What is TypeScript?

TypeScript (TS) is a **programming language developed by Microsoft** that is a **superset of JavaScript**.

This means:

- Every valid JavaScript code is also valid TypeScript code.
- TypeScript adds **extra features** on top of JavaScript, primarily **static typing**.

Key Features of TypeScript

1. **Static Typing**
   - You can declare types for variables, function parameters, and return values.
   - Helps catch errors **before running the code**.

```
2. let age: number = 25;   // TypeScript knows age must be a number
```

3. **Compile-time Checking**
   - TypeScript code is **compiled to JavaScript** before running.
   - Errors are caught **at compile-time**, reducing runtime bugs.
4. **Supports Modern JavaScript**
   - Works with **ES6+ features** like `classes`, `modules`, `arrow functions`, `template literals`, etc.
5. **Interfaces and Enums**
   - Define the **structure of objects** with `interfaces`.
   - Define **named constants** with `enums`.
6. **Optional Types**
   - You can write as strict or as flexible code as you like.
   - Using `any` allows any type temporarily.
7. **Classes and OOP**
   - Supports **object-oriented programming** features like classes, inheritance, access modifiers (`public`, `private`, `protected`).
8. **Tooling Support**
   - IDEs like **VS Code** provide **autocomplete, error checking, and refactoring** for TypeScript.

Why Use TypeScript?

- **Catch errors early** → fewer bugs at runtime.
- **Better code readability** → easier for teams.
- **Improved tooling** → autocomplete and type hints.
- **Safe scaling** → great for large projects.

Let's do a **full comparison of TypeScript vs JavaScript**, step by step, with all the examples we discussed. I'll explain **exactly what changes** and why, and give you runnable code with expected output.

---

1. Basic Types

TypeScript

```
let age: number = 25;
let name: string = "Alice";
let isStudent: boolean = true;
let hobbies: string[] = ["Reading", "Swimming", "Coding"];

console.log(age, name, isStudent, hobbies);
```

- **Type annotations** (`: number`, `: string`, `: boolean`, `: string[]`) ensure type safety.
- Compile-time errors if types are wrong.

Output:

```
25 Alice true ["Reading", "Swimming", "Coding"]
```

JavaScript

```
let age = 25;
let name = "Alice";
let isStudent = true;
let hobbies = ["Reading", "Swimming", "Coding"];

console.log(age, name, isStudent, hobbies);
```

- **No type annotations**.
- JavaScript is dynamically typed — types are checked at runtime, not compile-time.

Output:

```
25 Alice true ["Reading", "Swimming", "Coding"]
```

 Comparison: TypeScript catches errors before running; JS allows anything at runtime.

---

2. Functions with Types

TypeScript

```
function add(a: number, b: number): number {
    return a + b;
}
console.log(add(5, 3)); // 8
```

- `a: number`, `b: number` → types for parameters
- `: number` → return type
- Compile-time error if you try `add(5, "3")`.

JavaScript

```
function add(a, b) {
    return a + b;
}
console.log(add(5, 3));   // 8
console.log(add(5, "3")); // 53 (runtime concatenation!)
```

- No type safety; `"3"` is concatenated as string at runtime.

 Comparison: TS prevents type mistakes, JS executes whatever you give.

---

3. Interfaces / Object Shapes

TypeScript

```
interface Person {
    name: string;
    age: number;
    greet(): void;
```

```
}

const user: Person = {
    name: "Bob",
    age: 30,
    greet() { console.log(`Hello, my name is ${this.name}`); }
};

user.greet();
```

- `interface Person` enforces object structure.
- Compile-time error if a property is missing or of wrong type.

JavaScript

```
const user = {
    name: "Bob",
    age: 30,
    greet() { console.log(`Hello, my name is ${this.name}`); }
};

user.greet();
```

- No interface; object can have any structure.
- JS won't warn if you forget `age` or `greet`.

Output for both:

```
Hello, my name is Bob
```

 Comparison: Interfaces are only in TypeScript; they help maintain structure.

---

4. Classes & Inheritance

TypeScript

```
class Animal {
    constructor(public name: string) {}
    move(distance: number) {
        console.log(`${this.name} moved ${distance}m.`);
    }
}

class Dog extends Animal {
    bark() { console.log("Woof! Woof!"); }
}
```

```
const dog = new Dog("Buddy");
dog.bark();
dog.move(10);
```

- `public name: string` → property with type
- `distance: number` → typed parameter

Output:

```
Woof! Woof!
Buddy moved 10m.
```

JavaScript

```
class Animal {
    constructor(name) {
        this.name = name;
    }
    move(distance) {
        console.log(`${this.name} moved ${distance}m.`);
    }
}

class Dog extends Animal {
    bark() { console.log("Woof! Woof!"); }
}

const dog = new Dog("Buddy");
dog.bark();
dog.move(10);
```

- Types removed, `public` keyword removed.
- Same behavior at runtime.

Output:

```
Woof! Woof!
Buddy moved 10m.
```

 Comparison: TypeScript ensures type safety; JavaScript just runs.

---

5. Enums

TypeScript

```
enum Direction { Up, Down, Left, Right }
let dir: Direction = Direction.Left;
console.log(dir); // 2
```

- Enums create named constants.
- Type-safe — you cannot assign arbitrary numbers.

