Week 5: Introduction to TypeScript Basics

FH TECHNIKUM WIEN
University of Applied Sciences

Week 5: Introduction to TypeScript Basics

This week we'll cover:

- What is TypeScript?
- Basic Types in TypeScript
- Using Interfaces and Objects
- Setting up TypeScript in a Project
- Converting JavaScript Code to TypeScript

What is TypeScript?

- **Definition**: TypeScript is a strongly typed superset of JavaScript.
  - Adds **static typing** to JavaScript.
  - Compiles to plain JavaScript, running in any JavaScript environment.

Why Use TypeScript?

1. **Error Prevention**: Type safety helps catch errors early.
2. **Improved Readability**: Type annotations make code more understandable.
3. **Enhanced Tooling**: Autocompletion, code navigation, and error checking in IDEs.

TypeScript vs. JavaScript

- **JavaScript**: Dynamically typed, no type annotations.
- **TypeScript**: Statically typed, types are checked at compile-time.

Example Comparison:

```
// JavaScript
let message = "Hello, JavaScript!";
message = 42; // Allowed in JavaScript
```

```
// TypeScript
let message: string = "Hello, TypeScript!";
message = 42; // Error: Type 'number' is not assignable to type 'string'
```

Explanation: TypeScript enforces type safety, preventing `message` from being assigned a different type.

Transitioning from JavaScript to TypeScript

Similarities and Main Differences

Similarities:

1. **Syntax**: TypeScript and JavaScript share the same core syntax. JavaScript code is valid TypeScript code (unless types are enforced).
2. **JS Functions and Classes**: TypeScript uses JavaScript functions, classes, and ES6+ features in the same way.
3. **Event Handling and DOM Manipulation**: Both languages handle events and the DOM similarly, but TypeScript enforces types for better error handling.

Main Differences:

1. **Type Annotations**:
    - TypeScript adds optional **type annotations** (`number`, `string`, etc.) to variables, function parameters, and return values.
    - TypeScript enforces **strict type-checking**, reducing runtime errors.
2. **Interfaces and Types**:
    - TypeScript introduces **interfaces** and **types** to define the shape and structure of objects.
    - These features make it easier to manage large codebases and ensure consistent object structures.
3. **Compile-Time Error Checking**:
    - TypeScript detects errors during **compilation**, helping developers catch issues early.
    - JavaScript, on the other hand, only shows errors at runtime.

How to Level Up from JavaScript to TypeScript

Easy Steps to Start with TypeScript

1. **Add TypeScript Gradually**:
    - Start with basic **type annotations** on variables and function parameters.
    - Slowly introduce **interfaces** for complex objects as you become more comfortable.
2. **Use `any` as a Temporary Solution**:
    - If you're unsure about a type, use `any` initially, then refine as you learn more about TypeScript's type system.
3. **Refactor Existing JavaScript Code**:
    - Begin by converting small parts of your JavaScript project to TypeScript.
    - Start with converting functions and classes to TypeScript, and add types as you go.

4. **Leverage TypeScript's IDE Support**:
    - Use an editor like **Visual Studio Code**, which has excellent TypeScript support with autocompletion, type hints, and inline error checking.
5. **Experiment with `tsc --watch`**:
    - Use `tsc --watch` to continuously compile TypeScript files as you make changes, helping you catch errors in real-time.

Practice: Try Converting Simple JavaScript Functions

```
// JavaScript version
function greet(name) {
    return "Hello, " + name;
}
```

```
// TypeScript version with type annotation
function greet(name: string): string {
    return "Hello, " + name;
}
```

Explanation: Adding a type annotation to `name` ensures that `greet` always receives a string, reducing the chance of unexpected errors.

Setting Up TypeScript

Step 1: Install TypeScript

- Install TypeScript globally:

```
npm install -g typescript
```

Step 2: Initialize a TypeScript Project

- Run the following command to create a TypeScript configuration file (`tsconfig.json`):

```
tsc --init
```

Step 3: Compile TypeScript Files

- Use `tsc` to compile a `.ts` file to JavaScript:

  ```
  tsc filename.ts
  ```

`tsconfig.json`: A file used to configure TypeScript project settings, like target ECMAScript version and compilation options.

Basic Types in TypeScript

Primitive Types

1. `string`: Represents text.

   ```
   let message: string = "Hello!";
   ```

2. `number`: Represents all numbers, both integer and floating-point.

   ```
   let count: number = 10;
   ```

3. `boolean`: Represents `true` or `false`.

   ```
   let isActive: boolean = true;
   ```

4. `any`: Disables type checking, allowing any type. Use sparingly.

