TypeScript Introduction - Part1

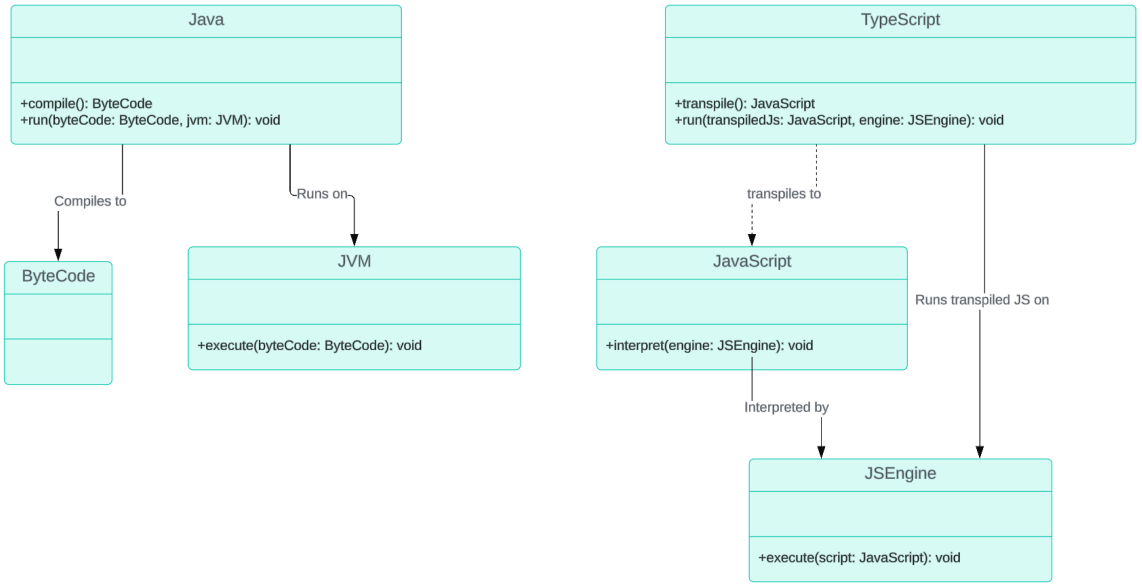
Baraneetharan Ramasamy

What is TypeScript?

TypeScript is a strongly-typed programming language that builds on JavaScript, giving you better tooling at any scale.

Java vs Javascript vs Typescript. How it runs

- **Java** is compiled and runs on the JVM.
- **JavaScript** is interpreted directly by browsers or Node.js.
- **TypeScript** is transpiled to JavaScript before execution



Key Differences Between JavaScript and TypeScript

The main difference between TypeScript and JavaScript is typing, as JavaScript has dynamic typing and TypeScript has static typing. However, we have covered some more differences between them in the table below.

Feature	JavaScript	TypeScript
Typing	Dynamic typing	Static typing
Compilation	Interpreted by browsers/Node.js	Compiled into JavaScript
Error Detection	Runtime errors	Compile-time errors
Tooling Support	Basic	Advanced (autocompletion, refactoring, etc.)
Prototypal Inheritance	Uses prototypes	Supports classes and classical inheritance
Use Cases	Small to medium projects, quick prototyping	Large projects, complex applications
Code Maintainability	Can be harder in large codebases	Easier due to static typing and interfaces
Interfaces	Not natively supported	Supported, and improved code structure
Type Inference	Not available	Available, reduces the need for explicit types

Feature	JavaScript	TypeScript
Access Modifiers	Not supported	Supports private, public, and protected modifiers
Asynchronous Programming	Callbacks, Promises, async/await	Same as JavaScript, with type safety

Why TypeScript?

Static Typing: One of the key features of TypeScript is its static typing system. It allows for catching potential errors and bugs at compile time, before the code even runs. This can improve code quality and maintainability, especially in larger projects.

Tooling and Editor Support: TypeScript provides excellent tooling support, including static code analysis, intelligent code completion, and refactoring tools. This improves developer productivity and makes it easier to work with complex codebases.

Compatibility with JavaScript: TypeScript is a superset of JavaScript, which means that any valid JavaScript code is also valid TypeScript code. This allows for easy adoption and integration with existing JavaScript codebases.

Scalability: TypeScript is designed to scale and handle large codebases effectively. It provides better organization, structure, and maintainability, making it easier to manage complex applications.

Community and Ecosystem: TypeScript has a large and active community, which means there are plenty of resources, libraries, and frameworks available to help with your projects.

When to use TypeScript?

Large-scale Projects: TypeScript is particularly beneficial for large-scale projects with multiple developers. The static typing and tooling support help catch errors early on and improve collaboration.

