guide # typescript

Building & Running TypeScript

TCSS 460 — Client/Server Programming

You know how to write TypeScript — type annotations, interfaces, unions, and the rest of the type system from the TypeScript Essentials guide. But what actually happens when you run your code? This guide covers the pipeline from `.ts` source to running program: what `tsc` produces, why types disappear from the output, how source maps reconnect output to source, and which development workflows you will use daily.

---

1 From Source to Runtime — The Big Picture

Every typed language follows the same fundamental pattern: you write source code, a tool transforms it, and a runtime executes the result. You have been doing this in Java since TCSS 142.

1.1 The Java Pipeline You Already Know

In Java, the pipeline looks like this:

```
YourApp.java  →  javac  →  YourApp.class  →  JVM
```

You write `.java` files. The Java compiler (`javac`) transforms them into `.class` files containing bytecode — a binary format that the Java Virtual Machine (JVM) understands. You cannot open a `.class` file in a text editor and read it. It is not meant for human eyes.

1.2 The TypeScript Pipeline

TypeScript follows the same "write, transform, run" pattern:

```
app.ts  →  tsc  →  app.js  →  Node.js
```

You write `.ts` files. The TypeScript compiler (`tsc`) transforms them into `.js` files — plain JavaScript source code. Node.js then executes the JavaScript.

1.3 Pipeline Comparison

The following table puts these pipelines side by side:

Step	Java	TypeScript
You write	`.java` source files	`.ts` source files
Tool transforms	`javac` (Java Compiler)	`tsc` (TypeScript Compiler)
Output format	`.class` bytecode (binary)	`.js` source code (text)
Runtime executes	JVM (Java Virtual Machine)	Node.js (or a browser)
Can you read the output?	No — bytecode is binary	Yes — it is JavaScript

The key observation: both pipelines follow "write, transform, run." But the outputs are fundamentally different. Java's compiler produces a binary format that only the JVM can interpret. TypeScript's compiler produces source code that you can open, read, and even edit by hand. This difference has profound implications for how the tools work and how you think about the build process.

---

2 Compiling vs. Transpiling

The TypeScript tool is named `tsc` — the **TypeScript Compiler**. Microsoft's official documentation calls the process "compilation." But what `tsc` does is not quite the same as what `javac` does, and understanding the distinction helps you reason about the entire toolchain.

2.1 Compilation

Compilation transforms source code into a *lower-level* representation — something closer to what the machine actually executes. The output is typically not human-readable.

Examples of compilation:

- **Java:** `.java` source → `.class` bytecode (binary instructions for the JVM)
- **C:** `.c` source → machine code (binary instructions for the CPU)

- **Rust:** `.rs` source → machine code

In each case, the output is at a fundamentally different abstraction level than the input. You cannot look at a `.class` file and see Java syntax.

2.2 Transpilation

Transpilation transforms source code into *another source language at roughly the same abstraction level*. The output is human-readable source code.

Examples of transpilation:

- **TypeScript:** `.ts` source → `.js` source
- **Sass:** `.scss` stylesheets → `.css` stylesheets
- **JSX:** `.tsx` files → `.js` files with function calls instead of HTML-like syntax

The key distinction: you can open the output file in a text editor and read it. The output looks almost identical to the input, with certain syntax removed or transformed.

> **Compiler or Transpiler?**
>
> Microsoft's official docs call `tsc` a "compiler" and the process "compilation." Technically, `tsc` performs transpilation — a source-to-source transformation. The industry uses both terms interchangeably for TypeScript. This course uses "compile" to match official documentation and the command name (`tsc` = TypeScript **C**ompiler), but understanding the distinction helps you reason about what is actually happening under the hood.

2.3 Why This Distinction Matters

Because `tsc` produces readable source code rather than binary, the transformation is relatively straightforward: strip type annotations, transform certain syntax features, and output JavaScript. The logic of your program passes through largely unchanged.

