TypeScript Fundamentals

Scale JavaScript Development with TypeScript Basics

Scott Tremaine
Software Developer and Educator

Breakpoint Coding Tutorials

Contents

Introduction to TypeScript

TypeScript, a superset of JavaScript, was created by Microsoft in 2012 to address some of JavaScript's inherent challenges with large-scale development. Led by Anders Hejlsberg, who previously designed C# and Delphi, TypeScript was developed to offer features that would make JavaScript more structured and manageable for building complex applications.

JavaScript is a highly flexible language, which allows variables to change types dynamically, leading to runtime errors that can be difficult to debug. Additionally, JavaScript's lack of built-in tooling for advanced code completion and refactoring often leads to slower workflows and less predictable outcomes in complex projects.

How TypeScript Improves Upon JavaScript

Static Typing System: The static typing system identifies potential type-related errors at compile time, minimizing runtime problems. By catching errors early, many common bugs are prevented from reaching production. The static type system also enhances the developer's productivity by empowering the IDE to offer features like autocompletion, refactoring suggestions, and inline error checking. This results in faster, more accurate coding and a greater level of confidence in the final product.

Enhanced Code Structure: Classes and interfaces, borrowed from object-oriented programming, help developers build reusable and structured code templates that maintain consistency throughout the application. Interfaces define clear data contracts that must be adhered to, bringing predictability to data handling. Additionally, namespaces and modules provide cohesive units of organization that make the code easier to navigate, understand, and maintain, especially as projects grow in complexity.

Support for Modern JavaScript Features: The language stays current by adopting the latest ECMAScript features, enabling developers to use new capabilities like async/await. These modern features help write asynchronous code that's more readable and maintainable. Early access to such features allows developers to write forward-compatible code that works across various browsers and environments.

Seamless Migration and Compatibility: The compilation into plain JavaScript ensures compatibility with existing JavaScript projects, tools, and browsers. This backward compatibility makes TypeScript suitable for teams transitioning their existing codebases. Adopting TypeScript can be done incrementally by using it for new modules or gradually refactoring older components. As a result, it seamlessly integrates with existing JavaScript, allowing developers to mix TypeScript and JavaScript without disrupting workflows or dependencies.

Advantages of TypeScript

The structured and modular nature of TypeScript leads to several advantages:

Reducing Runtime Errors: By catching errors before code execution, the static typing system helps prevent issues that often occur at runtime. Developers are immediately alerted to potential bugs, reducing the risk of faulty code reaching production environments. This early detection simplifies debugging, making applications more reliable and predictable.

Maintainability, Scalability, and Collaboration: The clear syntax and strong typing system significantly improve maintainability. Developers can more easily understand and modify the codebase, which is essential for long-term upkeep. The modular structures and namespaces allow the code to remain scalable, ensuring that projects grow smoothly and remain manageable even when large teams are involved. Interfaces and data contracts establish clear guidelines and expectations, helping teams collaborate efficiently and ensuring consistent data usage across different components and modules.

Key Differences from JavaScript

TypeScript introduces notable differences that enhance JavaScript development:

Strong Typing vs. Dynamic Typing: With strong typing, data structures are consistent and predictable, which helps prevent errors that are common in dynamically typed languages like JavaScript. Developers can be confident that their variables and functions maintain expected types throughout the application, leading to fewer unexpected behaviors and runtime issues.

Additional Features:

- **Interfaces**: Define consistent data structures that the code must adhere to, enforcing uniform data usage and making the intent clearer.
- **Enums**: Provide named sets of constants for easier reference, making the code more readable and reducing the chances of errors when working with repeated values.
- **Namespaces**: Allow developers to group related code components together, enhancing modularity and enabling better organization of the project as it scales.

Setting Up the Development Environment

Installing Node.js and TypeScript Compiler

To work with TypeScript efficiently, you'll need a development environment that includes Node.js and the TypeScript compiler. Node.js provides a server-side runtime for JavaScript, while the TypeScript compiler (often referred to as `tsc`) converts TypeScript code into JavaScript. Here's a step-by-step guide for setting up these essential tools.

