Javascript and Node.js

Autonomous Software Agents - Lab

Marco Robol - marco.robol@unitn.it


JAVASCRIPT
We don't have to worry about complex stuff like memory management
JAVASCRIPT IS A HIGH-LEVEL, OBJECT-ORIENTED, MULTI-PARADIGM PROGRAMMING LANGUAGE. 🤯
Based on objects, for storing most kinds of data
We can use different styles of programming
Instruct computer to *do* things
JS

Javascript

> https://developer.mozilla.org/javascript - JavaScript (JS) is a lightweight, interpreted, or just-in-time compiled programming language with first-class functions. While it is most well-known as the scripting language for Web pages, many non-browser environments also use it, such as Node.js, Apache CouchDB and Adobe Acrobat. JavaScript is a prototype-based, multi-paradigm, single-threaded, dynamic language, supporting object-oriented, imperative, and declarative (e.g. functional programming) styles. ...

> https://www.w3schools.com/js/ - This tutorial will teach you JavaScript from basic to advanced.

Node.js

Node.js is a server-side platform built on Google Chrome's JavaScript Engine (V8 Engine). Node.js uses an event-driven, non-blocking I/O model that makes it lightweight and efficient, perfect for data-intensive real-time applications that run across distributed devices.

You can install Node.js by following the instructions from the Node.js project webpage (https://nodejs.org/en/).

Software needed for coding

To start coding Javascript and run the scripts in the Node.js we need:

- Text editor (e.g. Visual Studio Code, Brackets, Sublime Text,...)
- Node.js (https://nodejs.org/it/download/) (with npm included)

In addition, we will use:

- Git CLI (https://git-scm.com/downloads)
- github.com (you can activate a pro account creating a new account with the @unitn email)
- Browser web (e.g. Chrome)

For today, you can quickly develop your code on replit.com

Basic scripting

Let's open our editor and create a file named *hello.js*

```
/* Hello World! program in Node.js */
console.log("Hello World!");
```

Running the script

```
$ node hello.js
```

As you can see, we are simply echo-ing the contents in the console. We can achieve the same using the interactive console by simply typing *node* in our terminal. For example:

```
$ node
> console.log("Hello World!");
Hello World!    // this is the result of executing console.log()
undefined       // this is the returned value from console.log()
```

Javascript basics

Structure of a JS script:

Sequence of statements:

- declaration of a variable
  - const, var, let (similar to var but with scope limited to block)
  - JavaScript is an untyped language
- declaration of a function
- function call
- control flow statements
  - if, switch, for, while

