Seminar 1 – Node.js

Technologies of Networked Information Systems

Index

1. Introduction
2. JavaScript
3. Node.js
4. Learning Results
5. References
6. Appendix 1: Object Orientation
7. Appendix 2: Promises

Objectives

- To use Node.js (JavaScript) as a basic tool for developing distributed system components.
- To identify the main JavaScript/Node.js characteristics and its advantages for application development: event-driven, asynchronous actions,...
- To describe some of the Node.js modules to be used in this course.

Index

1. Introduction
2. JavaScript
3. Node.js
4. Learning Results
5. References
6. Appendix 1: Object Orientation
7. Appendix 2: Promises

1. Introduction

- The rest of this presentation introduces…
  - The JavaScript programming language
  - The Node.js interpreter
- This is not a JavaScript or Node.js reference. Only some of their aspects (those relevant for this subject) are described.
  - Promises and classes are summarised in the appendices
    - Although they are important, they are not mandatory in TSR.
      - Those appendices provide a short reference for the interested reader.

1. Introduction

- JavaScript is a scripting language, interpreted, dynamic and portable
  - High level of abstraction
    - Simple programmes
    - Fast development
- Programming language initially designed for providing dynamic behaviour to web pages
  - Current browsers include an interpreter of this language
- Event-driven with asynchronous interactions supported with "callbacks"
  - This boosts both throughput and scalability
- No support for multi-threading
  - No shared objects. No need for synchronisation mechanisms
  - But we should take care about when a variable gets its value
    - Callback management
- It supports both functional and object-oriented programming

1. Introduction

- Node.js:
  - Development platform based on the JavaScript interpreter (known as V8) being used by Google in its Chrome browser
    - Node.js provides a series of modules that facilitates the development of distributed applications
  - It defines:
    - Programming interfaces
    - Common utilities
    - Interpreter
    - Module management
    - …
  - Most technologies being considered to set the learning results and competences of TSR can be easily integrated or developed using Node.js

Index

1. Introduction
2. JavaScript
3. Node.js
4. Learning Results
5. References
6. Appendix 1: Object Orientation
7. Appendix 2: Promises

2. JavaScript. Main characteristics

- Imperative and structured
  - Syntax similar to that of Java.
- Multi-paradigm
  - Functional programming:
    - Functions are "objects" and can be used as arguments to other functions.
  - Object-oriented programming:
    - Based on prototypes, instead of regular classes and inheritance.
    - However, prototypes may emulate object orientation.
- Related programming languages

Java	syntax, primitive values vs. objects
Scheme	functional programming
Self	prototype-based inheritance
Perl and Python	string, array and regular expressions

2. JavaScript

- How to run its programs? Two basic alternatives:
  1. Using the interpreter included in web browsers.
     - Writing "script" elements in the HTML of a web page:
       - <script type='text/javascript'> ... code ... </script>
       - <script type='text/javascript' src='file.js'></script>
     - Or using the JavaScript console in your browser.
  2. Using an external interpreter
     - For instance, "node"
       - This is the approach to be used in this course.
       - The interpreter can be downloaded and installed from http://nodejs.org
         - **node** is the command that runs this interpreter

2. JavaScript. Syntax

- Similar to Java
  - Case-sensitive.
  - Comments using // (up to the end of line) or /* … */ (for blocks of multiple lines).
  - Identical control structures (if/else, while, do/while, for, switch…)
  - Identical operators.
  - Unicode character set.
  - Blocks defined with braces: {…}
- But…
  - Variable and function declarations do not follow a Java-like syntax.
  - Semicolons at the end of sentence are optional.
- Take a look at the "JavaScript syntax" entry in the wikipedia!

2. JavaScript. Values

- As Java, JavaScript divides the values (i.e., the elements that can be assigned to a variable) in two groups: primitive and compound (i.e., objects)
  - Primitive
    - They are passed by value
    - Two primitive elements are considered equal when they have the same value
    - They are immutable
  - Compound (Objects)
    - Internally, they may hold multiple elements (properties)
    - They are passed by reference
    - Single identity. In a strict sense, each object is only equal to itself
    - They can be modified (object properties may be modified, eliminated or added)

2. JavaScript. Primitive values

- null
  - It means 'no object'
- undefined
  - Meanings: uninitialised variable, argument without value, or inexistent property
- Boolean
  - Logical expressions interpret as **false** these values: null, undefined, false, 0 and NaN (for numbers) and "" (for strings). Any other value is **true**
- Number
  - Examples: 3.12, 45, -.23e-5 (Java syntax)
  - Always maintained as floating point numbers. 64 bits.
  - Special values: NaN (not a number) and Infinity
    - They can be used in expressions
    - They may be generated when some operations are executed. Example number("abc") returns NaN
- String
  - Between simple or double quotes, e.g., "hello"
  - As in Java, they admit special characters (\t, \n ...)