Step-by-Step Guide for Different Platforms

Windows:

1. **Node.js:** Download the latest version from the official Node.js website. Run the installer and follow the instructions to complete the installation. Verify the installation by opening Command Prompt and typing

```
node -v
```

2. **TypeScript Compiler:** After installing Node.js, open Command Prompt and install TypeScript globally using `npm` (Node's package manager). Type:

```
npm install -g typescript
```

Confirm by typing

```
tsc -v
```

macOS:

1. **Node.js:** Download the Node.js installer from the official website. Run the installer and follow the prompts to complete the installation. Verify the installation by opening Terminal and typing.

```
node -v
```

2. **TypeScript Compiler:** In Terminal, install TypeScript globally using `npm`. Type:

```
npm install -g typescript
```

Confirm by typing

```
tsc -v
```

Linux:

1. **Node.js:** Depending on the Linux distribution, use the respective package manager or follow the instructions provided on the Node.js website. For Ubuntu-based distributions, open Terminal and run:

```
sudo apt-get update
sudo apt-get install nodejs npm
```

2. **TypeScript Compiler:** Once Node.js is installed, use `npm` to install TypeScript globally:

```
npm install -g typescript
```

Verify the installation by typing `tsc -v`.

Installing TypeScript Globally and Locally

Global Installation: Installing TypeScript globally (using the `-g` flag) makes it available system-wide. This is beneficial when you need to use `tsc` from any project directory.

Local Installation: For individual projects, it's also possible to install TypeScript locally as a project dependency, ensuring a specific TypeScript version is tied to that project. Inside your project's directory, run:

```
npm install typescript --save-dev
```

This will add TypeScript as a development dependency in the project's `node_modules` folder.

Verifying the Installation

After installation, confirm that TypeScript is properly installed by running

```
tsc -v
```

This command should print the installed version, indicating the compiler is ready for use. Now, your development environment is ready to start coding in TypeScript.

Development Tools and IDE Setup

Popular IDEs and Editors

To write TypeScript effectively, it's important to choose a suitable Integrated Development Environment (IDE) or code editor. The most popular choices include:

- **Visual Studio Code (VSCode):** A free and open-source editor by Microsoft, VSCode offers comprehensive support for TypeScript right out of the box. Its robust features include syntax highlighting, intelligent code completion, and inline error detection. The built-in TypeScript support can be extended further with extensions.
- **WebStorm:** A commercial IDE by JetBrains, WebStorm provides extensive features for TypeScript, including advanced code navigation, refactoring tools, and built-in debugging. It comes with strong support for front-end frameworks like Angular and React.
- **Sublime Text:** A lightweight, versatile text editor that can be enhanced with third-party TypeScript plugins. It offers basic code editing features and can be customized according to the developer's needs.
- **Atom:** Developed by GitHub, Atom is highly customizable with a range of packages for TypeScript support, offering syntax highlighting and linting features.

Enhancing Productivity with TypeScript-Specific Plugins

Regardless of the editor you choose, you can enhance your productivity by installing TypeScript-specific plugins:

- **TypeScript Language Features:** Many IDEs include built-in TypeScript support that provides features like autocomplete, error checking, and type inference. This makes it easier to write accurate and error-free code.
- **Linting Plugins:** Plugins like ESLint help enforce coding standards and best practices. By integrating with TypeScript, these linters can identify potential issues in your code, suggest improvements, and ensure uniform code style across the project.

- **Refactoring Tools:** Refactoring plugins automate code transformations to keep your codebase clean and maintainable. For instance, extracting interfaces, converting anonymous types to named types, or renaming symbols across the codebase.
- **Snippet Libraries:** Code snippets help automate repetitive tasks, improving coding speed. They can be customized to fit the project's conventions and make common patterns available as quick templates.
- **Debugging Integrations:** Some IDEs and plugins offer built-in debugging tools that work directly with TypeScript. They let you set breakpoints, inspect variables, and analyze the call stack in real time, streamlining the debugging process.

By leveraging these tools and plugins, you can maximize your productivity and ensure a smoother, more efficient development workflow in TypeScript.

First TypeScript Program

Creating a Simple TypeScript File

To begin writing your first TypeScript program, create a new file with a `.ts` extension. This extension signifies that it's a TypeScript file. For this example, let's name it `hello.ts` and place it in a directory of your choice. Open this file in your preferred IDE or editor and write the following TypeScript code:

```
/* hello.ts */
function greet(name: string): string {
    return `Hello, ${name}! Welcome to TypeScript.`;
}

console.log(greet("Developer"));
```

In this small program, we've defined a function named `greet` that accepts a string parameter `name` and returns a greeting message. The function uses TypeScript's type annotations to specify that `name` should be a string. We then call this function with the value `"Developer"` and print the result to the console.

Compiling to JavaScript Using the TypeScript Compiler (`tsc`)

After writing the TypeScript code, you need to compile it into plain JavaScript that can be executed in any JavaScript environment. The TypeScript compiler, `tsc`, will handle this conversion. Open a terminal or command prompt, navigate to the directory where `hello.ts` is located, and run:

```
tsc hello.ts
```

The compiler will generate a new file named `hello.js` in the same directory. This file contains the equivalent JavaScript code that can be executed by any JavaScript engine.

