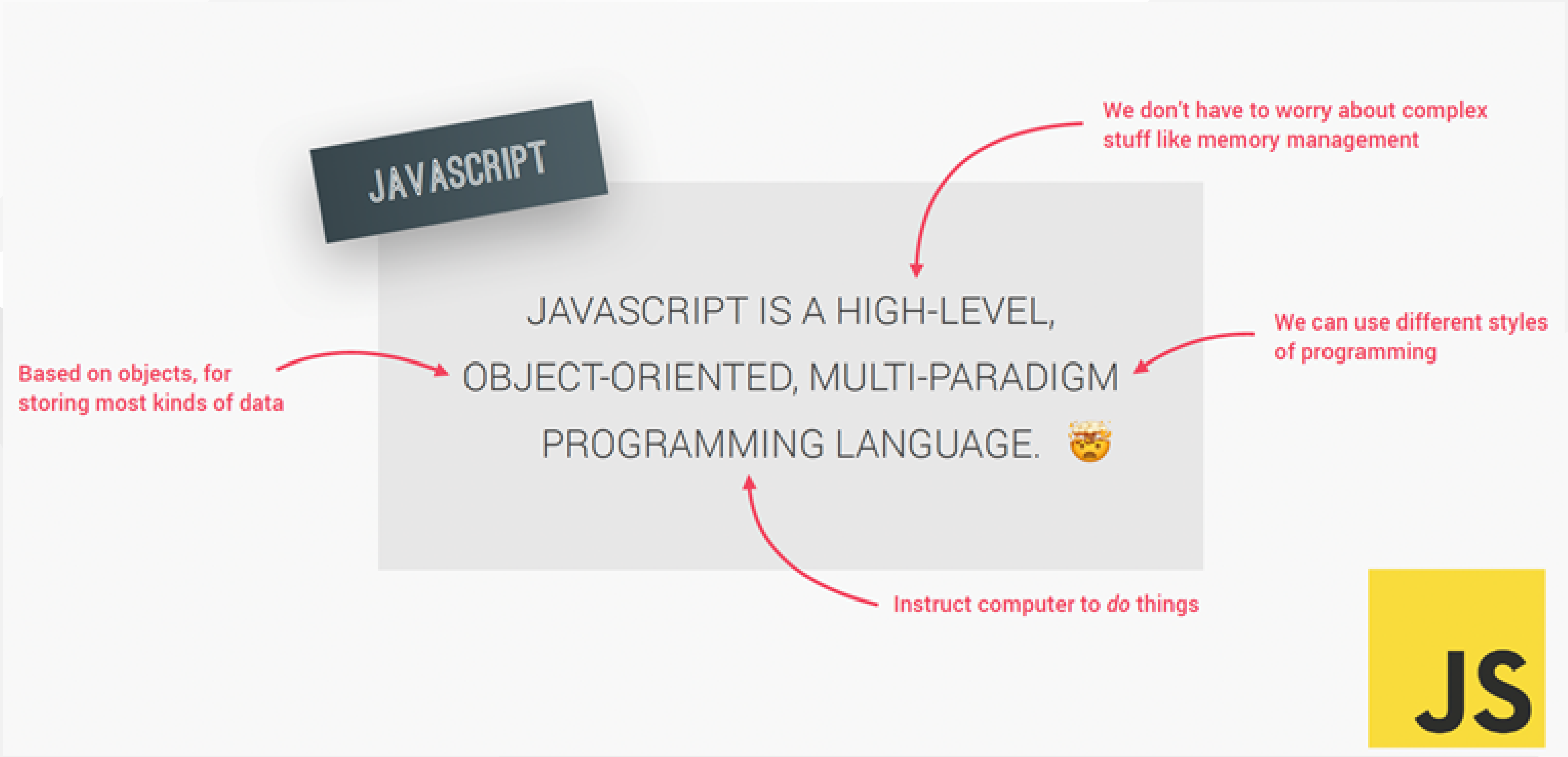
Javascript and Node.js

Autonomous Software Agents - Lab

Marco Robol - marco.robol@unitn.it



JAVASCRIPT

JAVASCRIPT IS A HIGH-LEVEL, OBJECT-ORIENTED, MULTI-PARADIGM PROGRAMMING LANGUAGE. 🤯

We don't have to worry about complex stuff like memory management

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/)

In addition, we will use:

