

ES6 Modules

- ES6 modules, introduced in ECMAScript 2015, are a way to organize JavaScript code by splitting it into separate files, known as modules.
- Each module encapsulates functionality, promoting modular programming where individual parts can be reused, maintained, and tested independently.
- ES6 modules offer a powerful way to structure applications, isolating code in separate files with imports and exports, making development and debugging easier, and enhancing code reusability and modularity.
- An ES6 module is a JavaScript file that executes strictly in strict mode. This means that any variables or functions declared in the module won't automatically be added to the global scope.
- Modules make code maintenance, debugging, and reuse easier. Each module represents a piece of code that executes once it's loaded.
- Modules only work via HTTP, not with local files. If we try to open a webpage locally, using the "file://" protocol, import/export directives won't function. To test them, use a local web server such as static-server, or enable the "Live Server" feature in your editor, such as the VS Code Live Server extension.
- The ES6 module transpiler tool takes ES6 modules and converts them into compatible ES5 code, either in the AMD (Asynchronous Module Definition, a specification for the JavaScript programming language) format or in the CommonJS style.
- Modules can load each other and use special directives, export and import, to share functionality, allowing one module to call functions from another.

Benefits of ES6 Modules

- **Code Reusability:** Functions, classes, and variables defined in one module can be used in others without duplication.
- **Better Organization:** Allows code to be split into logical pieces, making applications easier to navigate and understand.
- **Encapsulation:** Only explicitly exported variables and functions are accessible outside a module, hiding internal details.
- **Global Namespace Isolation:** Each module has its own scope, avoiding pollution of the global namespace.

Exporting in ES6

- Exporting allows you to define which parts of a module can be accessed from other modules.
- By marking functions, variables, classes, or objects with export, you signal that they can be imported elsewhere.
- This modularity is crucial in building applications that are organized, maintainable, and reusable.

There are two main types of exports in ES6:

- **Named Export:** Used when you want to export multiple items from a module.
- **Default Export:** Used when a module only needs to export one main item.

Named Export

- Named exports allow you to export multiple values from a module.
- With named exports, each export has a unique name, allowing the module to export several constants, functions, or classes.
- Named exports can be done inline, or you can list all exports at the end of a file.
- When importing named exports, you must use the exact name defined in the module.

Inline Named Export: You can export variables, functions, or classes directly as you define them.

Example

```
export const add = (a, b) => a + b;  
export const subtract = (a, b) => a - b;  
export function multiply(a, b) { return a * b; }
```

In the above example, add, subtract, and multiply are named exports. Each can be imported individually.

Exporting at the End of File: Another way to do named exports is by defining functions and variables first, and then exporting them at the end of the file. This approach is useful if you want to organize all exports in one place, especially when you have a long module file.

Example

```
const add = (a, b) => a + b;  
const subtract = (a, b) => a - b;  
const multiply = (a, b) => a * b;  
export { add, subtract, multiply };
```

Default Export

- A default export is useful when a module primarily exports one main value, function, or class.
- A module can only have one default export, which makes it easier to identify its primary functionality.
- You use export default before the item you want to export as the default.
- When importing a default export, you don't need curly braces {}, and you can give it any name.

Example: Default Export of a Function

```
export default function message() {  
  return "Hello, World!";  
}
```

In this example, message is the default export, and only one default export is allowed per module.

Example: Default Export of a Class

```
export default class User {  
  constructor(name) {  
    this.name = name;  
  }  
  
  sayHello() { // instance method of class 'User'*  
    return `Hello, ${this.name}`;  
  }  
}
```

Note: Named Exports are best when you have multiple things to export, while Default Exports are suitable for a single, primary export from a module. Both approaches can be combined in a single file for flexible module design, making ES6 modules highly versatile in modern JavaScript applications.

Importing in ES6

- The import statement in ES6 allows one module to access and use code from another module.
- It works together with export to enable code sharing and modularization.
- The import statement must always be at the top level of a file, meaning it can't be conditionally imported or placed inside a function.

