

# W1 PRACTICE


## *JSES6, JS Doc, Modules, File IO*

 *At the end of his practice, you should be able to...*


- ✓ Set up and run a Node.js project
- ✓ Manipulate **JSDoc** annotation
- ✓ Review key **modern JS** (ES6+)
- ✓ Work with **customs** and **core modules**
- ✓ Practice file system operations (**sync/async**)
- ✓ Manage dependencies and **package.json**

 *How to start?*

- ✓ Download **start code** from related MS Team assignment
- ✓ Run `npm install` when needed to install the dependencies

 *How to submit?*

- ✓ Submit your **code** on MS Team assignment

 *Are you lost?*

*You can read the following documentation to be ready for this practice:*

[https://www.w3schools.com/js/js\\_modules.asp](https://www.w3schools.com/js/js_modules.asp)

<https://www.freecodecamp.org/news/javascript-modules-explained-with-examples>

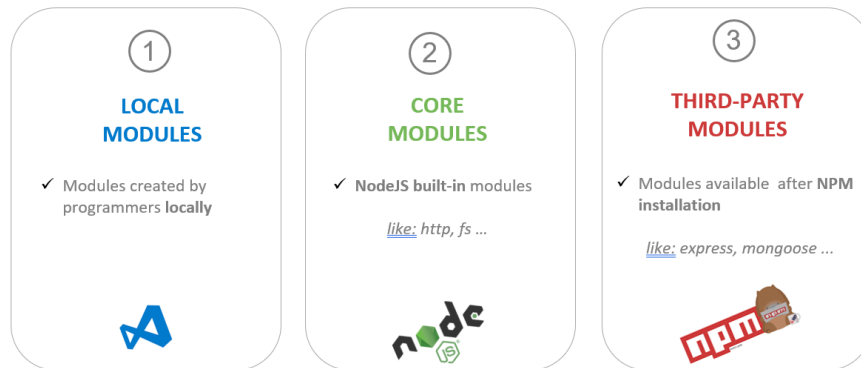
<https://www.scaler.com/topics/nodejs/require-vs-import-nodejs>

# BEFORE STARTING...

## JS Node modules

Do you remember the **usage of modules** you learned during Front End Development course ?

- ✓ Take some time [to review the slides](#) (handle *local*, *core* and *third-party* modules).



*Review the TERM 2 - WEEK 3 – Modules & Packages slides*

- ✓ You can also have a look on bellow documentation:  
[https://www.w3schools.com/js/js\\_modules.asp](https://www.w3schools.com/js/js_modules.asp)  
<https://www.freecodecamp.org/news/javascript-modules-explained-with-examples>  
<https://www.scaler.com/topics/nodejs/require-vs-import-nodejs>  
<https://webpack.js.org/guides/ecma-script-modules/>

## JS Doc

JSDoc's purpose is to document the API of your JavaScript application or library. It is assumed that you will want to document things like modules, namespaces, classes, methods, methods, parameters, and so on.

<https://jsdoc.app/about-getting-started>

- ✓ JSDoc comments should generally be placed immediately before the code being documented.
- ✓ Each comment must start with a `/**` sequence in order to be recognized by the JSDoc parser.
- ✓ Comments beginning with `/*`, `/**`, or more than 3 stars will be ignored.

```
jsdoc book.js
```

*Generate the doc from a well-documented code*

# EXERCISE 1 – File IO

For this exercise, you will start from scratch (not start code).

**Q1 -** Create a new project (EX-1) and initialize the package.json file:

```
npm init -y
```

**Q2 -** Open package.json and add "type": "module" :

```
"type": "module",
```

Now JavaScript files in your project should be treated as **ECMAScript Modules (ESM)** rather than **CommonJS (CJS)** modules.

**Q3 –** Install the following modules **globally**

Module	Goal
nodemon	Automatically restarts your Node.js app whenever you save changes to your files.
jsdoc	Generates HTML documentation from specially formatted comments in your JavaScript code.

**Q4 –** Create a new File Student.js with the following code

```
const filePath = "./hello.txt";

// Write to a file (synchronously)
fs.writeFileSync(filePath, "Hello, Node.js beginner!");

// Read the file (synchronously)
const content = fs.readFileSync(filePath, "utf8");
console.log("File content:", content);
```

- Run this code using the node.js command:

```
node ./Student.js
```

- *The code does not work as we missed the FS import, fix this issue !!!*

- Run the code with nodemon

```
nodemon ./Student.js
```

- *What's happen when you change and save you code now?*

Nodemon automatically restarts the app. You don't need to manually run the file again — it keeps watching for changes and reruns the program for you. This is very helpful for development.

