# 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
*Antonio Natali*
1 Introduction ........ 2
2 JavaScript and Node.js ........ 2
2.1 JavaScript ........ 2
2.2 Node.js ........ 3
2.2.1 Frameworks and altJS. ........ 3
2.3 ES6 ........ 3
3 Functions ........ 4
3.1 Definitions of Functions and call with the `()` Operator ........ 4
3.2 Variables, hoisting and scoping ........ 5
3.2.1 Immediately Invoked Function Expression (`IIFE`). ........ 5
3.3 About `this` in JavaScript ........ 5
3.4 About `this` in Node ........ 7
3.5 The variable *arguments* ........ 7
3.6 Strict mode ........ 7
3.7 Arrow functions ........ 8
3.8 Function.prototype.bind() ........ 8
3.8.1 bind ........ 9
3.9 Invoking with with call() and apply() ........ 10
3.10 Lexical closures ........ 11
3.10.1 Closure ........ 11
3.11 Callbacks ........ 11
3.12 Continuation passing style (CPS) ........ 12
3.13 Tail recursion and Trampoline ........ 13
4 Higher order functions ........ 16
4.1 Compose ........ 16
5 Asynchronous Operations ........ 18
5.1 setTimeout ........ 18
5.1.1 setTimeout at work ........ 19
5.1.2 setTimeout and scope ........ 20
5.1.3 setTimeout and bind ........ 21
5.2 Calling a function (psuedo-)asynchronously ........ 21
5.3 The callback hell ........ 22
6 Prototypes and inheritance ........ 24
6.1 Led as object ........ 24
6.2 Button as observable ........ 25
6.3 Led as technology-independent object ........ 26
6.4 A Button-Led system ........ 27
6.5 A Button-Led system for HTML ........ 27
7 Node.js ........ 29
7.1 Standard input and count down ........ 29
7.2 Node module.exports and require ........ 29
7.3 Browserify ........ 30
7.4 Event emitters and Event listeners ........ 30

7.5 process.nextTick ........ 30
7.6 setImmediate ........ 31
7.7 The `fs` library ........ 32
7.8 The `net` library ........ 32
7.8.1 TcpServer. ........ 33
7.8.2 TcpClient. ........ 33
7.9 The `http` library ........ 34
7.9.1 HTTP server that returns the content of a `JSON` file. ........ 35
7.9.2 Http CRUD Server. ........ 35
7.9.3 cURL ........ 36
8 Promises ........ 38
8.1 From callbacks to promises (again) ........ 38
8.2 Promises in Node.js ........ 38
8.3 Promises: an example ........ 39
8.3.1 An example: file read. ........ 39
8.4 Beyond continuation passing. ........ 40
8.4.1 Sequencing ........ 41
8.5 Promises at work ........ 43
8.6 Settled Promises. ........ 45
9 Node on RaspberryPi ........ 46
9.1 Blink a Led with JavaScript ........ 46
55

# 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[1].

Node.js[2] (written and introduced by Ryan Dahl in 2009[3]) 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[4]) programming language.

JavaScript[5] 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<br>– 1997: ECMAScript standard is established<br>– 1999: ES3 comes out and IE5 is all the rage<br>– 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).<br>– 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<br>– 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 |

[1] For some experiment, connect to the |jsfiddle| site
[2] See |nodejs.org/api|, |Tutorial node.js| and |Node University site|.
[3] Visit Wikipedia node.js to know the history of Node.js.
[4] Recent browsers perform just-in-time compilation.
[5] See |JavaScript wiki| and |w3schools|

## 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[6].

| **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.<br>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 (see Subsection **??**). The event loop can be viewed as a queue of callbacks (see Subsection 3.11) 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 Frameworks and altJS.** Plain JavaScript and the *jQuery* library[7] 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[8] are aimed specifically at "taming" asynchronous callbacks by allowing them to be written in a more synchronous style.

Moreover, many JavaScript frameworks[9] 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[10] 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.

## 2.3 ES6

ECMAScript (ES6, formally called ECMAScript2015) does introduce features that represent the foundation for modern JavaScript applications. Many JavaScript environments (in particular borwsers 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.

[6] To use a Node application within a browser, you can use the |browserify| utility.
[7] 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.
[8] See the sites |altjs| , |async|
[9] See for example |Javascript frameworks| and |Comparison of JavaScript frameworks|.
[10] See for example |nodejs frameworks|.

## 3 Functions

A JavaScript function is a block of code designed to perform a particular task; it is executed when "something" invokes it [11], by using (for example) the `()` Operator.

JavaScript functions are objects that have both properties and methods; they can be used the same way as we use variables, in all types of formulas, assignments, and calculations.

### 3.1 Definitions of Functions and call with the `()` Operator

The example that follows shows different ways to define a JavaScript function and different ways to invoke it:

- Invoking as a Function (e.g. line 13, 30)
- Invoking with a Function Constructor (always created in the global scope) (e.g. line 19, 42)
- Invoking as a Method (e.g. line 44)

```
1 /*
2 * ====================================
3 * FunctionDefIntro.js
4 * ====================================
5 */
6 /*
7  *  A function that belongs to the global object
8  */
9 function toCelsius(fahrenheit) {
10     return (5/9) * (fahrenheit-32);
11 }
12 //INVOKE as a Function
13 console.log("toCelsius(75)" , toCelsius(75));
14 /*
15 * A function defined with JavaScript built-in Function constructor
16 * If a function invocation is preceded with the new keyword, it is a constructor invocation.
17 * Functions created with the Function constructor do not create closures to their creation contexts;
18 * they always are created in the global scope.
19 */
20 var mul = new Function("a", "b", "return a * b");
21 //INVOKE as a Function
22 console.log("mul(2,3)=" , mul(2,3) );
23 /*
24  * A (anonymous) function defined using an expression.
25  * Functions defined using an expression are not hoisted.
26  * Hoisting is JavaScript's default behavior of moving declarations to the top of the current scope.
27  */
28 var sum = function(a,b){ return a + b; } ;
29 //INVOKE as a Function
30 console.log("sum(2,3)=" , sum(2,3) );
31 /*
32  * A function defined as the property of an object, is called a method to the object.
33  * A function designed to create new objects, is called an object constructor.
34  */
35 Point2D = function(i,j){
36     this.name = "p2D("+i+","+j+")";  //property
37     this.getX = function(){ return i; } //method
38     this.getY = function(){ return j; } //method
39 }
40 //INVOKE with a Constructor
41 p = new Point2D(1,1);
42 //INVOKE as a Method
43 console.log("p", "name="+p.name + " getX()=" + p.getX() + " getY()=" + p.getY() );
```

**Listing 1.1.** `FunctionDefIntro.js`

[11] JavaScript function can be invoked in 4 different ways. Each method differs in how `this` is initialized. See Subsection 3.3 and |w3schools function invocation|.

## 3.2 Variables, hoisting and scoping

With reference to *variables*, bindings rules[12] 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) introduce new scopes. Thus JavaScript variables *lexically function-scoped*, and blocks are ignored. Moreover, JavaScript *hoists* all variable declarations, i.e. it moves them to the beginning of their direct scope. ES6 does introduce block-level, lexical bindings with `let` and `const` declarations.

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.

A simulation of block scoping is usually done with a pattern named IIFE[13] (Immediately Invoked Function Expression).

**3.2.1 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.

```
/*
* ====================================
* iife.js
* ====================================
*/
let entity = function(name) {
    return {
        getName: function() { return name; }
    };
}("book");
console.log("iife=", entity.getName()); //iife=book
```

**Listing 1.2.** `iife.js`

## 3.3 About `this` in JavaScript

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:

| | |
|---|---|
| **this** | a keyword that evaluates to the value of the *ThisBinding* of the current execution context" (ğ11.1.1). *ThisBinding* 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:

[12] *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.
[13] pronounced 'iffy'

**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 3.9)
  - The bind method (see Subsection 3.8)
  - ES6 Arrow functions (see exploringjs.com)

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.
- When a function is used as a *constructor* (with the `new` keyword), its new 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.

