**Typescript**

**Introduction: From JavaScript to TypeScript**

We've successfully built a functional "Buy Me A Coffee" decentralized application using plain JavaScript. This application allows users to connect their wallets (like MetaMask), check the smart contract's balance, send ETH to the contract (fund), and withdraw ETH from the contract if they are the owner. It effectively interacts with our smart contract deployed on a local blockchain like Anvil, all through the browser.

While the JavaScript version works perfectly, our next step is to enhance the project's robustness and maintainability by transitioning the entire frontend codebase from JavaScript (index-js.js) to TypeScript. Although this might seem daunting, modern tools, including AI assistants, can significantly simplify the process.

**What is TypeScript and Why Use It?**

TypeScript, developed by Microsoft, is an open-source programming language that builds directly on top of JavaScript. It's often described as a **superset** of JavaScript. This means that any valid JavaScript code is also valid TypeScript code, but TypeScript adds extra capabilities.

The most significant feature TypeScript introduces is **static typing**. Unlike JavaScript, where variable types can change during runtime (dynamic typing), TypeScript allows (and encourages) you to define the expected types for variables, function parameters, and return values during development.

Why add this layer of complexity? The benefits become clear, especially in larger projects:

1. **Improved Readability & Maintainability:** Explicit types make the code's intent clearer. It's easier for you (and others) to understand what kind of data functions expect and return, making the codebase simpler to navigate and maintain over time.
2. **Early Error Detection:** TypeScript's compiler checks your code for type errors *before* you even run it in the browser (at compile time). This catches a whole class of bugs related to incorrect types that might only surface during runtime in JavaScript, often leading to subtle and hard-to-track issues. It addresses the sometimes "flimsy" feel of JavaScript by enforcing type consistency.

For a deeper dive into the motivations behind TypeScript, you can visit the official explanation: https://www.typescriptlang.org/why-create-typescript/

**The Browser Challenge: Understanding TypeScript Compilation**

There's a fundamental challenge when using TypeScript for web development: **web browsers do not understand TypeScript**. Browsers are built to execute JavaScript (.js files).

If you were to create a simple TypeScript file, say index-ts.ts, with typed code:

let myNum: number = 1;

And then try to include it directly in your index.html like this:

<!-- Incorrect: Browser cannot execute .ts files -->

<script src="./index-ts.ts" type="module"></script>

You would encounter an error in the browser's console. The error message might vary slightly, but it will typically indicate a failure to load the script due to an unexpected MIME type (like video/mp2t or similar), essentially telling you the browser doesn't recognize .ts as an executable script format.

The solution to this problem is **compilation**. The standard workflow involves:

1. **Write:** Develop your application logic using TypeScript (.ts files).
2. **Compile:** Use a special tool called the TypeScript compiler to translate your TypeScript code into equivalent JavaScript code (.js files).
3. **Use:** Include the *compiled* JavaScript file (.js) in your HTML file.

**Setting Up Your TypeScript Environment**

To compile TypeScript, you first need the TypeScript compiler (tsc). This tool runs within a NodeJS environment and is typically managed using a package manager. We'll use pnpm (Performant NPM) in this guide.

1. **Ensure NodeJS and pnpm are Installed:**
   * You need NodeJS installed on your system.
   * Check your pnpm version: pnpm --version.
   * If you don't have pnpm, follow the official installation guide: https://pnpm.io/installation.
   * *(Note for Windows Users):* Using WSL (Windows Subsystem for Linux) is generally recommended for a smoother NodeJS development experience compared to native Windows.\*
2. **Install TypeScript:**  
   Navigate to your project directory in your terminal and add TypeScript as a development dependency:

pnpm add typescript

This command performs several actions:

* + Downloads the typescript package and its dependencies.
  + Creates a node\_modules/ folder (if not already present) to store these packages locally. Previously, our JavaScript example imported viem directly from a URL (https://esm.sh/viem); now, dependencies will be managed locally.
  + Creates or updates a package.json file, which lists your project's dependencies (including typescript).
  + Creates or updates a pnpm-lock.yaml file, which locks down the specific versions of all dependencies for consistent installations.