   ```
   let variable: any = "Could be any type";
   ```

Arrays and Tuples

Arrays

- Define arrays with element types, e.g., `string[]` or `number[]`.
- Example:

```
let notes: string[] = ["C", "D", "E", "F"];
```

Tuples

- Tuples specify types for a fixed number of elements.
- Example:

```
let mixedTuple: [string, number] = ["Note", 5];
```

Explanation: Arrays allow flexible length, while tuples have a fixed length with specific types for each position.

Type Inference

- **Type Inference**: TypeScript can infer the type based on the assigned value, even without explicit annotation.
- Example:

```
let count = 10; // TypeScript infers 'number'
let name = "Alice"; // TypeScript infers 'string'
```

Note: Type inference is useful, but explicit types improve readability and maintainability, especially in complex codebases.

Defining Object Types with Interfaces

Interfaces

- **Interfaces** define the structure of an object, enforcing type checking on object properties.
- Example:

```
interface PianoKey {
    note: string;
    color: string;
    key: string;
}

const keyC: PianoKey = { note: "C", color: "white", key: "a" };
```

Explanation: `PianoKey` defines that each key object must have `note`, `color`, and `key` properties, all of type `string`.

Optional and Read-Only Properties

1. **Optional Properties**: Use `?` to indicate that a property is optional.

```
interface PianoKey {
    note: string;
    color: string;
    key?: string;
}
```

2. **Read-Only Properties**: Use `readonly` to prevent modifications.

```
interface PianoKey {
    readonly note: string;
    color: string;
}
```

Explanation: Optional properties are not required, while `readonly` properties cannot be changed after initialization.

Type Annotations for Functions

Function Parameter and Return Types

- TypeScript allows specifying types for function parameters and return values.
- Example:

```
function playSound(volume: number): string {
    return `Playing sound at volume ${volume}`;
}
```

Explanation: The parameter `volume` is a `number`, and the function returns a `string`. TypeScript checks for type consistency.

Type Annotations in Arrow Functions

- Type annotations can also be used with arrow functions.
- Example:

```
const multiply = (a: number, b: number): number => a * b;
```

Explanation: This arrow function takes two `number` parameters and returns a `number`.

Setting Up `tsconfig.json`

- `tsconfig.json` allows you to configure TypeScript project settings.
  - Common settings:
    - `"target"`: ECMAScript version to compile to.
    - `"outDir"`: Directory to output compiled JavaScript files.
    - `"strict"`: Enables strict type-checking.

Example `tsconfig.json` Setup

```
{
  "compilerOptions": {
    "target": "es6",
    "outDir": "./dist",
    "strict":
 true
  }
}
```

Explanation: This configuration sets the target to ES6, outputs files to `dist`, and enforces strict type-checking.

Hands-On Practice: Define Basic Types

1. **Define Variables with Explicit Types**
    - Define types for variables in the Piano Game (e.g., `volume`, `isPlaying`, `currentMelody`).

```
let volume: number = 0.5;
let isPlaying: boolean = false;
let currentMelody: string[] = ["C", "D", "E"];
```

2. **Use Type Annotations in Functions**
    - Define functions with parameter and return type annotations.

```
function playMelody(melody: string[]): void {
    melody.forEach(note => console.log(`Playing ${note}`));
}
```

Converting JavaScript Classes to TypeScript

- Refactor the `PianoKey` class to TypeScript, using type annotations and interfaces.

Define the PianoKey Interface

```
interface PianoKeyProps {
    note: string;
    color: string;
    key: string;
}
```

Convert `PianoKey` Class to TypeScript

```
class PianoKey {
    note: string;
    color: string;
    key: string;

    constructor({ note, color, key }: PianoKeyProps) {
        this.note = note;
        this.color = color;
        this.key = key;
    }

    playSound(): void {
        const audio = new Audio(`sounds/${this.note}.mp3`);
        audio.play();
    }
}

const keyE = new PianoKey({ note: "E", color: "white", key: "d" });
```

Adding Type Annotations to Event Listeners

- Add type annotations to event parameters in TypeScript.
- **Example**:

```
const volumeSlider = document.getElementById("volume-control") as HTMLInputElement;
volumeSlider.addEventListener("input", (event: Event) => {
    const target = event.target as HTMLInputElement;
    appState.volume = parseFloat(target.value);
});
```

Explanation: Adding types to DOM events improves readability and type safety in event handling.

Weekly Exercise: Refactor Piano Game to TypeScript

Goals

1. Set up TypeScript in your project using `tsconfig.json`.
2. Add type annotations for variables and functions.
3. Convert `PianoKey` class and `appState` to TypeScript.

Summary and Q&A

- **TypeScript Basics**: Understanding types, interfaces, and how to refactor JavaScript to TypeScript.
- **Benefits of TypeScript**: Type safety, code readability, and error prevention.

Q&A: Let’s clarify any questions about TypeScript basics!