# NIST - LAB 1 COURSE 2021/22

# INTRODUCTION TO THE LABS: JAVASCRIPT AND NODEJS

This lab consists of three sessions, with the following goals:

1. Present the needed tools and procedures to carry out the lab work for NIST.
2. Present the basic techniques for software development using javascript on NodeJS.

It is assumed that you have gone through the material made available to you as "**Lab 0**". Besides, you should also consider these other aspects:

- There are two documents that: (1) describe the DSIC labs and (2) introduce the NIST lab resources. They are available in the Lessons part of PoliformaT, as links to PDF files in Unit 2 (last part: presential methodology). Both documents describe how you should use the available resources in the Labs (virtual desktop, access to your virtual machine...).
- This bulletin assumes that the lab sessions are carried out through **Polilabs**, but you may easily adapt such assumption to other scenarios...
    - An equivalent virtual machine image to be used, e.g., with VirtualBox.
    - Your virtual machine accessed through the portal.
    - Directly on your personal computer (if you have installed all required resources).

# INTRODUCTION

## Tools

All lab programming work will be carried out using JavaScript on NodejS.

The computers available at the lab rooms use shared virtual machine environments. The consequence is that several resources, like **network ports**, are shared among various users, making the environment unsuitable for carrying out the exercises of NIST (with heavy usage of network interactions).

In order to avoid conflicts, the network port numbers mentioned in this wording should be adapted by the students. The range of ports from 50000 to 59999 is reserved for this subject in the labs. Since no exercise will need more than 10 ports (to be used as the units digit, from 0 to 9, in the port numbers), there are three other digits to distinguish different users. Our recommended criteria is to use the last three digits of your DNI or passport number in the thousand to ten digit of the port number.

- Thus, a student with DNI 29332481 can use ports 54810 to 54819 in each exercise.
- So, if an activity uses ports 8000, 8001 and 8002, that student should adapt those numbers and use, respectively, 54810, 54811 and 54812 instead of them.

The main characteristics of the shared virtual machine environments you will have acces to are:

- LINUX (64-bit Ubuntu), with access to your own account home directory.
- NodeJS (version 12) environment.
- Basic auxiliary libraries (fs, http, net, ...)
- NodeJS npm package management tool
    - If you use it in your own computer, run "npm init" before installing any package.
- We can use the pre-installed code editors (Visual Studio Code, available through the Programming menu).

Students can install similarly-capable environments (from the point of view of NodeJS and tools) on their own computers (Windows, LINUX o MacOS), check http://nodejs.org.

## Procedures

**Clause of good use of lab resources**

Resources are made available to students only for the purpose of supporting the educational goal fo the school. A correct utilization of them is the exclusive responsibility if the student. Consequently students commit to:

- Treat their passwords as confidential information, avoiding sharing them.
- Keep from interfering in the activity of other students.
- Use the resources for ends other tan those presented in the lab bulletins.

Our installations contain resource monitoring mechanisms that can be queried for usage information when needed.

JavaScript programs are text files with a **.**js extension. A program written in file **x.js** is executed with command **node x[.js] [optionalArgs]**

The three sessions of Lab 1 are organized in two parts, the first one focusing in JavaScript (the language); while the second focuses on the NodeJS environment.

As you proceed through the exercises presented to you it will be normal to encounter some errors. The most frequent ones are summarized in the following table.