Running the Compiled JavaScript Code

Now that you have the compiled JavaScript file (`hello.js`), you can execute it using Node.js. In the terminal or command prompt, run:

```
node hello.js
```

This will output the message:

```
Hello, Developer! Welcome to TypeScript.
```

Congratulations! You've successfully written and executed your first TypeScript program. You can now explore more complex examples or integrate TypeScript into existing JavaScript projects.

TypeScript Compilation and Configuration Files

Compiling TypeScript

When you write a TypeScript file, it must be compiled into JavaScript before it can run in a JavaScript environment. The TypeScript compiler (`tsc`) handles this process, converting TypeScript into JavaScript by following the language rules and type annotations you specify. The compiled output maintains compatibility with all JavaScript environments, allowing you to mix TypeScript and JavaScript seamlessly.

Understanding How TypeScript Converts to JavaScript

The compilation process transforms TypeScript into JavaScript according to the rules defined in the configuration file or via command-line options. Type annotations are stripped away since JavaScript doesn't support them natively, and TypeScript's advanced features are translated into equivalent JavaScript code.

Compilation Targets and Output File Structures

During compilation, you can specify which version of JavaScript (ECMAScript) to target. TypeScript supports a range of targets, from ES3 to the latest version. This ensures compatibility with different JavaScript environments. Additionally, you can customize the output file structure by specifying a root directory for the input source code and an output directory for the compiled files.

Configuration with `tsconfig.json`

To manage all the compiler settings, you can create a `tsconfig.json` file in your project directory. This file provides a structured way to define compiler options, source file inclusions, and output directory structures. When the TypeScript compiler detects `tsconfig.json`, it uses the options specified there for every compilation.

Explanation of Common Compiler Options

- **target:** Specifies the ECMAScript version to which TypeScript should compile (e.g., es5, es6/es2015, es2020).
- **module:** Determines the module system for the output (e.g., commonjs, esnext).
- **strict:** Enables strict mode, turning on all strict type-checking options.

- **rootDir:** Defines the root folder of the input TypeScript files.
- **outDir:** Specifies the directory where the compiled JavaScript files will be saved.

Creating and Configuring a `tsconfig.json` File

To create a `tsconfig.json` file automatically, navigate to your project directory in the terminal or command prompt and run:

```
tsc --init
```

This command generates a default configuration file. You can then customize it as needed. For instance:

```
/* tsconfig.json */
{
  "compilerOptions": {
    "target": "es5",
    "module": "commonjs",
    "strict": true,
    "rootDir": "./src",
    "outDir": "./dist"
  },
  "include": ["src/**/*"],
  "exclude": ["node_modules"]
}
```

In this configuration:

- **target** is set to `es5` to ensure compatibility with older browsers.
- **module** uses `commonjs` to align with Node.js module systems.
- **strict** mode is enabled to enforce best coding practices.
- **rootDir** defines `src` as the root directory for TypeScript source files.
- **outDir** specifies that compiled files should be output to the `dist` folder.

Using `include` and `exclude` for Better Control

The `include` and `exclude` fields refine which files and directories should be considered by the compiler. For instance, the configuration above includes all `.ts` files inside the `src` directory and ignores files inside `node_modules`. This helps reduce compilation time and ensures that only the intended files are processed.

With a well-configured `tsconfig.json`, you can streamline your TypeScript project's compilation process, making development more efficient and organized.

Type Annotations and Inference

Primitive Types

TypeScript allows you to specify explicit type annotations for variables and parameters. Primitive types include:

string: Represents text values. For example:

```
let username: string = "Alice";
```

number: Represents both integers and floating-point numbers.

```
let age: number = 30;
```

boolean: For true or false values.

```
let isStudent: boolean = true;
```

symbol: Used to create unique identifiers.

```
let uniqueId: symbol = Symbol("unique");
```

undefined and **null**: Represent the absence of value.

```
let data: undefined = undefined;
let value: null = null;
```

Arrays and Tuples

Arrays can store multiple values of the same type, while tuples allow you to define an ordered list of values with mixed types.

Array:

```
let scores: number[] = [85, 90, 95];
```

Tuple:

```
let person: [string, number] = ["Alice", 30];
```

Object Types and Functions

TypeScript also supports type annotations for objects and functions.

