TYPESCRIPT

**The Ultimate Beginner's Guide to TypeScript**

JavaScript is great, but let’s be honest—it can get messy. One tiny mistake, like using the wrong variable type, can break your entire app. That’s where TypeScript comes in. It’s JavaScript’s smarter, more disciplined cousin, giving your code structure, better error-checking, and an overall smoother development experience.

If you’ve ever wished JavaScript had better tools to prevent bugs, this guide is for you!

**What is TypeScript?**

TypeScript is a **superset** of JavaScript, meaning it builds on JavaScript but adds extra features—most importantly, static typing. With TypeScript, you can:

* Catch errors before running your code.
* Use modern JavaScript features with full support.
* Scale your projects more easily.
* Enjoy better autocompletionand documentation in your editor.

It compiles down to plain JavaScript, so it works everywhere JavaScript does. Pretty cool, right?

**Setting Up TypeScript**

To get started, install TypeScript globally:

npm install -g typescript

Check if it’s installed:

tsc --version

To compile a TypeScript file (.ts), run:

tsc filename.ts

This creates a filename.js file, which is standard JavaScript.

**1. TypeScript Basics: Variables & Types**

**Declaring Variables**

TypeScript introduces static types, which means you must specify the type of a variable when declaring it.

let message: string = "Hello, TypeScript!";

let age: number = 25;

let isStudent: boolean = false;

But TypeScript is smart! If you assign a value, it infers the type automatically:

let city = "New York"; // TypeScript knows this is a string.

**Common Types in TypeScript**

| **Type** | **Example** |
| --- | --- |
| string | "hello" |
| number | 42, 3.14 |
| boolean | true, false |
| null | null |
| undefined | undefined |
| any | *(Disables type checking)* |

**2. Functions in TypeScript**

Functions in TypeScript allow you to define the expected parameter and return types, leading to fewer runtime errors and better maintainability.

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

return a + b;

}

**Optional & Default Parameters**

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

console.log(`Hello ${name}`);

}

function greet(name: string = "Guest") {

console.log(`Hello ${name}`);

}

**Arrow Functions**

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

**3. Objects & Interfaces**

Objects help structure data in TypeScript, while interfaces allow defining consistent object shapes across your codebase.

**Using Objects**

let user: { name: string; age: number } = {

name: "Alice",

age: 30

};

**Interfaces (Better Object Structures)**

interface User {

name: string;

age?: number; // Optional property

}

let user: User = {

name: "John",

age: 25

};

**Readonly Properties**

interface User {

readonly id: number;

name: string;

}

**4. Classes & Object-Oriented Programming (OOP)**

Classes in TypeScript bring traditional OOP concepts like encapsulation, inheritance, and polymorphism, making your code more modular and reusable.

**Creating a Class**

class Person {

name: string;

age: number;

constructor(name: string, age: number) {

this.name = name;

this.age = age;

}

greet(): void {

console.log(`Hello, my name is ${this.name}.`);

}

}

const person = new Person("Alice", 30);

person.greet();

**Access Modifiers**

| **Modifier** | **Description** |
| --- | --- |
| public | Accessible anywhere (default). |
| private | Accessible only within the class. |
| protected | Accessible within the class and its subclasses. |

**Inheritance**

class Animal {

constructor(public name: string) {}

makeSound() {

console.log("Some sound...");

}

}

class Dog extends Animal {

makeSound() {

console.log("Woof! Woof!");

}

}

**5. Advanced TypeScript Features**

TypeScript provides powerful features like enums, generics, and utility types to create robust and flexible applications.

**Enums**

enum Status {

Pending,

Approved,

Rejected

}

**Generics (Reusable Code)**

function identity<T>(value: T): T {

return value;

}

**6. Modules in TypeScript**

Modules help break large applications into smaller, manageable files, promoting code reusability and maintainability.

**Named Exports**

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

return a + b;

}

import { add } from "./math";

**Default Exports**

export default function greet(name: string) {

return `Hello, ${name}!`;

}

import greet from "./greet";

**7. TypeScript with React**

Set up React with TypeScript:

npx create-react-app my-app --template typescript

React Component with TypeScript:

import React from "react";

interface Props {

name: string;

}

const Greeting: React.FC<Props> = ({ name }) => {

return <h1>Hello, {name}!</h1>;

};

export default Greeting;

**Conclusion**

TypeScript **supercharges JavaScript** with static typing, better tooling, and more maintainable code. Whether you're building small apps or enterprise-level projects, it’s a game-changer.

💡 **Next Steps:**

* Try TypeScript in a real project.
* Learn about tsconfig.json for project setup.
* Explore TypeScript with frameworks like React or Node.js.