This is why alternative tools like **esbuild** and **SWC** can be dramatically faster than `tsc` — if the primary job is "remove type annotations from source code," you do not need the full infrastructure of a traditional compiler. These tools skip type checking entirely and focus on the transformation step alone. (You will not use esbuild or SWC in this course, but you should know they exist — modern build tools like Vite use them internally.)

Compare this with `javac`: the Java compiler must transform source code into a completely different binary format with its own instruction set, constant pools, and class file structure.

That transformation is fundamentally more complex than removing type annotations from text.

---

3 What `tsc` Actually Produces

The best way to understand what `tsc` does is to look at a concrete example. Here is a TypeScript file that uses several type system features:

3.1 A Complete TypeScript File

```
// user-service.ts

interface User {
    id: number;
    name: string;
    email: string;
}

type UserRole = "admin" | "editor" | "viewer";

function createGreeting(user: User, role: UserRole): string {
    const prefix: string = role === "admin" ? "Administrator" : "User";
    return `${prefix}: ${user.name} (${user.email})`;
}

const alice: User = {
    id: 1,
    name: "Alice",
    email: "alice@example.com"
};

const greeting: string = createGreeting(alice, "admin");
console.log(greeting);
```

This file uses an interface, a type alias with a union type, typed function parameters, a typed return value, and typed variable declarations.

3.2 The JavaScript Output

Run `npx tsc user-service.ts` and open the resulting `user-service.js`:

```
// user-service.js (produced by tsc)

function createGreeting(user, role) {
    var prefix = role === "admin" ? "Administrator" : "User";
```

```
    return prefix + ": " + user.name + " (" + user.email + ")";
}

var alice = {
    id: 1,
    name: "Alice",
    email: "alice@example.com"
};

var greeting = createGreeting(alice, "admin");
console.log(greeting);
```

3.3 Line-by-Line Walkthrough

Compare the two files:

TypeScript	JavaScript	What happened?
`interface User { ... }`	*(gone)*	Interface declaration completely removed
`type UserRole = "admin" | "editor" | "viewer"`	*(gone)*	Type alias completely removed
`function createGreeting(user: User, role: UserRole): string`	`function createGreeting(user, role)`	Parameter types and return type stripped
`const prefix: string = ...`	`var prefix = ...`	Type annotation stripped; `const` became `var`
`const alice: User = { ... }`	`var alice = { ... }`	Type annotation stripped
Template literal `` `${prefix}: ...` ``	`prefix + ": " + ...`	Template literal converted to concatenation
`console.log(greeting)`	`console.log(greeting)`	Identical — runtime code unchanged

The runtime logic — the conditional, the string building, the object literal, the function call — is identical. Everything that was *type syntax* is gone. Everything that was *runtime behavior* remains.

> **Note**
>
> The `const` to `var` conversion and template literal to concatenation changes happen because `tsc` defaults to targeting an older JavaScript version (ES3/ES5). With `"target": "ES2020"` or later in your `tsconfig.json` (which our starter projects use), `const` stays as `const` and template literals stay as template literals. The type erasure behavior is the same regardless of target.

3.4 Type Erasure Explained

This process is called **type erasure**. Types exist in two places:

1. In your `.ts` source files (for you and your teammates to read)
2. In the compiler's memory (for `tsc` to check correctness)

Types never exist in the running program. They are not encoded in the output. They have zero runtime cost — no extra memory, no extra CPU cycles, no extra bytes in the deployed file.

If you have used Java generics, you have seen a limited form of type erasure: `ArrayList<String>` becomes just `ArrayList` at runtime. The JVM does not know the generic type parameter. But Java retains the class structure, method signatures, and other metadata in bytecode. TypeScript erasure is far more thorough — **all** type syntax is removed. Interfaces, type aliases, type annotations, generic type parameters, union types — none of it survives into the output.

3.5 The Enum Exception

There is one notable exception to "types disappear": **enums**. Unlike interfaces and type aliases, TypeScript enums produce runtime JavaScript code.