**Q5 –** Convert the read and write into **async versions**

## EXERCISE 2 – Handle Durations

For this exercise, you will start with the start code.

You will handle an **immutable Duration module** in JavaScript.



**Q1 -** Complete the Duration class:

- ✓ The Duration class represents a duration of time in seconds.
- ✓ The Duration constructor takes a number of seconds as parameter

Complete the constructor and the static builder method:

```
class Duration {  
    /**  
     * Total duration in seconds.  
     * @type {number}  
     * @private  
     */  
    _totalSeconds;  
  
    /**  
     * Creates a new Duration object.  
     * @param {number} [seconds=0] - The number of seconds.  
     */  
    constructor(seconds) {  
        // YOUR CODE  
    }  
  
    /**  
     * Creates a new Duration from a number of minutes and seconds.  
     * @param {number} [minutes=0] - The number of minutes.  
     * @param {number} [seconds=0] - The number of seconds.  
     * @returns {Duration} A new Duration instance.  
     */  
    static fromMinutesAndSeconds(minutes = 0, seconds = 0) {  
        // YOUR CODE  
    }  
}
```

**Q4-** Add **toString** method to return a human readable string **in minutes and seconds**

```
/**
 * Converts the duration into a human-readable string, e.g., "2m 30s".
 * @returns {string} the formatted duration string.
 */
toString = () => {
    // YOUR CODE
}
```

**Q5-** Add the **minus** and **plus** method to minus or add 2 Durations. Also complete the **comments**

```
/**
 * Returns a new Duration by adding another duration.
 * @param {Duration} other - Another duration to add.
 * @returns {Duration} A new Duration representing the sum.
 */
plus = (other) => {
    // YOUR CODE
}

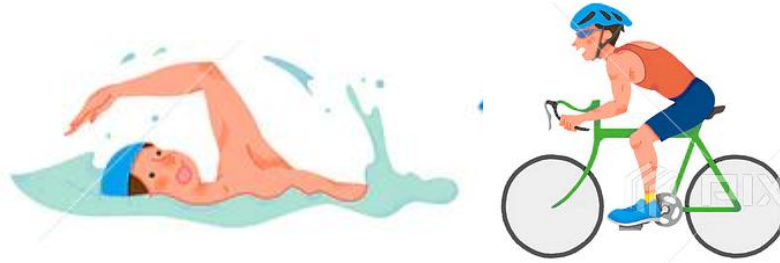
/**
 * YOUR COMMENT
 */
minus = (other) => {
    // YOUR CODE
}
```

**Q6-** Comple the main file:

1. Make sure the Duration module can be imported
2. Test your Duration module

## EXERCISE 3 – Manage Race scores

In this exercise, you will create a module that represents **race results**.



It should:

- ✓ **Add a race result** (store participant ID, sport type, time).
- ✓ **Save the race list** to a JSON file.
- ✓ **Read the race list** from a JSON file.
- ✓ Retrieve the **time for a specific participant** in a given sport.
- ✓ Compute the **total time** for a participant by **summing all of their race times**.

As an example, here is how we should be able to use your RaceResults module :

### *mainInit.js*

```
import { Duration } from "../duration/Duration.js";
import { RaceResults } from "../race-scores/RaceScores.js";

// Initialize RaceResults
const raceManager = new RaceResults();

raceManager.addRaceResult(new RaceResult("participant1", "swim", Duration.fromMinutesAndSeconds(2, 30)));
raceManager.addRaceResult(new RaceResult("participant1", "run", Duration.fromMinutesAndSeconds(1, 45)));
raceManager.addRaceResult(new RaceResult("participant2", "swim", Duration.fromMinutesAndSeconds(3, 15)));
// Save results to file
raceManager.saveToFile("../race-scores/data/raceScores.json");
```

### *mainLoad.js*

```
import { Duration } from '../Duration.js';
import RaceResults from '../raceResults.js';

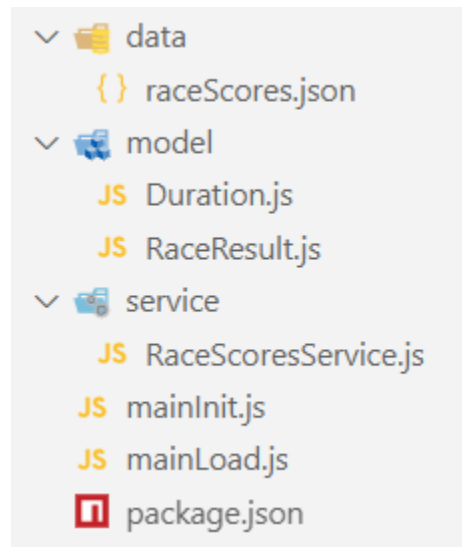
// Load results from file
raceManager.loadFromFile('raceResults.json');

// Retrieve time for a participant and sport
const time1 = raceManager.getTimeForParticipant('participant1', swim);
console.log(time1.toString()); // "2m 30s"

// Compute total time for a participant
const totalTime1 = raceManager.getTotalTimeForParticipant('participant1');
console.log(totalTime1.toString()); // "4m 15s"
```