To use the exported functionalities, import is used in the following ways:

- **Named Import:** Access specific exported items by name, wrapped in curly braces {}.

```
import { add, subtract } from './file.js';  
console.log(add(2, 3));
```

- **Default Import:** Import a default export without curly braces.

```
import multiply from './file.js';  
console.log(multiply(4, 5));
```

- **Renaming Imports and Exports:** If you want to avoid name conflicts or add clarity, you can rename exports.

```
// exportt.js
```

```
export { add as addition, subtract as subtraction };
```

```
// importt.js
```

```
import { addition, subtraction } from './exportt.js';
```

- **Dynamic Imports:** With import(), you can load modules asynchronously, which is useful for lazy loading.

```
import('./file.js').then(calculate => {  
    console.log(calculate.add(4, 2));  
});
```

Es6 Modules Examples

1) Named Exports & Imports

- Named exports are distinguished with their names.
- The class, variable, or any function which is exported by using the named export can only be imported by using the same name.
- Multiple variables, functions & classes can be imported and exported by using the named export.

Import a single export from a module / Import a single binding

Example - For Variables

index.html

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-
scale=1.0">
  <title>Document</title>
</head>
<body>
  <script src="exportt.js" type="module"> </script>
  <script src="importt.js" type="module"> </script>
</body>
</html>
```

exportt.js

```
let a = 2;
let b = 4;
let c = a + b;
export { c }
```

importt.js

```
import { c } from "./exportt.js";
console.log(c);
```

Example - For Functions

index.html

same as previous

exportt.js

```
function calc(n) {  
  return n * 2  
}  
export { calc };
```

importt.js

```
import { calc } from "./exportt.js";  
console.log(calc(2))
```

Example - For Classes

index.js

same as previous

exportt.js

```
class Person {  
  message() {  
    return "Hello User this is ES6 Classes";  
  }  
}  
export { Person }
```

importt.js

```
import { Person } from "./exportt.js";  
const info = new Person();  
console.log(info.message())
```

Import multiple exports from a module / Import multiple bindings

Example -- For Variables

index.html

same as previous

exportt.js

```
let name = "Learn2Earn Labs Training Institute";  
let location = "Sikandra, Agra";  
export { name, location };
```

importt.js

```
import { name, location } from "./exportt.js";  
console.log(name);  
console.log(location);
```

Example -- For Functions

index.html

same as previous

exportt.js

```
let fname = "Vandana";  
let lname = "Singh";  
function prntName(fname, lname) {  
  console.log(`Hello student, your name is ${fname} ${lname}`)  
}  
  
export { fname, lname, prntName };
```

importt.js

```
import { fname, lname, prntName } from "./exportt.js";  
console.log(fname);  
console.log(lname);  
prntName("Shubham", "Agarwal");
```

Example -- For Classes

index.js

same as previous

exportt.js

```
let message = "Hello I'm Variable";  
function display() {  
    console.log("Hello I'm Function")  
}  
class Person {  
    displayMsg() {  
        console.log("Hello I'm Person Class")  
    }  
}  
export { message, display, Person };
```

importt.js

```
import { message, display, Person } from "./exportt.js";  
console.log(message);  
display();  
let info = new Person();  
info.displayMsg();
```


2) Default Exports & Imports

- Unlike the named export, we cannot simultaneously make more than one export statement in default export.
- The Default exports can be done without curly braces.
- A default export can be imported with any name.

Default Export without changing their name

Example -- For Variables

index.js

same as previous

exportt.js

```
let one = "Hello";  
let two = "Students";  
let result = `${one} ${two}`;  
export default result;
```

importt.js

```
import result from "./exportt.js";  
console.log(result)
```

Example -- For Functions

index.js

same as previous

exportt.js

```
function display() {  
  return "Hello Default Export for function";  
}  
export default display;
```

importt.js

```
import display from "./exportt.js";  
console.log(display())
```

Example -- For Classes

index.js

same as previous

exportt.js