- Git CLI - get it from https://git-scm.com/downloads
- Github - you can upgrade your account by applying to education program at https://education.github.com/students and verifying your @*studenti.unitn.it* mail
- Browser web (e.g. Chrome)

To start, you can quickly develop your code on https://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

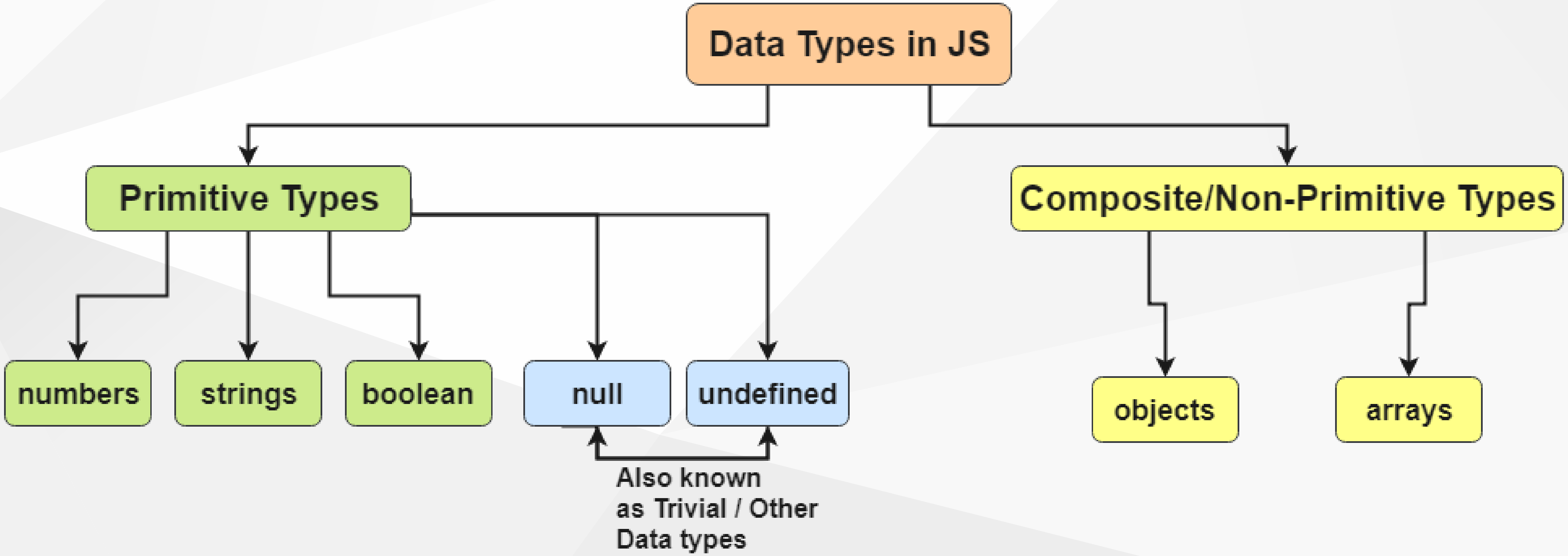
```
$ node hello.js
```

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

> https://javascript.info/first-steps
>
> Types, Functions, Control flow statements, Objects and Classes


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 // Array.isArray(myvar) // true
myvar = {key1: "value1"};             // object
myvar = null;                         // object
myvar = function(n){return n+1};      // function
```

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

Scope of variables (& constants): global, block, function

```
const global_const = 'global_const';  // globally-scoped constant
var global_var = 'global_var';        // globally-scoped variable

function myFn () {

    console.log(global_const)         // 'global_const'
    console.log(global_var)           // 'global_var'

    if ( true ) {
        const constant = 'constant';  // block-scoped constant
        let local = 'local';          // block-scoped (local) variable
        var variable = 'variable';    // function-scoped variable
    }

    console.log(constant)             // ReferenceError
    console.log(local)                // ReferenceError
    console.log(variable)             // 'variable'
}

console.log(variable)                 // ReferenceError
```

Control flow statements

```
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
```

Logical and Mathematical operators

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

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

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

Functions

How to define a function

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

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

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

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 associated to the object
}
console.log(car);
console.log(car.description());
```

`this` keyword

> `this` keyword behaves a little differently in JavaScript compared to other languages. In most cases, the value of this is determined by how a function is called (runtime binding) and it may be different each time the function is called. The bind() method can set the value of a function's this regardless of how it's called, and arrow functions don't provide their own this binding (it retains the this value of the enclosing lexical context). See .bind() and .call()

```
function sum (a) {
    return this + a;
}

var bindedSum = description.bind(2);
bindedSum(3); // 6

description.call(2,3) // 6
```