Scope of variables

global scope

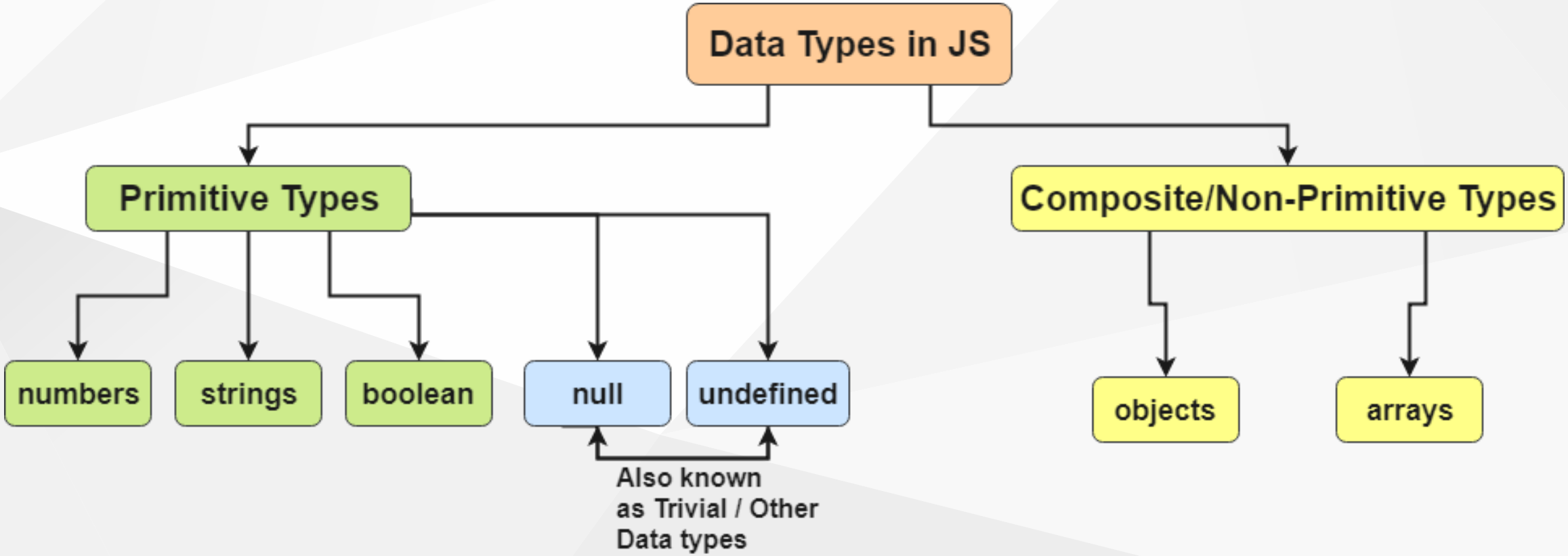
```
const me = 'me';     // outside of any function or block
```

function scope

```
function myF () { const me = 'me'; }
console.log(me)      // ReferenceError
```

block scope

```
if (year>2022) { const me = 'me'; var you = 'you'; }
console.log(me)      // ReferenceError
console.log(you)     // you
```


Data Types in JS
Primitive Types
Composite/Non-Primitive Types
numbers
strings
boolean
null
undefined
objects
arrays
Also known as Trivial / Other Data types

Types

```
var myvar;
console.log(typeof (myvar));              // undefined

myvar = 'Pippo';                          // string
myvar = 5;                                // number
myvar = true;                             // boolean
myvar = [1,2,3];                          // object
myvar = {key1: "value1"};                 // object
myvar = null;                             // object
myvar = function(n){return n+1};          // function

console.log(Array.isArray([1,2,3]));      // true
```

Lists

```
var list = ["apple", "pear", "peach"];  //list of elements
list[0]                                 // accessing an element by id
list.indexOf("pear")                    // checking the index of "pear" in the array
list.push("banana");                    // Adding a new element
list.pop()                              // Taking the last element from the array
list.shift()                            // Taking the first element
list.length                             // checking the number of elements
list.slice(start, end)                  // copy a subportion of the original array
list.join('separator')                  // return string by concatening elements
```

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array

Logical operator

```
// equality
2 == 2    // true
2 == '2'  // true

// inequality
2 != 2    // false
2 != '2'  // false

// strict equality
2 === 2   // true
2 === '2' // false

// strict inequality
2 !== 2   // false
2 !== '2' // true
```

Matematical operators

```
// greater than
3 > 2   // true
// less than
3 < 5   // true
// greater than or equal to
3 >= 3  // true
// less than or equal to
3 <= 3  // true
```

Loops

```
while (condition) { console.log('do') }
```

```
for (var i=0; i< 100; i++) {
    if ((i%2)==0) continue;      // if even, skip to the next cycle
    console.log(i);              // else, print i
    if (i>=10) break;            // when greater equal then 10, quit the loop
}
```

```
for (let value of ['first','second']) {
    console.log(value)           // value is the item in the array
}
```

```
[1,2,3].forEach( console.log )      // callback receives the item as parameter

[1,2,3].forEach( v=>console.log(v) )// callback defined as an arrow function
```