- Boolean, Number and String have got some properties and methods. They are "objects"
  - For instance: "hello".length
- **null** and **undefined** are simple constants. They have not got any method or property

2. JavaScript. Compound values (objects)

- Explicit objects
  - Literal.- {k:v, k':v', ...}.
    - Example: {color:'red', brand:'seat', model:'exeo', year:2008}
  - Each k, k', … is a property and it can be an identifier, a string or a number
  - Each v, v', … is the value associated to the property. It can be any value, including a function
  - They are accessed as object.k (if k is a valid identifier) or object[k]
- Arrays
  - Literal.- [v, v', ...]
    - Example: [1,2,3]
  - It is an ordered sequence of values: v, v', …
  - They are accessed per position (i.e., indexed access, first index is 0). Syntax: array[index]
- Regular expressions
  - Literal.- /pattern/flags
  - For processing strings; e.g., parsing strings
- Functions
  - Literal.- **function** (args) {code}
  - Functions may be defined inside other functions (nesting)

2. JavaScript. Variables

- Variables should be declared before being used
  - Syntax
    - var x // The type cannot be declared
  - They can be initialised in their declaration
    - var x=12
    - const pi = 3.141592 // constant (read-only variable)
  - Assignments using =
    - Combined with arithmetic operators, as Java does: +=, -=, *=, ...
  - They may be declared inside any function or with a global scope
    - Declaration scope: the current function
- Variable identifiers follow the Java rules
  - There is a set of reserved identifiers: **if**, **while**, **var**, ...

2. JavaScript. Variables

- JavaScript is not a "strongly typed" language
  - A variable is declared (with "**var**") before its first use, but without any specification of type
    - **var** x // No type is given
    - **var** y= 'tsr' // A variable may be initialised in its declaration. Since 'tsr' is a String, **y** holds now a String.
  - A variable may hold, in an execution, elements of different types (i.e., its type is "dynamic").
    - x=4 // **x** is now a number…
    - x='Text' //…, later a String…
    - x= {colour:'red', brand:'Seat', model:'Toledo', year:2016} // …now, an object…
    - x = [1,2,3,'test',6] // …, here, an array…
    - x = function() {return 'Example'} //…at this point, a function…
    - var y = x() // What is held in **y**?
  - Objects are heterogeneous
    - Their values may belong to different types

2. JavaScript. Variables

- JavaScript type management is weak
- We should take care of its implicit type conversions
- For instance…:
  - **var** x = “7” // Value of x is “7” (a String)
  - x == 7 // **true** (implicit type conversion)
  - x === 7 // **false** (strict comparison)
  - x + 23 // Its result is “723” (+ concatenates strings)
  - x + “2” // Its result is “72”
  - x * 2 // Its result is 14 (x is taken as a number) since the operator * has no meaning for strings

2. JavaScript. Variables

- Operators to obtain the dynamic type of an element (they return a string)
  - typeof
    - e.g., if (typeof(x) == 'Number') …
  - instanceof
    - for objects
    - x instanceof y iff x was created using the y constructor
- Values may have properties
  - Example: var s="hello"; s.length;
  - If a property is a function, then it is a method. Example: "hello".toUpperCase()
- Example.- For a String value (e.g., "hello")...
  - Properties
    - length, ...
  - Methods
    - charAt(i), charCodeAt(i), concat, fromCharCode, indexOf, lastIndexOf, localeCompare, match, replace, search, slice, split, substr, substring, toLocaleLowerCase, toLocaleUpperCase, toLowerCase, toString, toUpperCase, turn, valueOf

2. JavaScript. Scope

- Lexic scope
  - The scope of a variable is…
    - Local to the function where it has been declared (using **var**)
    - Global (entire file) when…
      - It is not declared inside a function
        - Equivalent to assume an implicit global function that holds the entire file
      - Or when its is declared in a function without using **var**
        - Example: x = 3. **Not recommended!!**
  - A statement…
    - may access all variables that have been defined in the scopes that include that statement
    - variables are searched from the inner to the outer scope
- Be careful!!
  - In Java and other languages, scopes are defined by blocks delimited by braces…
  - …but in JavaScript the scopes are defined **only by functions**!!

2. JavaScript. Closures

- Closure = function + connection to variables in outer scopes
  - Functions remember the scope where they have been created

```
function createFunc() {
  var name= "Mozilla"
  return function() {console.log(name)}
}
var myFunc = createFunc()
myFunc() // it shows "Mozilla"
```

  - Another example

```
function multiplyBy(x) {
  return function(y) {return x*y}
}
var triplicate = multiplyBy(3)
y = triplicate(21) // Returns 63
```

- Additional details in [1]

2. JavaScript. Scope

- Example taken from [3] in order to show different variable scopes:
  - Read [1] in order to get more information about the scope in JavaScript.

```
function alert(x) {  // Needed in Node.js in order
   console.log(x);    // to print messages to stdout.
}

var global = 'this is global';

function scopeFunction() {
  alsoGlobal = 'This is also global!';
  var notGlobal = 'This is private to scopeFunction!';

  function subFunction() {
    alert(notGlobal); // We can still access notGlobal
                       // in this child function.
    stillGlobal = 'No var keyword so this is global!';
    var isPrivate = 'This is private to subFunction!';
  }

  alert(stillGlobal); // This is an error since we
                       //haven't executed subfunction

  subFunction();   // Execute subfunction
  alert(stillGlobal); // This will output 'No var
                       // keyword so this is global!'
  alert(isPrivate); // This generates an error since
                       // isPrivate is private to
                       // subfunction().
  alert(global);     // It outputs: 'this is global'
}

alert(global);       // It outputs: 'this is global‘

alert(alsoGlobal); // It generates an error since
                    // we haven't run scopeFunction yet.

scopeFunction();

alert(alsoGlobal); // It outputs: 'This is also global!';

alert(notGlobal); // This generates an error.
```