Let us start by using *?* in a top-level function:

```
function testThis(){
    return this;
}
var global = testThis();
console.log("testThis() " + global ); //testThis() [object global]
console.log("NUMBER_OF_PROCESSORS= " + global.process.env.NUMBER_OF_PROCESSORS ); //NUMBER_OF_PROCESSORS= 4 (in Node)
//[object Window] in a browser. See http://www.w3schools.com/jsref/obj_window.asp
```

**Listing 1.3.** `this global`

Next, we use *this* in a method of an object (literal):

```
age = 52;
person = {
        name : "Bob",
        age  : 35,
        getSomeAge : function(){
            return age;    //refers to the global age
        },
        getName : function(){
            console.log("person name=" + this.name)
            return this.name;
        },
        getAge : function(){
            console.log("person age=" + this.age)
            return this.age;
        }
}
console.log("person.age=" + person.age);              //35
console.log("person.getName()=" + person.getName());  //Bob
console.log("person.getSomeAge()=" + person.getSomeAge()); //52
```

**Listing 1.4.** `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`.

Finally, let us use *this* within a constructor function:

```
function student( name, age ){
    this.name   = name;
    this.age    = age;
```

```
    this.getName = function(){ return this.name; }
}
alice = new student( "alice", 22 );
console.log("method call=" + alice.getName()); //method call=alice
```

**Listing 1.5.** `this in a constructor`

### 3.4 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.addEventListener("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.)

### 3.5 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 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 *slice*.

### 3.6 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.

In normal code, `eval("var x;")` introduces a variable `x` into the surrounding function or the global scope. This means that, in general, in a function containing a call to eval every name not referring to an argument or local variable must be mapped to a particular definition at runtime (because that eval might have introduced a new variable that would hide the outer variable). In strict mode eval creates variables only for the code being evaluated, so eval can't affect whether a name refers to an outer variable or some local variable:

```
/*
* ====================================
* strictIntro.js
* ====================================
*/
// Whole-script strict mode syntax
//"use strict";

```

```
//Script mode per function syntax
function strictF(){
    "use strict";
    strictV = "strict mode local var"
}
//console.log("test:", strictF() ); //ReferenceError: strictV is not defined

//Eval in strict mode
var x = 17;
var evalX = eval("'use strict'; var x = 42; x");
console.assert(x === 17);
console.assert(evalX === 42);
```

**Listing 1.6.** `strictIntro.js`

For the details, see |JavaScript Strict mode|.

### 3.7 Arrow functions

Arrow functions are introduced in ES6 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.

```
/*
* ====================================
* arrowfun.js
* ====================================
*/
//No argument
let noArg = () => "hello from noArg"
console.log("noArg:" , noArg() ) //noArg: hello from noArg

//Single argument
let oneArg = v => v
console.log("oneArg:" , oneArg(3) ) //oneArg: 3

//More arguments
let moreArgs = (v1, v2) => v1 + v2
console.log("moreArgs:" , moreArgs(2,3) ) //moreArgs: 5

//With body
let withBody = (v1, v2) => { return v1 - v2; } //return must explicit
console.log("withBody:" , withBody(4,3) ) //withBody: 1

//IIFE
let thing = ( (name) => { //note the sourrounding ()
    return {
        getName: function() { return name; }
    };
} )("book");
console.log("iife=", thing.getName());  //iife= book
```

**Listing 1.7.** `Examples of arrow functions`

### 3.8 Function.prototype.bind()

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.

### 3.8.1 bind

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.

When bound function is called, it calls internal method [[Call]][a] 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.

[a] 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.

Let us show an example:

```
/*
 * ====================================
 * bind.js
 * ====================================
 */
x = 9;   //GLOBAL VARIABLE
/*
 * Literal object Point
 */
var Point = {
    x : 81,
    getX :  function() { return x; },
    getMyX : function() { return this.x; }
};
//New programmers might confuse the global x with module's property x
console.log( "Point.getX()=" + Point.getX() + " Point.getMyX()=" + Point.getMyX() );

var retrieveX = Point.getMyX;
console.log( "retrieveX=" + retrieveX() );  //9: The function gets invoked at the global scope

// Create a new function with 'this' bound to Point
var boundGetX = retrieveX.bind(Point);
console.log( "boundGetX=" + boundGetX() ); // 81: call a new function with 'this' bound to Point

/*
```

**Listing 1.8.** `bind.js`

The *bind()* method creates a new function that, when called, has its `this` keyword set to the provided value, with a given sequence of arguments preceding any provided when the new function is called.

The output is:

```
/*
Point.getX()=9 Point.getMyX()=81
retrieveX=9
boundGetX=81
*/
```

**Listing 1.9.** `bind.js result`

### 3.9 Invoking with with call() and apply()

The next example will show the case in which a function is invoked with a `Function`[14] Method. To understand the code, we recall that:

- When a function is called without an owner object, the value of `this` is the global object. In JavaScript there is always a default global object[15].
- In JavaScript, functions are objects that have properties and methods.
- `call()` and `apply()` are predefined JavaScript `Function` methods. Both methods can be used to invoke a function, and both methods must have the *owner* object as first parameter.

With *call()* or *apply()* we can set the value of `this`, and invoke a function as a new method of an existing object. The `Apply` and `Call` methods are two of the most often used Function methods in JavaScript: they allow us to borrow functions and set the `this` value (see Subsection 3.3) in function invocation.

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[16]

```
/*
* =====================================
* FunctionCallIntro.js
* =====================================
*/
/*
 * --------------------------------------
 * CALLING A FUNCTION via call, apply
 * --------------------------------------
 */
/*
 * When a function is called without an owner object, the value of this is the global object.
 * In JavaScript there is always a default global object.
 * In a web browser, the global object is the browser window.
 *
 * In JavaScript, functions are objects that have properties and methods.
 * call() and apply() are predefined JavaScript function methods.
 * Both methods can be used to invoke a function, and both methods must have the owner object
 * as first parameter.
 *
 * With call() or apply() we can set the value of this, and invoke a function as a
 * new method of an existing object.
 */
name = "globalName"
console.log("initial name=" , name ); //initial name= globalName

var getGlobalName = function( ){
    return name;
}
var getContextualName = function( ){
    return this.name;
}
Point2D = function(i,j){
    this.name = "p2D("+i+","+j+")";
    this.getX = function(){ return i; }
    this.getY = function(){ return j; }
}
console.log("getGlobalName()=" , getGlobalName() ); //getGlobalName()= globalName
console.log("getContextualName()=" , getContextualName() ); //getContextualName()= globalName
p = new Point2D(1,1) ;
```

[14] See |JavaScript Function|
[15] In a web browser, the global object is the browser window
[16] A variadic function takes varying number of arguments.

```
console.log("p.name=" , p.name ); //p.name= p2D(1,1)
/*
 * CALLS
 */
console.log("getContextualName call 1=" , getContextualName.call( p ) ); //getContextualName call 1= p2D(1,1)
console.log("getContextualName call 2=" , getContextualName.call( new Point2D(2,2) ) ); //getContextualName call 2=
      p2D(2,2)

/*
 * In JavaScript strict mode, the first argument becomes the value of this in the invoked function,
 * even if the argument is not an object.
 * In "non-strict" mode, if the value of the first argument is null or undefined,
 * it is replaced with the global object.
 */
```

**Listing 1.10.** `FunctionCallIntro.js`

### 3.10 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.

#### 3.10.1 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.

```
/*
* ====================================
* ClosureExample.js
* ====================================
*/
function addClosure(x){
   function incrementBy(y){
       return x + y
    }
    return incrementBy
}

var c1 = addClosure(10) ;
var c2 = addClosure(20) ;

console.log( c1(1) );
console.log( c2(1) );
```

**Listing 1.11.** *Execution of ClosureExample.js*

The result is:

```
/*
 11
 21
 */
```

**Listing 1.12.** *Execution of ClosureExample.js*

### 3.11 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[17], and therefore, the use of a callback function is also known as a *callback pattern*.

```
/*
 * ===================================
 * SynchCallback.js
 * ===================================
 */
var start = new Date;

function workAndCallback( callback ){
    var n = 0;
    while( n < 10 ){ n = new Date - start };
    callback( n );
 }

workAndCallback( function(k){ console.log("hello "+k) } ) //hello 10
```

**Listing 1.13.** `SynchCallback.js`

### 3.12 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

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:

```
/*
 * ===================================
 * Continuation.js
 * ===================================
 */
 function evalValue(){
    return 10;
 }
 function workNormal(){
    var v0 = 5;
    console.log( "v0=" + v0 ); //v0 = 5
    var n = evalValue( );
    //--------------------
    var k = n + v0 ;
    v0 = v0 -1;
    console.log( "k normal=" + k + " v0= " + v0 );
 }
  workNormal(); //k normal=15 v0= 4
```

**Listing 1.14.** `workNormal in Continuation.js`

[17] The term 'pattern' is used here to denote an established solution to a common problem.

