guide # typescript

Modules & Imports

TCSS 460 — Client/Server Programming

As your projects grow beyond a single file — whether an Express API or a React front end — you need a way to split code into separate files and connect them together. TypeScript's module system lets you do exactly that — export functions, interfaces, and classes from one file and import them into another. If you've used Java's `import` statements and one-class-per-file convention, you already understand the motivation. This guide shows you how modules work in TypeScript and how they'll shape every project you build in this course.

---

1 Why Modules?

Every non-trivial program needs to be split across multiple files. A single 2,000-line file is hard to navigate, hard to test, and hard for a team to work on simultaneously. Modules solve this by giving each file a clear responsibility and an explicit contract for what it shares with the rest of the project.

1.1 The Java Model You Already Know

In Java, the organizational unit is the **class**, and the convention is one public class per file:

```java
// User.java
package com.example.models;

public class User {
    private String name;
    private String email;

    public User(String name, String email) {
        this.name = name;
        this.email = email;
    }

    public String getName() { return name; }
    public String getEmail() { return email; }
}
```

```java
// UserService.java
package com.example.services;

import com.example.models.User;

public class UserService {
    public User findById(int id) {
        // ...
    }
}
```

Java's `import` statement tells the compiler where to find the `User` class. The package structure mirrors the directory structure. You've been doing this since TCSS 142.

1.2 The TypeScript Model

TypeScript follows a similar idea, but with more flexibility. Instead of one class per file, **any file can export any number of values** — functions, interfaces, types, constants, or classes. The file itself is the module.

```typescript
// validation.ts
export function validate(email: string): boolean {
    return email.includes("@");
}

export interface User {
    name: string;
    email: string;
}
```

```typescript
// userService.ts
import { validate, User } from "./validation";

function createUser(name: string, email: string): User {
    if (!validate(email)) {
        throw new Error("Invalid email");
    }
    return { name, email };
}
```

The key difference: Java organizes code around **classes**. TypeScript organizes code around **files**. A TypeScript file can export a single class, or it can export a mix of functions, interfaces, and constants — whatever makes sense for that file's responsibility.

1.3 One File, One Concern

The principle is the same in both languages: **each file should have a clear, focused purpose**. In an Express project, that typically looks like this:

```
src/
├── controllers/
│   ├── userController.ts    ← route handler functions for /users
│   └── movieController.ts   ← route handler functions for /movies
├── routes/
│   ├── userRoutes.ts        ← route definitions for /users
│   └── movieRoutes.ts       ← route definitions for /movies
├── middleware/
│   ├── authMiddleware.ts    ← JWT verification
│   └── errorHandler.ts     ← centralized error handling
├── services/
│   └── tmdbService.ts       ← TMDB API client
├── interfaces/
│   └── user.ts              ← User interface and related types
└── index.ts                 ← app entry point, wires everything together
```

Each file exports what other files need, and imports what it depends on. The `import` and `export` keywords are the glue that holds the project together.

---

2 Named Exports

A **named export** attaches a specific name to a value and makes it available to other files. You can have as many named exports per file as you need.

2.1 Exporting Functions

Add the `export` keyword before a function declaration:

```
// validation.ts
export function validateEmail(email: string): boolean {
    return email.includes("@") && email.includes(".");
}

export function validatePassword(password: string): boolean {
    return password.length >= 8;
}
```

Both functions are now available to any file that imports from `"./validation"`.

2.2 Exporting Interfaces and Types

Interfaces and type aliases can be exported the same way:

```ts
// types.ts
export interface User {
    id: number;
    name: string;
    email: string;
}

export interface Movie {
    id: number;
    title: string;
    releaseYear: number;
}

export type ApiResponse<T> = {
    data: T;
    message: string;
};
```

This is a common pattern in Express projects — you define your data shapes in one file and import them wherever they're needed. It keeps your types consistent across route handlers, services, and middleware.

