**TypeScript:**

**What is** [**TypeScript**](https://chatgpt.com?q=TypeScript)**?**

[TypeScript](https://chatgpt.com?q=TypeScript) is an **open-source programming language** developed and maintained by [Microsoft](https://chatgpt.com?q=Microsoft). It is a **superset of JavaScript**, which means all JavaScript code is valid TypeScript code—but with additional powerful features.

**Key Characteristics:**

* **Statically Typed**: TypeScript allows you to define types for variables, function parameters, and return values. This helps catch errors at **compile-time** rather than at **runtime**.
* **Compiled Language**: TypeScript code is compiled into standard [JavaScript](https://chatgpt.com?q=JavaScript) so it can run in any browser or JavaScript engine.
* **Object-Oriented**: TypeScript supports features like **classes**, **interfaces**, **inheritance**, and **access modifiers** (public, private, protected), making it ideal for large-scale applications.
* **Tooling Support**: Offers excellent support in code editors like [Visual Studio Code](https://chatgpt.com?q=Visual%20Studio%20Code) with features like IntelliSense, autocompletion, and real-time type checking.

**Example:**

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

return a + b;

}

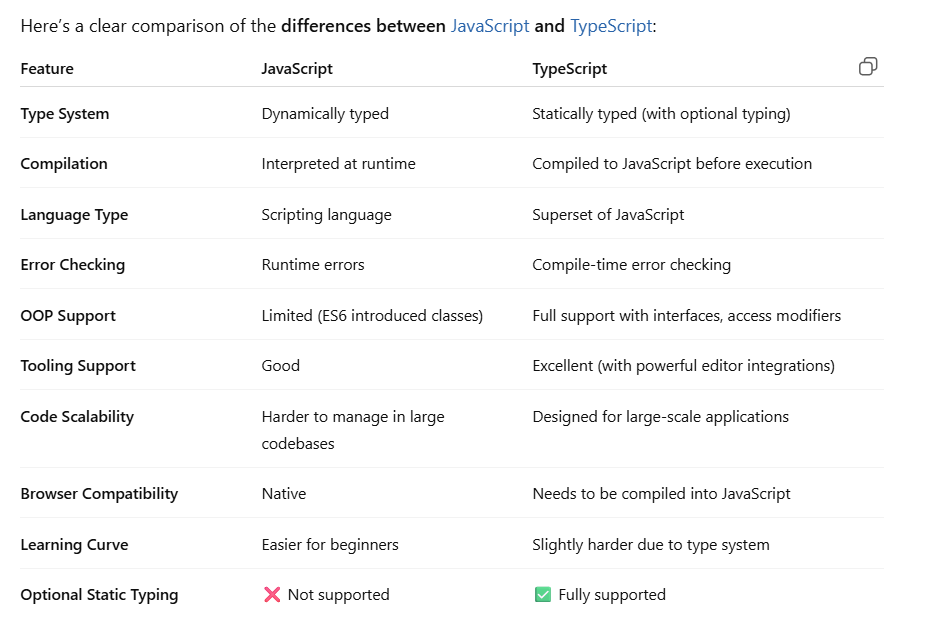
console.log(add(5, 3)); // Output: 8

Here, a and b are both typed as numbers, and the function is guaranteed to return a number.

**Why Use TypeScript?**

* Early error detection
* Better code documentation
* Improved readability and maintainability
* Scalability for large projects
* Enhanced collaboration in teams

**Differences between** [**JavaScript**](https://chatgpt.com?q=JavaScript) **and TypeScript**



**Example Comparison:**

**JavaScript:**

function greet(name) {

return "Hello " + name;

}

greet(123); // Works, but may cause bugs

**TypeScript:**

function greet(name: string): string {

return "Hello " + name;

}

greet(123); // ❌ Error at compile-time

**Note:**

* Use **JavaScript** for smaller projects or rapid prototyping.
* Use **TypeScript** when working on **large codebases**, in **team environments**, or when you want **robust tooling and error prevention**.