Complex Applications: If you're working on applications with complex logic and interactions, TypeScript can provide better structure and organization,

making the code easier to understand and maintain.

When Type Safety is Important: In projects where type safety is critical, such as financial applications or mission-critical systems, TypeScript can help prevent runtime errors and improve reliability.

Existing JavaScript Codebases: If you have an existing JavaScript codebase and want to add type checking, TypeScript is a seamless choice as it doesn't require rewriting your code from scratch.

Learning and Teaching: TypeScript can also be useful for learning and teaching purposes, as it helps developers understand type systems and improves their overall coding skills.

Setting up a TypeScript Development Environment

Step 1: Install Node.js and npm

Before installing TypeScript, you need to have Node.js and its package manager, npm (Node Package Manager), installed on your computer. You can download the latest version of Node.js from the official Node.js website (https://nodejs.org). Node.js comes bundled with npm, so you'll get both installed at once.

Step 2: Install TypeScript

Open a terminal or command prompt and run the following command to install TypeScript globally on your system:

```
npm install -g typescript
```

The "-g" flag installs TypeScript globally, allowing you to access the "tsc" (TypeScript Compiler) command from any project folder.

Step 3: Create a New Project

Create a new folder for your TypeScript project. You can use a code editor or IDE of your choice, such as Visual Studio Code, which has excellent support for TypeScript.

Step 4: Initialize a TypeScript Project

Navigate to your project folder in the terminal and run the following command:

```
tsc --init
```

This command will create a "tsconfig.json" file in your project folder. This file contains the configuration settings for your TypeScript project, including compiler options and include/exclude patterns for files.

Step 5: Write Your First TypeScript Code

Create a new file with a ".ts" extension, such as "index.ts," in your project folder. You can use a basic "Hello, World!" program to start:

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

const message = greet("Alice");
console.log(message);
```

Step 6: Compile TypeScript to JavaScript

In your terminal, run the following command to compile your TypeScript code into JavaScript:

```
tsc
```

This command will invoke the TypeScript compiler, which will read your ".ts" files, perform type checking, and generate equivalent JavaScript code in a ".js"

file.

Step 7: Run Your JavaScript Code

After compilation, you'll find the generated JavaScript file (e.g., "index.js") in your project folder. You can run this JavaScript code using Node.js:

```
node index.js
```

You should see the output of your program, such as "Hello, Alice!" in this example.

Step 8: Configure Your Editor

If you're using a code editor like VS Code, you can take advantage of its built-in TypeScript support. Install the "TypeScript Extension for VS Code" if it's not already installed. This extension provides syntax highlighting, type checking, and code suggestions within your editor.

Step 9: Explore tsconfig.json

Open the "tsconfig.json" file and explore the various compiler options. You can configure settings like target ECMAScript version, module system, include/exclude patterns, and more. Adjust these settings according to your project's needs.

Step 10: Integrate with Build Systems

If you're using a build system like Webpack or Gulp, you can integrate TypeScript compilation into your build process. These tools have plugins or configurations that allow you to compile TypeScript alongside other assets.

tsconfig.json file

Projects

- `incremental`: Save `.tsbuildinfo` files to allow for incremental compilation of projects.
- `composite`: Enable constraints that allow a TypeScript project to be used with project references.
- `tsBuildInfoFile`: Specify the path to `.tsbuildinfo` incremental compilation file.

- `disableSourceOfProjectReferenceRedirect`: Disable preferring source files instead of declaration files when referencing composite projects.
- `disableSolutionSearching`: Opt a project out of multi-project reference checking when editing.
- `disableReferencedProjectLoad`: Reduce the number of projects loaded automatically by TypeScript.

Language and Environment

- `target`: Set the JavaScript language version for emitted JavaScript and include compatible library declarations.
- `lib`: Specify a set of bundled library declaration files that describe the target runtime environment.
- `jsx`: Specify what JSX code is generated.
- `experimentalDecorators`: Enable experimental support for legacy experimental decorators.
- `emitDecoratorMetadata`: Emit design-type metadata for decorated declarations in source files.
- `jsxFactory`: Specify the JSX factory function used when targeting React JSX emit.
- `jsxFragmentFactory`: Specify the JSX Fragment reference used for fragments when targeting React JSX emit.
- `jsxImportSource`: Specify module specifier used to import the JSX factory functions when using 'jsx: react-jsx*'.
- `reactNamespace`: Specify the object invoked for 'createElement' when targeting React JSX emit.
- `noLib`: Disable including any library files, including the default lib.d.ts.
- `useDefineForClassFields`: Emit ECMAScript-standard-compliant class fields.
- `moduleDetection`: Control what method is used to detect module-format JS files.

Modules

- `module`: Specify what module code is generated.
- `rootDir`: Specify the root folder within your source files.

- `moduleResolution`: Specify how TypeScript looks up a file from a given module specifier.
- `baseUrl`: Specify the base directory to resolve non-relative module names.
- `paths`: Specify a set of entries that re-map imports to additional lookup locations.
- `rootDirs`: Allow multiple folders to be treated as one when resolving modules.
- `typeRoots`: Specify multiple folders that act like './node_modules/@types'.
- `types`: Specify type package names to be included without being referenced in a source file.
- `allowUmdGlobalAccess`: Allow accessing UMD globals from modules.
- `moduleSuffixes`: List of file name suffixes to search when resolving a module.
- `allowImportingTsExtensions`: Allow imports to include TypeScript file extensions.
- `resolvePackageJsonExports`: Use the package.json 'exports' field when resolving package imports.
- `resolvePackageJsonImports`: Use the package.json 'imports' field when resolving imports.
- `customConditions`: Conditions to set in addition to the resolver-specific defaults when resolving imports.
- `resolveJsonModule`: Enable importing.json files.
- `allowArbitraryExtensions`: Enable importing files with any extension, provided a declaration file is present.
- `noResolve`: Disallow 'import's,'require's or '<reference>'s from expanding the number of files TypeScript should add to a project.

JavaScript Support

- `allowJs`: Allow JavaScript files to be a part of your program.
- `checkJs`: Enable error reporting in type-checked JavaScript files.
- `maxNodeModuleJsDepth`: Specify the maximum folder depth used for checking JavaScript files from 'node_modules'.

Emit

- `declaration`: Generate.d.ts files from TypeScript and JavaScript files in your project.
- `declarationMap`: Create sourcemaps for d.ts files.
- `emitDeclarationOnly`: Only output d.ts files and not JavaScript files.
- `sourceMap`: Create source map files for emitted JavaScript files.
- `inlineSourceMap`: Include sourcemap files inside the emitted JavaScript.
- `outFile`: Specify a file that bundles all outputs into one JavaScript file.
- `outDir`: Specify an output folder for all emitted files.
- `removeComments`: Disable emitting comments.
- `noEmit`: Disable emitting files from a compilation.
- `importHelpers`: Allow importing helper functions from tslib once per project.
- `downlevelIteration`: Emit more compliant, but verbose and less performant JavaScript for iteration.
- `sourceRoot`: Specify the root path for debuggers to find the reference source code.
- `mapRoot`: Specify the location where debugger should locate map files instead of generated locations.
- `inlineSources`: Include source code in the sourcemaps inside the emitted JavaScript.
- `emitBOM`: Emit a UTF-8 Byte Order Mark (BOM) in the beginning of output files.
- `newLine`: Set the newline character for emitting files.
- `stripInternal`: Disable emitting declarations that have '@internal' in their JSDoc comments.
- `noEmitHelpers`: Disable generating custom helper functions like '__extends' in compiled output.
- `noEmitOnError`: Disable emitting files if any type checking errors are reported.
- `preserveConstEnums`: Disable erasing 'const enum' declarations in generated code.
- `declarationDir`: Specify the output directory for generated declaration files.

Interop Constraints

- `isolatedModules`: Ensure that each file can be safely transpiled without relying on other imports.
- `verbatimModuleSyntax`: Do not transform or elide any imports or exports not marked as type-only.
- `isolatedDeclarations`: Require sufficient annotation on exports so other tools can trivially generate declaration files.
- `allowSyntheticDefaultImports`: Allow 'import x from y' when a module doesn't have a default export.
- `esModuleInterop`: Emit additional JavaScript to ease support for importing CommonJS modules.
- `preserveSymlinks`: Disable resolving symlinks to their realpath.
- `forceConsistentCasingInFileNames`: Ensure that casing is correct in imports.

Type Checking

- `strict`: Enable all strict type-checking options.
- `noImplicitAny`: Enable error reporting for expressions and declarations with an implied 'any' type.
- `strictNullChecks`: When type checking, take into account 'null' and 'undefined'.
- `strictFunctionTypes`: When assigning functions, check to ensure parameters and the return values are subtype-compatible.
- `strictBindCallApply`: Check that the arguments for 'bind', 'call', and 'apply' methods match the original function.
- `strictPropertyInitialization`: Check for class properties that are declared but not set in the constructor.
- `noImplicitThis`: Enable error reporting when 'this' is given the type 'any'.
- `useUnknownInCatchVariables`: Default catch clause variables as 'unknown' instead of 'any'.
- `alwaysStrict`: Ensure 'use strict' is always emitted.
- `noUnusedLocals`: Enable error reporting when local variables aren't read.
- `noUnusedParameters`: Raise an error when a function parameter isn't read.

- `exactOptionalPropertyTypes`: Interpret optional property types as written, rather than adding 'undefined'.
- `noImplicitReturns`: Enable error reporting for codepaths that do not explicitly return in a function.
- `noFallthroughCasesInSwitch`: Enable error reporting for fallthrough cases in switch statements.
- `noUncheckedIndexedAccess`: Add 'undefined' to a type when accessed using an index.
- `noImplicitOverride`: Ensure overriding members in derived classes are marked with an override modifier.
- `noPropertyAccessFromIndexSignature`: Enforces using indexed accessors for keys declared using an indexed type.
- `allowUnusedLabels`: Disable error reporting for unused labels.
- `allowUnreachableCode`: Disable error reporting for unreachable code.

Completeness

- `skipDefaultLibCheck`: Skip type checking.d.ts files that are included with TypeScript.
- `skipLibCheck`: Skip type checking all.d.ts files.

TS Node

TS Node (TypeScript Node) is a package that allows you to run TypeScript files directly with Node.js, without the need to compile them to JavaScript files first.

TS Node is a wrapper around the Node.js runtime that allows you to execute TypeScript files as if they were JavaScript files. It uses the TypeScript compiler to compile the TypeScript code on the fly, and then runs the resulting JavaScript code using Node.js.

Here are some benefits of using TS Node:

1. **No need to compile TypeScript files**: With TS Node, you don't need to compile your TypeScript files to JavaScript files before running them. This saves you an extra step in your development workflow.

2. **Faster development cycle**: Since TS Node compiles your TypeScript code on the fly, you can make changes to your code and see the results immediately, without having to recompile your code.
3. **Better error reporting**: TS Node provides better error reporting than the standard Node.js runtime, since it uses the TypeScript compiler to analyze your code.

To use TS Node, you need to install it as a package using npm or yarn:

```
npm install -g ts-node
```

Once installed, you can run your TypeScript files using the `ts-node` command:

```
ts-node your_file.ts
```

This will run the `your_file.ts` file as if it were a JavaScript file, using the Node.js runtime.

TS Node also supports many of the same command-line options as Node.js, such as `-r` for requiring modules, `-e` for evaluating expressions, and `-p` for printing the result of an expression.

Here are some examples of using TS Node:

```
ts-node -r tsconfig/register your_file.ts
ts-node -e "console.log('Hello, World!')"
ts-node -p "require('./your_file').default"
```

Overall, TS Node is a convenient and powerful tool for developing and running TypeScript applications with Node.js.

TypeScript types

TypeScript types! This is a fundamental concept in TypeScript, and understanding types is crucial to getting the most out of the language.

What are types in TypeScript?

In TypeScript, a type represents the structure of a value, including the type of its properties, methods, and behavior. Types are used to define the shape of an

object, function, or variable, and to ensure that the code is correct and safe at compile-time.

Types in TypeScript can be categorized into several groups:

1. **Primitive types**: These are the basic building blocks of TypeScript, and include:
   - `number`: A numeric value, such as 42 or 3.14.
   - `string`: A sequence of characters, such as "hello" or 'hello'.
   - `boolean`: A true or false value.
   - `null`: A null value.
   - `undefined`: An undefined value.
2. **Object types**: These represent complex data structures, such as:
   - `object`: A generic object type, which can have any properties.
   - `array`: An array of values, such as `number[]` or `string[]`.
   - `tuple`: A fixed-length array of values, such as `[number, string]`.
3. **Function types**: These represent functions, including:
   - `function`: A generic function type, which can take any arguments and return any value.
   - `() => void`: A function that takes no arguments and returns no value.
   - `(x: number) => string`: A function that takes a single `number` argument and returns a `string` value.
4. **Union types**: These represent a value that can be one of several types, such as:
   - `number | string`: A value that can be either a `number` or a `string`.
   - `boolean | null`: A value that can be either a `boolean` or `null`.
5. **Intersection types**: These represent a value that must satisfy multiple types, such as:
   - `{ x: number } & { y: string }`: An object that must have both an `x` property of type `number` and a `y` property of type `string`.
6. **Literal types**: These represent a specific value, such as:
   - `42`: A literal type representing the value 42.

- `"hello"`: A literal type representing the string "hello".

7. **Enum types**: These represent a set of named values, such as:
   - `enum Color { Red, Green, Blue }`: An enum type representing the values Red, Green, and Blue.
8. **Type aliases**: These are aliases for existing types, such as:
   - `type StringOrNumber = string | number;`: A type alias for the union type `string | number`.

How to use types in TypeScript:

1. **Type annotations**: You can add type annotations to your code using the `:` syntax, such as:

```
let x: number = 42;
```

2. **Type inference**: TypeScript can often infer the types of variables and function parameters automatically, without the need for explicit type annotations.
3. **Interfaces**: You can define interfaces to describe the shape of an object, such as:

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

4. **Class**: You can define class to describe the shape of an object, such as:

```
class Student
{
    RollNo: number;
    Name: string;
    constructor(_rollNo: number, _name: string)
    {
        this.RollNo = _rollNo;
        this.Name = _name;
```

```
    }
    showDetails()
    {
        console.log(this.RollNo + " : " + this.Name);
    }
}

// Create an instance of the Student class
let student = new Student(1, "Alice");

// Call the showDetails() method on the student instance
student.showDetails();
```

5. **Type guards**: You can use type guards to narrow the type of a value within a specific scope, such as:

```
if (typeof x ==='string') {
  // x is now known to be a string
}
```

6. **Castings**: You can use castings to explicitly convert a value from one type to another, such as:

```
let x: any = 'hello';
let y: string = x as string;
```

This is just a brief introduction to the world of TypeScript types. There's much more to explore, but I hope this gives you a solid foundation to build upon!

Variables in TypeScript

Variable Declaration Syntax & Description
var name:string = ”mary” The variable stores a value of type string
var name:string; The variable is a string variable. The variable’s value is set to undefined by default

Variable Declaration Syntax & Description
var name = ”mary” The variable’s type is inferred from the data type of the value. Here, the variable is of the type string
var name; The variable’s data type is any. Its value is set to undefined by default.

```
var name1:string = "John";
var score1:number = 50;
var score2:number = 42.50
var sum = score1 + score2

console.log("name"+name1)
console.log("first score: "+score1)
console.log("second score: "+score2)
console.log("sum of the scores: "+sum)
```

Variable Declaration in TypeScript: "var", "let", and "const”

In TypeScript, "var", "let", and "const" are keywords used to declare variables, and they have different characteristics and use cases. Here's a lesson on how to use each of them:

Lesson: Using "var", "let", and "const" in TypeScript

1. "var"

"var" is the traditional way to declare variables in JavaScript.

Variables declared with "var" are function-scoped or globally scoped if declared outside of any function.

"var" allows variable redeclaration and reassignment.

Example:

```
function exampleVar() {
var x = 10; // Function-scoped
if (true) {
var x = 5; // Valid, redeclares and shadows the previous x
console.log(x); // Outputs: 5
```

```
}
console.log(x); // Outputs: 10 Why?
}
exampleVar();
// 5
// 5

// var x = 10; // Function-scoped
// function exampleVar() {
//     if (true) {
//         var x = 5; // Valid, redeclares and shadows the pr
//         console.log(x); // Outputs: 5
//     }
// }

// exampleVar();
// console.log(x); // Outputs: 10 Why?

// 5
// 10
```

2. "let"

"let" is a newer keyword introduced in ES6 (ECMAScript 2015) and supported in TypeScript.

Variables declared with "let" are block-scoped, meaning they are limited to the block ({}) they are declared in.

"let" allows reassignment but not redeclaration.

Example:

```
function exampleLet() {
let y = 20; // Block-scoped
if (true) {
let y = 15; // Valid, different block scope
console.log(y); // Outputs: 15
}
console.log(y); // Outputs: 20
```

```
}
exampleLet();
```

3. "const"

"const" is also introduced in ES6 and supported in TypeScript.

Variables declared with "const" are block-scoped.

"const" variables must be initialized during declaration and cannot be reassigned. However, mutable objects (like arrays and objects) declared with "const" can have their properties modified.

Example:

```
function exampleConst() {
const z = 30; // Block-scoped and constant
z = 25; // Error: Assignment to constant variable
const colors = ["red", "green"];
colors.push("blue"); // Allowed, modifies the array's propert
console.log(z); // Outputs: 30
console.log(colors); // Outputs: ['red', 'green', 'blue']
}
exampleConst();
```

Summary:

Use "**var**" for function-scoped or global variables, but be aware of its limitations, such as variable hoisting and lack of block scoping.
Prefer "
let" for most variable declarations, as it provides block scoping and prevents accidental redeclarations.
Use "
const" for variables that should not be reassigned, ensuring immutability and helping catch potential errors.