```
class calc {  
  add(a, b) {  
    return a + b;  
  }  
  sub(a, b) {  
    return a - b;  
  }  
  div(a, b) {  
    return a / b;  
  }  
  multi(a, b) {  
    return a * b;  
  }  
}  
export default calc;
```

importt.js

```
import calc from "./exportt.js";  
let addition = new calc();  
console.log(addition.add(4, 5));  
let subtraction = new calc();  
console.log(subtraction.sub(9, 5));  
let multiplication = new calc();  
console.log(multiplication.multi(4, 5));  
let division = new calc();  
console.log(division.div(20, 5));
```

Default Export by changing their name

Example -- For Variables

index.js

same as previous

exportt.js

```
let first = "Hello";  
let second = "students,";  
let third = "How are you?";  
let result = first + second + third;  
export default result;
```

importt.js

```
import res from "./exportt.js";  
console.log(res);
```

Example -- For Functions

index.js

same as previous

exportt.js

```
function display() {  
  return "I am displaying a message";  
}  
export default msg;
```

importt.js

```
import display from "./exportt.js";  
console.log(display());
```

Example -- For Classes

index.js

same as previous

exportt.js

```
class calc {  
  add(a, b) {  
    return a + b;  
  }  
  sub(a, b) {  
    return a - b;  
  }  
  div(a, b) {  
    return a / b;  
  }  
  multi(a, b) {  
    return a * b;  
  }  
}  
export default calc;
```

importt.js

```
import calculator from "./exportt.js";  
let addition = new calculator();  
console.log(addition.add(4, 6));  
let subtraction = new calculator();  
console.log(subtraction.sub(9, 5));  
let multiplication = new calculator();  
console.log(multiplication.multi(4, 5));  
let division = new calculator();  
console.log(division.div(20, 5));
```

3) Wildcard Imports / Namespace Imports / Import an entire module as an object

If a module has numerous named exports, you can import them all at once using `*` with an alias. This approach is useful for grouping all exports under a single namespace.

Example -- For Variables

`index.js`

same as previous

`exportt.js`

```
let one = "Variable One",
    two = "Variable two",
    three = "Variable three",
    four = "Variable four",
    five = "Variable five",
    six = "Variable six";

// Here all variable are declared using 'let'
export { one, two, three, four, five, six };
```

`importt.js`

```
import * as variable from "./exportt.js";
console.log(typeof variable);
console.log(variable.one);
console.log(variable.two);
console.log(variable.three);
console.log(variable.four);
console.log(variable.five);
console.log(variable.six);
```

Example -- For Functions

index.js

same as previous

exportt.js

```
function add(a, b) {  
    return a + b;  
}  
function multi(a, b) {  
    return a * b;  
}  
function sub(a, b) {  
    return a - b;  
}  
function div(a, b) {  
    return a / b;  
}  
export { add, mul, sub, div };
```

importt.js

```
import * as calc from "./exportt.js";  
console.log(calc.add(2, 3))  
console.log(calc.mul(3, 3))  
console.log(calc.sub(9, 3))  
console.log(calc.div(9, 3))
```

Example -- For Classes

index.js

Same as previous

exportt.js

```
class add {  
    addNum(a, b) {  
        return a + b;  
    }  
}  
class sub {  
    subNum(a, b) {  
        return a - b;  
    }  
}  
class mul {  
    multiNum(a, b) {  
        return a * b;  
    }  
}  
class div {  
    divNum(a, b) {  
        return a / b;  
    }  
}  
export { add, sub, mul, div };
```

importt.js