<table>
<tr><td colspan="3">Syntax errors</td></tr>
<tr><td>Definition</td><td>Detection/solution</td><td>Example</td></tr>
<tr><td>Illegal syntax (invalid identifier, structures incorrectly nested, etc.)</td><td>The interpreter indicates the error type and its location in the code.<br>Correct on code source</td><td>> "Potato"();<br>TypeError: string is not a function<br>at repl:1:9<br>at REPLServer.defaultEval (repl.js:132:27)<br>at bound (domain.js:254:14)<br>at REPLServer.runBound [as eval] (domain.js:267:12)<br>at REPLServer.<anonymous> (repl.js:279:12)</td></tr>
<tr><td colspan="3">Errors in the logic</td></tr>
<tr><td>Definition</td><td>Detection/solution</td><td>Example</td></tr>
<tr><td>Programming error (try to access a non-existent property or 'undefined', pass arguments of an incorrect data type, etc.)<br><br>JavaScript is not strongly type-oriented: some errors, which other languages catch at compilation time, are only shown when the program is run.</td><td>Wrong results when executing. Modify the code. It suits that in the code verify always the restrictions to apply on the arguments of the functions</td><td>> function add(array) {return array.reduce(function(x,y){return x+y})}<br>undefined<br>> add([1,2,3])<br>6<br>> add(1)<br>TypeError: undefined is not a function<br>at add (repl:1:36)<br>at repl:1:1<br>at REPLServer.defaultEval (repl.js:132:27)<br>at bound (domain.js:254:14)<br>at REPLServer.runBound [as eval] (domain.js:267:12)</td></tr>
<tr><td colspan="3">Operational error</td></tr>
<tr><td colspan="2">Definition</td><td>Detection/solution</td></tr>
<tr><td colspan="2">Exceptional situations that can arise during the execution of the program.<br><br>They can be due to:<br>• The environment (e.g., memory exhaustion, too many open files)<br>• Current system configuration (e.g., there is no route to a remote host)<br>• Use of the network (e.g., problems with the use of sockets)<br>• Access problems to a remote service (e.g., I cannot connect to the server)<br>• Wrong or inconsistent input data<br>• Etc.</td><td>Use of try, catch, and throw, similar to those of Java<br><br>Strategy:<br>- If it is clear how to resolve the error, manage it (e.g., error when opening a file for writing -> create it as a new file)<br>- If the management of the error is not responsibility of this fragment of the program, propagate the error to the caller.<br>- For transitory errors (e.g., network-related problems), retry that operation<br><br>If we cannot manage neither propagate the error:<br>- If it prevents the program from going on, abort it.<br>- In another case, write the error down in a log file.</td></tr>
</table>

In order to minimize errors, we recommend:

- Use the strict mode: **`node --use_strict file.js`**
- Document correctly each function
    - Meaning and type of each argument, as well as any additional constraint
    - Which type of operational errors can appear, and how to manage them
    - Its intended return value

# PART ONE

## Running JavaScript Programs

Attached to the documentation prepared to this lab, you can find a folder with name **`javascript`** containing several simple programs illustrating various aspects of the JavaScript language whose command are required for NIST.

- These programs illustrate some basic characteristics of JavaScript, such as object reference ***this,*** builtin function method ***bind,*** *closures, asynchronous programming,* etc
- A list of the available programs can be found at the end of this bulletin.

The activity for this part consists of the study, analysis and execution of those programs, reaching the pertinent conclusions.

Your professor can direct you through those codes he/she deems more appropriate to emphasize.

# PART TWO

## Introduction to NodeJS

### Accessing files

All methods related to files appear in the 'fs' module. The operations are asynchronous by default. However, for each asynchronous function **xx**, there is also a synchronous variant **xxSync**. The following pieces of code can be directly copied to a js file and executed.

- Read asynchronously the contents of a file (**`read1.js`**)

```
const fs = require('fs');
fs.readFile('/etc/hosts', 'utf8', function (err,data) {
  if (err) {
    return console.log(err);
  }
```

```
  console.log(data);
});
```

- Write asynchronously content into a file (**`write1.js`**)

```
const fs = require('fs');
fs.writeFile('/tmp/f', 'contenido del nuevo fichero', 'utf8',
  function (err) {
  if (err) {
    return console.log(err);
  }
  console.log('se ha completado la escritura');
});
```

- Modifications on reads and writes: using modules.

We define module **`fiSys.js`**, based on **`fs.js`**, to access files, along with some usage examples.

(**fiSys.js**)

```
//Module fiSys
//Ejemplo de módulo de funciones adaptadas para el uso de ficheros.
//(Podrían haberse definido más funciones.)

const fs=require("fs");

function readFile(fichero,callbackError,callbackLectura){
	fs.readFile(fichero,"utf8",function(error,datos){
		if(error) callbackError(fichero);
			else callbackLectura(datos);

	});
}


function readFileSync(fichero){
	var resultado; //retornará undefined si ocurre algún error en la lectura
	try{
		resultado=fs.readFileSync(fichero,"utf8");
	}catch(e){};
	return resultado;
}


function writeFile(fichero,datos,callbackError,callbackEscritura){
	fs.writeFile(fichero,datos,function(error){
			if(error) callbackError(fichero);
				else callbackEscritura(fichero);
	});
}

exports.readFile=readFile;
exports.readFileSync=readFileSync;
exports.writeFile=writeFile;
```