**Compiling TypeScript Manually with**tsc

With TypeScript installed, you can now use its compiler, tsc. The compiler is accessible via your package manager runner (pnpm in this case).

To compile a single TypeScript file into its JavaScript equivalent, run:

pnpm tsc path/to/your/file.ts

For example, if we compile our simple index-ts.ts file containing let myNum: number = 1;:

pnpm tsc index-ts.ts

This command will generate a new file, index-ts.js, in the same directory. The content of index-ts.js will be the JavaScript equivalent:

// Compiled output in index-ts.js

var myNum = 1;

(Note: The exact output JavaScript might vary slightly depending on your compiler configuration, but the core principle is the conversion from typed TypeScript to standard JavaScript).

**Accelerating Conversion with AI**

Manually rewriting a substantial JavaScript codebase into TypeScript can be time-consuming. Thankfully, modern AI tools (like DeepSeek, Claude, ChatGPT, etc.) are remarkably effective at this task.

The process is straightforward:

1. **Copy:** Select and copy the entire JavaScript code from your original file (e.g., index-js.js).
2. **Prompt:** Paste the code into your chosen AI chat interface and provide a clear prompt, such as: "Rewrite this JavaScript code in TypeScript".
3. **Generate:** The AI will process the code and generate the TypeScript version, often adding type annotations automatically.

**Crucially, always review AI-generated code.** While AI provides a fantastic starting point, it's not infallible, especially in specialized domains like blockchain development where precision is paramount. Treat the AI output as a draft that needs careful verification and potential refinement.

**Refining the TypeScript Code: Types, Imports, and Constants**

After pasting the AI-generated TypeScript code into your index-ts.ts file (or writing it yourself), several adjustments are typically needed to make it fully functional and type-safe:

1. **Dependency Imports:**
   * JavaScript URL imports (import ... from "https://esm.sh/viem";) must change to package imports (import ... from "viem";).
   * This requires installing the actual package:

pnpm add viem

1. **Handling**window.ethereum**:**
   * TypeScript doesn't know about the window.ethereum object injected by browser wallets like MetaMask by default. Attempting to access it directly will cause a type error.
   * The viem library provides type definitions for this. Add the following import statement, usually near the top of your file:

import "viem/window";

* + *(Troubleshooting): If you still see errors related to window.ethereum after adding this, try restarting the TypeScript language server in your code editor (e.g., in VS Code, open the Command Palette (Ctrl+Shift+P) and search for "TypeScript: Restart TS server").*

1. **Converting Constants:**
   * If you have a separate constants file (e.g., constants-js.js containing your contract address and ABI), it also needs conversion to TypeScript (constants-ts.ts).
   * The primary change is adding the export keyword before each constant:

// constants-ts.ts (example)

export const contractAddress = "0x...";

export const abi = [/\* ... ABI array ... \*/];

* + Update the import statement in index-ts.ts to reference the new file:

import { contractAddress, abi } from "./constants-ts";

*(Note: Depending on your compiler configuration, you might import from ./constants-ts or ./constants-ts.js - the setup often handles this resolution automatically).*

1. **Adding Type Annotations:**
   * **DOM Elements:** When getting elements using document.getElementById, TypeScript only knows it's an HTMLElement or null. Use type assertions (as) to specify the exact type:

const connectButton = document.getElementById("connectButton") as HTMLButtonElement;

const ethAmountInput = document.getElementById("ethAmount") as HTMLInputElement;

* + **Viem Clients:** Declare variables for viem clients with their specific types, importing the types from viem:

import { createWalletClient, createPublicClient, custom, http, type WalletClient, type PublicClient, type Chain, type Address } from "viem";

// ...

let walletClient: WalletClient;

let publicClient: PublicClient;

* + **Function Signatures:** Specify parameter types and return types for functions. This is especially important for async functions, which return Promises:

async function connect(): Promise<void> { /\* ... \*/ }

​

async function getBalance(): Promise<void> { /\* ... \*/ }

​

async function getCurrentChain(client: WalletClient): Promise<Chain> { /\* ... \*/ }

Notice the use of imported types like WalletClient, Promise<void> (for functions that don't explicitly return a value), Chain, and Address.