Interfaces and Inline Object Type Declarations: Interfaces define the structure and data types of objects, while inline types allow quick type definitions.

```
interface User {
    name: string;
    age: number;
    isActive: boolean;
}
let user: User = { name: "Alice", age: 30, isActive: true };
```

Function Parameter and Return Types: You can specify the types for function parameters and return values.

```
function greet(name: string): string {
    return `Hello, ${name}!`;
}
```

Enums

Enums are a way to define a set of named constants, which can be numeric or string-based.

Numeric Enums: By default, numeric enums assign sequential values starting from zero.

```
enum Direction {
    North,
    East,
    South,
    West
}
let travel: Direction = Direction.North;
```

String Enums: String enums map each constant to a specified string.

```
enum Status {
    Success = "SUCCESS",
    Failure = "FAILURE"
}
let operation: Status = Status.Success;
```

Enums are useful for switch statements and control flow to maintain readable and predictable logic.

Type Inference

Type inference is the feature that allows TypeScript to automatically infer the type of a variable if an annotation is omitted. For example:

```
let greeting = "Hello, TypeScript!"; // inferred as string
let counter = 42; // inferred as number
```

Benefits and Risks of Relying on Inference

The benefit of type inference is that it reduces verbosity, making code easier to read and write. TypeScript still ensures that variables maintain consistent types throughout their usage.

However, relying entirely on inference can be risky, especially with complex data types. When types are not explicitly stated, inferred types might not align with your expectations, potentially leading to subtle bugs. It's advisable to use explicit annotations for function parameters and return types, as well as any data structures with significant complexity.

Declaring Variables

TypeScript supports three primary ways to declare variables: `var`, `let`, and `const`, each with different behaviors concerning scope and mutability.

var: Variables declared with `var` have function scope, meaning they are visible throughout the entire function, even before their actual declaration due to hoisting. This behavior can sometimes lead to unexpected bugs and makes `var` less desirable.

```
function example() {
    var age = 30;
    if (true) {
        var age = 40; // This redeclares the 'age' variable in the
   function scope
    }
    console.log(age); // Output: 40
}
```

let: The `let` keyword declares variables with block scope, meaning their visibility is limited to the nearest enclosing block or statement. This is useful to avoid issues with `var`'s hoisting and makes it the preferred way to declare mutable variables.

```
function example() {
    let age = 30;
    if (true) {
        let age = 40; // This is a separate variable from the outer '
   age'
    }
    console.log(age); // Output: 30
}
```

const: The `const` keyword also declares block-scoped variables, but with immutability. This means that once a value is assigned to a `const` variable, it cannot be reassigned or changed.

```
const birthYear = 1990;
birthYear = 1991; // Error: Cannot reassign a constant
```

When to Use Each Declaration

- **var:** `var` should generally be avoided due to its function scoping and hoisting behavior, which can lead to confusing code.
- **let:** Use `let` for variables that need to be mutable and are scoped to a particular block.
- **const:** Use `const` for variables that will remain constant throughout their use.

Special Types and Guards

any: The `any` type disables TypeScript's strict type-checking. It's useful when working with dynamic or third-party libraries but should be used sparingly as it bypasses TypeScript's benefits.

```
let data: any = "Hello";
data = 42; // No type error, even though the value changes type
```

unknown: The `unknown` type is similar to `any` but requires explicit type checks before usage, which makes it safer. It's useful for safely handling values from external sources.

```
let input: unknown = "Some input";
if (typeof input === "string") {
    console.log(input.toUpperCase());
}
```

never: The `never` type represents code paths that cannot be reached or values that should never occur. For example, functions that always throw errors or infinite loops would return `never`.

```
function throwError(message: string): never {
    throw new Error(message);
}
```

Using Type Guards and Assertions for Error Handling

Type guards help ensure that a value is of a specific type. They can involve operators like `typeof`, `instanceof`, or custom user-defined functions.

typeof: Detects primitive types such as `string`, `number`, and `boolean`.

```
function printMessage(value: string | number) {
    if (typeof value === "string") {
        console.log(value.toUpperCase());
    } else {
        console.log(value.toFixed(2));
    }
}
```

instanceof: Checks if an object is an instance of a specific class.

```
class Car {}
let myCar = new Car();
console.log(myCar instanceof Car); // Output: true
```

Type Assertions: When TypeScript cannot infer a specific type, you can use type assertions to specify it explicitly.

```
let inputValue: unknown = "Hello TypeScript";
let trimmedValue = (inputValue as string).trim(); // Asserted as a
   string
```

Type assertions should be used carefully, as they can bypass TypeScript's type-checking and potentially introduce errors.

Functions in TypeScript

Parameter and Return Types

In TypeScript, you can specify types for function parameters and return values to ensure type consistency and catch errors early.