Here is a TypeScript enum:

```
enum Direction {
    Up,
    Down,
    Left,
    Right
}

const move: Direction = Direction.Up;
console.log(move);        // 0
console.log(Direction[0]); // "Up"
```

And here is what `tsc` produces:

```
var Direction;
(function (Direction) {
    Direction[Direction["Up"] = 0] = "Up";
    Direction[Direction["Down"] = 1] = "Down";
    Direction[Direction["Left"] = 2] = "Left";
    Direction[Direction["Right"] = 3] = "Right";
})(Direction || (Direction = {}));

var move = Direction.Up;
console.log(move);          // 0
console.log(Direction[0]);  // "Up"
```

The `Direction` enum becomes a real JavaScript object that exists at runtime. It creates a two-way mapping: `Direction.Up` returns `0`, and `Direction[0]` returns `"Up"`. This is fundamentally different from an interface or type alias, which leave no trace in the output.

> ⚠ **Enums Are the Exception, Not the Rule**
>
> Most TypeScript type constructs (interfaces, type aliases, union types, type annotations) are erased completely. Enums are the primary exception — they generate runtime code because they define *values*, not just *types*. In this course, you will use interfaces far more often than enums. When you see an `enum` in TypeScript, remember that it carries a runtime cost that interfaces do not.

> 🔥 **Try It Yourself**
>
> 1. Create a file called `erasure-demo.ts`
> 2. Add an interface, a type alias, a typed function, and an enum
> 3. Run `npx tsc erasure-demo.ts`
> 4. Open `erasure-demo.js` in your editor
> 5. Answer these questions: What happened to the interface? What happened to the `: string` annotations? Why is the function body identical? What did the enum become?

---

4 Source Maps — Connecting Output Back to Source

4.1 The Problem

When your program throws an error at runtime, the error message references line numbers in the file that Node.js is actually executing — the `.js` file. But you wrote `.ts`. The line numbers

will not match because type annotations, interfaces, and other erased content shift the line positions.

Consider a runtime error in your Express API:

```
TypeError: Cannot read properties of undefined (reading 'name')
    at createGreeting (user-service.js:3:42)
```

The error points to `user-service.js` line 3. But you need to find the bug in `user-service.ts`, which might have the corresponding code on line 12 because the interface and type alias occupy lines 3 through 10. Hunting for the right line in your source file is tedious and error-prone.

4.2 Source Maps Solve This

A **source map** is a file (with a `.js.map` extension) that contains a lookup table mapping positions in the compiled `.js` file back to positions in the original `.ts` file. When a runtime error occurs, tools that understand source maps translate the error location automatically.

To enable source maps, add this to your `tsconfig.json`:

```
{
    "compilerOptions": {
        "sourceMap": true
    }
}
```

With this enabled, `tsc` produces three files for every `.ts` input:

File	Purpose
`user-service.js`	The JavaScript output (what Node.js runs)
`user-service.js.map`	The source map (lookup table)
`user-service.ts`	Your original source (unchanged)

4.3 What a Source Map Contains

If you open a `.js.map` file, you will see JSON that looks something like this:

```
{
    "version": 3,
```

```
    "file": "user-service.js",
    "sourceRoot": "",
    "sources": ["user-service.ts"],
    "mappings": "AAQA,SAAS,eAAe..."
}
```

The `mappings` field is a compact encoded string that maps every position in the `.js` output to the corresponding position in the `.ts` source. You do not need to understand the encoding — the important thing is that tools read this file automatically.

4.4 Source Maps in Practice

In most development scenarios, source maps work invisibly:

- **Node.js** reads source maps automatically (Node.js 12+ with `--enable-source-maps` flag, enabled by default in recent versions)
- **Browser DevTools** detect source maps and show your original TypeScript in the Sources panel
- **VS Code / WebStorm debuggers** use source maps to set breakpoints in `.ts` files and step through TypeScript code

> **Note**
>
> Tools like `ts-node` and `ts-node-dev` compile TypeScript in memory and handle source map translation internally. You do not see `.js` or `.js.map` files on disk when using these tools — the mapping still happens, just behind the scenes.

---

5 Understanding `package.json`

Before diving into development workflows, you need to understand the file that ties a Node.js project together. Every project you work with in this course has a `package.json` at its root — it is the manifest file that describes your project, its dependencies, and how to run it.

