Working with Modules and Enums in TypeScript

Modules in TypeScript

Modules are a key concept in JavaScript and TypeScript for structuring and organizing code. A module is essentially a file containing related pieces of code (e.g., functions, variables, or classes) that can be exported and imported into other modules or files.

In TypeScript, this concept works similarly to how it does in JavaScript. TypeScript files get compiled into multiple JavaScript files that can be linked together. This modular approach improves maintainability and clarity in larger applications.

Setting Up Your `tsconfig.json`

Before you begin working with modules in TypeScript, you need to configure your `tsconfig.json` file to enable modern import/export functionality. Here's how you can do that:

```
{
  "compilerOptions": {
    "target": "es2016",
    "module": "es2015"
  }
}
```

Alternatively, for Node.js projects, it's common to set `"module": "CommonJS"`, since Node.js does not fully support ES modules yet.

Using Modules in HTML

When using modules in the browser, you'll need to adjust the `<script>` tag in your HTML file. Instead of the typical script tag, you'll add a `type="module"` attribute to indicate that the script uses ES module syntax.

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

Importing and Exporting Modules

With this configuration in place, you can start using `import` and `export` syntax to share code between different TypeScript files. For example, if you have a function in one file and want to use it in another, you can export it like this:

```
// src/hello.ts
export function sayHello() {
  console.log('Hello, world!');
}
```

Now, in another file, you can import and use the `sayHello` function:

```
// src/main.ts
import { sayHello } from './hello.js';

sayHello(); // Output: Hello, world!
```

Note: Even when working with TypeScript files, when importing, always use the `.js` extension.

Named Exports vs Default Exports

There are two main types of exports in TypeScript: named exports and default exports. In the previous example, we used a named export. Here's a quick overview of both:

Named Exports: You can export multiple variables, functions, or classes from a file and import them using curly braces.

```
// utils.ts
export const PI = 3.14159;
export function calculateArea(radius: number) {
  return PI * radius * radius;
}

// main.ts
import { PI, calculateArea } from './utils.js';

console.log(PI); // 3.14159
console.log(calculateArea(5)); // 78.53975
```

Default Exports: You can export a single value as the default from a module, which can be imported without curly braces.

```
// math.ts
export default function add(a: number, b: number): number {
  return a + b;
}

// main.ts
import add from './math.js';

console.log(add(2, 3)); // 5
```

Organizing Code with Modules

Modules help organize your application code into different concerns. For example, in a large project, you could split code into modules like `user.ts`, `product.ts`, and `order.ts`. Each module can handle specific parts of the functionality.

Creating a Utility Module

Imagine you have a set of utility functions that can be used across your project. You can organize them into a `utils.ts` file.

```
// utils.ts
export function formatDate(date: Date): string {
  return date.toISOString().split('T')[0];
}

export function capitalize(str: string): string {
  return str.charAt(0).toUpperCase() + str.slice(1);
}
```

You can then import these functions wherever needed:

```
// main.ts
import { formatDate, capitalize } from './utils.js';

console.log(formatDate(new Date())); // 2024-09-23
console.log(capitalize('hello')); // Hello
```

Re-Exporting

TypeScript also allows re-exporting, which means you can import something and immediately export it from a module. This can be useful when you want to group multiple exports into a single module:

```
// moduleA.ts
export function foo() {
  return 'foo';
}

// moduleB.ts
export function bar() {
  return 'bar';
}

// index.ts (Re-export both)
export { foo } from './moduleA.js';
export { bar } from './moduleB.js';
```

Now, you can import `foo` and `bar` from `index.ts`:

```
import { foo, bar } from './index.js';

console.log(foo()); // foo
console.log(bar()); // bar
```

Module Resolution

TypeScript uses several strategies to resolve modules. The most common ones are:

1. **Relative Imports**: If the file is in the same directory or a nested one, you use `./` or `../` in the import path.
2. **Non-Relative Imports**: Used for modules installed via npm (e.g., `import * as express from 'express'`).
3. **Path Mapping**: You can define custom module paths in your `tsconfig.json` using `paths` and `baseUrl`.

Example of path mapping in `tsconfig.json`:

```
{
  "compilerOptions": {
    "baseUrl": "./",
    "paths": {
      "@utils/*": ["src/utils/*"]
    }
  }
}
```

Now you can import from `src/utils/` like this:

```
import { formatDate } from '@utils/date.js';
```

Dynamic Imports

Another cool feature TypeScript inherits from JavaScript is dynamic imports. Sometimes you may want to load a module only when it's needed, rather than at the start of your script. This can be achieved using the `import()` function:

```
// main.ts
async function loadModule() {
  const { sayHello } = await import('./hello.js');
  sayHello();
}

loadModule();
```

This approach can be useful for optimizing large applications by lazy loading code when it's required.

Summary

Modules are a powerful way to organize your code and make it reusable. By using TypeScript's module system, you ensure that your code remains clean, maintainable, and efficient.

- Always configure `tsconfig.json` for proper module resolution.
- Use `export` and `import` syntax to share functionality across files.

- Named and default exports help structure the way you expose functions or variables.
- Dynamic imports provide flexibility in loading code when needed.

With these tools, you can manage large codebases and ensure that every piece of your application works seamlessly together.

Enums in TypeScript

Enums are a special feature of TypeScript that allow developers to define a set of named constants, which can be either numeric or string values. Enums provide a way to organize related values under meaningful names, making the code more readable and reducing potential bugs. Let's dive into how they work and how they can enhance your TypeScript projects.