Patterns to simulate classes using functions

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

```javascript
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()); // this keyword get bounded to fiat500rossa

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

Patterns to simulate classes using prototypes

In programming, we often want to take something and extend it. *Prototypal inheritance* is a language feature that helps in that. https://javascript.info/prototypes

```
let animal = {
    eats: true,
    walk() { console.log("Animal walk"); }
};
let rabbit = {
  jumps: true,
  __proto__ = animal // or later do rabbit.__proto__ = animal
};

// we can find both properties in rabbit now:
console.log( rabbit.eats ); // true
console.log( rabbit.jumps ); // true
// walk is taken from the prototype
rabbit.walk(); // Animal walk
```

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

```javascript
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());
```

Hands-on

- 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

Es 1 - Arrays - Compact - Write a method that clears array from all unnecessary elements, like false, undefined, empty strings, zero, null

```
/**
  * Task description: Write a method that clears array from all
  * unnecessary elements, like false, undefined, empty strings, zero, null
  * Expected Result: [0, 1, false, 2, undefined, '', 3, null] => [1, 2, 3]
  * Task Complexity: 1 of 5
  * @param {Array} array - An array of any elements
  * @returns {Array}
*/
const compact = (array) => {
 throw new Error('Put your solution here');
}
const data = [0, 1, false, 2, undefined, '', 3, null];
console.log(compact(data)) // [1, 2, 3]
```

> https://medium.com/@andrey.igorevich.borisov/10-javascript-exercises-with-arrays-c44eea129fba Es 3

Es 2 - Arrays - Flatten - Write a function that turns a deep array into a plain array. Please, do not use array.flat() to make this task more enjoyable.

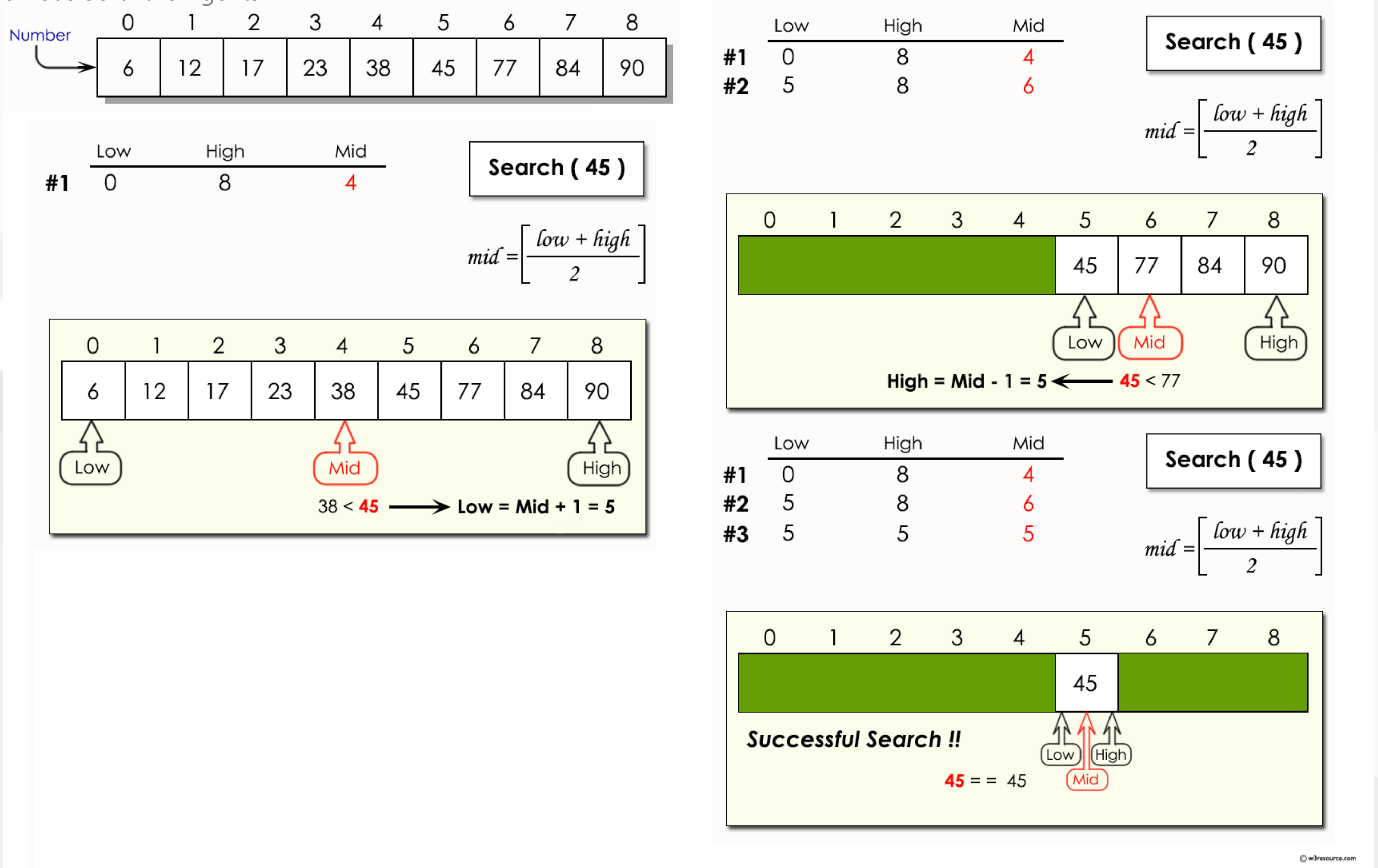
```
/**
  * Task description: Write a method that turns a deep array into a plain array
  * Expected Result: [1, 2, [3, 4, [5]]] => [1, 2, 3, 4, 5]
  * Task complexity: 3 of 5
  * @param {Array} array - A deep array
  * @returns {Array}
*/
const flatten = (array) => {
 throw new Error('Put your solution here');
}
const data = [1, 2, [3, 4, [5]]];
console.log(flatten(data)); // [1, 2, 3, 4, 5]
```

> https://medium.com/@andrey.igorevich.borisov/10-javascript-exercises-with-arrays-c44eea129fba Es 8

Es 3 - Functions - Write a function for searching JavaScript arrays with a binary search

Note : A binary search searches by splitting an array into smaller and smaller chunks until it finds the desired value.

> https://www.w3resource.com/javascript-exercises/javascript-functions-exercises.php Es 18



	Low	High	Mid
#1	0	8	4

	Low	High	Mid
#1	0	8	4
#2	5	8	6

	Low	High	Mid
#1	0	8	4
#2	5	8	6
#3	5	5	5

$$mid = \left\lfloor \frac{low + high}{2} \right\rfloor$$

Asyncronous programming

An example - accessing the file system synchronously