```
import * as calc from "./exportt.js";  
let addition = new calc.add();  
console.log(addition.addNum(5, 3))  
let subtraction = new calc.sub();  
console.log(subtraction.subNum(9, 3))  
let multiplication = new calc.mul();  
console.log(multiplication.mulNum(5, 3))  
let division = new calc.div();  
console.log(division.divNum(25, 5))
```

4) Import without any bindings

We can directly use the module variable, functions & classes without importing them as bindings. But make sure we cannot access them in other modules by their names.

Example

index.html

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-
scale=1.0">
  <title>Document</title>
</head>
<body>
  <script type="module" src="exportt.js"> </script>
  <script type="module" src="module1.js"> </script>
  <script type="module" src="module2.js"> </script>
</body>
</html>
```

module1.js

```
import "./exportt.js";
```

module2.js

```
import "./exportt.js";
```

exportt.js

```
function display() {
  return "Hello Students";
}
console.log(display());
```


5) Importing both default and non-default bindings

In order to import both default and non-default bindings, so in this case :-

- The default binding must come first.
- The non-default binding must be surrounded by curly braces.

Example

index.html

```
<script src="exportt.js" type="module"> </script>
<script src="importt.js" type="module"> </script>
```

exportt.js

```
let name = "Tushar Gupta";
function add(a, b) {
  return a + b;
}
class Person {
  msg() {
    return "Good Luck to all the students";
  }
}
export { name, add };
export default Person;
```

importt.js

```
import Person, { name, add } from "./exportt.js";
console.log(name);
console.log(add(5, 5));
let info = new Person();
console.log(info.msg());
```

6) Importing both default and namespace bindings

In order to import both default and namespace bindings, so in this case :-

- a. The default binding must come first.
- b. The namespace binding comes after default binding.

Example

index.html

Same as previous

exportt.js

Same as previous

importt.js

```
import Person, * as content from "./exportt.js";  
console.log(content.name);  
console.log(content.add(15, 15));  
let info = new Person();  
console.log(info.msg());
```

7) Direct Imports

Example

index.html

```
<script type="module">
  import {name} from "./exportt.js";
  console.log(name)
</script>
```

exportt.js

```
let name = "Neha Rathore";
export { name };
```

7) Imports for promises

We have special import statements for promises.

Example

```
let promise = import("module-name");
```

Renaming / Aliasing Module

- JavaScript allows us to create aliases for variables, functions, or classes when we export and import.
- In order to renaming variables, functions or classes we use 'as' keyword.

a) Renaming Named Exports

Example

index.html

```
<script src="exportt.js" type="module"> </script>
<script src="importt.js" type="module"> </script>
```

exportt.js

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

importt.js

```
import { sum } from "./exportt.js"
console.log(sum(4, 4));
```

Example

index.html

same a previous

exportt.js

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

importt.js

```
import { add as sum } from "./exportt.js"
console.log(sum(5, 4));
```

Example

index.html

same as previous

exportt.js

```
let name = "Vandana Singh";
function add(a, b) {
    return a + b;
}
class Person {
    msg() {
        return "Hello dear students";
    }
}
export { name, add, Person };
```

importt.js

```
import { name, add as sum, Person as classPerson } from "./exportt.js"
console.log(name);
console.log(sum(10, 6));
let info = new classPerson();
console.log(info.msg())
```

b) Renaming Default Exports

Example

index.html

Same as previous

exportt.js

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

importt.js

```
import { default as sum } from "./exportt.js"  
console.log(sum(5, 5))
```



3) Renaming Namespace Exports

Example

index.html

Same as previous

exportt.js

```
let name = "Tushar Gupta";  
function add(a, b) {  
    return a + b;  
}  
class Person {  
    msg() {  
        return "Hello Students";  
    }  
}  
export { name, add, Person };
```

importt.js

```
import * as content from "./exportt.js";  
console.log(content.name);  
console.log(content.add(5, 10));  
let info = new content.Person();  
console.log(info.msg());
```

Re-exporting a binding / aggregate exports

- It's possible to export bindings that you have imported.
- This is called re-exporting.

Example

index.html

```
<script src="exporttt.js" type="module"> </script>
<script src="module1.js" type="module"> </script>
<script src="module2.js" type="module"> </script>
```

exporttt.js

```
export function display() {
  console.log("Welcome to Learn2Earn Labs Training Institute");
}
```

module1.js

```
export { display } from "./exporttt.js";
```

module2.js

```
import { display } from "./module1.js";
display();
```


Difference Between Modules & Regular Scripts

a) Modules always in "use strict" mode by default

Assigning to an undeclared variable will give an error.