2. JavaScript. Operators

- With Java syntax
  - Logical operators
    - && and, || or, ! not (&& and || with short-circuit evaluation)
  - Relational operators
    - ==, !=, <, >, <=, >=, ===, !==
    - === and !== for strict comparison (avoiding implicit type conversion)
  - Arithmetic operators
    - *, +, -, /, %, ++, --, -(negate)
    - +x (conversion into number)
  - Bit operators
    - &, |, ~, ^, <<, >>, >>>
  - String operators
    - + (concatenate)
- Other
  - delete Deletes an object, a property in an object or an element in an array.
  - void a Evaluates expression **a** without returning any value
  - typeof, instanceof Already discussed
  - return Identical to Java

2. JavaScript. Statements

- Basically, JavaScript statements behave as Java statements
  - But any returned value may be interpreted as a Boolean
    - Already explained (slide 13)
  - Conditionals
    - **if/else** (Java syntax)
    - **switch** (Java syntax)
    - cond **?** ifTrue **:** ifFalse (**?:** operator, Java syntax)
  - Loops
    - **while** (Java syntax)
    - **do/while** (Java syntax)
    - **break/continue** (Java syntax)
    - **for(;;)** (Java syntax)
    - **for(**variable **in** object**).** Loops onto the properties (keys) of the given object
  - Exception management
    - **try/catch/finally** (Java syntax)
  - Other
    - expr1**,**expr2 (comma). Evaluates 2 expressions, returning the result of expr2

2. JavaScript. Functions

- Anonymous functions
  - function (args) {…}
  - It is a value that can be assigned, passed as an argument,...
    - Example: var double = function (x) {return 2*x}
  - To be invoked as identifier(args), returning a single value
    - Example: var x = double(28)
- Declaration
  - function name(args) {...}
  - Equivalent to: var name = function (args) {…}
    - function double(x) {return 2*x} ...is equivalent to...
    - var double = function (x) {return 2*x}
- They can be declared everywhere, even inside another function (i.e., they can be nested)
- They provide the scope for variable definition
- Arguments are passed by value (as in Java)
  - But objects are actually passed by reference
- Functions are objects
  - with properties and methods
- A single return value, but it may be a composed element (i.e., an object)

2. JavaScript. Functions (arity)

- Arity (number of arguments)
  - A function with n arguments may be invoked using…
    - Exactly n values
    - Less than n values. The remaining arguments receive the "undefined" value
    - More than n values. The unexpected arguments are ignored
  - Arguments are accessed…
    - by name
    - or using the "arguments" pseudo-array

```
function greetings() {
   for(var i=0; i<arguments.length; i++) {
    console.log("Hello, " + arguments[i])
   }
}
greetings("Diana", "John", "Paul")  // 'Hello, Diana', 'Hello, John'. 'Hello, Paul'
```

  - The arity may be enforced
    - function f(x,y) {if (arguments.length != 2)… }
  - Or default values may be assigned
    - function f(x,y) {x = x||defaultValueX; y=y||defaultValueY; ..}
  - There cannot be two functions with the same name, even when they are defined with different arities

2. JavaScript. Arrays

- Array = Heterogeneous sequence of elements. Accessed per slot.
  - Indexes: 0...
- Syntax: [v,v,...]
  - v,v',… can be arbitrary values (including other arrays)
  - Example: var a=[1,2,3]

- They are objects, with properties and methods
  - Property
    - length — a value can be assigned, modifying its capacity
  - Methods
    - slice(from,to) and slice(from) — copies a fragment of the array
    - push(x) — appends an element (at the end)
    - pop() — removes the last element
    - shift() — removes the first element
    - unshift(x) — inserts at the beginning
    - indexOf(x) — searches an element, returning its position
    - join, join(sep) — concatenates

2. JavaScript. Functional Programming

- Since functions may be regular arguments in calls to other functions, the functional programming paradigm may be used in some cases.
- Example:
  - Assuming an array, we could build different functions to operate with its contents:
    - function sum(a) { var r=0; for (i in a) r += a[i]; return r}
    - function prod(a) { var r=1; for (i in a) r *= a[i]; return r }
      - We may obtain an overall value applying any binary operator to each pair of elements
  - Instead of having multiple functions with similar code...
    - We extract a general algorithm from all those solutions: to loop over the array applying the binary operation
    - The binary operation should be passed as an argument to the function that implements such general algorithm
  - In JavaScript, the "reduce()" method provides such functionality
    - [4,3,6,7,8].reduce(function(a,b){return a+b}) // Returns the sum of [4,3,6,7,8]: 28
    - [4,3,6,7,8].reduce(function(a,b){return Math.max(a,b)}) // Returns the maximum: 8

2. JavaScript. Functional Programming

- There are other predefined functions that are useful in these cases. All of them require another function f2 as their argument. For instance:
  - map.- Applies f2 onto each array element, returning a new array.
  - filter.- Maintains in the array the elements returning true when f2 is invoked, removing the other elements.
  - some.- Returns true when any of the array elements obtains true when f2 is invoked.
  - every.- Returns true when every array element obtains true when f2 is invoked.