2.3 Exporting Constants and Variables

Constants are also commonly exported:

```ts
// config.ts
export const PORT = 3000;
export const DATABASE_URL = process.env.DATABASE_URL ??
"postgresql://localhost:5432/mydb";
export const JWT_SECRET = process.env.JWT_SECRET ?? "dev-secret";
```

2.4 Export After Declaration

You can also declare first and export later using an export list at the bottom of the file:

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

function multiply(a: number, b: number): number {
    return a * b;
}

const PI = 3.14159;

export { add, multiply, PI };
```

Both styles — inline `export` and bottom-of-file export lists — are valid. Inline exports are more common in practice because they make it immediately clear which values are public.

Try It Yourself

1. Create a file called `mathUtils.ts` with two exported functions: `add` and `multiply`
2. Create a second file called `main.ts` that imports and uses both functions
3. Run `npx ts-node main.ts` to verify it works
4. Try removing `export` from one function and see what error you get when importing it

---

3 Named Imports

To use an exported value, you **import** it by name using curly braces.

3.1 Basic Named Imports

```
import { validateEmail, validatePassword } from "./validation";

const isValidEmail = validateEmail("alice@example.com");  // true
const isValidPass = validatePassword("short");             // false
```

The names inside the curly braces must match the exported names exactly. This is different from Java, where the class name is determined by the file — in TypeScript, a single file can export many things, and you pick the ones you need.

3.2 Importing Selectively

You don't have to import everything a file exports. Import only what you need:

```
// Only need the User interface, not Movie or ApiResponse
import { User } from "./types";
```

This is good practice — it makes your dependencies explicit and keeps your imports clean.

3.3 Renaming Imports

Sometimes two files export values with the same name, or you want a more descriptive name in the current context. Use `as` to rename:

```
import { validate as validateUser } from "./userValidation";
import { validate as validateMovie } from "./movieValidation";

validateUser(userData);
validateMovie(movieData);
```

You can also rename during export:

```
// validation.ts
function checkEmail(email: string): boolean {
    return email.includes("@");
}

export { checkEmail as validateEmail };
```

3.4 Namespace Imports

If you need many exports from one file, you can import everything under a single namespace:

```
import * as validators from "./validation";

validators.validateEmail("alice@example.com");
validators.validatePassword("secret123");
```

This is similar to how you might use a class with static methods in Java:

```
// Java equivalent concept
Validators.validateEmail("alice@example.com");
Validators.validatePassword("secret123");
```

Use namespace imports sparingly. Named imports are usually clearer because they show exactly which values you depend on.

⚠ Common Mistake: Forgetting the Curly Braces

```
// WRONG — this tries to import a default export (Section 4)
import validateEmail from "./validation";

// CORRECT — curly braces for named imports
import { validateEmail } from "./validation";
```

Forgetting the curly braces is one of the most common import errors. Without them, TypeScript looks for a **default export**, which is a different mechanism entirely. If your file uses named exports and you forget the braces, you'll get an error like: `Module '"./validation"' has no default export.`

---

4 Default Exports

A **default export** is a single value that represents the "main thing" a file provides. Each file can have at most one default export.

4.1 Exporting a Default

```
// logger.ts
export default class Logger {
    private prefix: string;

    constructor(prefix: string) {
        this.prefix = prefix;
    }

    log(message: string): void {
        console.log(`[${this.prefix}] ${message}`);
    }

    error(message: string): void {
        console.error(`[${this.prefix}] ERROR: ${message}`);
    }
}
```

4.2 Importing a Default

When importing a default export, you **don't use curly braces**, and you can name it anything you want:

```
import Logger from "./logger";
// or
import MyLogger from "./logger";
// or
import AppLogger from "./logger";

const log = new Logger("App");
log.log("Server started");
```

The importer chooses the name. This is fundamentally different from named exports, where the name is fixed by the exporter.

4.3 Combining Default and Named Exports

A file can have both a default export and named exports:

```
// database.ts
export default class Database {
    async connect(): Promise<void> {
        // connect to PostgreSQL
    }

    async query(sql: string): Promise<unknown[]> {
        // execute a query
    }
}

export interface DatabaseConfig {
    host: string;
    port: number;
    database: string;
}

export const DEFAULT_PORT = 5432;
```

Importing both:

```
import Database, { DatabaseConfig, DEFAULT_PORT } from "./database";
```

4.4 Named vs. Default: Which Should You Use?

The short answer: **prefer named exports**.

Aspect	Named Exports	Default Exports
Per file	Unlimited	One maximum

Aspect	Named Exports	Default Exports
Import syntax	`import { X } from "..."`	`import X from "..."`
Name control	Exporter controls the name	Importer chooses the name
Refactoring	Rename propagates automatically	Rename doesn't propagate
IDE support	Better auto-import, auto-complete	Less reliable auto-import
Tree shaking	Bundlers can remove unused exports	Harder to optimize

Named exports are easier to refactor. If you rename a named export, your IDE can update all import sites automatically. With default exports, every file that imports it chose its own name — renaming the export doesn't update those.

> **Course Convention**
>
> In this course, **prefer named exports** for functions, interfaces, types, and constants. Default exports are fine for files that genuinely export one primary thing (like a class), but named exports are the default choice.

You'll encounter default exports frequently in the Node.js ecosystem — many npm packages use them. Express itself uses a default export:

```
import express from "express";
```

So you need to understand both, even if you prefer named exports in your own code.

---

5 Re-exporting (Barrel Files)

As your project grows, import paths can get long and repetitive. **Re-exporting** — also called creating a **barrel file** — solves this by gathering exports from multiple files into a single entry point.

5.1 The Problem

Imagine an Express project with several route files:

```
// In index.ts (the app entry point)
import { userRoutes } from "./routes/userRoutes";
import { movieRoutes } from "./routes/movieRoutes";
import { authRoutes } from "./routes/authRoutes";
import { searchRoutes } from "./routes/searchRoutes";
```

Every time you add a new route file, you add another import line here. The file that creates these routes (`index.ts`) has to know the exact file path of every single route module.

5.2 The Solution: An Index File

Create an `index.ts` inside the `routes/` directory that re-exports everything:

```
// routes/index.ts
export { userRoutes } from "./userRoutes";
export { movieRoutes } from "./movieRoutes";
export { authRoutes } from "./authRoutes";
export { searchRoutes } from "./searchRoutes";
```

Now the app entry point imports from the directory:

```
// index.ts
import { userRoutes, movieRoutes, authRoutes, searchRoutes } from "./routes";
```

When TypeScript sees an import from a directory (like `"./routes"`), it automatically looks for an `index.ts` file inside that directory. This is called **index resolution**.

5.3 Re-export Syntax Variations

Re-export specific named exports:

```
export { validateEmail, validatePassword } from "./validation";
```

Re-export everything from a file:

```
export * from "./validation";
```

Re-export with renaming:

```
export { validate as validateUser } from "./userValidation";
export { validate as validateMovie } from "./movieValidation";
```

5.4 A Real Express Project Structure

Here's how barrel files work in practice for an Express API:

```
src/
├── routes/
│   ├── index.ts              ← barrel: re-exports all route modules
│   ├── userRoutes.ts
│   ├── movieRoutes.ts
│   └── authRoutes.ts
├── middleware/
│   ├── index.ts              ← barrel: re-exports all middleware
│   ├── authMiddleware.ts
│   └── errorHandler.ts
├── interfaces/
│   ├── index.ts              ← barrel: re-exports all interfaces
│   ├── user.ts
│   └── movie.ts
└── index.ts                  ← app entry point
```

The route barrel file:

```
// routes/index.ts
export { userRoutes } from "./userRoutes";
export { movieRoutes } from "./movieRoutes";
export { authRoutes } from "./authRoutes";
```

The interface barrel file:

```
// interfaces/index.ts
export { User, CreateUserRequest } from "./user";
export { Movie, MovieSearchResult } from "./movie";
```

The app entry point becomes clean and organized:

```
// index.ts
import express from "express";
import { userRoutes, movieRoutes, authRoutes } from "./routes";
import { authMiddleware, errorHandler } from "./middleware";

const app = express();
app.use(express.json());
app.use("/api/users", authMiddleware, userRoutes);
app.use("/api/movies", movieRoutes);
app.use("/api/auth", authRoutes);
app.use(errorHandler);
```

Try It Yourself

1. Create a `src/utils/` directory with two files: `stringUtils.ts` and `numberUtils.ts`
2. Export a function from each (e.g., `capitalize` and `clamp`)
3. Create `src/utils/index.ts` that re-exports both
4. In a `main.ts` file, import both functions from `"./utils"` (not from the individual files)
5. Verify it works with `npx ts-node main.ts`

Circular Dependencies

Barrel files can accidentally create **circular dependencies** — where File A imports from File B, which imports from File A (possibly through re-exports). TypeScript will sometimes compile circular imports without error, but you'll get `undefined` values at runtime. If you see unexpected `undefined` where you expected a function or class, check for circular imports.

---

6 Module Resolution

When you write `import { X } from "..."`, TypeScript needs to figure out which file `"..."` refers to. The rules depend on whether the path starts with `.` or not.

6.1 Relative Imports

Paths starting with `./` or `../` are **relative imports** — they point to files in your project, relative to the current file's location.