Example

```
<script type="module">
    msg = "Hello Students";
    console.log(msg)
</script>
```

b) Each module have its own scope

- Each module has its own top-level scope. In other words, top-level variables and functions from a module are not seen in other scripts.
- In order to access variables, functions & classes from one module to another then we use 'import' & 'export' keywords in order to use them. We cannot use variables, functions & classes directly from one module to another like in normal scripts.
- The top-level module code is mostly used for initialization, creation of internal data structures, and if we want something to be reusable export it.

Example

index.html

Same as previous

module1.js

```
const abc = "Hello Variable";
```

module2.js

```
console.log(abc); // Error
```

c) Every module is executed first time

If the same module is imported into multiple other modules, its code is executed only the first time, then exports are given to all importers.

index.html

```
<script type="module" src="module1.js"> </script>
<script type="module" src="module2.js"> </script>
```

exportt.js

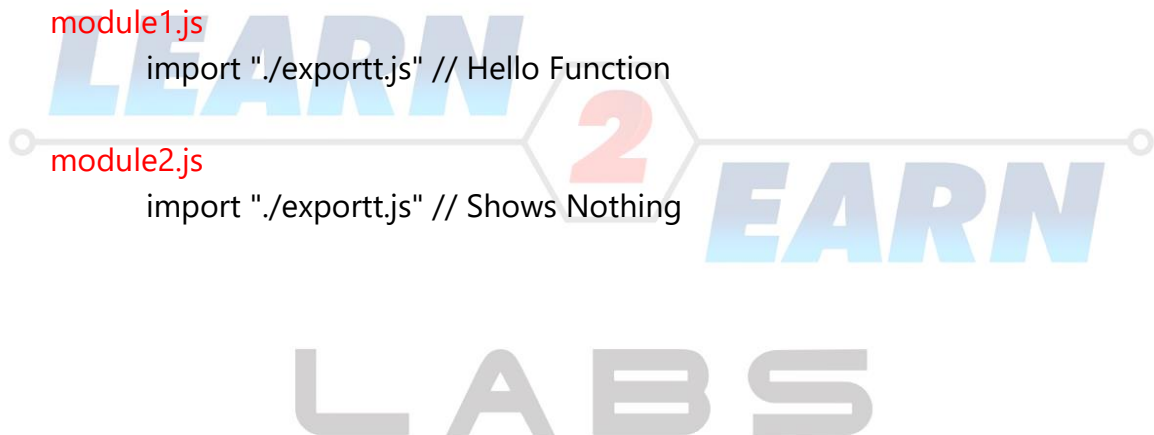
```
function hello() {
  return "Hello Funtion";
}
console.log(hello());
```

module1.js

```
import "./exportt.js" // Hello Function
```

module2.js

```
import "./exportt.js" // Shows Nothing
```



d) Exporting an Object

If this module is imported from multiple files, the module is only evaluated the first time, admin object is created, and then passed to all further importers.

index.html

```
<script type="module" src="module1.js"> </script>
<script type="module" src="module2.js"> </script>
```

exportt.js

```
let info = {
  fname: "Vandana",
  lname: "Singh"
}
export { info }
```

module1.js

```
import { info } from "./exportt.js"
info.fname = "Tushar";
info.lname = "Gupta"
console.log(`Hello ${info.fname} ${info.lname} -- Module 1`)
```