2. JavaScript. Objects

- Object literal:
  - {k:v, k':v', ...}
  - It is a set of properties k, k', … with values v, v',…
    - k,k',… may be identifiers, strings or numbers.
    - v,v',… may be any value, even another object (object nesting) or a function (method)
      - A method implicitly defines a special variable (**this**) that represents the object on which the method is applied
  - Properties are accessed as obj.k (when k is an identifier) or as obj[k] (for strings and numbers)
  - Properties may be removed with: **delete** obj.k
  - Test whether a property exists with: k **in** obj
  - The **for (…in…)** loop iterates on the object properties.
  - Example:

```
var dog= {
      name: 'Toby',
      legs: 4,
      state: function() {console.log(this.name + " is OK! ")}
}
dog.state() // writes "Toby is OK!"
var f = dog.state
f() // f isn't bound to an object (undefined this) → error. Writes "undefined is OK1"
f2 = f.bind(dog); f2() //ok (the binding may be done in this way)
for (var k in dog) console.log(k) // Properties may be accessed in this way
```

2. JavaScript. Object orientation

- JS doesn't provide an object model based on classes, but on prototypes
  - It is not based on creating a data type (class) to declare objects (instances)
    - Classes (with their common meaning), class inheritance and interfaces do not exist in JavaScript
  - Each object may be based on another existent object (prototype)
- This model provides more flexibility in some cases:
  - Singleton objects may be declared without defining any class
  - Each object may individually modify its behaviour (for instance, for processing incoming messages)
  - …
- But this is not equivalent to other class-based programming languages (e.g., Java)
  - Appendix 1 describes how to emulate classes and inheritance
    - It is only a syntactic mechanism. At low level, object prototyping is still used.
    - There are other alternatives:
      - Node.js v6 introduces a new class syntax (taken from ECMAScript 6)

2. JavaScript. Serialization. JSON

- JSON (JavaScript Object Notation)
  - It is a text format that may be easily transferred through the network.
  - It allows the "serialization" of any JavaScript object.
- Two operations:
  - JSON.stringify(obj)
    - Serializes the object "obj", returning a JSON string.
      - Example:

```
var ob = {"x":23,"y":{"a":45,"b":[5,0]}}
var s = JSON.stringify(ob);
console.log(s); // Shows: {"x":23,"y":{"a":45,"b":[5,0]}}
```

  - JSON.parse(str)
    - Deserializes "str": It returns an object taking the JSON string "str" as its input argument.

2. JavaScript. Callbacks

- A "callback function" is…:
  - …a reference to a function that is passed as an argument to another function B. B invokes that callback when it is terminating its execution.
  - Example: Let us assume a fadeIn() method that progressively vanishes an element that is displayed.
    - It is called as: element.fadeIn(speed, function() {…})
    - The second argument is a callback function that will be invoked when "element" has completely disappeared.
- Callback functions allow asynchronous invocations:
  - An agent calls B(args,C), being C a callback
  - When B is terminated, it calls C
    - Thus, B reports its completion and provides its result

2. JavaScript. Events

- JavaScript is single-threaded
  - But multiple activities may be executed
  - Setting them as events
- There is an event queue that…
  - accepts external interactions
  - holds pending activities
  - is turn-based
- Each kind of event may be managed in a different way
  - But all event answers are executed by the same thread
  - This imposes a sequence-based management
    - i.e., a new event isn't processed until the current one is finished

```
function fibo(n) {
   return (n<2) ? 1 : fibo(n-2) + fibo(n-1)
}

console.log("Starting...");

// Writes a message in 10 ms
setTimeout( function() {
  console.log( "M1: Something is written..." )
}, 10 );

// This statement lasts more than 5 seconds...
var j = fibo(40);
function anotherMessage(m,u) {
   console.log( m + ": The result is: " + u )
}

// M2 is written before M1 since the "main" thread is never interrupted
anotherMessage("M2",j)

// M3 is written after M1
setTimeout( function() {
 anotherMessage("M3",j)
}, 1 )
```

2. JavaScript. Asynchronous execution. Callbacks

- Asynchronous executions may be built using callbacks
- But there are some constraints:
  - Exceptions in nested callbacks
    - When an exception isn't managed, it is propagated to its caller
    - Without a uniform management in all program operations…:
      - Some exceptions might be lost
      - Some exceptions could be managed in operations that do not expect them
  - The resulting code may be almost unreadable
    - In most cases, execution order will not be intuitive
  - Uncertainty on which is the turn for the execution of each callback
    - Asynchronous callbacks are run when its encompassing function has terminated
    - In many cases, that termination depends on external events
      - e.g., message arrival in case of a receive() operation on a socket

