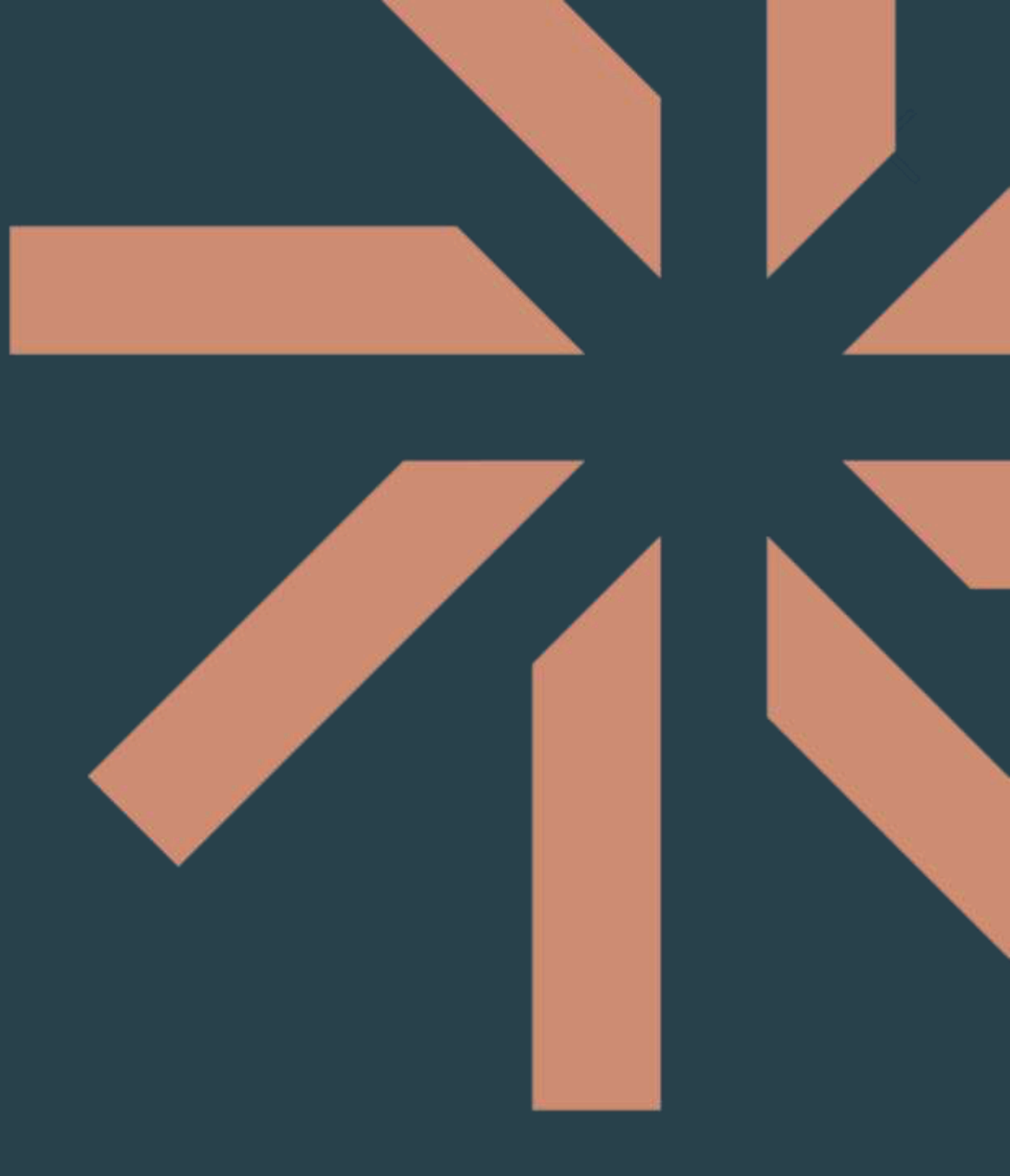
Modules and I/O

Qinshift
Academy

WHAT ARE MODULES?

- A module is a reusable piece of JavaScript code. It can be a single .js file or a directory containing multiple .js files. You can export the content of these files and use them in other files.
- Modules help developers adhere to the DRY (Don’t Repeat Yourself) principle in programming. They also assist in breaking down complex logic into small, simple, and manageable chunks.



Types of Modules

There are three main types of Node.js modules:

1. Built-in modules
2. Local modules
3. Third-party modules

Built-in Modules

Node.js comes with several modules out of the box. These modules are available for use upon installation of Node.js. Some common examples include:

- http
- url
- path
- fs (file system)

Local Modules

When working with Node.js, you create local modules to load and use in your program. To create a module:

1. Write your module code
2. Export the desired functionality (object, array, function, or any data type)
3. Use a specific syntax to export the module

```
function sayHello(userName) {
    console.log(`Hello ${userName}!`)
}

module.exports = sayHello
```

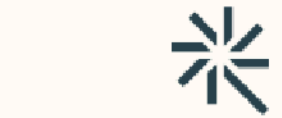
Types of Exports

The export declaration is used to export values from a JavaScript module. There are two types of exports:

1. Named exports
2. Default exports

You can have multiple named exports per module but only one default export.