Functions

How to define a function

```
function add(a, b) {
    return a + b;
};

var mult = function (a, b) { return a * b; };

var arrowFunction = (a,b) => a * b;
```

How to call a function

```
add(1,2)                // 3
mult(1,2)               // 2
arrowFunction(2,2)      // 4
```

Objects

Define an object without defining the class

```
var car = {
    type : 'Fiat',
    model : '500',
    color : 'red',
    description : function() {
        return this.color + ", " + this.model + ", " + this.type;
    }
    // methods cannot be defined using arrow functions!
    // in the case of arrow functions, context 'this' is not assotiated to the object
}
console.log(car);
console.log(car.description());
```

Patterns to simulate classes using functions

Define a class by using a function. Instantiate a new object using the function constructor.

```
function Car(type, model, color) {
    this.type = type;
    this.model = model;
    this.color = color;
    this.description = function() {
        return this.color + ", " + this.model + ", " + this.type;
    };
}
var fiat500rossa = new Car('Fiat', '500', 'red');
console.log(fiat500rossa);
console.log(fiat500rossa.description());

// Never call a constructor function directly
// e.g. Car('Fiat', '500', 'white');
```

Patterns to simulate classes using prototypes

```
function Car2(type, model, color) {
    this.type = type;
    this.model = model;
    this.color = color;
}

Car2.prototype.description = function() {
    return this.color + ", " + this.model + ", " + this.type;
};
var fiat500bianca = new Car2('Fiat', '500', 'white');
console.log(fiat500bianca);
console.log(fiat500bianca.description());
```

Define a class by using the new reserved 'class' keyword of ES6

```
class Car3 {
    constructor(type, model, color) {
        this.type = type;
        this.model = model;
        this.color = color;
    }
    description() {
        return this.color + ", " + this.model + ", " + this.type;
    };
}
var fiatPuntobianca = new Car3('Fiat', 'Punto', 'white');
console.log(fiatPuntobianca);
console.log(fiatPuntobianca.description());
```

Extend a class with ES6

```
class Suv extends Car3 {
    description() {
        return this.color + ", " + this.model + ", " + this.type + ", SUV";
    };
}
var NissanQuashqai = new Suv('Nissan', 'Quashqai', 'black');
console.log(NissanQuashqai);
console.log(NissanQuashqai.description());
```

Exercises

Arrays: n3 and n8 - https://medium.com/@andrey.igorevich.borisov/10-javascript-exercises-with-arrays-c44eea129fba

Functions: n18 - https://www.w3resource.com/javascript-exercises/javascript-functions-exercises.php

Advanced

Blocking vs Non-Blocking

https://nodejs.org/en/docs/guides/blocking-vs-non-blocking/

Blocking is when the execution of additional JavaScript must wait until a non-JavaScript operation completes. This happens because the event loop is unable to continue running JavaScript while a blocking operation is occurring. Blocking methods execute synchronously and non-blocking methods execute asynchronously.

```
const fs = require('fs');
const data = fs.readFileSync('/file.md'); // blocks here until file is read
```

```
const fs = require('fs');
fs.readFile('/file.md', (err, data) => {
  if (err) throw err;
}); // continue executing the javascript code while waiting for the file
```

Callbacks

https://nodejs.org/en/knowledge/getting-started/control-flow/what-are-callbacks/

In a synchronous program, you would write something along the lines of:

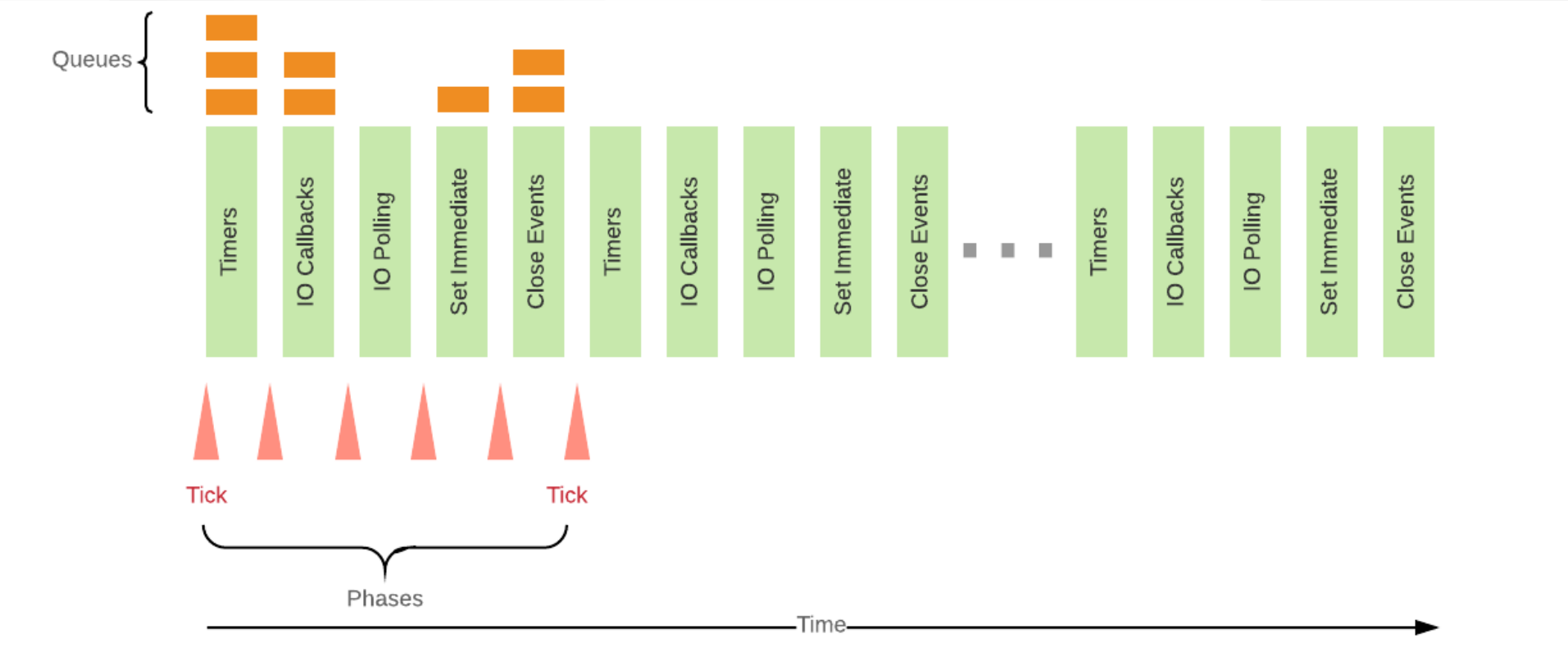
```
var data = fetchData (); // block the whole program waiting for the data
console.log(data); // do something with the fetched data
```

A callback is a function called at the completion of a given task; this prevents any blocking, and allows other code to be run in the meantime.

```
fetchData(function (data) {
    console.log(data); // do something with the fetched data
});
```

Node.js, being an asynchronous platform, uses callbacks to avoid waiting for things like file I/O to finish.

The Node.js Event Loop



https://medium.com/the-node-js-collection/what-you-should-know-to-really-understand-the-node-js-event-loop-and-its-metrics-c4907b19da4c

Promises

There are different ways to handle operations in NodeJS or in JavaScript. For asynchronous execution, different processes run simultaneously and are handled once the result of each process is available. There are different ways to handle the asynchronous code in NodeJS or in JavaScript which are:

-Callbacks
-Promises https://web.dev/promises/
-Async/Await

https://www.geeksforgeeks.org/difference-between-promise-and-async-await-in-node-js/#:~:text=Promise is an object representing,– resolved%2C rejected and pending.