2. JavaScript. Asynchronous execution. Promises

- Asynchronous executions may be also built using promises
  - Operation calls follow the traditional format (easy to read)
    - There is no callback argument
  - The result of that call is a "promise" object.
    - It represents a future value on which we may associate operations and manage errors
    - It may be in one of the following states
      - pending. Initial state. The operation has not yet concluded (unknown result).
      - resolved. The operation has terminated and we can get its result. This is a final state that cannot change.
        - rejected. The operation has terminated with error. The reason is given.
        - fulfilled. The operation has terminated successfully. A value is returned.
  - A function is associated to each final state (rejected vs fulfilled). Such function is run when the main thread finishes its current turn.
    - Actually, it is enqueued in a new turn as a future event.
- Appendix 2 provides additional information on promises
  - Promises are not required for implementing the lab projects

2. JavaScript. Callbacks vs promises

- Example: Asynchronous read of a file
  - The version based on promises needs that an asynchronous function (in this case, readFilePromisified) returns a promise
  - Appendix 2 explains how to define that kind of functions

Callbacks	Promises

<pre>fs.readFile('jsonFILE',
    function (error, text) {
        if (error) {
            console.error('error')
        } else {
            try {
                const obj = JSON.parse(text)
                console.log(JSON.stringify(obj))
            } catch(e) {
                console.error('error')
            }
        }
    })</pre>
	<pre>readFilePromisified('jsonFILE`)
	.then(function(text) {
	    const obj = JSON.parse(text)
	    console.log(JSON.stringify(obj))
	})
	.catch(function(error) {console.error('error')})</pre>

Index

1. Introduction
2. JavaScript
3. Node.js
4. Learning Results
5. References
6. Appendix 1: Object Orientation
7. Appendix 2: Promises

3. Node.js

- Node.js is a special JavaScript interpreter:
  - Independent. Valid for writing server agents.
    - Not embedded in a web browser.
    - Available in: http://nodejs.org/download/ (interpreter), http://nodejs.org/api/ (documentation).
  - Using "**require**()" other modules can be included in a program.
  - Most methods in Node.js modules allow asynchronous interactions.
    - Method returns immediately.
    - Results are provided via "*callbacks*".
    - An "asynchronous programming" model is followed:
      - Single-threaded: no concurrency, no shared variables, no critical sections,… Very efficient. No concurrency "dangers".
      - This single thread is not blocked in I/O operations nor when other traditionally blocking OS services are called.
    - Those asynchronous methods also have other blocking versions (without "callbacks").
      - e.g., fs.readFile() is asynchronous, but there is also an fs.readFileSync()

3. Node.js

- How is this asynchrony achieved??
  - Programmers see a single thread, but...
    - A queue of "function closures" is handled by the node runtime.
      - It is the "turn queue".
    - At each time, the Node runtime dequeues the first turn and executes it.
      - This action defines a "turn".
      - NOTE: setTimeout(f,0) stores function f() in the queue.
        - Useful when we need to execute f() once the current activity was finished.
  - Asynchronous modules are based on the **libuv** [6] library.
    - **libuv** maintains a "*thread pool*".

3. Node.js

- When a blocking operation is called...
  1. A thread T is taken from the "*pool*".
  2. Invocation arguments are given to T, including the "callback" scope.
  3. The invoking thread returns and our program goes on.
     - T remains in the "ready-to-run" state.
  4. T executes all operation sentences.
     - It might block in some of them.
  5. When T finishes that operation, it calls its associated *"callback"*...
     1. T creates a scope for such "callback", passing the needed arguments.
     2. T stores such scope in the turn queue.
     3. T comes back to the "*pool*".
     4. The "*callback*" is executed in a future turn.
        - When it becomes the first in the turn queue.
        - This avoids any race condition.

3.1. Module management

- Exports
  - Programmers should decide which objects and method are exported by a module.
  - Each of those elements should be declared as a property of the "module.exports" object (or, simply, "exports").
    - Example:

```
// Module Circle.js

exports.area = function(r) {
  return Math.PI * r * r;
}
```

```
exports.circumference = function(r) {
  return 2 * Math.PI * r;
}
```

- Require
  - Modules are imported using "require()".
  - The module global object may be assigned to a variable. This names its context/scope.
    - Example 1: **var** HTTP = require('http');
    - Example 2: **var** st = require('./Circle.js');
      console.log("Area of a circle with radius 5:" + st.area(5));

3.2. Events module

- The **events** module is needed for implementing event generators.
  - Generators should be instances of EventEmitter.
  - A generator throws events using its method **emit(event [,arg1][,arg2][...])**.
    - emit() executes the event handlers in the current turn.
    - If we do not want such behavior...
      - setTimeout(function() {emit(event,...);},0)
- Event "*listeners*" may be registered in the event emitters:
  - Using method **on(event,listener)** from the emitter.
    - **addListener(event, listener)** does the same.
    - A "*listener*" is a "*callback*".
  - The "*listener*" is invoked each time the event is thrown.
  - There may be multiple "*listeners*" for the same event.
- Documentation available in:
  - http://nodejs.org/api/events.html