**Benefits of using TypeScript**

**✅ 1. Static Typing**

* Helps catch errors at **compile-time**, before running the code.
* Reduces bugs and improves code quality.

let age: number = 25;

// age = "twenty-five"; // ❌ Compile-time error

**✅ 2. Improved Code Readability & Maintainability**

* Types serve as **self-documentation**, making the code easier to understand.
* Great for onboarding new developers in a team.

**✅ 3. Excellent IDE Support**

* Features like **IntelliSense**, **autocompletion**, **type hints**, and **refactoring tools**.
* Works seamlessly with editors like [Visual Studio Code](https://chatgpt.com?q=Visual%20Studio%20Code).

**✅ 4. Object-Oriented Programming Features**

* Supports **classes**, **interfaces**, **inheritance**, **access modifiers** (private, public, protected) and **abstract classes**.

class Person {

private name: string;

constructor(name: string) {

this.name = name;

}

}

**✅ 5. Early Error Detection**

* Detects issues **before** running the code, reducing debugging time.
* Especially useful for large-scale applications.

**✅ 6. Better Collaboration**

* Clearly defined types make it easier for teams to understand how to use functions, objects, and components.

**✅ 7. Rich Ecosystem & Compatibility**

* Works with existing JavaScript code.
* Compatible with popular frameworks like [React](https://chatgpt.com?q=React), [Angular](https://chatgpt.com?q=Angular), and [Vue](https://chatgpt.com?q=Vue).

**✅ 8. Easier Refactoring**

* Refactor code with confidence due to static type checks.
* Safely rename variables and restructure components.

**✅ 9. Improved Project Scalability**

* Ideal for building **enterprise-level** or **long-term** projects.
* Reduces technical debt over time.

**✅ 10. Optional Typing**

* TypeScript is flexible — you can gradually adopt it in JavaScript projects using **type inference** and **JSDoc comments**.

**Installing TypeScript (npm install -g typescript)**

**Step-by-Step Guide: Installing TypeScript Globally**

**1. Make sure Node.js and npm are installed**

TypeScript requires **Node.js** and **npm**. To check if you already have them:

node -v

npm -v

If you don't have them, download and install Node.js from the official site: <https://nodejs.org>

**2. Install TypeScript globally using npm**

Run the following command in your terminal or command prompt:

**npm install -g typescript**

* The -g flag means it will install TypeScript **globally**, so you can use the tsc command anywhere.

**3. Verify the installation**

After installation, check the installed TypeScript version:

**tsc -v**

**Example output:**

Version 5.4.3

**🛠️ What does this do?**

* Installs the TypeScript compiler (tsc).
* Allows you to compile .ts files into .js using:

**tsc filename.ts**

**First TypeScript program**

Here’s how to write and run your **first** [**TypeScript**](https://chatgpt.com?q=TypeScript) **program** in just a few easy steps:

**✅ Step 1: Create a TypeScript File**

Create a new file named hello.ts:

// hello.ts

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

console.log(message);

**✅ Step 2: Compile the TypeScript Code**

Open your terminal or command prompt in the folder where hello.ts is saved, then run:

**tsc hello.ts**

This will compile your TypeScript file into a JavaScript file named hello.js.

**✅ Step 3: Run the Output JavaScript File**

Use Node.js to run the compiled JavaScript file:

**node hello.js**

You should see this output:

Hello, TypeScript!



**Setting up a TypeScript project**

**Step-by-Step Setup: TypeScript Project**

**✅ 1. Create a new project folder**

mkdir my-typescript-project

cd my-typescript-project

**✅ 2. Initialize a Node.js project**

This creates a package.json file:

npm init -y

**✅ 3. Install TypeScript as a dev dependency**

npm install --save-dev typescript

**✅ 4. Create a tsconfig.json file**

npx tsc --init

This creates a default tsconfig.json file. Modify it like this:

{

"compilerOptions": {

"target": "es6",

"module": "commonjs",

"outDir": "./dist",

"rootDir": "./src",

"strict": true,

"esModuleInterop": true

},

"include": ["src"]

}

**✅ 5. Create source folders and files**

mkdir src

touch src/index.ts

Then in src/index.ts:

const greet = (name: string): string => {

return `Hello, ${name}!`;

};

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

**✅ 6. Compile TypeScript**

npx tsc

* This compiles .ts files from src to .js files in dist.