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.

```
function workByContinuation( callback ){
    var n = evalValue( );
    callback( n );
 }
continuation = function(n){
    var k = n + v0 ;
    v0 = v0 -1;
    console.log( "k continuation=" + k + " v0= " + v0 );
}

 var v0 = 5;
 console.log( "v0=" + v0 ); //v0=4
 workByContinuation(
     function(n){  //callback
        var k = n + v0 ;
        v0 = v0 -1;
        console.log( "k continuation=" + k + " v0= " + v0 ); //k continuation=15 v0= 4
     }
  );
  console.log( "v0=" + v0 ); //v0=4
```

**Listing 1.15.** `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 8.4 we will see techniques and constructs to overcome CPS.

### 3.13 Tail recursion and Trampoline

Javascript (before ES6) does not implement tail call optimization[18]. 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 *factTrampolinedl* at line 57)

```
1 /*
2 * ====================================
3 * Trampoline.js
4 * ====================================
```

[18] ES6 will probably have support for tail call optimization.

```
5  */
6  /*
7   * Fact recursive
8   */
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 factTrampolined(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("factTrampolined(3)=" , factTrampolined(3) );  //6
66 /*
67 You can visualize the stack like a bouncing trampoline:
68
69   ---|---  ---|---  ---|---
70 ---      ---      ---
71 */
72 console.log("factTrampolined(15711)=" , factTrampolined(15711) );  //Infinity
```

**Listing 1.16.** `Trampoline.js`

Note that the call `factTrampolined(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`[19] in order to avoid the `Infinity` result. Moreover, we use (line 21) the *bind* operation (see Subsection 3.8) 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 1.17.** `TrampolineBigdata.js`

[19] The function `require` is defined by NodeJs only: see Subsection 7.2

## 4 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 3.11) that a *callback* is a functions that is passed to another (higher-order) function as argument.

### 4.1 Compose

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 1.18.** `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:

```
/*
square(0.5)= 0.25
squarePair(0.5)= [ 0.25, 'square' ]
sine(Math.PI/4)= 0.7071067811865475
sinePair(Math.PI/4)= [ 0.7071067811865475, 'sin' ]
COMPOSE_1= 0.4999999999999999
COMPOSE_2= [ 0.4999999999999999, 'square' ]
COMPOSE_3= [ NaN, 'square' ]
*/
```

**Listing 1.19.** `hofcompose.js`

## 5 Asynchronous Operations

In Section 3 we introduced the concept of (JavaScript) function as 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*. [20]

- 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 3.11) 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[21]; we have to leverage on a technology provided natively, such as `setTimeout` or `setInterval` that can be found in any JavaScript environment,

### 5.1 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<br>`long setTimeout(function f, unsigned long timeout, any args...)`<br>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>`. |
|---|---|

[20] WebWorkers do introduce multi-threading in JavaScript but without shared stated and with message-based interaction based on callbacks that are run from the event queue, like a conventional I/O.

[21] 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.

#### 5.1.1 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.

```
/*
 * ====================================
 * SetTimeoutBasic.js
 * ====================================
 */
var data; // global variable

showData = function() {
    console.log(data);
};
var setData = function() {
    setTimeout(function() {
        data = 10;
    }, 200);
};
function sequence() {
    setData();
    showData(); // here data is still undefined
}
```

**Listing 1.20.** A sequence call in *SetTimeoutBasic.js*

Let us introduce now two other asynchronous functions in CPS style (see Subsection 3.12): *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.

```
/*
 * CPS style
 */
setDataVal = function(v,dt,callback) {
    setTimeout(function() {
        data = v;
        callback(data);
    }, dt);
};

getDataVal = function(dt, callback) {
    setTimeout(function() {
        callback("getVal done=", data)
    }, dt);
}

function sequenceCPS() {
    var dt = 100;
    setDataVal(10, dt, showData );
    setDataVal(20, dt, function(){ showData(); } );
    setDataVal(30, dt, function(){ getDataVal( 0, console.log); } );
}
```

**Listing 1.21.** A sequence CPS call in *SetTimeoutBasic.js*

Finally we introduce a *main* function[22].

[22] We use here the Node `process` object - see |process.html| - to avoid the call of *main* if the current program is not our file.

```
function main(){
    console.log("data1=", data);
    sequence();    // shows undefined
    console.log("data2=", data);
    sequenceCPS(); // will shows 10 20 30
    console.log("data3=", data);
}

//Conditional call to main
if( process.argv[1].toString().includes("SetTimeoutBasic") ) main();
```

**Listing 1.22.** A main for *SetTimeoutBasic.js*

The global result is:

```
/*
data1= undefined
undefined
data2= undefined
data3= undefined
10
20
getVal done= 30
 */
```

**Listing 1.23.** *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`.

#### 5.1.2 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;
```

```
        console.log('Time elapsed:', end - start, 'ms');
    }, 500);

//here we keep the control for 800 msec
while( new Date - start < 800) { };
```

**Listing 1.24.** *SetTimeoutExample.js*

The result is:

```
/*
i should be 4: 4
i should be 4: 4
i should be 4: 4
x= 1
x= 2
x= 3
IIFE k= 1
IIFE k= 2
IIFE k= 3
Time elapsed: 3031 ms
*/
```

**Listing 1.25.** *Execution of SetTimeoutExample.js*

#### 5.1.3 setTimeout and bind

Another example using the *bind* operation (see Subsection 3.8) for a `Counter` object:

```
/*
* ====================================
* setTimeoutBind.js
* ====================================
*/
var val = 100;

function Counter() {
  this.val = 0;
  this.inc = function(){ this.val += 1; }
}

Counter.prototype.getVal = function() {
  setTimeout(this.show.bind(this), 500);
};

Counter.prototype.show = function() {
  console.log('I am a counter with val= ' + this.val );
  this.inc();
};

var count = new Counter();
count.getVal();
count.getVal();
```

**Listing 1.26.** *SetTimeoutBind.js*

### 5.2 Calling a function (psuedo-)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.

```
/*
* ====================================
* FactAsynch.js
```

```
* ===================================
*/
factAsynch = function( n, callback ){
    factIterAsynch(n,n,1,callback);
}
factIterAsynch = function( n, n0, v, callback ){
    var res = n*v; //ACCUMULATOR
    console.log( "factIterAsynch n0=" + n0 + " n=" + n, " v=" + v + " res=" + res);
    if( n == 1 ) callback( "factIterAsynch(" + n0 + ") RESULT="+res );
    else setTimeout( function(){ factIterAsynch(n-1, n0, res, callback) ; } , 0 );
 }

console.log("START");
console.log("CALL= ", factAsynch(4, console.log) );
factAsynch(6,console.log);
console.log("END");
```

**Listing 1.27.** *FactAsynch.js*

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

```
/*
START
factIterAsynch n0=4 n=4 v=1 res=4
CALL= undefined
factIterAsynch n0=6 n=6 v=1 res=6
END
factIterAsynch n0=4 n=3 v=4 res=12
factIterAsynch n0=6 n=5 v=6 res=30
factIterAsynch n0=4 n=2 v=12 res=24
factIterAsynch n0=6 n=4 v=30 res=120
factIterAsynch n0=4 n=1 v=24 res=24
factIterAsynch(4) RESULT=24
factIterAsynch n0=6 n=3 v=120 res=360
factIterAsynch n0=6 n=2 v=360 res=720
factIterAsynch n0=6 n=1 v=720 res=720
factIterAsynch(6) RESULT=720
*/
```

**Listing 1.28.** *Execution of FactAsynch.js*

### 5.3 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|).

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[23] to avoid the call of *main* if the current program is not our file.