**Configuring the Compiler:**tsconfig.json

While pnpm tsc index-ts.ts works for single files, compiling larger projects or using advanced TypeScript features often requires a configuration file: tsconfig.json.

If you run pnpm tsc index-ts.ts on the more complex, converted code, you might encounter errors related to missing type definitions (e.g., for Map) or language features (e.g., Promise constructor) not being recognized for your target JavaScript version.

The solution is to create a tsconfig.json file in the root of your project. You can often generate a basic one using AI or copy a standard configuration from a reliable source (like the course repository). This file tells the tsc compiler how to behave. Key options within the compilerOptions section include:

* target: Specifies the ECMAScript version to compile down to (e.g., "ES2021").
* lib: Includes standard built-in type definition libraries (e.g., ["ES2022", "DOM"] to include modern JavaScript features and browser DOM types).
* module: Defines the module system for the output JavaScript (e.g., "NodeNext" or "ESNext").
* outDir: Specifies the directory where compiled .js files should be placed (e.g., "./dist").
* strict: Enables a suite of strict type-checking options (highly recommended: true).
* include: An array of patterns specifying which .ts files the compiler should process (e.g., ["\*.ts"]).

With a tsconfig.json in place, you can often just run pnpm tsc without specifying input files, and the compiler will use the configuration file to find and compile all relevant TypeScript files.

**Streamlining Development with Vite**

Manually running pnpm tsc after every code change becomes tedious during active development. Modern frontend build tools automate this process, providing a much smoother workflow. **Vite** is a popular, fast, and modern option.

1. **Install Vite:**

pnpm add vite

1. **Run the Vite Development Server:**

pnpm vite

1. **How Vite Works:**
   * Vite starts a local development server (usually accessible at http://localhost:5173 or a similar address).
   * It analyzes your index.html file.
   * When the browser requests a .ts file linked in your HTML, Vite intercepts the request.
   * It compiles the TypeScript code to JavaScript **on the fly**, just in time for the browser.
   * Crucially, Vite provides **Hot Module Replacement (HMR)** or fast page reloads. When you save changes in your TypeScript code, Vite automatically updates the running application in the browser, often without needing a full page refresh.
2. **Update HTML:** With Vite handling the compilation, you can now directly link your main TypeScript file in index.html:

<script src="./index-ts.ts" type="module"></script>

Vite takes care of serving the correct JavaScript to the browser behind the scenes.

If you now run pnpm vite, open the provided localhost URL, and make a change (e.g., add a console.log("Live reload!")) in index-ts.ts and save, you'll see the change reflected almost instantly in the browser console, demonstrating the power of the development server.

**Next Steps and Practice**

You now have a fully functional "Buy Me A Coffee" dApp frontend written in TypeScript, benefiting from improved type safety and maintainability, along with a streamlined development workflow using Vite.

To solidify your understanding of this transition, consider these approaches:

1. **AI-Assisted Review:** Use an AI tool to generate the TypeScript code from the original JavaScript version, then carefully review and refine the output, comparing it to the final version.
2. **Study the Final Code:** Obtain the final TypeScript code (index-ts.ts, constants-ts.ts, tsconfig.json) from the project's source repository (e.g., Cyfrin/html-ts-coffee-cu on GitHub) and study how it's structured and typed.
3. **Rewrite from Scratch (Recommended):** The most effective way to learn is by doing. Try to rewrite the index-js.js file into index-ts.ts yourself, adding types and handling imports. Refer to the final code or use AI for hints only when you get stuck.

Remember the adage: repetition is the mother of skill. Actively engaging with the code, typing it out, and troubleshooting issues is the best path to mastering TypeScript in web3 development.