Priyanshu Dhyani

TypeScript Handbook

Introduction to TypeScript

Brief Overview

TypeScript is a statically typed superset of JavaScript developed and maintained by Microsoft. It adds static typing to JavaScript, allowing developers to catch errors early in the development process and improve code quality. TypeScript code compiles down to plain JavaScript, ensuring compatibility with all JavaScript environments.

Why Use TypeScript?

TypeScript offers several advantages over plain JavaScript:

- **Static Typing:** Helps catch errors at compile time, improving code reliability and maintainability.
- **Enhanced IDE Support:** Provides better autocompletion, navigation, and refactoring capabilities in most modern IDEs.
- **Advanced Features:** Introduces features such as interfaces, enums, and generics that are not available in JavaScript.
- **Scalability**: Makes it easier to manage large codebases by providing clear contracts and modularity.

Getting Started

Installation

To install the TypeScript compiler (tsc), you need Node.js and npm (Node Package Manager) installed on your system. Once you have these prerequisites, you can install TypeScript globally using npm:
npm install -g typescript

Setting Up a New TypeScript Project

Initialize a new project:
mkdir my-typescript-project
cd my-typescript-project
npm init -y

Install TypeScript locally:
npm install --save-dev typescript

Create a tsconfig.json file to configure TypeScript:
npx tsc --init

Create your first TypeScript file:
// src/index.ts
const message: string = "Hello, TypeScript!";
console.log(message);

Priyanshu Dhyani

Compile the TypeScript file:
npx tsc

Run the compiled JavaScript file:
node dist/index.js

Integrating TypeScript with Existing JavaScript Projects
You can gradually migrate an existing JavaScript project to TypeScript by renaming .js files to .ts and fixing type errors.
Use the allowJs option in tsconfig.json to allow TypeScript to compile JavaScript files:

```
{
  "compilerOptions": {
    "allowJs": true,
    "outDir": "./dist"
  },
  "include": ["src//*"]
}
```

Basic Syntax and Types
TypeScript vs. JavaScript Syntax
TypeScript syntax is similar to JavaScript, with the addition of type annotations. Here's a comparison:

JavaScript:

```
let message = "Hello, JavaScript!";
console.log(message);
```

Typescript:

```
let message: string = "Hello, TypeScript!";
console.log(message);
```

Basic Data Types
TypeScript supports the same data types as JavaScript, with additional type annotations:

number:

```
let count: number = 42;
```

String:

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

Boolean:

```
let isDone: boolean = true;
```

null and undefined:

```
let nothing: null = null;
let notDefined: undefined = undefined;
```

Priyanshu Dhyani

Type Annotations and Type Inference
Type Annotations: Explicitly specify types.
let age: number = 25;

Type Inference: TypeScript infers the type based on the value.
let greeting = "Hello"; // Inferred as string

Static Typing
Explanation and Benefits
Static typing means variable types are known at compile time, reducing runtime errors and improving code quality. Benefits include early error detection, better code readability, and enhanced refactoring capabilities.

Declaring Variable Types
Use type annotations to declare variable types:
let isValid: boolean = true;

Type Inference
TypeScript can automatically infer types based on the assigned values:
let score = 100; // Inferred as number

Interfaces
Definition and Usage
Interfaces define the structure of an object, specifying its properties and their types:

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

let user: Person = {
  name: "John",
  age: 30
};
```

Optional and Read-Only Properties
Optional Properties: Use ? to mark properties as optional.

