Introduction to TypeScript

**What is TypeScript?**

TypeScript is a strongly typed superset of JavaScript developed by Microsoft. It extends JavaScript by adding static types, which helps developers catch errors during development rather than at runtime. TypeScript code is transpiled into plain JavaScript, which runs in any JavaScript environment.

**Why Use TypeScript with React?**

1. **Static Typing**: Detect errors during development, such as passing incorrect props to components.
2. **Improved Code Readability**: Define clear contracts (interfaces and types) for your components and data structures.
3. **Enhanced Tooling**: Better IntelliSense, autocompletion, and refactoring in IDEs.
4. **Scalability**: Makes managing large codebases easier by reducing the risk of type-related bugs.

**Key Advantages of TypeScript**

1. **Error Prevention**:
   * Example:
   * const add = (a: number, b: number): number => {
   * return a + b;
   * };
   * add(5, "10"); // Error: Argument of type 'string' is not assignable to parameter of type 'number'.
   * In JavaScript, this would fail silently or cause runtime errors.
2. **Improved Developer Experience**:
   * Example:
   * const greet = (name: string): string => `Hello, ${name}!`;

IDEs can autocomplete name and show what the function returns.

1. **Documentation with Types**:
   * The type definitions act as self-documenting code.

**JavaScript vs. TypeScript: A Comparison**

|  |  |  |
| --- | --- | --- |
| Feature | JavaScript | TypeScript |
| Typing | Dynamic typing (no type enforcement). | Static typing (types enforced during dev). |
| Tooling Support | Basic. | Advanced (IntelliSense, autocomplete). |
| Error Detection | Errors show up during runtime. | Errors caught during development. |
| Code Scalability | Harder to scale as the app grows. | Easier to manage in larger projects. |

**Code Examples**

1. **JavaScript Code**:

function multiply(a, b) {

return a \* b;

}

console.log(multiply(5, "10")); // NaN - bug not caught at compile time.

1. **TypeScript Code**:

function multiply(a: number, b: number): number {

return a \* b;

}

// multiply(5, "10"); // Error: Argument of type 'string' is not assignable to parameter of type 'number'.

console.log(multiply(5, 10)); // 50

**Take Away**

* TypeScript enforces rules (types) that JavaScript doesn't.
* It improves reliability and makes your code predictable.
* Combined with React, TypeScript helps type-check props, state, and events, making UI development safer and more efficient.

**TypeScript Basics**

#### ****Getting Started: How to Run TypeScript in VS Code****

1. **Install TypeScript Compiler**:
   * Open a terminal and install TypeScript globally:

npm install -g typescript

* + Verify installation:

tsc --version

1. **Set Up a TypeScript Project**:
   * Create a new project folder and open it in VS Code.
   * Initialize a tsconfig.json file:

tsc --init

* + This file allows you to configure TypeScript settings.

1. **Write and Compile TypeScript**:
   * Create a file named example.ts and add the following code:

const greet = (name: string): string => `Hello, ${name}!`;

console.log(greet("World"));

* + Compile the file to JavaScript:

tsc example.ts

* + Run the generated example.js file:

node example.js

1. **Enable TypeScript Auto-Compilation**:
   * Run:
   * tsc --watch
   * This will automatically compile .ts files when you save changes.

#### ****Key Concepts****

##### **1. Type Annotations**

Type annotations explicitly define the type of a variable or function.

* **Basic Types**:

let age: number = 25;

let name: string = "Dinesh";

let isStudent: boolean = true;

* **Special Types**:
  + **any**: Disables type checking (use sparingly):

let data: any = 42;

data = "Now a string"; // No error

* + **unknown**: Safer alternative to any:

let input: unknown = "Hello";

if (typeof input === "string") {

console.log(input.toUpperCase()); // Safe

}

**void**: For functions that do not return anything:

const logMessage = (message: string): void => {

console.log(message);

* + };

##### **2. Arrays and Tuples**

**Arrays**:

let numbers: number[] = [1, 2, 3];

let strings: string[] = ["one", "two", "three"];

**Tuples**:

let tuple: [string, number] = ["Dinesh", 25];

##### **3. Interfaces vs. Types**

* **Interfaces**: Used to define the shape of objects.

interface User {

id: number;

name: string;

}

const user: User = {

id: 1,

name: "Dineshkumar",

};

* **Types**: Similar to interfaces but more flexible.

type Product = {

id: number;

price: number;

};

const product: Product = {

id: 101,

price: 25.5,

};

* **Key Difference**: Interfaces can be extended; types are used for more complex constructs like unions.