A promise is an object having three possible states:

- Pending: Initial State, before the event has happened.
- Resolved: After the operation completed successfully.
- Rejected: If the operation had error during execution, the promise fails.

A Promise in NodeJS is similar to a promise in real life. It is an assurance that something will be done. Promise is used to keep track of whether the asynchronous event has been executed or not and determines what happens after the event has occurred.

```
var myPromise = new Promise(function (res) {
    // async code to be executed, when ready, call res() to resolve the promise
    res('any_value');
});
// when not yet resolved
console.log(myPromise);      //Promise { <pending> }
// once resolved
console.log(myPromise);      //Promise { 'any value' }
```

For a successfully resolved promise, we use .then() method and for rejected promise, we use .catch() method. To run a code after the promise has been handled using .then() or .catch() method, we can .finally() method. The code inside .finally() method runs once regardless of the state of the promise.

```
promise
    .then(function () {
      console.log("Promise resolved successfully");
    })
    .catch(function () {
      console.log("Promise is rejected");
    });
```

Callbacks vs. Promises:

```
callbackBasedFunction(callbackWithErrorHandling)

promiseBasedFunction.then(callback).catch(errorHandling)
```

Example with timeout

Suppose we want a sequence of timeouts. With callback-based setTimeout() we have:

```
setTimeout(()=>{
    setTimeout(()=>{
            console.log(123)
        }, 5000)    // later, wait for another 5 seconds
}, 1000)    // first wait 1 second
```

After promisifing the setTimeout:

```
myPromisifiedTimeout = function (time) {
    return new Promise( (res) => setTimeout(res, time) )
}
```

```
myPromisifiedTimeout(1000)                  //first wait for 1 second
.then(()=>{return myPromisifiedTimeout(5000)})   // later wait for additional 5
.then(()=>console.log(123))
```

Example: While accessing a file, the Promise is in a pending state. The Promise is resolved after reading the file, or rejected if file could not be read.

```
const fs = require('fs');

const readPromisify = function (file) {
    return new Promise(function (resolve, reject) {
        fs.readFile(file, (err, data) => {
            if (err) throw reject(err);
            resolve(data); data
        });
    });
}

readPromisify('/file.md')
.then(function (data) {
      console.log("Promise resolved successfully");
})
.catch(function (err) {
    console.log("Promise is rejected");
});
```

Error Handling of Promises: For a successfully resolved promise, we use .then() method and for rejected promise, we use .catch() method. To run a code after the promise has been handled using .then() or .catch() method, we can .finally() method. The code inside .finally() method runs once regardless of the state of the promise.

```
promise
    .then(function (value) {
      console.log("Promise resolved successfully");
    })
    .catch(function (err) {
      console.log("Promise is rejected");
    });
```

Asynch/await

https://javascript.info/async-await

The word “async” before a function means one simple thing: a function always returns a promise. Other values are wrapped in a resolved promise automatically.

```
async function f() { return 1; }
f().then(alert); // 1
```

Await works only inside async functions. It waits for a promise to resolve, then return resolved value.

```
async function f() {
  let result = await promise; // wait until the promise resolves (*)
  alert(result); // "done!"
}
```

Timeout example

```
function example() {
    myPromisifiedTimeout(1000)                  //first wait for 1 second
    .then(()=>{return myPromisifiedTimeout(5000)})   // later wait for additional 5
    .then(()=>console.log(123);)
}
example();
```

```
async function asyncExample() {
    await myPromisifiedTimeout(1000);   //first wait for 1 second
    await myPromisifiedTimeout(5000);   // later wait for additional 5
    console.log(123);
}
asyncExample();
```

Generator function

```
function* generator(i) {
  yield i;
  yield i + 10;
}

const gen = generator(10);

console.log(gen.next().value);
// expected output: 10

console.log(gen.next().value);
// expected output: 20
```