Basic Numeric Enums

The simplest form of an enum is a numeric enum, where the values are automatically assigned incremental numbers, starting from `0` unless otherwise specified. Here's an example:

```
enum Status {
  PENDING,
  IN_PROGRESS,
  COMPLETED,
  CANCELLED
}

console.log(Status.PENDING);      // 0
console.log(Status.COMPLETED);    // 2
console.log(Status.CANCELLED);    // 3
```

In this example, TypeScript automatically assigns the value `0` to `PENDING`, `1` to `IN_PROGRESS`, and so on. This automatic incrementing behavior can simplify code where sequential numeric values are meaningful.

Customizing Numeric Enums

You can customize the starting value of an enum by explicitly setting the value of the first element. The following elements will continue incrementing from the initial value:

```
enum OrderStatus {
  NEW = 1,
  PROCESSING,
  SHIPPED,
  DELIVERED
}

console.log(OrderStatus.NEW);       // 1
console.log(OrderStatus.PROCESSING);// 2
console.log(OrderStatus.DELIVERED); // 4
```

Here, `NEW` starts at `1`, and the subsequent values increment automatically. If you need specific values for each constant, you can manually assign them too:

```
enum ResponseCode {
  SUCCESS = 200,
  UNAUTHORIZED = 401,
  NOT_FOUND = 404,
  SERVER_ERROR = 500
}

console.log(ResponseCode.SUCCESS);      // 200
console.log(ResponseCode.SERVER_ERROR); // 500
```

This is useful when you want to directly match enums to predefined numeric values, such as HTTP status codes or error codes.

String Enums

In addition to numeric enums, TypeScript also supports string-based enums. These are useful when the enum values represent more descriptive, non-numeric values:

```
enum Direction {
  North = 'NORTH',
  South = 'SOUTH',
  East = 'EAST',
  West = 'WEST'
}

console.log(Direction.North); // "NORTH"
console.log(Direction.West);  // "WEST"
```

String enums provide more meaningful output and are commonly used when you need more readable and user-friendly constants. Unlike numeric enums, string enums don't increment values automatically, so you need to assign each value explicitly.

Enum Properties and Reverse Mappings

TypeScript's numeric enums also have a reverse mapping feature. This means that you can get the name of an enum member by its value:

```
enum Color {
  Red = 1,
  Green,
  Blue
}

console.log(Color.Red);    // 1
console.log(Color[2]);     // "Green"
```

In this example, accessing `Color[2]` returns the name `Green`, thanks to TypeScript's reverse mapping for numeric enums. Note that this feature is only available for numeric enums, not for string enums.

Use Cases for Enums

Enums are particularly helpful when you have a set of related values that don't change often, as:

- **Statuses**: For tracking different states in a workflow, like `PENDING`, `COMPLETED`, or `FAILED`.
- **Directions**: Like `NORTH`, `SOUTH`, `EAST`, `WEST` in a navigation system.
- **Error Codes**: Matching error codes to meaningful constants.
- **Roles or Permissions**: Like `ADMIN`, `USER`, `GUEST` to represent different access levels.

Enums can also make your code easier to maintain and prevent mistakes, because tools like IntelliSense will provide you with the possible options as soon as you start typing.

```
enum Role {
  Admin = 'ADMIN',
  User = 'USER',
  Guest = 'GUEST'
}
```

```
function checkAccess(role: Role) {
  if (role === Role.Admin) {
    console.log('Full access granted.');
  } else if (role === Role.User) {
    console.log('Limited access.');
  } else {
    console.log('No access.');
  }
}

checkAccess(Role.Admin); // Full access granted.
```

Constant Enums

In cases where you want to avoid the overhead of generating an object at runtime, TypeScript provides constant enums. These are completely removed during the compilation phase, and the values are inlined into your code.

```
const enum Days {
  Monday,
  Tuesday,
  Wednesday
}

let day: Days = Days.Monday;
console.log(day); // 0
```

Using `const enum` improves performance by reducing the footprint of your final JavaScript output. However, you lose features like reverse mapping, since the enum doesn't exist as an object in the compiled JavaScript.

Heterogeneous Enums

Though less common, TypeScript allows heterogeneous enums, where you mix string and numeric values:

```
enum Result {
  Success = 1,
  Failure = 'FAIL'
}
```

```
console.log(Result.Success); // 1
console.log(Result.Failure); // "FAIL"
```

This type of enum can be helpful in very specific scenarios where some values must be numbers and others need to be descriptive strings, but it's generally best to stick to either all string or all numeric values for consistency and readability.

Enum as Flags

Enums can also be used to represent **bitwise flags**. This is especially useful when you need to store multiple related options in a single value. Here's how it works:

```
enum Permissions {
  Read = 1,       // 0001
  Write = 2,      // 0010
  Execute = 4     // 0100
}

let userPermissions = Permissions.Read | Permissions.Write;

function hasPermission(perm: Permissions, checkPerm: Permissions): boolean {
  return (perm & checkPerm) === checkPerm;
}

console.log(hasPermission(userPermissions, Permissions.Read));   // true
console.log(hasPermission(userPermissions, Permissions.Execute));// false
```

In this case, we use the bitwise OR (|) operator to combine permissions and the bitwise AND (&) operator to check if a specific permission exists.

Summary

Enums in TypeScript are a powerful feature that can help make your code more readable, maintainable, and less prone to errors. Whether you're working with numeric values, descriptive strings, or using enums as bitwise flags, they give you a clean and concise way to manage sets of related constants.

With their versatility and the ability to easily integrate into your project, enums are a great addition to your TypeScript toolbox. Don't forget to use them whenever you need a fixed set of options or states!