##### **4. Union and Intersection Types**

* **Union**: A variable can hold one of multiple types.

type ID = string | number;

let userId: ID = 123;

userId = "ABC";

* **Intersection**: Combines multiple types into one.

type A = { id: number };

type B = { value: string };

type AB = A & B;

const obj: AB = {

id: 1,

value: "Hello",

};

**Problem Statement: E-commerce Order Processing System**

**Background:**

You are building a small e-commerce application where customers can place orders for various products. The application needs to handle both customer and guest users, and it needs to process orders by combining information about the customer and the product they are purchasing.

**Requirements:**

1. **Order Processing**:
   * You need to implement a function that takes an order as a **tuple** where the first element is the product name (string) and the second element is the quantity (number). This function should return a message confirming the order details.
2. **Customer and Product Objects**:
   * Create a **Customer** object using an interface that includes:
     + id: A unique identifier for the customer (number).
     + name: The customer's full name (string).
     + email: The customer's email address (string).
   * Create a **Product** object using a type that includes:
     + id: A unique identifier for the product (number).
     + name: The product name (string).
     + price: The price of the product (number).
3. **User Management**:
   * Define a **User** type that can represent either:
     + A Customer (with all the properties: id, name, and email).
     + A Guest user (with only name and a guest flag indicating their guest status).
4. **Order Details**:
   * Create a type that combines both the Customer and Product types into an **OrderDetails** type. This type should represent an order that contains both the customer information and the details of the product being purchased.
5. **Output**:
   * Implement the following functionality:
     + When an order is placed, print a message confirming the product and quantity.
     + Print out the details of a customer, a product, and an order using the Customer, Product, User, and OrderDetails types.

**Constraints:**

* The customer must be able to place an order for a product, and the system should handle both customer and guest users.
* Ensure that the code is structured using **tuples**, **interfaces**, **types**, **union types**, and **intersection types** as shown in the example.

Sample Input:

const order: [string, number] = ["Laptop", 2];

const customer: Customer = {

  id: 1,

  name: "Dineshkumar",

  email: "Dineshkumar@example.com",

};

const product: Product = { id: 101, name: "Laptop", price: 1200 };

const user1: User = customer; // A customer

const user2: User = { name: "Divya", guest: true }; // A guest user

console.log(processOrder(order)); // Order received for 2 of Laptop.

console.log(

  `Customer: ${customer.name}, Product: ${product.name}, Price: $${product.price}`

);

console.log(user1);

console.log(user2);

const orderDetails: OrderDetails = { ...customer, ...product };

console.log(orderDetails);

Output:

Order received for 2 of Laptop.

Customer: Dineshkumar, Product: Laptop, Price: $1200

{ id: 1, name: 'Dineshkumar', email: 'Dineshkumar@example.com' }

{ name: 'Divya', guest: true }

{ id: 101, name: 'Laptop', email: 'Dineshkumar@example.com', price: 1200 }

**Functions in TypeScript**

In TypeScript, functions are strongly typed, which allows for better error checking and code completion.

#### ****1. Typing Function Parameters and Return Types****

TypeScript allows you to explicitly define types for function parameters and return values. This helps prevent bugs by ensuring the correct types are passed to and returned from functions.

**Syntax**:

function functionName(param1: type1, param2: type2): returnType {

// function logic

return result;

}

**Example**:

// Function that adds two numbers and returns a number

function add(a: number, b: number): number {

return a + b;

}

console.log(add(3, 5)); // Output: 8

// console.log(add("3", "5")); // Error: Argument of type 'string' is not assignable to parameter of type 'number'

* a and b are of type number.
* The return type is also number.

**Another Example**:

// Function that greets a user

function greet(name: string): string {

return `Hello, ${name}!`;

}

console.log(greet("Dinesh")); // Output: "Hello, Dinesh!"

#### ****2. Optional and Default Parameters****

TypeScript also supports **optional** and **default** function parameters. These features allow you to make parameters optional or provide a default value for parameters that are not passed.

##### **Optional Parameters**:

An **optional parameter** is denoted by a ? in the function signature. The parameter can either be passed or omitted when calling the function.

**Syntax**:

function functionName(param1: type1, param2?: type2): returnType {

// function logic

}

**Example**:

function greetUser(name: string, age?: number): string {

if (age) {

return `Hello, ${name}! You are ${age} years old.`;

} else {

return `Hello, ${name}!`;

}

}

console.log(greetUser("Dinesh")); // Output: "Hello, Dinesh!"

console.log(greetUser("Divya", 25)); // Output: "Hello, Divya! You are 25 years old."

* age is an optional parameter. It’s not required when calling the function.

##### **Default Parameters**:

A **default parameter** is given a default value in case the caller does not provide it.

**Syntax**:

function functionName(param1: type1, param2: type2 = defaultValue): returnType {

// function logic

}

**Example**:

function greet(name: string, greeting: string = "Hello"): string {

return `${greeting}, ${name}!`;

}

console.log(greet("Dinesh")); // Output: "Hello, Dinesh!"

console.log(greet("Divya", "Hi")); // Output: "Hi, Divya!"