Required Parameters: By default, parameters are required. If you don't pass the required parameters to a function, TypeScript will generate an error.

```
function multiply(x: number, y: number): number {
    return x * y;
}
```

Optional Parameters: Optional parameters are denoted by a question mark (`?`) after the parameter name. TypeScript infers the type as the union of the declared type and `undefined`.

```
function greet(name: string, title?: string): string {
    return title ? `${title} ${name}` : `Hello, ${name}`;
}
```

Default Parameters: Default parameters have a value assigned to them if not explicitly passed.

```
function applyDiscount(price: number, discount: number = 0.1): number {
    return price * (1 - discount);
}
```

Return Type Annotations and Inference: Functions can explicitly declare the return type to ensure the function returns the correct type of value. If no return type is specified, TypeScript will infer it from the function body.

```
function add(x: number, y: number): number {
    return x + y; // Explicit return type annotation
}

function sayHello(): void {
    console.log("Hello!"); // Void indicates no return value
}
```

Function Overloads

TypeScript allows you to define multiple function signatures for a single function implementation, known as function overloads. These signatures guide TypeScript's type-checking and autocompletion.

Defining Multiple Signatures: Overload signatures are defined before the function implementation.

```
function display(value: string): void;
function display(value: number): void;
function display(value: string | number): void {
    if (typeof value === "string") {
        console.log(`String: ${value}`);
    } else {
        console.log(`Number: ${value}`);
    }
}
```

Applying Overloads in Practical Scenarios: Function overloads are particularly useful when a function can handle multiple types or scenarios, such as handling different input types in logging or formatting functions.

Arrow Functions

Arrow functions are a shorthand syntax introduced in ES6 that allows you to write more concise functions. They also handle the `this` keyword differently compared to traditional functions.

Arrow Function Syntax: Arrow functions use the `=>` syntax and are useful for simple expressions.

```
const square = (n: number): number => n * n;
```

this Keyword Behavior: Unlike traditional functions, arrow functions do not bind their own `this`. Instead, they inherit `this` from the surrounding context, known as lexical scoping.

```
class Timer {
    private seconds = 0;

    start() {
        setInterval(() => {
            this.seconds++; // Inherits 'this' from the 'Timer' class
            console.log(this.seconds);
        }, 1000);
    }
}

const timer = new Timer();
timer.start();
```

Typing Arrow Functions: Arrow functions can have parameter and return type annotations, just like regular functions.

```
const multiply = (x: number, y: number): number => x * y;
```

Using arrow functions is particularly beneficial when working with asynchronous callbacks, as their lexical scoping avoids issues with incorrect `this` references.

TypeScript Coding Examples and Exercises

Practicing with Primitives, Arrays, Objects, and Functions

Declare variables for different primitive types and log their values:

```
const age: number = 25;
const isStudent: boolean = true;
const firstName: string = "Alice";

console.log(`Age: ${age}, Student: ${isStudent}, Name: ${firstName}`);
```

Create an array of numbers and iterate through it using a loop:

```
const scores: number[] = [85, 90, 78, 92];

for (let score of scores) {
    console.log(`Score: ${score}`);
}
```

Define an object with a custom type and manipulate its properties:

```
interface Person {
    name: string;
    age: number;
    isEmployed: boolean;
}

let person: Person = { name: "Bob", age: 28, isEmployed: true };
person.age += 1; // Increment age
console.log(`Updated Age: ${person.age}`);
```

Refactoring Small JavaScript Snippets into TypeScript

Take a simple JavaScript function like this:

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

Refactor it into TypeScript:

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

The TypeScript version adds type annotations to the parameters and return value, making the function more predictable.

Mini Project

Building a Simple TypeScript Application from Scratch

Create a basic calculator application that adds and subtracts numbers:

```
// calculator.ts
interface Operation {
    type: string;
    value: number;
}

function calculate(initialValue: number, operations: Operation[]):
   number {
    let result = initialValue;

    for (let op of operations) {
        switch (op.type) {
            case "add":
                result += op.value;
                break;
            case "subtract":
                result -= op.value;
                break;
            default:
                console.log(`Unknown operation: ${op.type}`);
        }
    }

    return result;
}

const operations: Operation[] = [
    { type: "add", value: 10 },
    { type: "subtract", value: 5 }
];

const finalResult: number = calculate(0, operations);
console.log(`Final Result: ${finalResult}`);
```

In this project:

- **Interface Operation:** Defines the structure of an operation object.
- **calculate Function:** Applies multiple arithmetic operations based on the input array.
- **Switch Statement:** Implements addition and subtraction based on the operation type.