Modules in TypeScript

Before ES6, the files created in TypeScript had global access. This means, variables, functions and other entities declared in one file are easily accessed in another file. This global nature caused code conflicts and issues with execution at run-time.

But with ES6 modules, you have export and import module functionality which can be used to avoid global variable, function conflicts.

Consider the following example before ES6 **(without modules):**

Example test1.ts

```
let age : number = 25;
```

You have defined a variable age of type number in test1.ts.

Example test2.ts

In test2.ts file you are easily able to access the variable **age** defined in test1.ts and also modify it as shown below:

```
age = 30; // changed from 25 to 30.

let _new_age = age;
```

So, the above case can create a lot of problems as the variables are globally available and can be modified.

Changed situation with ES6:

1. With **Modules**, the code written remains locale to the file and cannot be accessed outside it.
2. To access anything from the file, it has to be exported using the export keyword.
3. **Import** is used when you want to access the exported variable, class, or interface or function. Doing this, code written remains intact within the file, and even if you define same variable names, they are not mixed up and behave local to the file where they are declared.

Using Export and Import

There are many ways to export and import. Let's discuss the syntax mostly used:

The syntax for import and export 1:

```
export  nameofthevariable or class name or interface name etc

//To import above variable or class name or interface you have to use import as shown below:

Import {nameof thevariable or class name or interfacename} from "file path here without.ts"
```

Here is a working example using export and import.

Example:

test1.ts

```
export let age: number = 25;
```

Export keyword is used to share age variable in another file.

test2.ts

```
import { age } from "./test1"

let new_age :number = age;
```

Import keyword is used to access the **age** variable, and you need to specify the file location as shown above.

Syntax for import and export 2:

There is another way to export, and import and the syntax for the same is as shown below:

```
export = classname;

import classname = require("file path of modulename")
```

When you are using **export =** to export your module, the import has to use require("file path of modulename") to import it.

Here is a working example showing the above case:

Customer.ts

```
class Customer {
    name: string;
    age: number;
    constructor(name: string, age: number) {
        this.name = name;
        this.age = age;
    }
    getName(): string {
        return this.name;
    }
}
export = Customer;
```

testCustomer.ts

```
import Customer = require("./Customer");

let a = new Customer("Harry", 30);
```

```
alert(a.getName());
```

Example 2:

File: mathUtils.ts

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

export function subtract(a: number, b: number): number {
  return a - b;
}
```

File: app.ts

```
import { add, subtract } from './mathUtils';

const sum = add(5, 3);
const difference = subtract(5, 3);

console.log(`Sum: ${sum}`);
console.log(`Difference: ${difference}`);
```

- In mathUtils.ts, two functions, add and subtract, are defined and exported using the export keyword.
- In app.ts, these functions are imported using the import statement, allowing their usage within the file.
- This modular approach promotes code reusability and clarity by separating functionalities into distinct files.

Output:

```
Sum: 8
Difference: 2
```

Re-telling the concept in a story-format:

The Old World: Global Scope

In the early days of JavaScript, everything lived in the global scope. You'd include multiple <script> tags in your HTML, and hope nothing clashed. If two files defined a User class, boom—conflict. If one overwrote window.config, chaos.

So yes, you could technically access anything from anywhere, but it was fragile, error-prone, and hard to scale.

The New World: Modules

Enter import and export. These keywords let you:

- **Encapsulate logic**: Each file becomes its own sandbox.
- **Avoid naming collisions**: You can have multiple User classes in different modules.
- **Control visibility**: Only exported entities are accessible elsewhere.

- **Enable tree-shaking**: Unused code can be stripped out during bundling.
- **Improve tooling**: TypeScript and modern IDEs thrive on module boundaries.

So when you say *“why enforce globals if we have import/export?”* — the answer is: **we don’t want to enforce globals anymore.** We only use them in rare cases (e.g., polyfills, extending window, or exposing APIs to external scripts).

Think of It Like This:

- **Global declarations** are like shouting across a crowded room.
- **Modules** are like passing notes in class—quiet, intentional, and scoped.

When Globals Still Matter

There are a few legit use cases for globals:

- Adding properties to window for browser plugins or analytics.
- Declaring ambient types for third-party libraries.
- Sharing config across multiple scripts in legacy systems.

But in modern TypeScript projects, **modules are the default**—and globals are the exception.