(**read2.js**)

```
//Lecturas de ficheros

const fiSys=require("./fiSys");
```

```
//Para la lectura asíncrona:
console.log("Invocación lectura asíncrona");
fiSys.readFile("/proc/loadavg",cbError,formato);
console.log("Lectura asíncrona invocada\n\n");

//Lectura síncrona
console.log("Invocación lectura síncrona");
const datos=fiSys.readFileSync("/proc/loadavg");
if(datos!=undefined)formato(datos);
        else console.log(datos);
console.log("Lectura síncrona finalizada\n\n");

//- - - - - - - - - - -
function formato(datos){
        const separador=" "; //espacio
        const tokens = datos.toString().split(separador);
        const min1 = parseFloat(tokens[0])+0.01;
        const min5 = parseFloat(tokens[1])+0.01;
        const min15 = parseFloat(tokens[2])+0.01;
        const resultado=min1*10+min5*2+min15;
        console.log(resultado);
}

function cbError(fichero){
        console.log("ERROR DE LECTURA en "+fichero);
}
```

(**`write2.js`**)

```
//Escritura asíncrona de ficheros

const fiSys = require('./fiSys');

fiSys.writeFile('texto.txt','contenido del nuevo fichero',cbError,cbEscritura);

function cbEscritura(fichero){
        console.log("escritura realizada en: "+fichero);
}

function cbError(fichero){
        console.log("ERROR DE ESCRITURA en "+fichero);
}
```

### Asynchronous programming: customized events

Usage of the **events** module. Review the following program and analyse its behaviour.

(**emisor1.js**)

```
const ev = require('events')              // library import (Using events module)

const emitter = new ev.EventEmitter()     // Create new event emitter
const e1='print', e2='read'               // identity of two different events

function handler (event,n) {     // function declaration, dynamic type args, higher-order
function
    return () => { // anonymous func, parameterless listener, closure
      console.log(event + ':' + ++n + ' times')
    }
}

emitter.on(e1, handler(e1,0)) // listener, higher-order func (callback)
emitter.on(e2, handler(e2,0)) // listener, higher-order func (callback)
emitter.on(e1, ()=>{console.log('something has been printed')}) //several listeners on e1

emitter.emit(e1) // emit event
emitter.emit(e2) // emit event

console.log('----------------------------')
setInterval(()=>{emitter.emit(e1)}, 2000) // asynchronous (event loop), setInterval
setInterval(()=>{emitter.emit(e2)}, 8000) // asynchronous (event loop), setInterval
console.log('\n\t========> end of code')
```

Analyse this second example (emisor2.js). When a program generates events, it can pass some associated values, as arguments to be received by the listener function.

(**emisor2.js**)

```
const ev = require('events')

const emitter = new ev.EventEmitter()
const e1='e1', e2='e2'

function handler (event,n) {
   return (incr)=>{ // listener with param
      n+=incr
      console.log(event + ':' + n)
   }
}

emitter.on(e1, handler(e1,0))
emitter.on(e2, handler(e2,'')) // implicit type casting

console.log('\n\n---------------------------- init\n\n')
for (let i=1; i<4; i++) emitter.emit(e1,i) // sequence, iteration, generation with param
console.log('\n\n---------------------------- intermediate\n\n')
for (let i=1; i<4; i++) emitter.emit(e2,i) // sequence, iteration, generation with param
console.log('\n\n---------------------------- end')
```

**Activity**: Complete file emisor3.js in order to deal with the requirements stated below...

(**`emisor3.js`**)

```
...
const e1='e1', e2='e2'
let inc=0, t

function rand() { // debe devolver valores aleat en rango [2000,5000) (ms)
   ... // Math.floor(x) devuelve la parte entera del valor x
   ... // Math.random() devuelve un valor en el rango [0,1)
}

function handler (e,n) { // e es el evento, n el valor asociado
   return (inc) => {..} // el oyente recibe un valor (inc)
}

emitter.on(e1, handler(e1,0))
emitter.on(e2, handler(e2,''))

function etapa() {
   ...
}

setTimeout(etapa,t=rand())
```

REQUIREMENTS: This program runs in an indeterminate number of stages. Each stage should have a random duration (from 2 to 5 seconds). At the end of each stage:

- Events e1 and e2 should be emitted. Their listeners should receive as an argument the value of variable **inc**.
- Variable **inc** is increased (+1).
- The duration of the stage should be shown on the terminal.