If you have used Maven in Java, `package.json` serves a similar role to `pom.xml`: it declares what your project needs and how to build it.

5.1 Key Fields

Here is a representative `package.json` from a lecture demo project:

```
{
    "name": "tcss460-lecture-demo",
    "version": "1.0.0",
    "scripts": {
        "dev": "ts-node-dev --respawn src/index.ts",
        "build": "tsc",
        "start": "node dist/index.js",
        "lint": "eslint src/",
        "format": "prettier --write src/",
        "test": "jest"
    },
    "dependencies": {
        "cors": "^2.8.5",
        "dotenv": "^16.4.0",
        "express": "^5.0.0"
    },
    "devDependencies": {
        "@types/cors": "^2.8.17",
        "@types/express": "^5.0.0",
        "eslint": "^9.0.0",
        "jest": "^29.7.0",
        "prettier": "^3.4.0",
        "ts-node-dev": "^2.0.0",
        "typescript": "^5.7.0"
    }
}
```

Field	Purpose
`name`	The project name (used by npm if you ever publish; mostly informational)
`version`	The project version (follows semantic versioning)
`scripts`	Custom commands you run with `npm run <name>`
`dependencies`	Packages your app needs to *run* (shipped to production)
`devDependencies`	Packages needed only during *development* (not shipped to production)

5.2 `npm install` and `node_modules/`

When you clone a project and run `npm install`, npm reads `package.json` and downloads every listed package (plus their dependencies) into a `node_modules/` folder. This folder can contain thousands of files — it is not committed to Git (the `.gitignore` excludes it).

This is analogous to Maven downloading `.jar` files into your local repository when you run `mvn install`. The `node_modules/` folder is your project's local library cache.

5.3 Scripts — Custom Commands

The `scripts` field defines shortcuts for common tasks. You run them with `npm run <script-name>`:

```
npm run dev       # Start the dev server with file watching
npm run build     # Compile TypeScript to JavaScript
npm run lint      # Check code for style issues
npm run format    # Auto-format all source files
npm run test      # Run the test suite
```

Two scripts have special shorthand: `npm start` (no `run` needed) and `npm test` (no `run` needed). All others require `npm run`.

5.4 dependencies vs. devDependencies

This separation matters for production deployment:

- **`dependencies`** — packages your application needs to function at runtime. Express handles HTTP requests. `cors` enables cross-origin requests. `dotenv` loads environment variables. Without these, your server cannot run.
- **`devDependencies`** — packages you use while writing code but that are not needed when the app runs in production. TypeScript compiles your code but is not needed after compilation. ESLint checks your style. Jest runs your tests. None of these ship to production.

When you deploy, the hosting platform runs `npm install --production` (or equivalent), which installs only `dependencies`. This keeps your production environment lean.

5.5 package-lock.json

You will notice a second file alongside `package.json`: `package-lock.json`. This file records the *exact* version of every installed package (and every transitive dependency). While `package.json` might say `"express": "^5.0.0"` (meaning "5.0.0 or any compatible newer version"), the lock file pins the exact version that was installed — for example, `5.0.1`.

Always Commit the Lock File

Always commit `package-lock.json` to Git. It ensures that every developer on your team — and every deployment — installs the exact same versions. Without it, one teammate might get Express 5.0.1 while another gets 5.0.3, leading to subtle "works on my machine" bugs.

---

6 Development Workflows

There are several ways to go from `.ts` source to a running program. Each workflow makes different trade-offs between speed, convenience, and production-readiness. This section covers the four workflows you will encounter in this course.

6.1 Compile-Then-Run: `tsc` + `node`

The most explicit workflow: compile first, then run the output.

```
npx tsc                # Compile all .ts files to .js (uses tsconfig.json)
node dist/index.js     # Run the compiled JavaScript
```

In a project with `package.json` scripts, this typically looks like:

```
npm run build          # Runs tsc, outputs to dist/
npm start              # Runs node dist/index.js
```

When to use: Production deployment, CI/CD pipelines, final testing before deployment.