3.2. Events module

- Example:
  - The event emitter should be created using "**new**"!!

```
var ev = require('events');
var emitter = new ev.EventEmitter;
// Names of the events.
var e1 = "print";
var e2 = "read";
// Auxiliary variables.
var num1 = 0;
var num2 = 0;

// Listener functions are registered in
// the event emitter.
emitter.on(e1, function() {
  console.log( "Event " + e1 + " has " +
    "happened " + ++num1 + " times.")});
emitter.on(e2, function() {
  console.log( "Event " + e2 + " has " +
    "happened " + ++num2 + " times.")});
```

```
// There might be more than one listener
// for the same event.
emitter.on(e1, function() {console.log(
  "Something has been printed!!");});

// Generate the events periodically...
// First event generated every 2 seconds.
setInterval( function() {
    emitter.emit(e1);}, 2000 );
// Second event generated every 3 seconds.
setInterval( function() {
    emitter.emit(e2);}, 3000 );
```

3.3. Stream module

- Stream objects are needed to access data streams.
- Four variants:
  - Readable: read-only.
  - Writable: write-only.
  - Duplex: allow both read and write actions.
  - Transform: similar to Duplex, but its writes usually depend on its reads.
- All they are EventEmitter. Managed events:
  - Readable: readable, data, end, close, error.
  - Writable: drain, finish, pipe, unpipe.
- Examples:
  - Readable: process.stdin, files, HTTP requests (server), HTTP responses (client), ...
  - Writable: process.stdout, process.stderr, files, HTTP requests (client), HTTP responses (server),...
  - Duplex: TCP sockets, files, ...
- Documentation available in:
  - http://nodejs.org/api/stream.html

3.3. Stream module

- Example:
  - Interactive version of the computation of the circumference given a radius.
  - process.stdin is a "Readable" *stream*.

```
var st = require('./Circle.js');

console.log("Radius of the circle: ");

// Needed for initiating the reads
// from stdin.
process.stdin.resume();
// Needed for reading strings instead of
// "Buffers".
process.stdin.setEncoding("utf8");

// Implemented as an endless loop.
// Every time we read a radius, its
// circumference is printed and a new
// radius is requested.
```

```
process.stdin.on("data", function(str) {
  // The string that has been read is "str".
  // Remove its trailing endline.
  var rd = str.slice(0,str.length-1);
  console.log("Circumference for radius " +
    rd + " is " + st.circumference(rd));
  console.log(" ");
  console.log("Radius of the circle: ");
});

// The "end" event is generated when
// STDIN is closed. [Ctrl]+[D] in UNIX.
process.stdin.on("end", function() {
  console.log("Terminating...");
});
```

3.4. Net module

- “net” module: management of TCP sockets:
  - net.Server: TCP server.
    - Generated using **net.createServer([options,][connectionListener])**.
      - “connectionListener”, when used, has a single parameter: a TCP socket already connected.
    - Events that may manage: listening, connection, close, error.
  - net.Socket: Socket TCP.
    - Generated using “new net.Socket()” or “net.connect(options [,listener])” or “net.connect(port [,host][,listener])”
    - Implements a Duplex Stream.
    - Events that may manage: connect, data, end, timeout, drain, error, close.
- Documentation available in:
  - http://nodejs.org/api/net.html

3.4. Net module

- Example (from the Node.js documentation):

Server

```
var net = require('net');
var server = net.createServer(
  function(c) { //'connection' listener
    console.log('server connected');
    c.on('end', function() {
      console.log('server disconnected');
    });
    // Send "Hello" to the client.
    c.write('Hello\r\n');
    // With pipe() we write to Socket 'c'
    // what is read from 'c'.
    c.pipe(c);
}); // End of net.createServer()
server.listen(9000,
  function() { //'listening' listener
    console.log('server bound');
  });
```

Client

```
var net = require('net');
// The server is in our same machine.
var client = net.connect({port: 9000},
  function() { //'connect' listener
    console.log('client connected');
    // This will be echoed by the server.
    client.write('world!\r\n');
  });
client.on('data', function(data) {
  // Write the received data to stdout.
  console.log(data.toString());
  // This says that no more data will be
  // written to the Socket.
  client.end();
});
client.on('end', function() {
  console.log('client disconnected');
});
```

3.6. HTTP Module

- To implement web servers (and also their clients).
- Consists of the following classes:
  - http.Server: EventEmitter that models a web server.
  - http.ClientRequest: HTTP request.
    - It is a Writable Stream and an EventEmitter.
    - Events: response, socket, connect, upgrade, continue.
  - http.ServerResponse: HTTP response.
    - It is a Writable Stream and an EventEmitter.
    - Events: close.
  - http.IncomingMessage: It implements the requests (for the web server) and the responses associated to ClientRequests.
    - It is a Readable Stream.
    - Events: close.
- Documentation available in:
  - http://nodejs.org/api/http.html