### Q1- Start Code explanation

- ✓ You are provided with a start code, structured as below:



File/ Folder	Role
Data/raceScores.json	Contain the race scores data (JSON format)
Model/Duration.js	Class defining the structure of a Duration
Model/RaceResult.js	Class defining the structure of a RaceResult
Servicer/RaceScoresService.js	Handle the operation of the race score (read/write and add score)
main_init.js	Create race results data and save it to the JSON file
main_load.js	Load the data from the JSON file and display for information

### Q1 - Model / Duration

*Copy your Duration class into this file*

### Q2- Model / RaceScore

A Race score is composed of:

- A participant id (string)
- A sport type (string)
- A duration (Duration)

Following the Duration class, define the

- Class attributes, *with proper JS Doc comments*
- Class constructor, *with proper JS Doc comments*
- The class RaceResult shall have a list of race results (array)

### Q3 - Service / RaceResultsService / Add

The RaceResultsService handles a list of RaceResults.

The service allows the users to:

- **Add** a new result
- **Save**, load from/to JSON
- **Query** the results

First, complete the add method as described in its comment.

```
/**
 * Adds a new race result to the race list.
 * @param {RaceResult} result - The race result to add.
 */
addRaceResult(result) {
  // TODO
}
```

### Q4- Service / RaceResultsService / Save

1 - Import CORE module FS:

```
import fs from 'fs';
```

2- Implement the method saveToFile to save the list of result to a given file

```
/**
 * Saves the race results list to a JSON file.
 * @param {string} filePath - The path to the file data should be saved.
 */
saveToFile(filePath) {
  // Your code
}
```

#### Notes:

1. convert the list of result into a JSON-formatted string

*JSON.stringify()* converts a JavaScript object (like an array, object, etc.) into a JSON-formatted string

(More information [here](#))

```
JSON.stringify(value);
```

2. Save your string value using the FS core module. More information [here](#).

```
fs.writeFileSync(filePath, data, 'utf8');
```

### Test your code

Run the mainInit.js file and ensure the score is well stored in the specified file



#### Q4- Service / RaceResultsService / Load

Implement the loadFromFile method to read the race results

```
/**
 * Loads the race results list from a JSON file.
 * @param {string} filePath - The path to the file to load data from.
 * @returns {boolean} True if loading was successful, false otherwise.
 */
loadFromFile(filePath) {
    // Your code
}
```

#### Notes:

1. Read the string from the JSON file, using the FS core module.

```
const data = fs.readFileSync(filePath, 'utf8');
```

2. Convert the string into a list of race results, using the JSON parse method

```
this._raceResults = JSON.parse(data);
```

3. To catch errors, **use a try/catch** and print error on console if any

#### Test your code

Run the mainLoad.js file (just the load part) and ensure the

#### Q5- Service / RaceResultsService / getTimeForParticipant

Implement the method getTimeForParticipant:

```
/**
 * Retrieves the race time for a given participant and sport.
 * @param {string} participantId - Participant ID.
 * @param {string} sport - Sport name.
 * @returns {Duration|null} Duration if found, else null.
 */
getTimeForParticipant(participantId, sport) {
    // Your code
}
```

#### Notes:

- [Use the function find](#) on the participant array to find the participant time for given ID and sport name.
- Return the time (duration) if any, or null if no data is found.
- Test your method using the mainLoad.js file

**Q6 - Service / RaceResultsService / getTotalTimeForParticipant**

Implement the method getTotalTimeForParticipant:

```
/**
 * Computes total time for a given participant by summing their race times.
 * @param {string} participant_id - The ID of the participant.
 * @returns {Duration} The total Duration object
 */
getTotalTimeForParticipant(participant_id) {
  // Your code
}
```

**Notes:**

- [Use the function filter](#) on the participant array to find all participant time for given ID
- Sum the durations (using the Duration method) and return it. Return a Duration of 0 seconds if nothing found
- Test your method using the mainLoad.js file

Participant ID	Sport	time
participant1	swim	2m 30s
Participant1	run	1m 15s
Participant2	swim	3m 15s

As example the method shall return **4m 15s** if called with participant ID 'participant1'