Introduction to TypeScript

What is TypeScript?

TypeScript is a superset of JavaScript, meaning it includes all the features of JavaScript along with additional functionalities like static typing. Static typing allows you to define the data types of variables and functions before the code is run. This helps catch potential errors early in the development process, leading to more reliable and maintainable code.

Why use TypeScript?

While JavaScript is a powerful language, it can be prone to errors due to its dynamic typing nature. TypeScript addresses this by offering several advantages:

Improved Code Clarity: Defining variable and function types makes your code more readable and understandable for yourself and other developers.

Early Error Detection: Static typing helps identify potential type-related errors during compilation, preventing runtime issues.

Better Refactoring Support: Refactoring code becomes easier with type information readily available.

Larger Codebase Management: TypeScript scales well for large projects with its ability to enforce consistency and prevent type mismatches.

Getting Started with TypeScript

Installation:

To use TypeScript, you need the TypeScript compiler (tsc). You can install it globally using npm or yarn:

Bash

```
npm install -g typescript
```

Setting Up a New Project:

For a new project, you can create a tsconfig.json file to configure the compiler options. This file is optional but provides granular control over how TypeScript compiles your code.

Integrating with Existing Projects:

You can gradually introduce TypeScript into an existing JavaScript project by converting files one at a time. TypeScript supports existing JavaScript code without any modifications.

Basic Syntax and Types

TypeScript Syntax:

TypeScript shares a large part of its syntax with JavaScript. However, it adds features for defining types:

TypeScript

```
let name: string = "John"; // Type annotation for string
```

```
let age = 30; // Type inferred from assignment
```

Basic Data Types:

TypeScript includes primitive data types similar to JavaScript:

number: Numeric values

string: Textual data

boolean: True or false values

null and undefined: Represent the absence of a value

Type Annotations and Inference:

You can explicitly define types for variables using type annotations. However, TypeScript can also infer types automatically based on the assigned value.

Static Typing

Benefits of Static Typing:

Static typing offers several benefits:

Early Error Detection: Type mismatches are caught during compilation, preventing runtime errors.

Improved Code Intelligence: IDEs can provide better code completion and suggestions based on type information.

Improved Documentation: Types act as documentation, clarifying expected data types throughout your code.

Declaring Variable Types:

You can declare variable types using type annotations:

TypeScript

```
let age: number = 30;
let isStudent: boolean = true;
```

Type Inference:

If you don't explicitly define a type, TypeScript can infer it based on the assigned value:

TypeScript

```
let message = "Hello World";  // Type inferred as string
```

Interfaces

What are Interfaces?

Interfaces define the structure of an object in TypeScript. They act as contracts outlining the expected properties and their types.

Using Interfaces:

TypeScript

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

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

Optional and Read-Only Properties:

Interfaces can have optional and read-only properties:

TypeScript

```
interface Product {
  name: string;
  price: number;
  readonly description?: string;  // Optional read-only property
}
```

Classes

Object-Oriented Programming (OOP) in TypeScript

TypeScript supports object-oriented programming concepts like classes. Classes are blueprints for creating objects with properties and methods.

Defining Classes:

TypeScript

```
class User {
 name: string;
 age: number;

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

 greet() {
  console.log("Hello, my name is " + this.name);
 }
}

let user1 = new User("Alice", 25);
user1.greet();
```

Constructors and Access Modifiers:

Constructors initialize objects when created. Access modifiers (public, private, protected) control access to properties and methods within classes and subclasses.

Inheritance and Overriding:

Classes can inherit properties and methods from parent classes using the `