Advantages:

- Catches all type errors upfront before any code runs
- Fast startup — Node.js runs plain JavaScript with no compilation overhead
- Produces the exact files that will be deployed

Disadvantages:

- Manual process — you must recompile after every change
- Two separate steps to remember

This is analogous to the Java workflow you know: `javac` to compile, `java` to run. You would not use this during active development because recompiling manually after every change is tedious.

6.2 Direct-Run: `ts-node`

`ts-node` compiles TypeScript in memory and runs it immediately — no `.js` files appear on disk.

```
npx ts-node src/index.ts
```

One command, and your TypeScript program runs. Internally, `ts-node` invokes `tsc` in memory, produces JavaScript, and feeds it directly to Node.js. The compiled output never touches your file system.

When to use: One-off scripts, quick experiments, testing a single file.

Advantages:

- Single command — no separate compile step
- No `.js` output files cluttering your project
- Handles source maps internally

Disadvantages:

- Slower startup — must compile before running every time
- Not suitable for production (compilation overhead on every start)
- Type errors surface at startup, not ahead of time

Think of `ts-node` as a convenience tool for development. It is similar to running `javac` and `java` in a single step — useful for quick iteration, not for deployment.

6.3 Watch Mode — Your Daily Driver

During active development, you want your server to automatically restart whenever you save a file. Watch mode tools handle this for you.

6.3.1 `ts-node-dev`

`ts-node-dev` combines `ts-node` (in-memory compilation) with file watching and automatic restart:

```
npx ts-node-dev --respawn src/index.ts
```

When you save any `.ts` file, `ts-node-dev` detects the change, recompiles in memory, and restarts your program. This is the tool configured in the course starter projects.

In `package.json`:

```
{
    "scripts": {
        "dev": "ts-node-dev --respawn src/index.ts"
    }
}
```

Then simply:

```
npm run dev
```

6.3.2 `tsc --watch`

TypeScript's built-in watch mode continuously compiles to disk whenever a file changes:

```
npx tsc --watch
```

This is useful when you want to inspect the compiled output or when another tool (like `nodemon`) handles restarting. However, it only compiles — it does not run your program. You would need a second terminal running `node` or a tool like `nodemon` watching the `dist/` folder.

6.3.3 `nodemon` with `ts-node`

`nodemon` is a general-purpose file watcher that can be configured to use `ts-node` for TypeScript:

```
npx nodemon --exec ts-node src/index.ts
```

`nodemon` is more configurable than `ts-node-dev` (you can specify which file extensions to watch, which directories to ignore, etc.), but requires more setup.

6.3.4 Comparing Watch Mode Tools

Tool	Compiles to disk?	Type checks?	Restarts on change?	Config needed?
`ts-node-dev --respawn`	No	Yes	Yes	Minimal
`tsc --watch`	Yes	Yes	No (compile only)	`tsconfig.json`

Tool	Compiles to disk?	Type checks?	Restarts on change?	Config needed?
`nodemon` + `ts-node`	No	Yes	Yes	`nodemon.json` or flags

For this course, `ts-node-dev` is the recommended default. It is already configured in the starter projects and requires the least setup.

6.4 Choosing the Right Workflow

Scenario	Workflow	Command
Deploying to production	Compile-then-run	`npm run build && npm start`
Quick experiment or script	Direct-run	`npx ts-node script.ts`
Active development (daily work)	Watch mode	`npm run dev` (ts-node-dev)
Inspecting compiled output	Watch + compile to disk	`npx tsc --watch`

During most of this course, you will use `npm run dev` and rarely think about the compilation step. But understanding what happens under the hood — that `ts-node-dev` is compiling TypeScript to JavaScript in memory before Node.js runs it — helps you debug problems when things go wrong.