```
var fs = require( "fs" );
var data = fs.readFileSync( "file.txt", "utf8" );
console.log( data );
console.log( "Program ended" );
```

```
$ node my_script.js
```

Operations are exececuted in **sequence**, you see the contents of the file and then the *Program ended* message.

An example - accessing the file system asynchronously

Let's try now an alternative implementation:

```
var fs = require( "fs" );
fs.readFile( "file.txt", "utf8", function(error, data) {
  console.log(data);
} );
console.log("Program ended.");
```

```
$ node my_script.js
```

In this case readFile expects a *callback* function, that is called when the file is ready. So that *Program ended* message cames first, followed by the contents of the file.

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
```

Asynchronous Programming with 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:

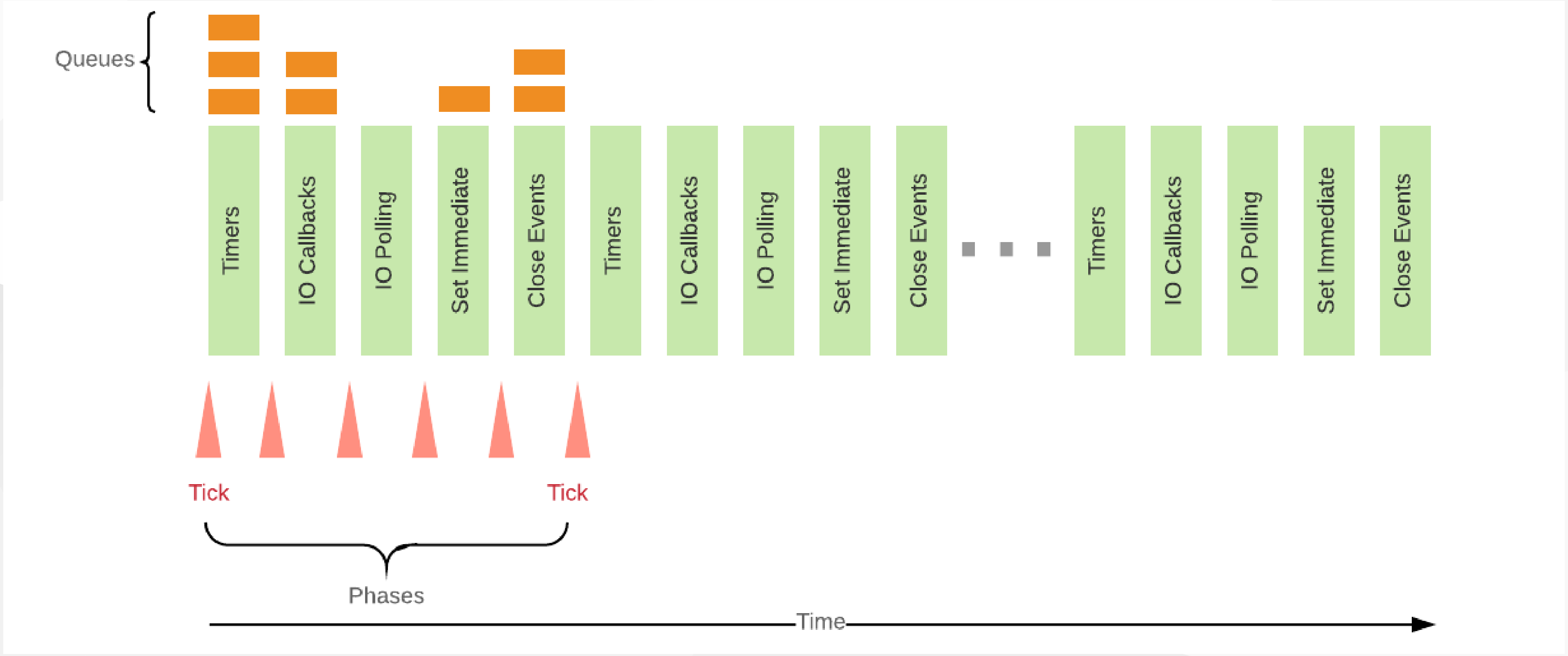
```
var data = fetchData (); // block the whole program waiting for the data
console.log(data); // do something with the fetched data
```

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

An example: setTimeout

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

```
setTimeout(()=>{
    console.log('1000ms');          // 1000ms
    setTimeout(()=>{
            console.log('5000ms')   // 5000ms
        }, 5000)                    // then, wait for another 5 seconds
}, 1000)                            // first, wait 1 second
```

... next time we will see how to improvement readability of asynchronous programming with *promises* and *async/await*.

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

Hands-on

Javascript promises - mastering the asynchronous - **codingame.com**

> Steps 1 to 4: https://www.codingame.com/playgrounds/347/

Modules and package mangement

> https://javascript.info/modules

Modules systems

As our application grows bigger, we want to split it into multiple files, so called "modules". A module may contain a class or a library of functions for a specific purpose. So the community invented a variety of ways to organize code into modules, special libraries to load modules on demand.

- **AMD** – initially implemented by the library require.js.
- **CommonJS** – the module system created for Node.js server.

> https://javascript.info/modules-intro

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");     // AMD
import fs from 'fs';        // CommonJS
```

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
export function userTemplate(user) {      // CommonJS
  return `Name: ${user.name}`;
}
module.exports = userTemplate;            // AMD
export {userTemplate as template};        // CommonJS
```

```
// index.js
const userTemplate = require('./user');    // AMD
import {userTemplate} from './user.js';    // CommonJS
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 a module

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 project dependencies

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

The way you **install** all the dependencies of the project is with the following command. This creates the `node_modules` folder with all the local dependency builds.

```
$ npm install
```

We can **uninstall** modules using the following command. This will removes the module from the `node_modules` folder and also from `package.json` project.

```
$ npm uninstall <module_name> --save
```

Hands-on: npm

`package` - Easy package.json exports - This module provides an easy and simple way to export package.json data - https://www.npmjs.com/package/package

- Create a new npm project
- Install `package` npm package
- Use the installed package:

```
var package = require('package')(module); // contains package.json data.
var yourAwesomeModule = {};
yourAwesomeModule.version = package.version;
```

- Commit code on git without node_modules folder

Thank you

Questions?

marco.robol@unitn.it