3.6. HTTP Module

- A minimal web server:
  - Given as example in: http://nodejs.org/about/

```
var http = require('http');
http.createServer(function (req, res) {
  // res is a ServerResponse.
  // Its writeHead() method sets the response header.
  res.writeHead(200, {'Content-Type': 'text/plain'});
  // The end() method is needed to communicate that both the header
  // and body of the response have already been sent. As a result, the response can
  // be considered complete. Its optional argument may be used for including the last
  // part of the body section.
  res.end('Hello World\n');
  // listen() is used in an http.Server in order to start listening for
  // new connections. It sets the port and (optionally) the IP address.
}).listen(1337, "127.0.0.1");
console.log('Server running at http://127.0.0.1:1337/');
```

Index

1. Introduction
2. JavaScript
3. Node.js
4. Learning Results
5. References
6. Appendix 1: Object Orientation
7. Appendix 2: Promises

5. Learning Results

- When this seminar is concluded, the student should be able to:
  - Identify JavaScript (with Node.js) as an example of programming language that admits asynchronous programming.
  - Identify JavaScript as a programming language that avoids multiple concurrency problems/dangers.
  - Build small programs in Node.js using an event-driven paradigm.
  - Know multiple sources in order to delve into Node.js and JavaScript programming.

Index

1. Introduction
2. JavaScript
3. Node.js
4. Learning Results
5. References
6. Appendix 1: Object Orientation
7. Appendix 2: Promises

6. References

Basic (Recommended)

1. Tim Caswell: "Learning JavaScript with Object Graphs". Available in: http://howtonode.org/object-graphs, 2011.
2. Tim Caswell: "Learning JavaScript with Object Graphs (Part II)". Available in: http://howtonode.org/object-graphs-2, 2011.
3. Patrick Hunlock: "Essential Javascript – A Javascript Tutorial". Available in: http://www.hunlock.com/blogs/Essential_Javascript_--_A_Javascript_Tutorial, 2007.
4. Joyent, Inc.: "Node.js v6.5.0 Documentation", available in: http://nodejs.org/api/, September 2016.

Advanced (Non-mandatory)

5. David Flanagan: "JavaScript: The Definitive Guide", 5th ed., O'Reilly Media, 1032 pgs., August 2006. ISBN: 978-0-596-10199-2 (printed edition), 978-0-596-15819-4 (ebook).
6. Nikhil Marathe: "An Introduction to libuv (Release 1.0.0)", July 2013. Available in: http://nikhilm.github.io/uvbook/index.html

Index

1. Introduction
2. JavaScript
3. Node.js
4. Learning Results
5. References
6. Appendix 1: Object Orientation
7. Appendix 2: Promises

A1. Classes

```
// Point constructor.
function Point(x,y) {this.x = x; this.y = y}

// Segment constructor.
function Segment(p1,p2) {this.p1 = p1;  this.p2 = p2}

// Auxiliar. Function that computes the distance between two Point objects.
function distance(a,b) {
    var dx = b.x-a.x, dy = b.y-a.y
    return Math.sqrt( dx*dx + dy*dy )
}

// length method shared by all Segment objects
Segment.prototype.length = function() {return distance(this.p1, this.p2)}

var p1 = new Point(10,0), p2 = new Point(5,5), p3 = new Point(3,6)
var se1 = new Segment(p1,p2), se2 = new Segment(p2,p3)

console.log("Length of segment 'se1' is: " + se1.length() )
console.log("Length of segment 'se2' is: " + se2.length() )
```

A1. Inheritance

- Inheritance (A inherits the methods from B):
  - A.prototype = Object.create(B.prototype) // method inheritance
  - A.prototype.constructor = A // Its constructor should be assigned
    - Reference [2] explains object management

```
// RegularPolygon constructor.
function RegularPolygon(ns,sl) {
  this.numSides = ns;
  this.sideLength = sl;
}

// perimeter() method for every RegularPolygon.
RegularPolygon.prototype.perimeter = function() {
  return this.numSides * this.sideLength;
}

// Square constructor.
function Square(sl) {
  this.numSides = 4;
  this.sideLength = sl;
}
```

```
// Inherit from RegularPolygon
Square.prototype = Object.create(RegularPolygon.prototype);
Square.prototype.constructor = Square;

// Create a Square with side 6.
example2 = new Square(6);

// Class Square has, additionally, an area() method.
Square.prototype.area = function() {
  return this.sideLength * this.sideLength;
}

// Check that all works as intended.
console.log( "Perimeter of a square with side 6 is: " +
  example2.perimeter() );
console.log( " and its area is: " + example2.area() );
```

Index

1. Introduction
2. JavaScript
3. Node.js
4. Learning Results
5. References
6. Appendix 1: Object Orientation
7. Appendix 2: Promises

A2. Promises

- Each asynchronous function that returns a promise may be called in any of these ways
  - The second column shows the new syntax for defining anonymous functions.

asyncFunc(args**)** **.**then**(function(**result**) {…})** **.catch(function(**error**) {…})**	**asyncFunc(**args**)** **.**then**(**result **=> {…})** **.catch(**error **=> {…})**

- It sets handlers for each state
  - fulfilled.- `then` is executed, receiving the promise result
  - rejected.- `catch` is executed, receiving the error
- The execution of these handlers is asynchronous
  - e.g.- `then` is executed once the promise is fulfilled
    - But once fulfilled, we don't know in which turn will be run
    - This depends on the event management
    - It is guaranteed that it will be run in a future turn (never in the current turn)
      - All state updates in the current turn are protected against potential race conditions with callbacks