**✅ 7. Run your app**

node dist/index.js

**✅ 8. Optional: Add npm scripts**

In package.json, add:

"scripts": {

"build": "tsc",

"start": "node dist/index.js",

"dev": "tsc --watch"

}

Now you can use:

npm run build # compile once

npm run dev # compile on file change

npm start # run the compiled JS

**Compiling .ts files to .js (tsc command)**

The tsc command is the **TypeScript compiler**. It converts your .ts (TypeScript) files into .js (JavaScript) so they can be run by browsers or Node.js.

**✅ Basic Syntax:**

tsc filename.ts

This compiles filename.ts and creates a filename.js in the same directory.

**📌 Example:**

**hello.ts**

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

console.log(message);

**Compile it:**

tsc hello.ts

**Output:**

A new file named hello.js is created:

var message = "Hello, TypeScript!";

console.log(message);

**✅ Compile an entire project using tsconfig.json**

If you have a tsconfig.json file (created with tsc --init), just run:

tsc

It will:

* Compile all .ts files inside the "include" or "rootDir" specified in the config.
* Output to the "outDir" (e.g., dist/).

**✅ Compile and Watch for Changes:**

Automatically recompile .ts files whenever you save them:

tsc --watch

**✅ Compile Multiple Files:**

tsc file1.ts file2.ts

tsconfig.json explained

The tsconfig.json file tells the TypeScript compiler **how to compile your project**. It defines compiler options, root files, and output settings.

**✅ Example tsconfig.json**

{

"compilerOptions": {

"target": "es6",

"module": "commonjs",

"rootDir": "./src",

"outDir": "./dist",

"strict": true,

"esModuleInterop": true

},

"include": ["src"],

"exclude": ["node\_modules"]

}

**📂 compilerOptions**

Controls how .ts files are compiled.



**Basic Types**

**number, string, boolean**

**number**

Represents **both integers and floating-point numbers**.

let age: number = 25;

let price: number = 99.99;

let hex: number = 0xff; // hexadecimal

let binary: number = 0b1010; // binary

let octal: number = 0o744; // octal

**string**

Represents **textual data** enclosed in quotes.

let firstName: string = "Abhinav";

let greeting: string = `Hello, ${firstName}!`; // template string

**boolean**

Represents a logical value: true or false.

let isLoggedIn: boolean = true;

let isAdmin: boolean = false;

**Basic Types** - any, unknown, void, null, undefined, never

**any**

* Disables type checking.
* Use it **sparingly** — often indicates a missing type.

let value: any = "hello";

value = 123; // ✅ Allowed

value = true; // ✅ Allowed

Use any when you’re dealing with **dynamic content**, like JSON from APIs.

**unknown**

* Like any, but **type-safe** — can’t use it without narrowing the type.

let input: unknown = "Hi";

input = 123; // ✅ Allowed

// ❌ Error: Object is of type 'unknown'

console.log(input.toUpperCase());

// ✅ Need type checking first:

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

console.log(input.toUpperCase());

}

Prefer unknown over any for values of unknown types.

**void**

* Used for functions that **do not return a value**.

function logMessage(message: string): void {

console.log(message);

}

Think of it as: "this function is just doing something — not returning anything."

**null**

* A special value that represents "no value".

let user: null = null;

Use in union types: string | null

**undefined**

* A variable that’s declared but not assigned a value.

let score: undefined = undefined;

Often returned by functions that don’t return anything explicitly.