```
/*
* ===================================
* sequencingHell.js
* ===================================
*/
var value; //GLOBAL
/*
 * A sets value asynchronously
 */
var A = function(callback) {
    setTimeout(function() {
```

[23] See |process.html|.

```
        console.log("A (before) value=", value );
        value = 10;
        callback();
    }, 200);
};
/*
 * B should run after A
 */
var B = function(callback) {
    value = value + 100;
    console.log("B value="+value );
    callback();
};
/*
 * C shows value without continuations
 */
var C = function() {
    console.log("C value=" , value );
};
/*
 * show shows value with continuations
 */
var show = function(callback) {
    console.log("show value=", value );
    callback();
};

/*
 * MAIN
 */
function main(){
    A( function(){
        B( function(){ C();}
    );
    });

    A( function(){
        show( function(){
                    B( function(){ C();}
            )}
        );
    })
}

//Conditional call to main
if( process.argv[1].toString().includes("sequencingHell") ) main();
```

**Listing 1.29.** `sequencingHell.js`

The result produce by *main* is:

```
/*
A (before) value= undefined
B value=110
C value= 110
A (before) value= 110
show value= 10
B value=110
C value= 110
*/
```

**Listing 1.30.** `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 3.12) .[24]

[24] Proper libraries, like *Async.js* (`https://github.com/caolan/async`), are introduced to avoid most "callback hell" scenarios in a Node code.

## 6 Prototypes and inheritance

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*.

### 6.1 Led as object

As an example, let us define a Led as an object that receives a name and a `guiId` at construction time:

```
/*
* ====================================
* Led.js
* ====================================
*/

/*
 * ********************************************
 * Led as a conventional 'object'
 * ********************************************
 */
//State (specific)
var Led = function(name, guiId){
    //Led Constructor: instance data
    this.name     = name;
    this.guiId    = guiId;
    this.ledState = false;
}

//Methods (shared)
Led.prototype.turnOn = function(){
    this.ledState = true;
}
Led.prototype.turnOff = function(){
    this.ledState = false;
    }
Led.prototype.switchState = function(){
    this.ledState = ! this.ledState;
    }
Led.prototype.getState = function(){
    return this.ledState;
}
Led.prototype.getName = function(){
    return this.name;
}
Led.prototype.getDefaultRep = function(){
    return this.name+"||"+ this.ledState
}
Led.prototype.showGuiRep = function(){
    if( typeof document != "undefined"){
        if( this.ledState ) document.getElementById(this.guiId).style.backgroundColor='#00FF33';
        else document.getElementById(this.guiId).style.backgroundColor='#FF0000';
    }
    else println( this.getDefaultRep() );
}

println = function ( v ){
    try{
        if( typeof document != "undefined" ) showMsg( 'outView', v+"<br/>" );
        else console.log( v );
    }catch(e){
        console.log( v );
    }
}
// EXPORTS
if(typeof document == "undefined") module.exports.Led = Led;
//To work is a browser, do: browserify Led.js -o LedBro.js
```

**Listing 1.31.** `Led.js`

## 6.2 Button as observable

In the following example, we will model a Button as an "observable" object that 'notifies' a set of possible 'registered' observers when it changes its state.

Let us introduce first of all an `Observable` that allows us to register two different types of computational entities:

- objects that implement the method *update*;
- functions (or better closures) that play the role of a "callbacks"

```
 * *********************************************
 * Observable prototype
 * *********************************************
 */
Observable = function(){
    this.nobs       = 0;
    this.nobsfunc   = 0;
    this.observerFunc = [ ];
    this.observer   = [ ];
    this.register   = function(obs){
        //println(" Observable register " + this.nobs);
        this.observer[this.nobs++] = obs;
    }
    this.registerFunc = function(func){
        //println(" Observable registerFunc " + this.nobs + " " + func);
        this.observerFunc[this.nobsfunc++] = func;
    }
    this.notify = function(){
        for(var i=0;i < this.observer.length;i++){
            //console.log(" Observable update " + this.observer[i] );
            this.observer[i].update();
        }
        for(var i=0;i < this.observerFunc.length;i++){
```

**Listing 1.32.** `ButtonObservable.js: the Observable`

A `Button` can now be introduced as an object that inherits from `Observable`:

```
 * ***********************************************************
 * Button as an 'object' that inherits from Observable
 * and works also as an event emitter.
 * project it.unibo.bls2016.qa
 * ***********************************************************
 */
var EventEmitter = require('events').EventEmitter;

function Button ( name ){
    this.emitter = new EventEmitter();
    this.name    = name;
    this.evId    = "pressed";
    this.count   = 0;
}
```

**Listing 1.33.** `ButtonObservable.js: the Button`

Note that the Button can work also as a Node.js event emitter: see Subsection 7.2 and Subsection 7.4. The operations provided by the Button can be defined as follows:

```
//Shared
Button.prototype     = new Observable();
Button.prototype.press = function(level){
    //console.log(" Button press " + level );
    this.notify();
    this.emitEvent() ;
}
Button.prototype.emitEvent = function(){
    console.log(" Button emits " + this.evId + " count=" + this.count++);
    this.emitter.emit(this.evId,'buttonPressed') ;
```

```
}
Button.prototype.getEmitter = function(){
    return this.emitter;
}
Button.prototype.setHandler = function( handler ){
    this.emitter.on(this.evId, handler );
}
Button.prototype.removeHandler = function( handler ){
```

**Listing 1.34.** `ButtonObservable.js: the Button`

The operation `press` notifies (executes) all the registered observers and emits an event with `id='pressed'`.

### 6.3 Led as technology-independent object

A Led can be introduced as a high-level entity that is configured at construction time with a specific implementation.

```
/*
* ===================================
* Led.js
* Led as a conventional 'object'
* ********************************************
*/
var Led = function(name, ledImpl){
    this.name     = name;
    this.ledImpl  = ledImpl;
    this.ledState = 0;
    this.turnOff();
}
Led.prototype.turnOn = function(){
    this.ledState = 1;
    this.ledImpl.turnOn();
}
Led.prototype.turnOff = function(){
    this.ledState = 0;
    this.ledImpl.turnOff();
}
Led.prototype.switchState = function(){
    this.ledState = (this.ledState + 1) % 2;
    if( this.ledState == 0 ) this.ledImpl.turnOff();
    else this.ledImpl.turnOn();
}
Led.prototype.getState = function(){
    return this.ledState;
}
Led.prototype.getName = function(){
    return this.name;
}
Led.prototype.getDefaultRep = function(){
    return this.name+"||"+ this.ledState
}
// EXPORTS
module.exports.Led = Led;
//To work is a browser, do: browserify Led.js -o LedBro.js
```

**Listing 1.35.** `Led.js: the Button`

A Led mock can be defined as follows:

```
/*
* ===================================
* LedImplPc.js
* Led implementation on a PC
* ===================================
*/
var LedImplPc = function( name ){
    this.name     = name;
    this.ledState = 0;
```

```
}
LedImplPc.prototype.turnOn = function(){
    this.ledState = 1;
    console.log("LED " + this.name + " ON");
}
LedImplPc.prototype.turnOff = function(){
    this.ledState = 0;
    console.log("LED " + this.name + " OFF");
}
// EXPORTS
if(typeof document == "undefined") module.exports.LedImplPc = LedImplPc;
```

**Listing 1.36.** `LedImplPc.js: the Button`

## 6.4 A Button-Led system

A Button-Led system can be defined as a test to be run with *nodeunit* (command: `nodeunit testBlsObservableObj.js` ):