A2. Promises. Chaining

- The result of **then** is a new promise
- This allows...
  - ...chaining a new call to **then**
  - ...create a sequence of operations to be executed asynchronously

```
// prints file contents and file length on screen using promises
// filename is received as command-line arg
fs = require("fs");

readFileAsync = function(filename) {
	return new Promise(function (resolve, reject) {
		fs.readFile(filename, function(error,text) {
			if (error) reject(error)
			else resolve(text)
		})
	})
}

// Use the promise, chaining multiple then()s, if needed.
readFileAsync(process.argv[2])
.then(text => {console.log(text); return text})
.then(text => {console.log(text.length); return text})
.catch(error => {console.log("Errors reading file..."); console.log(error)})
```

A2. Promises. Error management

- An exception activates the **catch** branch
  - This is specially useful when promises are nested
    - Errors are propagated through the chain
    - Exceptions in promise handlers are converted into rejected promises
    - If there is an error in any of these steps, all subsequently returned promises (following the promise chain) will be rejected
  - If **catch** is not used, nothing is run in case of error
    - Instead, an exception will be raised at the end of that chain and the process will be aborted
- Advantages (compared with callbacks)
  - Not all the callbacks have a parameter for error notification
    - When no parameter exists, error management becomes impossible
    - When there is any, the code may be difficult to read

A2. Promises. Error management. Example

<table>
<tr><th></th><th>With callbacks</th><th>With promises</th></tr>
<tr><td>Without error management</td><td><pre>getUser("someone",
function(err,user) {
   getBestFriend(user,
function(er,friend) {
              showBestFriend(friend)
   })
})</pre></td><td><pre>getUser("someone")
.then(getBestFriend)
.then(showBestFriend)</pre></td></tr>
<tr><td>With error management</td><td><pre>getUser("someone",
function(err,user) {
   if (err) showError(err)
   else {
    getBestFriend(user,
function(er,friend){
          if (er) showError(er)
          else {
             showBestFriend(friend)
          }
     })
   }
})</pre></td><td><pre>getUser("someone")
.then(getBestFriend)
.then(showBestFriend)
.catch(showError)</pre></td></tr>
</table>

A2. Promises. Generation

- With a constructor (see example in page 59)
  - **new** Promise(**function** (resolve, reject) **{...})**
  - This anonymous function should call
    - reject(condError) in case of error
      - **condError** is an error description
    - resolve(result) when the promise is fulfilled
      - **result** is its computed result
- Promise.resolve(x) returns a fulfilled promise with value x
- Promise.reject(err) returns a rejected promise, being err the cause of that rejection

```
asyncFunc = function(args) {
      return new Promise(
               function (resolve, reject) {
                     ....
                     if (error) reject(error)
                     else resolve(result)
               })
      })
}
```

A2. Promises. all()

- Promise.all(arg):
  - To be used when some code fragment should be executed once all the promises in the **arg** array have been fulfilled.
  - This method returns a promise that is...:
    - Rejected if any of the promises in the array has been rejected
    - Fulfilled when all the promises in the array have been fulfilled
  - See the example in the following page:
    - It is based on the example that reads a file
    - It has been extended for showing the length of all files whose names are passed as arguments and their total length (addition of all their lengths)
      - In order to show that total size we need that all reads have been completed

A2. Promises. all(). Example

```
// Prints the length of multiple files on screen.  All file names are given as arguments. Optimistic version (errors not handled)
var fs = require("fs");

readFileAsync = function(filename) {
      return new Promise(function (resolve, reject) {
            fs.readFile(filename, function(error,text) {
                  if (error) reject(error)
                  else resolve(text)
            })
      })
}

var names=process.argv.slice(2) //Array of file names
if (!names.length) {console.log("Introduce the file names as arguments, please!");  return}

var myFiles=[] // Array of promises
var numFiles=names.length, allLength=0; // Number of files to be processed, Accumulated length

function showAll( ) {console.log( "Total: " + allLength )}      // print total length
function showLength(name) {
   return function(text) {                              // print file name and length
      allLength+=text.length;
      console.log( name + ": " + text.length )
   }
}
function showErrors( err ) {console.log(err.message)}  // print error info
function showFinalError( err) {console.log( "errors accessing some files. Unable to compute overall
length")}

for (f in names) {
 var pr;
   myFiles.push( pr=readFileAsync(names[f]) )
   pr.then(showLength(names[f])).catch(showErrors))
}
Promise.all(myFiles).then(showAll).catch(showFinalError)
```

A2. Promises. all(). Comments on the example

- When there are no errors:
  - All promises will be fulfilled
    - A message is shown for each file
      - Showing its name and length
    - A closure has been used for reporting the correct file name
    - This functionality is provided by the then() call on each myFiles component
  - When the then() associated to the all() result is run...
    - The computed total length is shown
- If there has been any error (file not found, file permissions, ...):
  - The error message associated to that erroneous file is shown, using the **showErrors** function
  - all() returns immediately, shown a general error message
    - Using **showFinalError** to this end