Modules

JavaScript modules allow you to break up your code into separate files .This makes it easier to maintain a code-base. Modules are imported from external files with the `import` statement.

Why is a Module System Required?

Early development of JavaScript encountered several difficulties that made an organized approach to coding necessary

1. **Global Scope Issues:** When all variables and functions were first defined in the global scope, managing big codebases became challenging due to conflicts. When multiple scripts attempted to use the same variable names, this frequently led to issues.
2. **Maintainability:** Without a modular architecture, code maintenance became more and more challenging as applications expanded in size and complexity. Developers can divide larger, more complex applications into smaller, more manageable chunks with a module system.
3. **Collaboration:** A module system fosters a collaborative environment by enabling several developers to work on separate modules separately and without hindrance.
4. **Performance Optimization:** By loading only the necessary modules when needed, asynchronous loading (ES Modules), which is supported by modern module systems, can enhance application performance.
5. **Standardization:** JavaScript programs can now be structured consistently across various contexts thanks to the introduction of standardized module systems like ES Modules.

Common Js

The purpose of CommonJS was to create a JavaScript module system, specifically for Node.js server-side development. Since its introduction in 2009, it has gained widespread use, particularly within the Node.js ecosystem.

Key Features

- **Synchronous Loading:** CommonJS modules are loaded synchronously, which means that before executing code, it waits for the module to load. Applications may exhibit blocking behavior as a result of this.
- **Export Mechanism:** Modules can easily share functions, objects, or variables across other files by exporting values using module.exports or exports.
- **Runtime Environment:** CommonJS is mainly used for backend applications and is fully supported in all Node.js versions.

ECMAScript Modules (ESM)

In 2015, ESM a standardized module system for JavaScript was released alongside ECMAScript (https://www.guvi.in/blog/features-of-ecmascript/) 6 (ES6). It seeks to offer a more contemporary method of code modularization that functions flawlessly in both server-side and client-side contexts.

Key Features

- **Asynchronous Loading:** Modules can be loaded asynchronously with ESM, which improves performance by letting other processes go on while modules are loaded.
- **Static Imports/Exports:** ESM makes use of the import and export syntax to allow for the parse-time static analysis of dependencies. This makes it easier to perform optimizations like tree shaking, which removes unnecessary code during bundling.
- **Native Browser Support:** ESM is the recommended option for front-end development since it is natively supported by most current web browsers.

Comparison Table

Features	Loading	Export syntax
CommonJS	Synchronous	module.exports
ECMAScript Modules	Asynchronous	export / export default
Import syntax	require(‘module_name’)	Import {…} from ‘module_name’
Browser support	Not natively supported	Natively supported
Dependency Management	Dynamic loading with require	Static analysis at parse time
Default Behavior	Caches modules upon the first load	Does not cache by default
Use Cases	Server-side apps	Frontend applications

CJS

There are multiple ways to import and export in common js modules :

```
// 1. Exporting a Single Value (Module.exports)
module.exports = { sayHello: () => console.log('Hello'), value: 42 };

// 2. Adding Properties to Exports Individually
module.exports.property1 = 'Value 1';
module.exports.method1 = () => {};

// 3. Exports Alias (Direct Assignment)
module.exports.name = 'John'; exports.greet = () => 'Hi there';

// 4. Exporting with Destructured Alias
const originalFunction = () => 'Original';
module.exports = { renamedFunction: originalFunction };

// 5. Multiple Exports with Aliases
module.exports = { originalName: 'John', aliasedName: 'John', // Same value, different name 'hyphenated-name': 'Special Export' // Ke
```

```
// 1. Basic Import (Entire Module)
const utils = require('./utils');

// 2. Destructuring Import (Specific Properties)
const { specificFunction, value } = require('./utils');

// 3. Importing with Alias
const { originalName: renamedImport, method: aliasedMethod } = require('./module');

// 4. Partial Module Import
const { subModule: { nestedProperty } } = require('./complex-module');
```

ESM

There are multiple ways to import and export in ecmascript modules :

Export

Modules with **functions** or **variables** can be stored in any external file.

There are two types of exports: **Named Exports** and **Default Exports**.

Named Exports

Let us create a file named `person.js`, and fill it with the things we want to export.

You can create named exports two ways. In-line individually, or all at once at the bottom.

In-line individually:

`person.js`

```
export const name = "Jesse";
export const age = 40;
```

All at once at the bottom:

`person.js`

```
const name = "Jesse";
const age = 40;
export {name, age};
// export { myFunction as function1, myVariable as variable }; // we can also give alias names here
```

Default Exports

Let us create another file, named `message.js`, and use it for demonstrating default export.

You can only have one default export in a file.

`message.js`

```
const message = () => {
const name = "Jesse";
const age = 40;
return name + ' is ' + age + 'years old.';
};

export default message;
// export { message as default }; Same as above
```

`other.js`

```
import otherName from './message.js' // we can have any name for default export
```

Re-exporting / Aggregating / Barrel Files

A module can also "relay" values exported from other modules without the hassle of writing two separate import/export statements. This is often useful when creating a single module concentrating various exports from various modules (usually called a "barrel module").

```
export { default as function1, function2 } from "bar.js";

// same as

import { default as function1, function2 } from "bar.js";
export { function1, function2 };
```

We can combine multiple files so that when importing we don't have import every files one by one

```
export { TabList } from './tab-list'
export { TabPanel } from './tab-panel'
```

Cheat Sheet For Export

```
// Exporting declarations
export let name1, name2/*, … */; // also var
export const name1 = 1, name2 = 2/*, … */; // also var, let
export function functionName() { /* … */ }
export class ClassName { /* … */ }
export function* generatorFunctionName() { /* … */ }
export const { name1, name2: bar } = o;
export const [ name1, name2 ] = array;

// Export list
export { name1, /* …, */ nameN };
export { variable1 as name1, variable2 as name2, /* …, */ nameN };
export { variable1 as "string name" };
export { name1 as default /*, … */ };

// Default exports
export default expression;
export default function functionName() { /* … */ }
export default class ClassName { /* … */ }
export default function* generatorFunctionName() { /* … */ }
export default function () { /* … */ }
export default class { /* … */ }
export default function* () { /* … */ }

// Aggregating modules
export * from "module-name";
export * as name1 from "module-name";
export { name1, /* …, */ nameN } from "module-name";
export { import1 as name1, import2 as name2, /* …, */ nameN } from "module-name";
export { default, /* …, */ } from "module-name";
export { default as name1 } from "module-name";
```

Cheat Sheet For Import

```
import defaultExport from "module-name";
import * as name from "module-name";
import { export1 } from "module-name";
import { export1 as alias1 } from "module-name";
import { default as alias } from "module-name";
import { export1, export2 } from "module-name";
import { export1, export2 as alias2, /* … */ } from "module-name";
import { "string name" as alias } from "module-name";
import defaultExport, { export1, /* … */ } from "module-name";
import defaultExport, * as name from "module-name";
import "module-name";
```