```
/*
 * =======================================================
 * testPCBlsObservableObj.js for nodeunit
 * project it.unibo.bls2016.qa
 * USAGE: nodeunit testBlsObservableObj.js
 * =======================================================
 */
var LedHL   = require("./Led");
var ButtonHL = require("./ButtonObservable");

configureForPc = function(){
    var LedOnPc  = require("./LedImplPc");
    var l1pc     = new LedOnPc.LedImplPc("l1pc"); //a mock object
    //Global variables
    b1  = new ButtonHL.Button( 'b1' );
    l1  = new LedHL.Led("l1pc",l1pc);
//handler of the event 'pressed' emitted by the button
    b1.setHandler(
        function(msg){
            console.log(" handler msg=" + msg +" when led=" + l1.getState());
            l1.switchState();
    } );
    //Set another event handler (for the HL button as an emitter)
    b1.getEmitter().on( b1.evId, function(v){console.log(" %%% HL PC event handler: event content=" + v); } )
}
exports.testObservableButton=function(test){
    configureForPc();
    test.expect(3);       //we expect 3 run
    test.ok( l1.getState() == 0, "testObservableButton initial");
    b1.press();
    test.ok( l1.getState() == 1, "testObservableButton press 1");
    b1.press();
    test.ok( l1.getState() == 0, "testObservableButton press 2");
    test.done();
}
```

**Listing 1.37.** `testPCBlsObservableObj.js`

## 6.5 A Button-Led system for HTML

A version to be used within an HTML page can be:

```
/*
 * ====================================================
 * blsOopHtml.js
 * ====================================================
*/
var ButtonHL = require("./ButtonObservable");
```

```
var LedHL   = require("./Led");
// BUTTON HIGH-level
var b1       = new ButtonHL.Button( "b1" );

configureForHtml = function(){
    var LedHtml  = require("./LedImplGui");
    var l1       = new LedHtml.LedImplGui("ledGuiId");
    l1           = new LedHL.Led("l1",l1);
    b1.setHandler(
        function(msg){
            console.log("handler " + msg +" led=" + l1.getState());
            l1.switchState();
    } );
}
//EXPORT a function buttonPress
window.buttonPress =function(){ b1.press(); };

//Main
console.log('blsOopHtml STARTS' );
configureForHtml();
//RAPID CHECK
pressTheButton = function(){ b1.press(); }
for( i=1; i<=5; i++ ) {
    setTimeout( pressTheButton, 500*i );
}
console.log('blsOopHtml ENDS' );
//To work is a browser, run: browserify blsOopHtml.js -o blsOopHtmlBro.js
```

**Listing 1.38.** `blsOopHtml.js`

The led implementation is:

```
/*
* ====================================
* LedImplGui.js
* Led implementation for HTML
* ====================================
*/
 var LedImplGui = function( name ){
        this.name     = name;
        this.ledState = 0;
 }
 LedImplGui.prototype.turnOn = function(){
    this.ledState = 1;
    document.getElementById(this.name).style.backgroundColor='#00FF33';
  }
 LedImplGui.prototype.turnOff = function(){
    this.ledState = 0;
    //console.log(this.name + " OFF " + document.getElementById(this.name));
    document.getElementById(this.name).style.backgroundColor='#FF0000';
    document.getElementById(this.name)
 }
// EXPORTS
module.exports.LedImplGui = LedImplGui;
```

**Listing 1.39.** `LedImplGui.js`

The HTML page could be:

```
<html>
<body>
<h3>blsObjObservable</h3>
<!-- LED -->
<div id="ledGuiId" style="height:20px; width:3%; position: absolute; background-color:#00FF33"></div><br/><br/>
<!-- BUTTON -->
<div><button onclick="buttonPress()">BUTTON</button></div> <!-- buttonPress is made visible by our code -->
</body>
<script src="./blsOopHtmlBro.js"></script> <!-- buttonPress is made visible by our code -->
</html>
```

**Listing 1.40.** `blsObjObservable.html`

# 7 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 deign and understanding of the code.

## 7.1 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.

```
/*
* ====================================
* stdinExample.js
* ====================================
*/
// prepare for input from terminal
process.stdin.resume();
// when receive data do ...
process.stdin.on('data', function (data) {
    //console.log("input=" + data);
    goon = false;
});

/*
 * Count down
 */
var v0  = 10;
var goon = true;

var count = function(){
    if( v0 > 0 && goon ){
        v0 = v0 - 1;
        console.log("v0=" + v0);
        setTimeout( count, 1000 );
    }else if( ! goon ) console.log("counte down stopped");
};
//main
console.log("START with v0=" + v0 );
setTimeout( count, 1000 );
console.log("END" );
```

**Listing 1.41.** 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.

## 7.2 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.

```
/*
* ====================================
* NodeModuleIntro.js
* ====================================
*/
//var exports = module.exports = {}; //default

 console.log("initial this=" , this);
 //console.log("initial module" , module); //shows all module info
```

```
console.log("initial exports=" , exports);
console.log("initial module.exports=" , module.exports);
exports.SimpleMessage = 'Hello world';
console.log("current exports=" , exports);

/*
initial this= {}
initial exports= {}
initial module.exports= {}
current exports= { SimpleMessage: 'Hello world' }
*/
```

**Listing 1.42.** `NodeModuleIntro.js`

The function `require` is a synchronous operation[25] that locate a module, given its path, and loads its code into the program.

### 7.3 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:

```
window.op =function(){
  ...;
};
```

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

### 7.4 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

### 7.5 process.nextTick

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

```
/*
 * ====================================
 * nextTickExample.js
 * ====================================
 */
```

[25] It should be used with care in asynchronous applications, like an HTTP server (see Subsection 7.9).

```
f = function (data, callback) {
    process.nextTick( function(){
        callback(data) ;
    } );
};
//main
console.log("START" );
f(3, function(v){console.log('OUTPUT=' + v);} );
f(5, function(v){console.log('OUTPUT=' + v);} );
console.log("END" );

/*
START
END
OUTPUT=3
OUTPUT=5
*/
```

**Listing 1.43.** `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.

### 7.6 setImmediate

Callbacks passed to `setImmediates` 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. Immediates 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. |

```
/*
 * ====================================
 * setImmediateExample.js
 * ====================================
 */
f = function (data, callback) {
    process.nextTick( function(){
        callback(data) ;
    } );
};
//main
console.log("START" );
setImmediate( function(){console.log('IMMEDIATE1' );} );
f(3, function(v){console.log('OUTPUT=' + v);} );
setImmediate( function(){console.log('IMMEDIATE2' );} );
f(5, function(v){console.log('OUTPUT=' + v);} );
setImmediate( function(){console.log('IMMEDIATE3' );} );
console.log("END" );

/*
START
END
OUTPUT=3
OUTPUT=5
```

```
IMMEDIATE1
IMMEDIATE2
IMMEDIATE3
*/
```

**Listing 1.44.** `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.

### 7.7 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:

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

**Listing 1.45.** `a.json`

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

```
/*
* =====================================
* ReaderWithNode.js
* =====================================
*/
var fs = require('fs');

readFileCallback = function(err, data){
    if (err) console.log("error " + err);
    else console.log( data );
}

try{
    console.log("START" );
    fs.readFile('./a.json', 'utf8', readFileCallback);
    console.log("END" );
}catch(err){
    console.log("error " + err);
}

/*
START
END
{ "name":"John", "age":30, "city":"New York"}
{ "name":"Alice", "age":24, "city":"Los Angeles"}
*/
```

**Listing 1.46.** `ReaderWithNode.js`

The output shows that the reading is asynchronous. Once completed, the *readFile* operation calls the given *readFileCalback* 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`.

### 7.8 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.

### 7.8.1 TcpServer.

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

```
/*
* ====================================
* TcpServerNode.js
* ====================================
*/
var net = require('net');
var port = 23;
var server = net.createServer(
    function(conn) {
        console.log('connected');
        conn.write('TCP node server READY' );
        conn.on('data', function (data) {
            console.log(data + ' from ADDR=' + conn.remoteAddress + ' PORT=' + conn.remotePort);
            conn.write('Echo:' + data );
        });
        conn.on('close', function() {
        console.log('client closed connection');
    });
}).listen(port, function(){ console.log('bound to port '+port); } );

internalWork = function( ){
    console.trace( );
    setImmediate( function(){ internalWork( ) ; } );
}
//main
console.log('TcpServerNode START ' );
internalWork( );