---

7 Dev vs. Production — Why the Distinction Matters

7.1 Two Different Goals

Development and production have fundamentally different priorities:

Priority	Development	Production
Speed of feedback	Fast — see changes immediately	Not a concern
Startup time	Acceptable to be slower	Must be fast
Error detail	Verbose — stack traces, source maps	Minimal — log errors, do not expose internals
File size	Does not matter	Smaller is better
Dependencies	All tools installed	Only what the app needs to run

During development, you want `ts-node-dev` watching your files, source maps showing TypeScript line numbers in errors, and detailed error messages in your terminal. In production, none of that matters — you want your server to start quickly, run efficiently, and not ship development tools to your deployment environment.

7.2 `dependencies` vs. `devDependencies`

This is why `package.json` separates dependencies into two categories:

```
{
    "dependencies": {
        "express": "^5.0.0"
    },
    "devDependencies": {
        "typescript": "^5.7.0",
        "ts-node-dev": "^2.0.0",
        "@types/express": "^5.0.0"
    }
}
```

- **`dependencies`**: Packages your app needs to *run* (Express, Prisma, etc.)
- **`devDependencies`**: Packages you need only during *development* (TypeScript, ts-node-dev, type definitions, linters, test runners)

When you deploy to production, you install only `dependencies` (`npm install --production`). TypeScript itself is not needed in production because the compiled `.js` files are what actually runs.

7.3 The Production Build Process

The production workflow for a TypeScript project looks like this:

```
src/           →  tsc  →  dist/           →  Node.js
(your .ts)              (compiled .js)        (runs .js)
```

1. `tsc` compiles everything in `src/` to `dist/`
2. You deploy `dist/` along with `node_modules/` (production dependencies only)
3. Node.js runs `dist/index.js` directly — no TypeScript, no `ts-node`, no compilation at startup

In Java terms, this is the difference between running your application from IntelliJ during development versus building a `.jar` file and deploying it to a server. During development, IntelliJ compiles and runs incrementally. For deployment, you produce a self-contained artifact.

7.4 How the Starter Projects Encode This

The course starter projects have `package.json` scripts that encode both workflows:

```
{
    "scripts": {
        "dev": "ts-node-dev --respawn src/index.ts",
        "build": "tsc",
        "start": "node dist/index.js"
    }
}
```

Script	Purpose	When
`npm run dev`	Compile in memory + watch + restart	Daily development
`npm run build`	Compile `src/` to `dist/`	Before deployment
`npm start`	Run compiled JavaScript	Production

You will use `npm run dev` almost exclusively during this course. But when your group project deploys to a hosting platform, the platform runs `npm run build` followed by `npm start`. If you do not understand this distinction, you will encounter the classic "works on my machine, fails in production" problem — and now you know exactly why it happens.

> **Gen AI & Learning: Understanding Build Output**
>
> When a coding agent generates TypeScript for your project, it produces `.ts` files. But deployment platforms — Render, Railway, Vercel, AWS — run `.js` files. Understanding the build pipeline helps you debug the gap between "works locally" and "fails in production." If your agent adds a new route and it works with `npm run dev` but the deployed version returns a 404, the first question is: did `tsc` compile the new file into `dist/`? Did the deployment run `npm run build`? These are pipeline questions, not code questions — and they require understanding the build process, not just the TypeScript syntax.

---

8 Linting & Formatting

Two tools keep your codebase consistent and catch mistakes before they reach production: **ESLint** and **Prettier**. If you have used SpotBugs or Checkstyle in Java, these serve similar roles in the TypeScript ecosystem.

8.1 ESLint — Catching Bugs and Enforcing Rules

ESLint is a static analysis tool that scans your code for potential bugs, anti-patterns, and style violations. It does not run your code — it reads it and flags problems. Think of it as Checkstyle for TypeScript: it enforces rules like "no unused variables," "no unreachable code," and "always handle Promise rejections."

```
npm run lint
```

This checks all files in `src/` and reports issues without changing anything. You review the output and fix problems manually — or let your editor's ESLint integration highlight them in real time.

8.2 Prettier — Automatic Code Formatting

Prettier is an opinionated code formatter. It rewrites your code to enforce consistent style: indentation, semicolons, quote style, line length, trailing commas. Unlike ESLint (which flags problems for you to fix), Prettier fixes formatting automatically.

```
npm run format
```

This reformats every file in `src/` in place. After running it, your code looks exactly the way the team agreed it should — no more debates about tabs vs. spaces.