module2.js

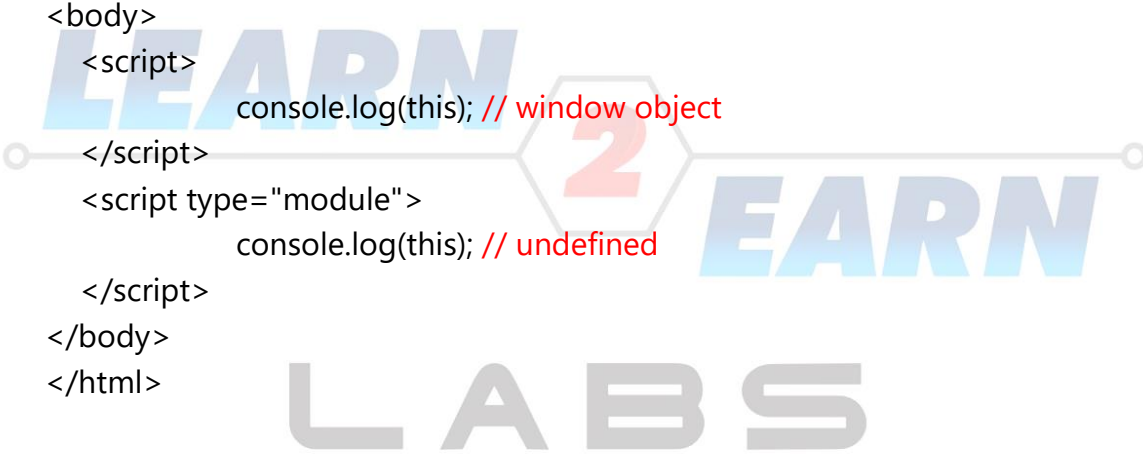
```
import { info } from "./exportt.js";
console.log(`Hello ${info.fname} ${info.lname} -- Module 2`)
```

e) In module 'this' is undefined

In a module, top-level this is undefined. But when we see it in a "non-module" script it is undefined.

Example

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Document</title>
</head>
<body>
  <script>
    console.log(this); // window object
  </script>
  <script type="module">
    console.log(this); // undefined
  </script>
</body>
</html>
```

A large, semi-transparent watermark is centered over the code. It features the words 'LEARN' and 'EARN' in blue, a large red '2' inside a hexagon, and the word 'LABS' in grey below them.

f) Module scripts are deferred

Module scripts are always deferred, same effect as defer attribute for both external and inline scripts.

index.html

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Document</title>
</head>
<body>
  <script type="module">
    console.log(typeof button); // object: the script can 'see' the button below
    // as modules are deferred, the script runs after the whole page is loaded
  </script>
```

Compare to regular script below:

```
<script>
  console.log(typeof button);
  // Error: button is undefined, the script can't see elements below
  // regular scripts run immediately, before the rest of the page is processed
</script>

<button id="button">Button</button>
</body>
</html>
```

g) In modules 'async' works on inline scripts

- For non-module scripts, the async attribute only works on external scripts.
- Async scripts run immediately when ready, independently of other scripts or the HTML document.
- For module scripts, it works on inline scripts as well.

Example

```
//index.html
<script async type="module">
  importing {name} from "./exportt.js";
  console.log(name);
</script>
```

```
// exportt.js
export const name = "Neha Rathore";
```

Using `<script async type="module">` is a modern approach to loading JavaScript in web applications. It enhances performance and maintainability, allowing developers to take advantage of the modular features of ES6 while ensuring non-blocking behavior for improved user experience.

h) Modules provide CORS Security

If a module script is fetched from another origin, the remote server must supply a header `Access-Control-Allow-Origin` allowing the fetch.

Example

```
<!-- another-site.com must supply Access-Control-Allow-Origin -->
<!-- otherwise, the script won't execute -->
<script type="module" src="http://another-site.com/their.js"> </script>
```

i) Compatibility

Old browsers do not understand type="module". Scripts of an unknown type are just ignored. For them, it's possible to provide a fallback using the nomodule attribute.

Example

```
<script type="module">  
  alert("Runs in modern browsers");  
</script>  
<script nomodule>  
  alert("Modern browsers know both type=module and nomodule, so skip this")  
  alert("Old browsers ignore script with unknown type=module, but execute  
this.");  
</script>
```