JavaScript

```
const Direction = { Up: 0, Down: 1, Left: 2, Right: 3 };
let dir = Direction.Left;
console.log(dir); // 2
```

- Manual object used instead of enum.
- No type safety.

 Comparison: TS enums are more readable and prevent invalid values.

---

6. Generics

TypeScript

```
function identity<T>(arg: T): T { return arg; }
console.log(identity<number>(42));   // 42
console.log(identity<string>("Hi")); // Hi
```

- `T` is a placeholder type.
- Compile-time type safety while being reusable.

JavaScript

```
function identity(arg) { return arg; }
console.log(identity(42));   // 42
console.log(identity("Hi")); // Hi
```

- JS can accept any type, but no type safety.

 Comparison: TypeScript gives both **flexibility + type safety**.

Step-by-Step to Run

1. **For TypeScript**

```
npm install -g typescript
tsc filename.ts     # Compile TS → JS
node filename.js    # Run JS
```

5. **For JavaScript**

```
node filename.js    # Run directly
```

7. **Optional** (for TS without compiling)

```
npm install -g ts-node
ts-node filename.ts
```

Summary of Key Differences

Feature	TypeScript	JavaScript
Typing	Static	Dynamic
Interfaces	Yes	No
Enums	Built-in	Use objects
Classes	Type-safe	Same at runtime
Generics	Yes	Not available
Compile-time errors	Yes	No, runtime only

Let's go **step by step** and cover **if-else, loops, and switch statements** in **TypeScript vs JavaScript**, with explanations, code, and comparisons.

---

1. If-Else Statements

TypeScript

```
let age: number = 20;

if (age < 18) {
    console.log("You are a minor.");
} else if (age < 60) {
    console.log("You are an adult.");
} else {
    console.log("You are a senior.");
}
```

Output

```
You are an adult.
```

- `age: number` ensures only numbers are allowed.
- If you try `let age: number = "20"`, TypeScript throws a **compile-time error**.

JavaScript

```
let age = 20;

if (age < 18) {
    console.log("You are a minor.");
} else if (age < 60) {
    console.log("You are an adult.");
} else {
    console.log("You are a senior.");
}
```

Output

```
You are an adult.
```

- JavaScript is dynamically typed → assigning `age = "20"` works but may produce unexpected results (`"20" < 18` is false because of type coercion).

 Comparison: TS prevents type mistakes; JS executes but may behave unexpectedly if types are wrong.

---

2. Loops

TypeScript (for, while, for-of)

```
// For loop
for (let i: number = 1; i <= 3; i++) {
    console.log(`For loop iteration: ${i}`);
}

// While loop
let j: number = 1;
while (j <= 3) {
    console.log(`While loop iteration: ${j}`);
    j++;
}

// For-of loop
let fruits: string[] = ["Apple", "Banana", "Cherry"];
for (let fruit of fruits) {
    console.log(fruit);
}
```

Output

```
For loop iteration: 1
For loop iteration: 2
For loop iteration: 3
While loop iteration: 1
While loop iteration: 2
While loop iteration: 3
Apple
Banana
Cherry
```

JavaScript (same code without types)

```
// For loop
for (let i = 1; i <= 3; i++) {
    console.log(`For loop iteration: ${i}`);
}

// While loop
let j = 1;
while (j <= 3) {
    console.log(`While loop iteration: ${j}`);
    j++;
```

```
}

// For-of loop
let fruits = ["Apple", "Banana", "Cherry"];
for (let fruit of fruits) {
    console.log(fruit);
}
```

Output

```
For loop iteration: 1
For loop iteration: 2
For loop iteration: 3
While loop iteration: 1
While loop iteration: 2
While loop iteration: 3
Apple
Banana
Cherry
```

 Comparison: Loops behave the same; TypeScript adds **type safety**.

---

3. Switch Statement

TypeScript

```
let day: number = 3;

switch(day) {
    case 1:
        console.log("Monday");
        break;
    case 2:
        console.log("Tuesday");
        break;
    case 3:
        console.log("Wednesday");
        break;
    default:
        console.log("Another day");
}
```

Output

```
Wednesday
```

- `day: number` ensures you only pass numbers.
- TypeScript will warn if you use a string accidentally (`day = "3"`).

JavaScript

```
let day = 3;

switch(day) {
    case 1:
        console.log("Monday");
        break;
    case 2:
        console.log("Tuesday");
        break;
    case 3:
        console.log("Wednesday");
        break;
    default:
        console.log("Another day");
}
```

Output

```
Wednesday
```

- Works the same way, but JS won't warn if types are wrong (e.g., `day = "3"`).

 Comparison: Switch statements function the same; TypeScript prevents type mistakes.

Quick Summary Table: TS vs JS for Control Flow

Feature	TypeScript	JavaScript
If-Else	Supports types (`number`, `string`) for safety	Dynamic typing; no type checking
Loops	Works the same; type annotations possible	Works the same; runtime only
Switch	Works the same; can type-check the value	Works the same; may allow unintended types
Error Checking	Compile-time errors if types mismatch	Errors only at runtime