8.3 ESLint vs. Prettier

Tool	Purpose	Analogy	Changes files?
ESLint	Catch bugs, enforce code rules	Checkstyle / SpotBugs	No (reports only)
Prettier	Auto-format code style	Code beautifier	Yes (rewrites files)

The two tools complement each other. ESLint catches logic and quality issues. Prettier handles formatting. The lecture demo projects include configuration files for both.

> **Run Before Committing**
>
> Get in the habit of running `npm run lint` and `npm run format` before every commit. The CI pipeline checks both — if your code has lint errors or inconsistent formatting, the pipeline will fail. Better to catch it locally than to push and wait for CI to tell you.

---

9 Running Tests

Every project in this course includes a test suite powered by **Jest**. You do not need to understand testing in depth yet — that comes later. For now, know the two commands that run your tests.

9.1 `npm test`

```
npm test
```

This runs the full test suite once and reports results. You will see output showing which tests passed, which failed, and a summary.

9.2 `npm run test:watch`

```
npm run test:watch
```

This starts Jest in watch mode — it re-runs relevant tests every time you save a file. During active development, this gives you instant feedback on whether your changes broke anything.

9.3 Where to Learn More

We cover testing in depth in the API Testing guide. That guide walks through writing tests, test structure, assertion patterns, and the full Jest workflow. For now, knowing that `npm test` runs all tests and `npm run test:watch` re-runs them as you code is enough to get started.

---

10 Summary

Concept	Key Point
Pipeline	`.ts` → `tsc` → `.js` → Node.js (analogous to `.java` → `javac` → `.class` → JVM)
Transpilation	Source-to-source transformation; `tsc` produces readable JavaScript, not binary bytecode
Type erasure	All TypeScript types are removed from the output; zero runtime cost
Enum exception	Enums produce runtime JavaScript code (unlike interfaces and type aliases)
Source maps	`.js.map` files connect compiled output back to `.ts` source for debugging
`package.json`	Project manifest — declares dependencies, scripts, and metadata (like Maven's `pom.xml`)

Concept	Key Point
`package-lock.json`	Locks exact dependency versions — always commit this file
`ts-node`	Compile and run in memory — no `.js` files written to disk
`ts-node-dev`	`ts-node` with file watching and auto-restart — your daily development tool
`tsc --watch`	Continuously compile to disk on file changes (compile only, does not run)
Dev vs. production	Development uses `ts-node-dev`; production uses `tsc` + `node`
`dependencies` vs. `devDependencies`	TypeScript and dev tools are `devDependencies` — not needed in production
ESLint	Static analysis — catches bugs and enforces code rules (`npm run lint`)
Prettier	Auto-formatter — enforces consistent code style (`npm run format`)
`npm test`	Runs the full Jest test suite; `npm run test:watch` re-runs on file changes

---

11 References

Official Documentation:

- TypeScript Handbook — Compiler Options — Complete reference for `tsconfig.json` settings including `sourceMap`, `outDir`, and `target`
- TypeScript Handbook — Modules — How TypeScript handles module compilation
- ts-node Documentation — Official docs for the `ts-node` runtime
- ts-node-dev on npm — File watching and auto-restart for TypeScript development

- MDN — Use a source map — Browser-focused explanation of source maps that applies to Node.js as well
- Node.js — Source Map Support — Node.js documentation for the `--enable-source-maps` flag

---

12 Further Reading

External Resources

- TypeScript Playground — Paste TypeScript in the left panel, see compiled JavaScript in the right panel instantly. The best way to experiment with type erasure and compiler output without any local setup.
- esbuild — A JavaScript/TypeScript bundler written in Go that performs transpilation orders of magnitude faster than `tsc` by skipping type checking. Used internally by Vite and other modern build tools.
- SWC — A Rust-based alternative to esbuild with similar speed advantages. Used internally by Next.js for TypeScript compilation.
- Source Map Specification (v3) — The formal specification for the source map format, if you want to understand the encoding details.

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