* greeting has a default value of "Hello". If not provided, it will use this value.

#### ****3. Arrow Functions with Types****

Arrow functions in TypeScript are an alternative to regular functions, offering a more concise syntax. They also support typing for both parameters and return values.

##### **Syntax**:

const functionName = (param1: type1, param2: type2): returnType => {

// function logic

}

**Example**:

// Regular function

function add(a: number, b: number): number {

return a + b;

}

// Arrow function with same behavior

const addArrow = (a: number, b: number): number => a + b;

console.log(addArrow(3, 4)); // Output: 7

* The arrow function addArrow does the same thing as the regular function add, but with a more compact syntax.

##### **Arrow Function with Optional and Default Parameters**:

You can also use optional and default parameters with arrow functions.

**Example**:

// Arrow function with optional parameter

const greetUser = (name: string, age?: number): string => {

return age ? `Hello, ${name}! You are ${age} years old.` : `Hello, ${name}!`;

};

console.log(greetUser("Dinesh")); // Output: "Hello, Dinesh!"

console.log(greetUser("Divya", 30)); // Output: "Hello, Divya! You are 30 years old."

// Arrow function with default parameter

const greet = (name: string, greeting: string = "Hi"): string => {

return `${greeting}, ${name}!`;

};

console.log(greet("Dinesh")); // Output: "Hi, Dinesh!"

console.log(greet("Divya", "Good Morning")); // Output: "Good Morning, Divya!"

### ****Key Points to Remember****:

1. **Function Parameters and Return Types**:
   * Always specify types for function parameters and return values to prevent errors and improve readability.
2. **Optional Parameters**:
   * Optional parameters are denoted by ? and can be omitted when calling the function.
3. **Default Parameters**:
   * Default parameters have default values, which will be used if no argument is passed for them.
4. **Arrow Functions**:
   * Arrow functions offer a shorter syntax and can also be typed like regular functions. They are particularly useful for inline functions.

React + TypeScript project using Vite

**Steps to Set Up React + TypeScript with Vite:**

1. **Install Node.js**: If you don’t have Node.js installed, download and install it from [nodejs.org](https://nodejs.org/).
2. **Create a New Vite Project**: Use the following command to create a new Vite project with React and TypeScript:

npm create vite@latest my-react-ts-app --template react-ts

* + my-react-ts-app: This is the name of your new project. You can change it to any name you prefer.
  + --template react-ts: This specifies that you want a template that includes React and TypeScript.

After running this command, Vite will create the project folder and install all necessary dependencies.

1. **Navigate to the Project Directory**:

cd my-react-ts-app

1. **Install Dependencies** (if not already done):

Vite should have already installed the required dependencies for you. However, if you skipped this or want to make sure everything is set up correctly, run:

npm install

1. **Run the Development Server**:

To start the development server and see your project in the browser, run:

npm run dev

This will start the development server and you should see a URL (usually http://localhost:5173) where your app is running.

**Project Structure**

Once your project is created, you’ll have the following basic structure:

my-react-ts-app/

├── node\_modules/

├── public/

├── src/

│ ├── assets/

│ ├── App.tsx

│ ├── main.tsx

│ └── vite-env.d.ts

├── package.json

├── tsconfig.json

├── index.html

└── vite.config.ts

* **src/App.tsx**: This is the main component of your application. You can start writing your React components here.
* **src/main.tsx**: This is the entry point where your React app is mounted into the DOM.
* **tsconfig.json**: This file contains the TypeScript configuration for the project.
* **vite.config.ts**: The Vite configuration file.

**Example of Running Components**

Now that you’ve set up the project, you can create React components with TypeScript. Below is a step-by-step guide to create and run the components we discussed earlier.

**1. Modify App.tsx to Test Your Components:**

Open the **src/App.tsx** file and replace its content with the following code to add a simple component that accepts props:

import React, { useState } from 'react';

// Define the type for props using an interface

interface GreetingProps {

name: string;

age: number;

}

const Greeting: React.FC<GreetingProps> = ({ name, age }) => {

return <p>Hello, {name}! You are {age} years old.</p>;

};

const App: React.FC = () => {

const [count, setCount] = useState<number>(0);

return (

<div>

<Greeting name="Dinesh" age={25} />

<p>Current count: {count}</p>

<button onClick={() => setCount(count + 1)}>Increment</button>

</div>

);

};

export default App;

* This example displays the Greeting component with props name and age and allows you to increment the counter.

**2. Test the Project:**

After saving the changes, open your browser and visit the development server (e.g., http://localhost:5173).

You should see the following:

* A greeting message: "Hello, Dinesh! You are 25 years old."
* A counter with an increment button.