Want to level up your TypeScript skills? Let’s go deeper! 🚀

REACT

**The Beginner's Guide to React: Build Amazing UIs with Ease**

React is one of the most popular JavaScript libraries for building user interfaces. Developed by Facebook, it makes creating interactive and dynamic web applications a breeze. If you've ever felt overwhelmed by JavaScript’s complexity, React offers a structured way to build apps with reusable components.

**Why Choose React?**

Here’s why React is a game-changer: ✔ **Component-Based** – Build reusable UI pieces. ✔ **Fast & Efficient** – Uses a Virtual DOM for optimized updates. ✔ **Strong Community** – Tons of resources and support. ✔ **Great for All Projects** – From simple websites to complex apps.

If you're ready to simplify your front-end development, let's dive in! 🚀

**1. Setting Up React**

**Using Create React App (Recommended)**

The easiest way to start a React project is using Create React App.

npx create-react-app my-app

cd my-app

npm start

This sets up everything for you, including Webpack, Babel, and a local development server.

**Using Vite (Faster Alternative)**

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

cd my-app

npm install

npm run dev

**2. Understanding JSX (JavaScript XML)**

JSX is a syntax extension that lets you write HTML-like code inside JavaScript. It makes UI code more readable and intuitive.

const element = <h1>Hello, React!</h1>;

JSX compiles to JavaScript:

React.createElement("h1", null, "Hello, React!");

**3. Components in React**

React apps are made up of **components**—small, reusable pieces of UI.

**Functional Components (Recommended)**

function Greeting() {

return <h1>Hello, World!</h1>;

}

**Class Components (Older Style)**

class Greeting extends React.Component {

render() {

return <h1>Hello, World!</h1>;

}

}

**4. Props (Passing Data to Components)**

Props allow components to receive dynamic data.

function Welcome(props) {

return <h1>Hello, {props.name}!</h1>;

}

export default function App() {

return <Welcome name="Alice" />;

}

**5. State and useState Hook**

State allows components to remember information.

import { useState } from "react";

function Counter() {

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

return (

<div>

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

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

</div>

);

}

**6. Handling Events**

React handles events like JavaScript but with JSX syntax.

function Button() {

function handleClick() {

alert("Button clicked!");

}

return <button onClick={handleClick}>Click Me</button>;

}

**7. Conditional Rendering**

Show or hide elements based on conditions.

function UserGreeting({ isLoggedIn }) {

return <h1>{isLoggedIn ? "Welcome back!" : "Please sign in."}</h1>;

}

**8. Lists and Keys**

Rendering dynamic lists in React.

function ItemList() {

const items = ["Apple", "Banana", "Cherry"];

return (

<ul>

{items.map((item, index) => (

<li key={index}>{item}</li>

))}

</ul>

);

}

**9. Forms in React**

Handling user input with controlled components.

function Form() {

const [text, setText] = useState("");

function handleChange(event) {

setText(event.target.value);

}

return (

<form>

<input type="text" value={text} onChange={handleChange} />

<p>You typed: {text}</p>

</form>

);

}

**10. React Router (Navigation)**

React Router allows multi-page navigation without reloading.

npm install react-router-dom

import { BrowserRouter as Router, Routes, Route, Link } from "react-router-dom";

function Home() {

return <h2>Home Page</h2>;

}

function About() {

return <h2>About Page</h2>;

}

function App() {

return (

<Router>

<nav>

<Link to="/">Home</Link> | <Link to="/about">About</Link>

</nav>

<Routes>

<Route path="/" element={<Home />} />

<Route path="/about" element={<About />} />

</Routes>

</Router>

);

}

**11. Fetching Data (API Calls with useEffect)**

Fetch data from an API when the component loads.

import { useState, useEffect } from "react";

function Users() {

const [users, setUsers] = useState([]);

useEffect(() => {

fetch("https://jsonplaceholder.typicode.com/users")

.then((response) => response.json())

.then((data) => setUsers(data));

}, []);

return (

<ul>

{users.map((user) => (

<li key={user.id}>{user.name}</li>

))}

</ul>

);

}

**12. Context API (Global State Management)**

Share state across components without prop drilling.

import { createContext, useContext, useState } from "react";

const UserContext = createContext();

function UserProvider({ children }) {

const [user, setUser] = useState("Guest");

return (

<UserContext.Provider value={{ user, setUser }}>

{children}

</UserContext.Provider>

);

}

function Profile() {

const { user } = useContext(UserContext);

return <h1>User: {user}</h1>;

}

function App() {

return (

<UserProvider>

<Profile />

</UserProvider>

);

}

**Conclusion**

React simplifies front-end development with reusable components, hooks, and state management. Whether you're building a small website or a complex app, React provides the flexibility and power you need.

💡 **Next Steps:**

* Explore React Hooks (useEffect, useReducer, etc.).
* Learn about performance optimization (Memoization, Lazy Loading).
* Experiment with advanced state management like Redux.

Happy coding! 🚀