/*
USAGE:
netcat 192.168.251.1 8050
telnet 127.0.0.1 (on port 23)
*/
```

**Listing 1.47.** `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.

### 7.8.2 TcpClient.

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

```
/*
* ====================================
* TcpClientNode.js
* ====================================
*/
var net = require('net');
var client = new net.Socket();
client.setEncoding('utf8');

```

```
// connect to server
client.connect ('8050','localhost', function () {
    console.log('connected to server');
    client.write('Line1');
});

// prepare for input from terminal
process.stdin.resume();
// when receive data, send to server
process.stdin.on('data', function (data) {
    client.write(data);
});
// when receive data back, print to console
client.on('data',function(data) {
    console.log(data);
});
// when server closed
//client.on('close',function() {
//console.log('connection is closed');
//});
```

**Listing 1.48.** `TcpClientNode.js`

### 7.9 The `http` library

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

```
/*
* ===================================
* HttpServerBase.js
* ===================================
*/
var http = require("http");

http.createServer(function(request, response) {
    //The request object is an instance of IncomingMessage (a ReadableStream and it's also an EventEmitter)
    var method = request.method;
    var url    = request.url;
    console.log("Server request method=" + method + " url="+ url);
    if (request.method === 'GET' && request.url === '/') {
        response.writeHead(200, {"Content-Type": "text/plain"});
        response.write("Hello World from the server");
        response.end();
    }
}).listen( 8080, function(){ console.log('bound to port 8080');} );

console.log('Server running on 8080');
```

**Listing 1.49.** *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:

```
/*
Server running on 8080
bound to port 8080
Server anwsers to GET url=/
Server anwsers to GET url=/favicon.ico
*/
```

**Listing 1.50.** *Execution of HttpServerBase.js*

### 7.9.1 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:

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

**Listing 1.51.** `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.

```
/*
* ====================================
* HttpServerFile.js
* ====================================
*/
var http = require('http');
var fs  = require('fs');

http.createServer(function (request, response) {
    var method = request.method;
    var url    = request.url;
    console.log("Server request method=" + method + " url="+ url);
    if (request.method === 'GET' && request.url === '/') {
        response.writeHead(200, {'Content-Type': 'text/plain'});

        //Read JSON file and use callback to define what to do with its contents
        fs.readFile('./a.json', 'utf8', function(err, data) {
            if (err){
                response.write('Could not find or open file for reading\n');
            }else{
                console.log("data=" + data );
                response.write(data);
            }
            response.end();
        });
    }
}).listen(8123, function() { console.log('bound to port 8123');});

console.log('Server running on 8123/');
```

**Listing 1.52.** *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:

```
/*
Server running on 8123/
bound to port 8123
Server request method=GET url=/favicon.ico
Server request method=GET url=/
data={ "name":"John", "age":30, "city":"New York"}
{ "name":"Alice", "age":24, "city":"Los Angeles"}

Server request method=GET url=/favicon.ico
 */
```

**Listing 1.53.** *Execution of HttpServerFile.js*

### 7.9.2 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.

```
/*
* ====================================
* HttpServer.js
* ====================================
*/
var http = require("http");
var dataStore = []; //Array of Buffers

http.createServer(function(request, response) {
    //The request object is an instance of IncomingMessage (a ReadableStream; it's also an EventEmitter)
    var headers  = request.headers;
    var method   = request.method;
    var url      = request.url;
    if( method == 'GET'){
        dataStore.forEach( function(v,i){
            response.write( i + ")" + v + "\n")
        });
        response.end();
    }//if GET
    if( method == 'POST'){
        var item = '';
        request.setEncoding("utf8"); //a chunk is a utf8 string instead of a Buffer
        request.on('error', function(err) {
            console.error(err);
          });
        request.on('data', function(chunk) { //a chunck is a byte array
            item = item + chunk;
          });
        request.on('end', function() {
                //dataStore = Buffer.concat(dataStore).toString();
                dataStore.push(item);
                console.log("dataStore=" + dataStore);
                response.on('error', function(err) { console.error(err); });
                response.statusCode = 200;
                response.setHeader('Content-Type', 'application/json');
                //response.writeHead(200, {'Content-Type': 'application/json'}) //compact form
                var responseBody = {
                  //headers: headers, //comment, so to reduce output
                  method: method,
                  url:   url,
                  dataStore:  dataStore
                };
                response.write( JSON.stringify(responseBody) );
                response.end();
                //response.end(JSON.stringify(responseBody)) //compact form
          });
    }//if POST
}).listen( 8080, function(){ console.log('bound to port 8080');} );

console.log('Server running on 8080');


/*
Server running on 8080
bound to port 8080
Server anwsers to GET url=/
Server anwsers to GET url=/favicon.ico
*/
```

**Listing 1.54.** Execution of `HttpServer.js`

### 7.9.3 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:

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

```
    {"method":"POST","url":"/","dataStore":["d1","d2"]}
curl -d d3 http://localhost:8080
    {"method":"POST","url":"/","dataStore":["d1","d2","d3"]}

curl -d data1 localhost:8080 //GET
    0)d1
    1)d2
    2)d3
```

# 8 Promises

The term *promise* comes from Daniel P. Friedman from 1976. The core idea behind promises is that a promise represents the result of an asynchronous operation. From MDN we read:

| | |
|---|---|
| **Promise** | A Promise is a proxy for a value not necessarily known when the promise is created. It allows us to associate handlers with an asynchronous action's eventual success value or failure reason. This lets asynchronous methods return values like synchronous methods: instead of immediately returning the final value, the asynchronous method returns a promise to supply the value at some point in the future. |

Thus, a promise is an object which is used for deferred and asynchronous computations. A promise represents an operation that hasn't completed yet, but is expected in the future. Promises are a way of organizing asynchronous operations in such a way that they appear synchronous.

More generally, promises allow us to handle "tricky" things (that might include asynchronous data, or null values, or something else) with a consistent approach.

## 8.1 From callbacks to promises (again)

In the earlier days of Node, asynchronous functionality was facilitated through the use of ***promises***. In the earlier Node implementation, a promise was an object that emitted exactly two events: *success* and *error*. A promise object ensured that the proper functionality was performed whenever the event finished - either the results could be manipulated, or the error processed.

The promise object was pulled from Node in version 0.1.30. As Ryan Dahl noted at the time, the reasoning was:

| | |
|---|---|
| **No promises** | Because many people only want a low-level interface to file system operations that does not necessitate creating an object, while many other people want something like promises but different in one way or another. So instead of promises we'll use last argument callbacks and consign the task of building better abstraction layers to user libraries. |

Rather than the promise object, Node incorporated the ***last argument callbacks***. All asynchronous methods feature a callback function as the last argument with a commonly used signature:

```
function(err, response){ ... }
```

The first argument in this callback function is always an error object.

Nowadays, Node follows the same JavaScript tour: from *Callbacks* to *Promises* to *Generators* to *Async/await* (ECMASCRIPT 7)

## 8.2 Promises in Node.js

Let us recall here some basic terminology:

- *promise*: is an object or function with a `then` method whose behavior conforms to this specification.
- *thenable*: is an object or function that defines a `then` method.
- *value*: is any legal JavaScript value (including `undefined`, a `thenable`, or a `promise`).
- *exception*: is a value that is thrown using the `throw` statement.
- *reason*: is a value that indicates why a promise was rejected.

A promise is in one of three different states:

- *pending*: The initial state of a promise.
- *fulfilled*: The state of a promise representing a successful operation.
- *rejected*: The state of a promise representing a failed operation.

Once a promise is fulfilled or rejected, it is immutable (i.e. it can never change again).

From https://github.com/.../states-and-fates.md we read:

**SF**

> **States**
> Promises have three possible mutually exclusive states: fulfilled, rejected, and pending.
>
> - A promise is fulfilled if promise.then(f) will call f "as soon as possible."
> - A promise is rejected if promise.then(undefined, r) will call r "as soon as possible."
> - A promise is pending if it is neither fulfilled nor rejected.
>
> We say that a promise is settled if it is not pending, i.e. if it is either fulfilled or rejected. Being settled is not a state, just a linguistic convenience.
>
> **Fates**
> Promises have two possible mutually exclusive fates: resolved, and unresolved.
>
> - A promise is resolved if trying to resolve or reject it has no effect, i.e. the promise has been "locked in" to either follow another promise, or has been fulfilled or rejected.
> - A promise is unresolved if it is not resolved, i.e. if trying to resolve or reject it will have an impact on the promise.
> - A promise can be "resolved to" either a promise or thenable, in which case it will store the promise or thenable for later unwrapping; or it can be resolved to a non-promise value, in which case it is fulfilled with that value.
>
> **Relating States and Fates**
> A promise whose fate is resolved can be in any of the three states:
>
> - Fulfilled, if it has been resolved to a non-promise value, or resolved to a thenable which will call any passed fulfillment handler back as soon as possible, or resolved to another promise that is fulfilled.
> - Rejected, if it has been rejected directly, or resolved to a thenable which will call any passed rejection handler back as soon as possible, or resolved to another promise that is rejected.
> - Pending, if it has been resolved to a thenable which will call neither handler back as soon as possible, or resolved to another promise that is pending.
>
> A promise whose fate is unresolved is necessarily pending.
> These relations are recursive, e.g. a promise that has been resolved to a thenable which will call its fulfillment handler with a promise that has been rejected is itself rejected.

The primary way of interacting with a promise is through its `then` method, which registers callbacks to receive either a promise's eventual value or the reason why the promise cannot be fulfilled. Promises also have a `catch` method that behaves the same as `then` when only a rejection handler is passed. Each call to `then()` or `catch()` *creates and returns another promise.* This second promise is resolved only once the first has been fulfilled or rejected.

New promises can be created in JavaScript ES6 by using a `Promise` constructor with a single argument: a function called the *Executor* that contains the code to initialize the promise. The *Executor* has two functions as arguments:

- the `resolve()` function, that is called when the executor has finished successfully to signal that the promise is ready to be resolved.
- the `reject()` function, that indicates that the executor has failed.

## 8.3 Promises: an example

### 8.3.1 An example: file read.

In the example that follows, the asynchronous read of a file (`fs.readFile`) is wrapped into a promise:

```
/*
* ===================================
* PromiseIntroFileRead.js
* ===================================
*/
var fs = require('fs');


var readAFile = function(fName){
//The native Node.js fs.readFile() asynchronous call is wrapped in a promise.
   return new Promise(
```

```
        function(resolve,reject){    //Executor
            console.log(" %%% readAFile promise starts for " + fName);
            fs.readFile(              //ansynch op
                fName, {encoding:"utf8"},
                function(err,contents){ //callback
                    if(err){ reject(err); return; }
                    console.log(" %%% readAFile promise resolving for " + fName);
                    resolve(contents); //triggers an asynch op
                }
            );
        });
}
// Main
main = function() {
    console.log("START");
    let promise = readAFile('anum.txt');
    let res    = promise.then( console.log );
    console.log("res=" + res); //res=[object Promise]
    promise.then( function(v){ console.log("promise then="+v); } );
    promise    = readAFile('nonexisting.txt');
    promise.catch( function(err){ console.log("promise err=",err.message);});
    console.log("END");
}

module.exports = { readAFile };
//Conditional call to main (to faciliatate importing of the module)
if( process.argv[1].toString().includes("PromiseIntroFileRead") ) main();

/*
START
        %%% readAFile promise starts for anum.txt
res=[object Promise]
        %%% readAFile promise starts for nonexisting.txt
END
promise err= ENOENT: no such file or directory, open 'C:\...\nonexisting.txt'
        %%% readAFile promise resolving for anum.txt
5
promise then=5
*/
```

**Listing 1.55.** `PromiseIntroFileRead.js`

The results show that each call to `then()` or `catch()` creates a new job (in a separate job queue that is reserved strictly for promises[26]) to be executed when the promise is resolved.

## 8.4 Beyond continuation passing.

Let us suppose to solve with asynchronous functions the problem of *evaluating the factorial of the number read from a file.*

A solution based on callbacks, in continuation passing style (see Subsection 3.12) can be expressed as follows:

```
/*
* ===================================
* ReadEvalCallBack.js
* contintuation passing style
* ===================================
*/
var fs = require('fs');

factIterAsynch = function( n, n0, v, callback ){
    var res = n*v; //ACCUMULATOR
    console.log( "factIterAsynch n0=" + n0 + " n=" + n, " v=" + v + " res=" + res);
    if( n == 1 ) callback(res); //callback( null, "factIterAsynch(" + n0 + ") RESULT="+res );
    else setImmediate( function(){ factIterAsynch(n-1, n0, res, callback) ; } );
}

```

[26] The precise details of this second job queue aren't important for understanding how to use promises.

```
var readEvalFactFile = function(fName, callback){
    fs.readFile(fName,
           function(err, val) {
               if (err) { throw err; }
               console.log("readEvalFactFile from file=" + fName);
               factIterAsynch( val, val, 1,
                      function( res) { callback(res); });
           });
};
//Utility
showIntValue = function( v, msg ){
    var tv = typeof v;
    msg   = msg || tv + "(" + v + ") >>>";
    console.log( msg, v );
    return Number(v);
}

/*
 * Main
 */
main = function() {
    console.log("START");
    readEvalFactFile("anum.txt",
        function(v){ showIntValue(v, "FACT="); }
    );
    //launch anther to show interleaving
    factIterAsynch( 10,10,1,function(v){ showIntValue(v, "FACT(10)="); });
    console.log("END");
};

module.exports = { factIterAsynch, showIntValue };
//Conditional call to main (to faciliatate importing of the module)
if( process.argv[1].toString().includes("ReadEvalCallBack") ) main();

/*
START
factIterAsynch n0=10 n=10 v=1 res=10
END
factIterAsynch n0=10 n=9 v=10 res=90
factIterAsynch n0=10 n=8 v=90 res=720
factIterAsynch n0=10 n=7 v=720 res=5040
factIterAsynch n0=10 n=6 v=5040 res=30240
factIterAsynch n0=10 n=5 v=30240 res=151200
factIterAsynch n0=10 n=4 v=151200 res=604800
factIterAsynch n0=10 n=3 v=604800 res=1814400
readEvalFactFile from file=anum.txt
factIterAsynch n0=5 n=5 v=1 res=5
factIterAsynch n0=10 n=2 v=1814400 res=3628800
factIterAsynch n0=5 n=4 v=5 res=20
factIterAsynch n0=10 n=1 v=3628800 res=3628800
FACT(10)= 3628800
factIterAsynch n0=5 n=3 v=20 res=60
factIterAsynch n0=5 n=2 v=60 res=120
factIterAsynch n0=5 n=1 v=120 res=120
FACT= 120
*/
```

**Listing 1.56.** `ReadEvalCallBack.js`

### 8.4.1 Sequencing

To reintroduce the flavour of sequential computations, we can recur to higher-order functions able to force the execution of a sequence of functions with argument (the first always a callback):

```
/*
* ====================================
* ReadEvalCbkSequential.js
* sequencing by using a Queue
* ====================================
*/
var fs      = require('fs');
```

```
var rdevcbck = require('./ReadEvalCallBack');

var beginWork = function( callback ){
    console.log(" %%% beginWork" );
    callback( "anum.txt");
}

var readEvalFactFile = function(callback, fName ){
    console.log(" %%% readEvalFactFile from file=" + fName);
    fs.readFile(fName,
            function(err, val) {
                if (err) { throw err; }
                console.log(" %%% done readEvalFactFile from file=" + fName + " val=" + val);
                callback( val );
            });
};

var factAsync = function(callback, v){
    console.log(" %%% factAsync v="+ v );
    rdevcbck.factIterAsynch( v,v,1,callback ) ;
}

var showResult = function(callback, v){
    rdevcbck.showIntValue( v, "RESULT" );
    callback();
}

var execSeqWithArgs = function(funcs, scope) {
    (function next() {
        if(funcs.length > 0) {
            var f = funcs.shift();
            //console.log("f=",f);
            //console.log("next arguments=",arguments); //arguments of next
            var arg = Array.prototype.slice.call(arguments, 0);
            //console.log("arg=",arg);
            var aa = [next].concat(arg);
            //console.log("aa=",aa);
            f.apply(scope, aa);
//            f.apply(scope, [next].concat(Array.prototype.slice.call(arguments, 0)));
        }else console.log("nothing to do");
    })();  //define and invoke
};