Named Exports



After the export keyword, you can use let, const, and var declarations, as well as function or class declarations. You can also use the `export { name1, name2 }` syntax to export a list of names declared elsewhere.

```
// export features declared elsewhere
export { myFunction2, myVariable2 };

// export individual features (can export var, let,
// const, function, class)
export let myVariable = Math.sqrt(2);
export function myFunction() {
  // …
}
```

Default Exports

The export default syntax allows any expression. Named exports are useful when you need to export several values. When importing this module, named exports must be referred to by the exact same name (optionally renaming with ‘as’), but the default export can be imported with any name.

```
// export feature declared elsewhere as default
export { myFunction as default };
// This is equivalent to:
export default myFunction;

// export individual features as default
export default function () { /* … */ }
export default class { /* … */ }
```

Loading Local Modules

You can load your local modules and use them in other files using:

1. The require function
2. The import declaration

Named import

Given a value named myExport which has been exported from the module my-module either implicitly as export * from 'another.js') or explicitly using the export statement, this inserts myExport into the current scope.

```
import { myExport } from "/modules/my-module.js";
```

You can import multiple names from the same module.

```
import { foo, bar } from "/modules/my-module.js";
```

Default import

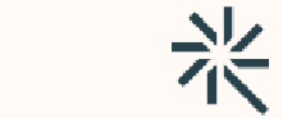
Default exports need to be imported with the corresponding default import syntax. The simplest version directly imports the default.

```
import myDefault from "/modules/my-module.js";
```

It is also possible to specify a default import with namespace imports or named imports. In such cases, the default import will have to be declared first.

```
import myDefault, * as myModule from "/modules/my-module.js";
// myModule.default and myDefault point to the same binding
```

Namespace import



The following code inserts myModule into the current scope, containing all the exports from the module located at /modules/my-module.js.

```
import * as myModule from "/modules/my-module.js";
```

Here, myModule represents a namespace object which contains all exports as properties. For example, if the module imported above includes an export doAllTheAmazingThings(), you would call it like this:

```
myModule.doAllTheAmazingThings();
```

Side-effect import

Import an entire module for side effects only, without importing anything. This runs the module's global code, but doesn't actually import any values.

```
import "/modules/my-module.js";
```

Node.js file paths

- Every file in the system has a path. On Linux and macOS, a path might look like: /users/joe/file.txt while Windows computers are different, and have a structure such as: C:\users\joe\file.tx
- You need to pay attention when using paths in your applications, as this difference must be taken into account.
- You include this module in your files using const path = require('path'); and you can start using its methods.

Working with paths

- You can join two or more parts of a path by using path.join()
- You can get the absolute path calculation of a relative path using path.resolve()

Getting information out of a path

Given a path, you can extract information out of it using those methods:

- **dirname**: get the parent folder of a file
- **basename**: get the filename part
- **extname**: get the file extension

Reading files

The simplest way to read a file in Node.js is to use the fs.readFile() method, passing it the file path, encoding and a callback function that will be called with the file data (and the error)

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

fs.readFile('/Users/joe/test.txt', 'utf8', (err, data) => {
  if (err) {
    console.error(err);
    return;
  }
  console.log(data);
});
```

Alternatively, you can use the synchronous version fs.readFileSync()

```
import fs from 'fs'

fs.readFileSync('Users/joe/test.txt', 'utf8', (err, data) => {
  if (err) {
    console.error(err);
    return;
  }
  console.log(data);
})
```

You can also use the promise-based fsPromises.readFile() method offered by the fs/promises module

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

async function example() {
  try {
    const data = await fs.readFile('/Users/joe/test.txt', { encoding: 'utf8' });
    console.log(data);
  } catch (err) {
    console.log(err);
  }
}
example();
```

Writing to files

The easiest way to write to files in Node.js is to use the fs.writeFile() API. This method will overwrite the file if the file already exists. The fs.writeFile() method is used to asynchronously write the specified data to a file. By default, the file would be replaced if it exists.

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

const content = 'Some content!';

fs.writeFile('/Users/joe/test.txt', content, err => {
  if (err) {
    console.error(err);
  }
  // file written successfully
});
```

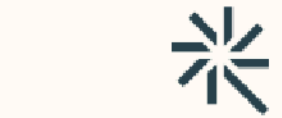
Alternatively, you can use the synchronous version fs.writeFileSync()

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

const content = 'Some content!';

try {
  fs.writeFileSync('/Users/joe/test.txt', content);
  // file written successfully
} catch (err) {
  console.error(err);
}
```

Writing to existing files



The fs.appendFile() method is used to synchronously append the data to the file.

```
// Append data to file
fs.appendFile('input.txt', data, 'utf8',

    // Callback function
    function(err) {
        if (err) throw err;

        //  If no error
        console.log("Data is appended to file successfully.")
});
```

Working with folders

- Use **fs.access()** (and its promise-based **fsPromises.access()** counterpart) to check if the folder exists and Node.js can access it with its permissions.
- Use **fs.mkdir()** or **fs.mkdirSync()** or **fsPromises.mkdir()** to create a new folder.
- Use **fs.readdir()** or **fs.readdirSync()** or **fsPromises.readdir()** to read the contents of a directory.
- Use **fs.rename()** or **fs.renameSync()** or **fsPromises.rename()** to rename folder. The first parameter is the current path, the second the new path.

EXERCISE

- Create a file named note.txt. It can have any content in it.
- Add some more content to the existing files (without replacing the current content).
- Read all the content from the file and log it in the console.

QUESTIONS?

You can find us at:

anetastankovskaane@gmail.com

igorveic7@gmail.com

Qinshift Academy