```
// From src/routes/userRoutes.ts

import { User } from "../interfaces/user";       // go up one directory, then
into interfaces/
import { validate } from "./helpers";             // same directory
import { authMiddleware } from "../middleware";    // go up, then into
middleware/ (index.ts)
```

Key rules:

- `./` means "same directory as this file"
- `../` means "parent directory"
- You can chain `../` to go up multiple levels: `../../config`

- **Omit the `.ts` extension** — TypeScript adds it automatically
- Importing a directory (e.g., `"../middleware"`) resolves to `index.ts` inside that directory

In Java, packages use dot notation (`com.example.models.User`) and are resolved from the classpath root. In TypeScript, paths are filesystem-relative — you navigate the actual directory structure.

6.2 Package Imports

Paths that **don't** start with `.` or `..` are **package imports** — they refer to installed npm packages in `node_modules/`:

```
import express from "express";                    // node_modules/express
import { Router } from "express";                 // named export from express
import { PrismaClient } from "@prisma/client";    // scoped package
import cors from "cors";                          // node_modules/cors
```

When TypeScript sees `import ... from "express"`, it looks in `node_modules/express/` for the package's entry point. You never write the `node_modules/` path yourself.

In Java, the equivalent is importing from your build tool's dependencies:

```
// Java — resolved from Maven/Gradle dependencies
import com.google.gson.Gson;
```

In TypeScript, npm is your Maven/Gradle, and the import path is just the package name.

6.3 Type Declaration Packages

Many npm packages were written in JavaScript and don't include TypeScript type information. The community maintains type declarations in packages prefixed with `@types/`:

```
npm install express           # the package itself (JavaScript)
npm install @types/express    # TypeScript type declarations
```

After installing `@types/express`, TypeScript automatically picks up the types — you don't need to import anything extra. Your `import express from "express"` just works with full type checking.

Not All Packages Need @types

Packages written in TypeScript (like Prisma) ship their own type declarations. You only need `@types/` packages for JavaScript libraries that don't include types. Your IDE will tell you — if you see a red squiggly under an import, try installing the corresponding `@types/` package.

6.4 Path Aliases (Brief Overview)

In larger projects, relative paths can get unwieldy:

```
import { User } from "../../../../interfaces/user";
```

TypeScript supports **path aliases** in `tsconfig.json` that let you write shorter paths:

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

Now you can write:

```
import { User } from "@/interfaces/user";
```

The `@/` prefix maps to the `src/` directory regardless of where you are in the project. This is a convenience feature — the starter repos in this course may or may not use path aliases, so check each project's `tsconfig.json`.

---

7 CommonJS vs. ES Modules

If you read Node.js tutorials or Stack Overflow answers, you'll see two different import syntaxes. Understanding why both exist will save you confusion.

7.1 A Brief History

When Node.js was created in 2009, JavaScript had no built-in module system. Node.js invented its own: **CommonJS**. It uses `require()` to import and `module.exports` to export:

```
// CommonJS (the old way)
const express = require("express");
const { Router } = require("express");

module.exports = { myFunction, myClass };
```

Years later, the JavaScript language standard added **ES Modules** (ESM) — the `import` / `export` syntax you've been learning in this guide:

```
// ES Modules (the modern way)
import express from "express";
import { Router } from "express";

export { myFunction, myClass };
```

7.2 Why Both Exist in Node.js

Node.js now supports both systems. Older packages and tutorials use CommonJS. Newer code uses ES Modules. You'll encounter both in the wild:

Feature	CommonJS	ES Modules
Syntax	`require()` / `module.exports`	`import` / `export`
Loading	Synchronous (blocking)	Asynchronous (non-blocking)
Tree shaking	Not possible	Bundlers can remove unused exports
Static analysis	Imports resolved at runtime	Imports resolved at compile time
Era	Node.js original (2009)	JavaScript standard (ES2015+)

7.3 What We Use in This Course

In this course, you **always write ES Module syntax** (`import` / `export`). TypeScript handles the conversion behind the scenes.

Here's what happens:

1. You write `import express from "express"` in your `.ts` file

2. The TypeScript compiler (`tsc`) compiles your code to JavaScript
3. Depending on your `tsconfig.json` settings, the output may use CommonJS `require()` calls or ES Module `import` statements
4. Node.js executes the compiled JavaScript

The `tsconfig.json` in your starter projects is already configured for this. You don't need to think about CommonJS when writing code — just use `import` and `export`.

> ⚠ **Don't Mix Syntaxes**
>
> Never use `require()` in a TypeScript file:
>
> ```
> // WRONG — don't do this in TypeScript
> const express = require("express");
>
> // CORRECT — always use import syntax
> import express from "express";
> ```
>
> If you see `require()` in a tutorial, mentally translate it to `import` syntax. The functionality is the same — the syntax is different.

7.4 Recognizing CommonJS in the Wild

You'll see CommonJS in older Node.js tutorials, Stack Overflow answers, and some npm package documentation. Being able to read it is useful even though you won't write it:

```
// CommonJS pattern you'll see in tutorials
const express = require("express");
const app = express();
const PORT = 3000;

app.get("/", (req, res) => {
    res.send("Hello World");
});

app.listen(PORT, () => {
    console.log(`Server running on port ${PORT}`);
});
```

The equivalent in our ES Module + TypeScript style:

```
// What you'd write in this course
import express from "express";

const app = express();
const PORT = 3000;
```

```
app.get("/", (req, res) => {
    res.send("Hello World");
});

app.listen(PORT, () => {
    console.log(`Server running on port ${PORT}`);
});
```

The only difference in this example is line 1. In more complex code, CommonJS uses `module.exports = { ... }` where you'd use `export { ... }`.

Gen AI & Learning: Import Syntax in AI-Generated Code

When you ask a coding agent to generate Node.js or Express code, pay attention to the import style it produces. Agents sometimes generate CommonJS `require()` syntax, especially if the prompt doesn't specify TypeScript. If you see `require()` in generated code, ask the agent to convert it to ES Module `import` syntax, or do the translation yourself. This is a quick check that becomes second nature.

---

8 Summary

Concept	Key Point
Module	Any TypeScript file with `import` or `export` is a module
Named export	`export function X()` — exporter controls the name, multiple per file
Named import	`import { X } from "..."` — curly braces required, names must match
Default export	`export default X` — one per file, importer chooses the name
Default import	`import X from "..."` — no curly braces
Re-export	`export { X } from "..."` — gather exports into a barrel file
Barrel file	An `index.ts` that re-exports from sibling files

Concept	Key Point
Relative import	`"./file"` or `"../file"` — your project files
Package import	`"express"` — npm packages in `node_modules/`
@types packages	Type declarations for JS packages (e.g., `@types/express`)
CommonJS	`require()` / `module.exports` — legacy Node.js syntax
ES Modules	`import` / `export` — modern standard, what we use
Path aliases	`@/...` shortcuts configured in `tsconfig.json`

---

9 References

Official Documentation:

- TypeScript Handbook — Modules — The authoritative reference for TypeScript's module system
- TypeScript Handbook — Module Resolution — How TypeScript finds imported files
- MDN Web Docs — JavaScript Modules — ES Module specification and browser support
- Node.js Documentation — Modules — ES Modules in Node.js, including interop with CommonJS

Tutorials:

- TypeScript Deep Dive — Modules — Practical guide to module patterns in TypeScript projects

---

10 Further Reading

External Resources

- Node.js — CommonJS vs. ES Modules — Official comparison of the two module systems
- TypeScript Handbook — tsconfig Reference — All compiler options, including module-related settings
- MDN Web Docs — import — Complete `import` statement reference
- MDN Web Docs — export — Complete `export` statement reference

This guide is part of TCSS 460 — Client/Server Programming, School of Engineering and Technology, University of Washington Tacoma.