**never**

* Indicates a value that **never occurs**.
* Typically used in:
  + functions that throw errors
  + infinite loops
  + exhaustive checks

function throwError(message: string): never {

throw new Error(message);

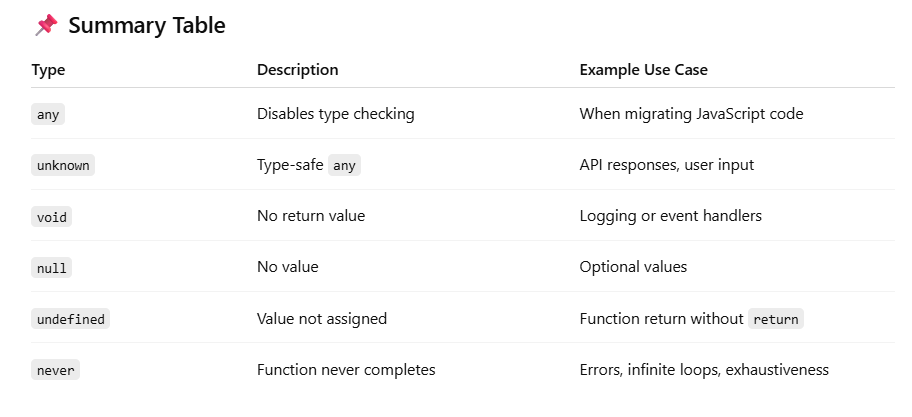
}

function loopForever(): never {

while (true) {}

}

Use never when a function **cannot return** under any circumstances.



**Arrays (number[], Array<string>)**

**Arrays in** [**TypeScript**](https://chatgpt.com?q=TypeScript)

In TypeScript, arrays are used to store **multiple values of the same type**. You can define an array in **two ways**:

**✅ 1. Using type[] syntax**

let numbers: number[] = [1, 2, 3, 4];

let names: string[] = ["Abhinav", "Sai", "Vannam"];

**✅ 2. Using Array<type> generic syntax**

let numbers: Array<number> = [10, 20, 30];

let names: Array<string> = ["Type", "Script"];

Both styles are **functionally identical** — it's a matter of preference.

**Common Operations:**

let fruits: string[] = ["apple", "banana"];

fruits.push("mango"); // Add

let firstFruit = fruits[0]; // Access

console.log(fruits.length); // Length

**🔁 Looping through an array:**

let scores: number[] = [80, 90, 100];

for (let score of scores) {

console.log(score);

}

**Tuples**

A **tuple** is a **fixed-length array** where each element can have a **different type**. Unlike normal arrays, the **types and order** of elements in a tuple are strictly defined.

**✅ Basic Syntax:**

let person: [string, number] = ["Abhinav", 25];

* person[0] must be a string
* person[1] must be a number

**🔁 Example:**

let user: [string, boolean] = ["admin", true];

console.log(user[0]); // "admin"

console.log(user[1]); // true

**❌ Type-Safety in Tuples:**

let coords: [number, number] = [10, 20];

// coords = [10, "north"]; // ❌ Error: second element must be a number

**🆕 Optional and Rest Elements in Tuples (ES6+):**

**✅ Optional Elements:**

let userInfo: [string, number?] = ["John"];

**✅ Rest Elements:**

let rgb: [number, number, ...number[]] = [255, 100, 50, 25];

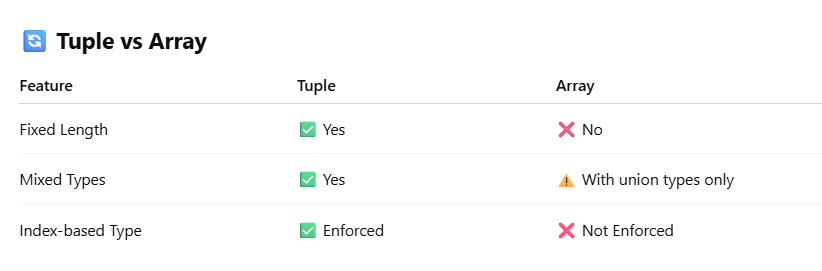
**Tuples in Functions:**

function getUser(): [string, number] {

return ["Alice", 30];

}

const [name, age] = getUser();



**Enums**