This is a possible execution example (Only its initial lines are shown):

```
e1 --> 0
e2 --> 0
duration: 3043 ms
e1 --> 1
e2 --> 01
duration: 3869 ms
e1 --> 3
e2 --> 012
duration: 2072 ms
e1 --> 6
e2 --> 0123
duration: 2025 ms
e1 --> 10
e2 --> 01234
duration: 4158 ms
...
```

## Client/server interaction

### HTTP MODULE

To develop web servers (i.e., HTTP servers)

E.g.- simple web server (**`ejemploSencillo.js`**) greeting users contacting the server

| Code | Comments |
|---|---|
| `const http = require('http');`<br><br>`function dd(i) {return (i<10?"0":"")+i;}`<br><br>`const server = http.createServer(`<br>`  function (req,res) {`<br>`  res.writeHead(200,{'Content-Type':'text/html'});`<br>`  res.end('<marquee>Node y Http</marquee>');`<br>`  var d = new Date();`<br>`  console.log('alguien ha accedido a las '+`<br>`  d.getHours() +":"+`<br>`  dd(d.getMinutes()) +":"+`<br>`  dd(d.getSeconds()));`<br>`}).listen(8000);` | It imports module http<br>dd(8) -> "08"<br>dd(16) -> "16"<br><br>It creates the server and associates it this function<br>that returns a fixed response<br>and<br>writes the current time to the console<br><br>The server listens to port 8000 |

This is the first example of program that uses network ports. Therefore, you should consider that port conflicts may arise and, because of this, port numbers should be adapted. Let us assume that your DNI is 29332481; so, you may use ports 54810 to 54819. Thus, the port number shown in this program could be replaced by the first one available in your range: 54810.

Please, run the server and use a web browser as its client:

- Access to the URL **http://localhost:8000** (i.e., http://localhost:54810 in the example discussed above)

- Check (in the browser) which is the response from the server, and in the console which is the message written by the server.

## NET MODULE

| Client (`netClient.js`) | Server (`netServer.js`) |
|---|---|
| `const net = require('net');`<br><br>`const client = net.connect({port:8000},`<br>`  function() { //connect listener`<br>`  console.log('client connected');`<br>`  client.write('world!\r\n');`<br>`  });`<br><br>`client.on('data',`<br>`  function(data) {`<br>`  console.log(data.toString());`<br>`  client.end(); //no more data written to the stream`<br>`  });`<br><br>`client.on('end',`<br>`  function() {`<br>`  console.log('client disconnected');`<br>`  });` | `const net = require('net');`<br><br>`const server = net.createServer(`<br>`  function(c) { //connection listener`<br>`  console.log('server: client connected');`<br>`  c.on('end',`<br>`  function() {`<br>`  console.log('server: client disconnected');`<br>`  });`<br>`  c.on('data',`<br>`  function(data) {`<br>`  c.write('Hello\r\n'+ data.toString()); // send resp`<br>`  c.end(); // close socket`<br>`  });`<br>`  });`<br><br>`server.listen(8000,`<br>`  function() { //listening listener`<br>`  console.log('server bound');`<br>`  });` |

### Accessing CLI arguments

When launching a program through the CLI, we are interacting with a Shell program. This Shell gathers all arguments passed to the execution command and passes them to the JS program, packed within an array which can be accessed with **process.argv** (e.g., attribute argv of object **process**). We can then count the number of arguments (length of the array) and access each argument positionally (index into the array):

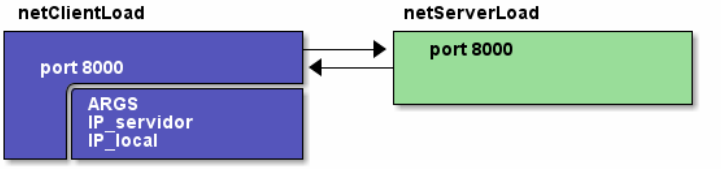
- `process.argv.length:` number of arguments on the CLI
- `process.argv[i]:` get i-th argument.
  - If we used command "`node program arg1` …" then `process.argv[0]` contains string `'node'`, `process.argv[1]` contains string `'program'`, `process.argv[2]` string `'arg1'`, etc.