//Main
main = function() {
    console.log("START");
    execSeqWithArgs([beginWork, readEvalFactFile, factAsync, showResult], undefined);
    //launch anther to show interleaving
    factIterAsynch( 10,10,1,function(v){ showIntValue(v, "FACT(10)="); });
    console.log("END");
};

//module.exports = { };
//Conditional call to main (to faciliatate importing of the module)

if( process.argv[1].toString().includes( "ReadEvalCbkSequential" ) ) main();
```

**Listing 1.57.** `ReadEvalCbkSequential.js`

The function `execSeqWithArgs` executes one after the sequence of functions given in its input array. The result is:

```
/*
START
        %%% beginWork
        %%% readEvalFactFile from file=anum.txt
factIterAsynch n0=10 n=10 v=1 res=10
END
factIterAsynch n0=10 n=9 v=10 res=90
factIterAsynch n0=10 n=8 v=90 res=720
factIterAsynch n0=10 n=7 v=720 res=5040
factIterAsynch n0=10 n=6 v=5040 res=30240
```

```
factIterAsynch n0=10 n=5 v=30240 res=151200
factIterAsynch n0=10 n=4 v=151200 res=604800
factIterAsynch n0=10 n=3 v=604800 res=1814400
       %%% done readEvalFactFile from file=anum.txt val=5
       %%% factAsync v=5
factIterAsynch n0=5 n=5 v=1 res=5
factIterAsynch n0=10 n=2 v=1814400 res=3628800
factIterAsynch n0=5 n=4 v=5 res=20
factIterAsynch n0=10 n=1 v=3628800 res=3628800
FACT(10)= 3628800
factIterAsynch n0=5 n=3 v=20 res=60
factIterAsynch n0=5 n=2 v=60 res=120
factIterAsynch n0=5 n=1 v=120 res=120
RESULT 120
nothing to do
*/
```

**Listing 1.58.** Result of ReadEvalCbkSequential.js

### 8.5 Promises at work

A solution based on promises can be introduced by extending our previous example of Subsection 8.3 as follows:

```
/*
* ====================================
* PromiseIntroReadEval.js
* ====================================
*/
var fpromise = require('./PromiseIntroFileRead');

function factPromise( n ){
    return new Promise(
        function(resolve,reject){    //Executor
            console.log(" %%% factPromise promise starts for n=" + n);
            factIterAsynch( n,n,1,
                function(res){    //callback of factIterAsynch
                    console.log(" %%% factPromise promise resolving for n=" + n);
                    resolve(res);
                }
        );
    });
}
factIterAsynch = function( n, n0, v, callback ){
    var res = n*v; //ACCUMULATOR
    console.log( "factIterAsynch n0=" + n0 + " n=" + n, " v=" + v + " res=" + res);
    if( n == 1 ) callback(res);
    else setImmediate( function(){ factIterAsynch(n-1, n0, res, callback) ; } );
}
//Utility
 showIntValue = function( v, msg ){
        var typev = typeof v;
        msg      = msg || typev + "(" + v + ") >>>";
        console.log( msg, v );
        return Number(v);
    }
//Error handlers
process.on("unhandledRejection", function(reason, promise) {
    console.log(" unhandledRejection " + promise + " reason=" + reason.message) ;
});
process.on("rejectionHandled", function(promise) {
    console.log(" rejectionHandled", promise) ;
});
process.on("uncaughtException", (err) => {
    console.log(" uncaughtException:", err.message) ;
});

readAndEval = function(){
    let filePromise = fpromise.readAFile('anum.txt');     //(1)
    //read the file
```

```
    let factProm = filePromise.then( factPromise );       //(2)
    //evaluate the factorial of the file content
    factProm.then( function(v){ showIntValue(v,"FACT=" );} ); //(3)
}
//Main
main = function(){
    console.log("START");
    readAndEval();
    //attempt to read an nonexisting file
    filePromise = fpromise.readAFile('nonexisting.txt');  //(4)

    //launch another factIterAsynch
    factIterAsynch(10,10,1,                               //(5)
            function(v){ showIntValue(v,"FACT(10)=" ); } );
    console.log("END");
}
if( process.argv[1].toString().includes("PromiseIntroReadEval") ) main();
```

**Listing 1.59.** `PromiseIntroReadEval.js`

This program:

1. calls a function `readAndEval` that: `(1)` creates a promise to read a file; `(2)` creates another promise to evaluate in asynchronous way the factorial of the number given as the result of the asynchronous read from the file; `(3)` create a promise to shows that result of the evaluation of the factorial;
2. `(4)` creates a promise to read a non-existing file; the resulting exception will captured by the callback handler associated (via `process.on`) to the `unhandledRejection` event;
3. `(5)` launches a task to to evaluate in asynchronous way the factorial of the number `10`, in order to show the interleaving of event-driven actions.

The results are:

```
    factIterAsynch(10,10,1,                               //(5)
            function(v){ showIntValue(v,"FACT(10)=" ); } );
    console.log("END");
}
if( process.argv[1].toString().includes("PromiseIntroReadEval") ) main();
/*
START
       %%% readAFile promise starts for anum.txt
       %%% readAFile promise starts for nonexisting.txt
factIterAsynch n0=10 n=10 v=1 res=10
END
factIterAsynch n0=10 n=9 v=10 res=90
  unhandledRejection [object Promise] reason=ENOENT: no such file or directory, open 'C:\...\nonexisting.txt'
factIterAsynch n0=10 n=8 v=90 res=720
factIterAsynch n0=10 n=7 v=720 res=5040
factIterAsynch n0=10 n=6 v=5040 res=30240
factIterAsynch n0=10 n=5 v=30240 res=151200
factIterAsynch n0=10 n=4 v=151200 res=604800
       %%% readAFile promise resolving for anum.txt
       %%% factPromise promise starts for n=5
factIterAsynch n0=5 n=5 v=1 res=5
factIterAsynch n0=10 n=3 v=604800 res=1814400
factIterAsynch n0=5 n=4 v=5 res=20
factIterAsynch n0=10 n=2 v=1814400 res=3628800
factIterAsynch n0=5 n=3 v=20 res=60
factIterAsynch n0=10 n=1 v=3628800 res=3628800
FACT(10)= 3628800
factIterAsynch n0=5 n=2 v=60 res=120
factIterAsynch n0=5 n=1 v=120 res=120
       %%% factPromise promise resolving for n=5
FACT= 120
*/
```

**Listing 1.60.** `Execution of PromiseIntroReadEval.js`

## 8.6 Settled Promises.

The `Promise` constructor is the best way to create *unsettled* promises due to the dynamic nature of what the promise executor does. But there are also two methods that create *settled* promises given a specific value[27]:

- The `Promise.resolve()` method accepts a single argument and returns a promise in the fulfilled state.
- The `Promise.reject()` method works like *Promise.resolve()* except the created promise is in the rejected state.

[27] If we pass a promise to *Promise.resolve()* or *Promise.reject()*, the promise is returned without modification.

# 9 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 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
```

## 9.1 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:

```
/*
 * ===================================
 * ledGpio.js
 * ===================================
 */

var onoff = require('onoff'); //#A

var Gpio = onoff.Gpio,
  led = new Gpio(4, 'out'), //#B
  interval;

interval = setInterval(function () { //#C
  var value = (led.readSync() + 1) % 2; //#D
  led.write(value, function() { //#E
    console.log("Changed LED state to: " + value);
  });
}, 2000);

process.on('SIGINT', function () { //#F
  clearInterval(interval);
  led.writeSync(0); //#G
  led.unexport();
  console.log('Bye, bye!');
  process.exit();
});

// #A Import the onoff library
```

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

**Listing 1.61.** Execution of `ledGpio.js`

More on this can be found in blsJavaScript.pdf.