> https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/function*

Iterating over generators

Since generators are iterables, you can implement an iterator in an easier way. Then you can iterate through the generators using the for...of loop.

```
function* generatorFunc() {
    yield 1;
    yield 2;
    yield 3;
}
const obj = generatorFunc();
// iteration through generator
for (let value of obj) {
    console.log(value);
}
```

> https://www.programiz.com/javascript/generators

Exercises

- Promises and async programming
  https://www.codingame.com/playgrounds/347/javascript-promises-mastering-the-asynchronous/what-is-asynchronous-in-javascript

Package mangement

Loading libraries

The Node.js installation comes with standard modules, e.g. 'fs' to access the file system. This module comes with the standard Node.js installation, so we do not need to install any third-party libraries (We'll get to that later in this tutorial).

```
var fs = require("fs");
```

The require instruction above loads the module "fs" and assigns an instance to the variable fs. Through this instance then we can have access to all the functions exported by that module.

> http://fredkschott.com/post/2014/06/require-and-the-module-system/.

Creating and Exporting a Module

./user.js

```
function userTemplate(user) {
  return `Name: ${user.name}`;
}
module.exports = userTemplate;
```

./index.js

```
const userTemplate = require('./user');
console.log( userTemplate({name:'marco'}) );
```

> https://www.sitepoint.com/understanding-module-exports-exports-node-js/

Package mangement with npm

NPM is a very powerful tool that can help you manage project dependencies and in general automate development workflows, much like `ant` or `make` in java and C.

The file `package.json` contains the metadata regarding your project, including name, version, license, and dependencies. Although you can install dependencies without a `package.json` file, it is the best way to keep track of your local dependencies.

> https://nodesource.com/blog/an-absolute-beginners-guide-to-using-npm/

Package.json

How do we start? We execute the command below and follow the instructions prompted.

```
npm init
```

This generates the `package.json` file.

Installing modules

To install an external module, we can use the `npm install` command

```
npm install express
```

The save params indicates npm to add the module to the list of dependencies in the `package.json` file. Indeed, if you check its contents, you'll now see:

```
{
  "name": "hello",
  ...
  "dependencies": {
    "express": "^4.16.3"
  }
  ...
}
```

Installing all npm dependencies

When someone shares the source code of their project (on a github, other source code management system, but even on a memory stick), they will not put their local dependency builds with their source code but give you only the `package.json` dependecies.

Let us "uninstall" express for a second, using `npm uninstall express` (if you add --save you'll also remove it from `package.json` but that's not what we want in this case). This removes the module from our project, and put it at the state you'll find any project on github. The way you install the dependencies of the project is then with the following command.

```
npm install
```

Project

How to shift from object-oriented to agent-oriented programming paradigm?

- How can we implement BDI agents and run them in a shared environment?
- What are the key concepts we want to consider?

How to implement a BDI agent

- **Reasoning loop** - Must be non-blocking. Implemented on top of the node.js event loop. For example, consider using setTimeout().
- **Representation of the world** (belief) - facts composed by a predicate and by parameters. Example: light_on(room1), in_room(marco, kitchen)
- **Plan set** - Given a desire, the agent can search in his plans set, looking for an applicable plan. Different plans could be available to achieve the same desire, each applicable in a different initial situation.
- Goal structured into **Subgoals** - A plan (intention)can push to the agent himself a subgoal (desire) and pausing its execution until the subgoal is successful.
- **Introspection**: during the execution of intentions, the agent can reason on his acting and thinking
- **Modularity**: agent can be build as a set of intentions/plans

How to implement the environment

- **Sensors** - Based on observability of events/facts - Agents can subscribe to given facts and being notified when changes happen.
- **Agent actions** create effects on the environments. **The environment applies the effects**. The agent himself cannot force effects to be applied on the environment.
- **Knowledge Representation** - Fact-based representation of the environment should include house structure and devices (status and actions). For example: door(kitchen, living_room), stairs(kitchen, garage), light_on(bedroom), car_charging(car1)

Thank you

Questions?

marco.robol@unitn.it