```
interface Car {
  model: string;
  year?: number;
}

let myCar: Car = { model: "Toyota" };
```

Read-Only Properties: Use readonly to make properties immutable

```
interface Book {
  readonly title: string;
  author: string;}
```

Priyanshu Dhyani

```
let novel: Book = { title: "1984", author: "George Orwell" };
// novel.title = "Animal Farm"; // Error: Cannot assign to 'title' because it is a read-only property.
```

Classes
Object-Oriented Programming Concepts
TypeScript supports object-oriented programming with classes, inheritance, and access modifiers.

Defining Classes
Define classes with properties and methods:

```
class Animal {
  name: string;

  constructor(name: string) {
    this.name = name;
  }

  speak(): void {
    console.log(${this.name} makes a noise.);
  }
}

let dog = new Animal("Dog");
dog.speak(); // Dog makes a noise.
```

Constructors and Access Modifiers
Constructors: Initialize class properties.
Access Modifiers: Control access to properties and methods (public, private, protected).

```
class Person {
  private id: number;
  public name: string;
  protected age: number;

  constructor(id: number, name: string, age: number) {
    this.id = id;
    this.name = name;
    this.age = age;
  }

  public getDetails(): string {
    return ID: ${this.id}, Name: ${this.name}, Age: ${this.age};
  }
}

let person = new Person(1, "Alice", 30);
console.log(person.getDetails()); // ID: 1, Name: Alice, Age: 30
```

Priyanshu Dhyani

Inheritance and Method Overriding
Extend classes using inheritance and override methods:

```
class Animal {
  name: string;

  constructor(name: string) {
    this.name = name;
  }

  speak(): void {
    console.log(${this.name} makes a noise.);
  }
}

class Dog extends Animal {
  constructor(name: string) {
    super(name);
  }

  speak(): void {
    console.log(${this.name} barks.);
  }
}

let dog = new Dog("Dog");
dog.speak(); // Dog barks.
```

Generics
Introduction to Generics
Generics allow you to create reusable components that work with any data type:

```
function identity<T>(arg: T): T {
  return arg;
}

let output = identity<string>("Hello");
```

Creating Reusable Components
Use generics to create flexible and reusable components:

```
class Box<T> {
  contents: T;

  constructor(contents: T) {
    this.contents = contents;
  }

  getContents(): T {
```

```
    return this.contents;
  }
}

let box = new Box<string>("Books");
console.log(box.getContents()); // Books
```

Generic Constraints

Use constraints to restrict generic types:

```
interface Lengthwise {
  length: number;
}

function logLength<T extends Lengthwise>(arg: T): void {
  console.log(arg.length);
}

logLength({ length: 10, value: "Hello" }); // 10
```

Advanced TypeScript Concepts

Union and Intersection Types

Union Types: A value that can be one of several types.

```
let value: string | number;
value = "Hello";
value = 42;
```

Intersection Types: Combine multiple types into one.

```
interface ErrorDetails {
  message: string;
}

interface Logging {
  log(): void;
}

type LoggableError = ErrorDetails & Logging;

let error: LoggableError = {
  message: "An error occurred",
  log() {
    console.log(this.message);
  }
};

error.log(); // An error occurred
```

Priyanshu Dhyani

Type Aliases and Type Assertions
Type Aliases: Create custom type names.
type ID = string | number;
let userId: ID;
userId = 123;
userId = "ABC";

Type Assertions: Tell TypeScript the specific type of a value.
```
let someValue: any = "Hello";
let strLength: number = (someValue as string).length;
```

Type Guards
Use type guards to work with union types:
```
function isString(value: any): value is string {
  return typeof value === "string";
}

function printValue(value: string | number): void {
  if (isString(value)) {
    console.log(String value: ${value});
  } else {
    console.log(Number value: ${value});
  }
}

printValue("Hello"); // String value: Hello
printValue(42); // Number value: 42
```

Conditional Types and Mapped Types
Conditional Types: Types that depend on a condition.
```
type NonNullable<T> = T extends null | undefined ? never : T;

type Example = NonNullable<string | number | null>; // string | number
```

Mapped Types: Create new types by transforming properties.
```
type Readonly<T> = {
  readonly [P in keyof T]: T[P];
};

interface User {
  name: string;
  age: number;
}

type ReadonlyUser = Readonly<User>;
```