TypeScript Tutorial: Adding Type Safety to JavaScript

Introduction

- As we learned from Sarah's story, TypeScript saved her from countless runtime errors and debugging sessions.
- This tutorial will teach you how TypeScript adds type safety to JavaScript while maintaining all of JavaScript's power and flexibility.

What is TypeScript?

- TypeScript is a **superset of JavaScript** that adds **static type checking**.
- Remember Sarah's 3 AM debugging session? TypeScript would have caught those errors at compile time!

Why TypeScript?

```
// JavaScript - Error found at runtime (in production!)
function calculatePriority(task) {
    return task.priority.level + task.urgency;  // What if priority is undefined?
}

// TypeScript - Error found during development
interface Task {
    priority?: { level: number };  // Optional property
    urgency: number;
}

function calculatePriority(task: Task): number {
    // TypeScript ERROR: Object is possibly 'undefined'
    return task.priority.level + task.urgency;
}
```

Setting Up TypeScript

We need to install Typescript compiler (tsc)!

```
# Install TypeScript globally
npm install -g typescript

# Create a TypeScript file
echo 'console.log("Hello, TypeScript!");' > hello.ts

# Compile to JavaScript
tsc hello.ts

# Run the compiled JavaScript
node hello.js
```

We need `tsconfi.json` to configure typescript compiler:

```
{
  "compilerOptions": {
    "target": "ES2018",
    "module": "es2015",
    "lib": [
      "ES2018",
      "DOM"
    ],
    "outDir": "./dist",
    "rootDir": "./src",
    "strict": true,
    "noFallthroughCasesInSwitch": true
  },
  "include": [
    "src/**/*"
  ],
  "exclude": [
    "node_modules"
  ]
}
```

Basic Syntax and Type Annotations

Variables with Types

```
// Type annotations - explicitly declare types
let studentName: string = "Alice";
let age: number = 20;
let isEnrolled: boolean = true;

// Type inference - TypeScript figures out the type
let courseName = "CSC640";  // TypeScript knows this is string
let creditHours = 3;          // TypeScript knows this is number

// Constants with types
const PASSING_GRADE: number = 60;
const UNIVERSITY_NAME: string = "Tech University";
```

Type Errors at Compile Time

```
let score: number = 85;

// This won't compile!
score = "Eighty-five";
// Error: Type 'string' is not assignable to type 'number'

// This helps catch bugs early
let studentCount: number = 25;
// studentCount = studentCount + "1";
// Error! Would be "251" in JavaScript
studentCount = studentCount + 1;      // Correct: 26
```

Union Types

```
// Variable can be multiple types
let result: number | string;
result = 100;          // OK
result = "Perfect";  // OK
// result = true;     // Error!

// Useful for function returns
function getGrade(score: number): string | number {
    if (score >= 90) return "A";
    if (score >= 80) return "B";
    return score;  // Return numeric score if below B
}
```

Type System

Basic Types

```
// Primitives
let studentName: string = "Sarah";
let age: number = 22;
let isActive: boolean = true;
let data: null = null;
let notDefined: undefined = undefined;

// Arrays
let scores: number[] = [85, 92, 78];
let names: string[] = ["Alice", "Bob", "Carol"];
let mixed: Array<string | number> = ["Alice", 20, "Bob", 21];
```

```typescript
// Tuple - fixed length array with known types
let student: [string, number] = ["Alice", 20];
// student = [20, "Alice"];  // Error! Wrong order

// Enum - named constants
enum Grade {
    A = 90,
    B = 80,
    C = 70,
    D = 60,
    F = 0
}

let myGrade: Grade = Grade.B;
```

Type Aliases

```
// Create custom type names
type StudentID = number;
type GPA = number;
type CourseName = string;

// Use them for clarity
let id: StudentID = 12345;
let gpa: GPA = 3.85;
let course: CourseName = "TypeScript 101";

// Complex type alias
type Grade = "A" | "B" | "C" | "D" | "F";
let finalGrade: Grade = "A";
// finalGrade = "E";  // Error! "E" is not a valid grade
```

Interfaces

```typescript
// Define object structure
interface Student {
    id: number;
    name: string;
    email: string;
    gpa: number;
    courses: string[];
}

// Use the interface
const alice: Student = {
    id: 1,
    name: "Alice Johnson",
    email: "alice@university.edu",
    gpa: 3.8,
    courses: ["CS101", "MATH201"]
};

// TypeScript ensures all properties are present
// const bob: Student = {
//     id: 2,
//     name: "Bob Smith"
//     // Error! Missing email, gpa, courses
// };
```

Optional and Readonly Properties

```
interface Course {
    code: string;
    name: string;
    credits: number;
    instructor?: string;      // Optional property
    readonly id: number;      // Can't be changed after creation
}

const mathCourse: Course = {
    id: 101,
    code: "ASE285",
    name: "Software Engineering",
    credits: 3
    // instructor is optional, so this is valid
};

// mathCourse.id = 102;
// Error! Cannot assign to 'id' because it is a read-only property
mathCourse.instructor = "Dr. Smith";
// OK - adding optional property
```