NOTE: you can use **process.argv.slice(2)** to obtain an array which has discarded 'node' and the program path, so that you are left with just the actual arguments passed to the JS program.

## Query a computer load

Some scalable client/server systems, replicate their servers (using horizontal scalability), and deliver the requests among them according to their current workload level.

We create the embryo for a system of this type, with a single client and a single server (possibly on different machines) communicating through port 8000




- The server is called **netServerLoad.js**, and it does not receive command-line arguments.
  - The following function **getLoad** computes its current load. That function reads the data from the file **/proc/loadavg**, filters its interesting values (adds one hundredth of a second to them in order to avoid the confusion between value 0 and an error), and processes them calculating a weighted average (with weight 10 to the load of the last minute, weight 2 to the last 5 minutes, and weight 1 to the last 15)

```
function getLoad(){
  data=fs.readFileSync("/proc/loadavg"); //requiere fs
  var tokens = data.toString().split(' ');
  var min1  = parseFloat(tokens[0])+0.01;
  var min5  = parseFloat(tokens[1])+0.01;
  var min15 = parseFloat(tokens[2])+0.01;
  return min1*10+min5*2+min15;
};
```

- The client file is **netClientLoad.js**: it receives as its command-line arguments the IP address of the server and its local IP address.
- Protocol: When the client sends a request to the server, it includes its own IP, the server computes its load and returns a response to the client that includes its own IP (i.e., that of the server) and the workload level (i.e., the result of the call to **getLoad**).

To help you test and debug your code, we are keeping an active server at **tsr1.dsic.upv.es**

Important:

- Your starting point to develop these programs consists of the code for netClient.js and netServer.js seen earlier
- You must make sure that all processes terminate (e.g. using `process.exit()`)
- You can use either the domain name or the IP address for a server machine. Use the **ip addr** command to get the local IP address.
- Complete both programs, place them in different computers (with the help of a classmate), and test their communication through port 8000: `netServerLoad` must compute the load as an answer to each request from a client. Correspondingly, `netClientLoad` must use its console to show the answer received from the server.

## Transparent proxy

A proxy is a server that, when invoked by a client, forwards the request to a third party (the real server), who, after processing the request, returns it to the proxy, which, in turn routes it to the original client that sent the request.

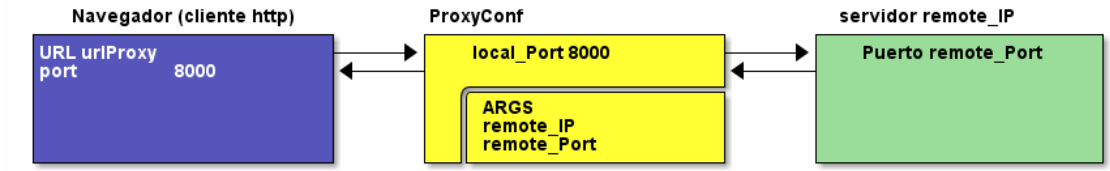
- From a client's point of view, it works as the real server (hides the real server)
- It can run on an arbitrary machine (unrelated to the client's or server's)
- The proxy can use different ports/addresses, but does not modify the messages in transit.

More info in http://en.wikipedia.org/wiki/Proxy_server

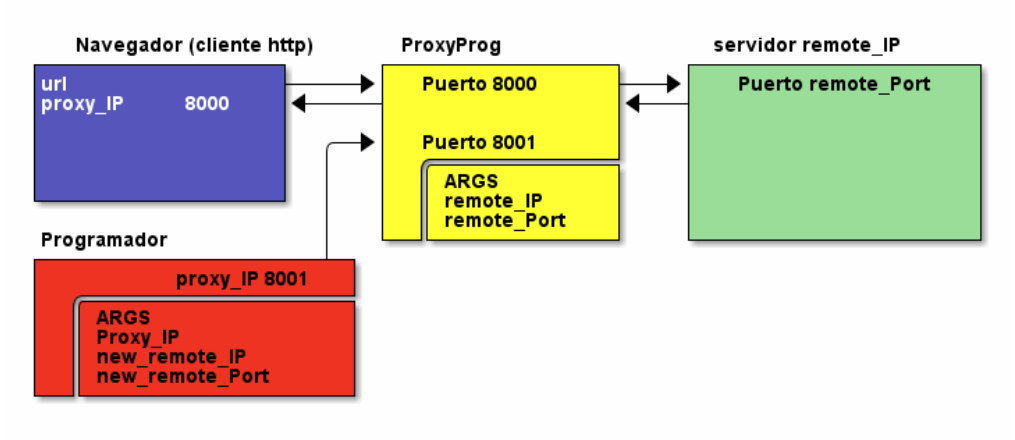
You should progress through three incremental stages:

1. Basic proxy (**Proxy**). When an HTTP client (e.g., a browser) contacts proxy's port 8000, the proxy forwards that request to the "memex" web server (158.42.186.55 port 8080), returning each server answer to the client


2. Configurable Proxy (**ProxyConf**): instead of hard coding the IP and port of the real target server in its code, it gets them through command line arguments




3. Programmable Proxy (**ProxyProg**). Initially, the IP address and port number are passed through command line arguments, but they can be altered at run time, sending an appropriate message to port 8001 of the proxy.




Important:

- We provide the code for Proxy.js (although with different IP address and port number, to be changed by you): you should study it until you understand how it works.
  - Verify it works correctly by pointing a browser to URL http://proxy_address:8000/

| Code (Proxy.js) | Comments |
|---|---|
| `const net = require('net');`<br><br>`const LOCAL_PORT = 8000;`<br>`const LOCAL_IP = '127.0.0.1';`<br>`const REMOTE_PORT = 80;`<br>`const REMOTE_IP = '158.42.4.23'; // www.upv.es`<br><br>`const server = net.createServer(function (socket) {`<br>`  const serviceSocket = new net.Socket();`<br>`  serviceSocket.connect(parseInt(REMOTE_PORT),`<br>`  REMOTE_IP, function () {`<br>`  socket.on('data', function (msg) {`<br>`  serviceSocket.write(msg);`<br>`  });`<br>`  serviceSocket.on('data', function (data) {`<br>`  socket.write(data);`<br>`  });`<br>`  });`<br>`}).listen(LOCAL_PORT, LOCAL_IP);`<br>`console.log("TCP server accepting connection on port: " + LOCAL_PORT);` | **Uses a socket to talk with the client (socket) and another to talk with the server (serviceSocket)**<br><br>1.- It opens a connection to the server<br>2.- It reads a message (`msg`) from client<br>3.- It writes a copy of the message<br>4.- It waits for an answer, and copies it to the client. |

- In scenario 2, you must test the following cases:
  - Access to **UPV**'s server should still work
  - ProxyConf proxying between netClientLoad and netServerLoad
    - If ProxyConf runs in the same machine as netServerLoad, their ports will collide -> modify netServerLoad's code to use a different port.
    - If you get an "EADDRINUSE" error when running the proxy, another program is using the proxy's port.

- In scenario 3 we need to code the program used to reconfigure the proxy. We call it **programador.js**
  - **programador.js** gets the IP of the proxy, as well as the IP and port of the new target server through command line arguments.
  - **programador.js** encodes the new server's IP and port, and sends them as a message to port 8001 of the proxy's machine, after which it terminates.
  - The format of **programador.js** messages should be like this:

```
var msg = JSON.stringify ({'remote_ip':"158.42.4.23", 'remote_port':80})
```

You can test your program using as target servers those already mentioned in this bulletin (memex.dsic.upv.es, IP 158.42.186.55, port 8080; and www.upv.es, IP 158.42.4.23, port 80) or any other that uses HTTP (instead of HTTPS), like www.aemet.es (IP 212.128.97.138, port 80).

## REFERENCES

1. **`Laboratorio/Pract 0 (autoaprendizaje)`**, in PoliformaT, within the *resources* and *lessons* tabs for NIST
2. **`Laboratorio/Pract1`**. List of examples within **`javascript`** folder also available at the *lessons* tab for NIST, in the PoliformaT site.

| | |
|---|---|
| `j00.js` | Functions, objects, reference ***this*** and function method ***bind***(). |
| `j01.js` | Variable declaration. Using functions and closures. |
| `j02.js`, `j03.js` | Closures with variables and functions. |
| `j04.js`, `j05.js`, `j06.js`, `j07.js`, `j08.js`, `j09.js`, `j10.js`, `j11.js` | Asynchronous operations, using function *setTimeout*. |
| `j12.js` | Summary exercise (async operations and closures) |
| `j13.js` | Fork-join like execution of concurrent async operations. |
| `j14.js` | Serialized execution of async operations. |