Functions with Types

Function Type Annotations

```
// Parameters and return types
function calculateGPA(grades: number[], credits: number[]): number {
    let totalPoints = 0;
    let totalCredits = 0;

    for (let i = 0; i < grades.length; i++) {
        totalPoints += grades[i] * credits[i];
        totalCredits += credits[i];
    }

    return totalPoints / totalCredits;
}

// Arrow function with types
const getLetterGrade = (percentage: number): string => {
    if (percentage >= 90) return "A";
    if (percentage >= 80) return "B";
    if (percentage >= 70) return "C";
    if (percentage >= 60) return "D";
    return "F";
};
```

Optional and Default Parameters

```
// Optional parameter
function greetStudent(name: string, title?: string): string {
    if (title) {
        return `Hello, ${title} ${name}!`;
    }
    return `Hello, ${name}!`;
}

console.log(greetStudent("Alice"));          // "Hello, Alice!"
console.log(greetStudent("Bob", "Dr."));     // "Hello, Dr. Bob!"

// Default parameters
function createStudent(
    name: string,
    age: number = 18,
    major: string = "Undeclared"
): Student {
    return {
        id: Date.now(),
        name,
        email: `${name.toLowerCase()}@nku.edu`,
        gpa: 0.0,
        courses: []
    };
}
```

Function Types

```
// Define a function type
type MathOperation = (a: number, b: number) => number;

// Use the function type
const add: MathOperation = (a, b) => a + b;
const multiply: MathOperation = (a, b) => a * b;

// Function that takes a function as parameter
function calculate(
    a: number,
    b: number,
    operation: MathOperation
): number {
    return operation(a, b);
}

console.log(calculate(5, 3, add));        // 8
console.log(calculate(5, 3, multiply));  // 15
```

Overloading

```
// Function overloading - different parameter types
function processGrade(grade: number): string;
function processGrade(grade: string): number;
function processGrade(grade: number | string): string | number {
    if (typeof grade === "number") {
        // Convert number to letter grade
        if (grade >= 90) return "A";
        if (grade >= 80) return "B";
        return "C";
    } else {
        // Convert letter grade to number
        switch(grade) {
            case "A": return 90;
            case "B": return 80;
            default: return 70;
        }
    }
}

console.log(processGrade(85));   // "B"
console.log(processGrade("A"));  // 90
```

Typed Arrays and Objects

Typed Arrays

```
// Simple typed array
let numbers: number[] = [1, 2, 3, 4, 5];
// numbers.push("six");  // Error!

// Array of objects
interface Task {
    id: number;
    title: string;
    completed: boolean;
}

let tasks: Task[] = [
    { id: 1, title: "Learn TypeScript", completed: false },
    { id: 2, title: "Build an app", completed: false }
];
```

ReadonlyArray - can't be modified

```
let readOnlyScores: ReadonlyArray<number> = [95, 87, 92];
// readOnlyScores.push(88);  // Error!
// readOnlyScores[0] = 96;   // Error!
```

Typed Object Methods

```
interface StudentRecord {
    name: string;
    grades: number[];

    // Method signatures
    addGrade(grade: number): void;
    getAverage(): number;
}

class StudentImpl implements StudentRecord {
    name: string;
    grades: number[];

    constructor(name: string) {
        this.name = name;
        this.grades = [];
    }

    addGrade(grade: number): void {
        if (grade >= 0 && grade <= 100) {
            this.grades.push(grade);
        } else {
            throw new Error("Grade must be between 0 and 100");
        }
    }

    getAverage(): number {
        if (this.grades.length === 0) return 0;
        const sum = this.grades.reduce((a, b) => a + b, 0);
        return sum / this.grades.length;
    }
}
```

Generic Types

```
// Generic function
function getFirstElement<T>(array: T[]): T | undefined {
    return array[0];
}

const firstNumber = getFirstElement([1, 2, 3]);
// returns 1
const firstName = getFirstElement(["Alice", "Bob"]);
// returns "Alice"
```

```typescript
// Generic interface
interface Container<T> {
    value: T;
    getValue(): T;
    setValue(value: T): void;
}

class NumberContainer implements Container<number> {
    value: number;

    constructor(value: number) {
        this.value = value;
    }

    getValue(): number {
        return this.value;
    }

    setValue(value: number): void {
        this.value = value;
    }
}
```

Control Flow with Types

Type Guards

```
// typeof type guard
function processValue(value: string | number) {
    if (typeof value === "string") {
        // TypeScript knows value is string here
        console.log(value.toUpperCase());
    } else {
        // TypeScript knows value is number here
        console.log(value.toFixed(2));
    }
}
```

```
// instanceof type guard
class Student {
    name: string;
    constructor(name: string) {
        this.name = name;
    }
}

class Teacher {
    subject: string;
    constructor(subject: string) {
        this.subject = subject;
    }
}

function greetPerson(person: Student | Teacher) {
    if (person instanceof Student) {
        console.log(`Hello student ${person.name}`);
    } else {
        console.log(`Hello teacher of ${person.subject}`);
    }
}
```

Discriminated Unions

```
// Common pattern for handling different cases
interface SuccessResult {
    type: "success";
    data: any;
}

interface ErrorResult {
    type: "error";
    message: string;
}

type Result = SuccessResult | ErrorResult;

function handleResult(result: Result) {
    switch (result.type) {
        case "success":
            console.log("Data:", result.data);
            break;
        case "error":
            console.error("Error:", result.message);
            break;
    }
}
```

Null and Undefined Handling

```
// strictNullChecks helps catch null errors
interface User {
    name: string;
    email?: string;  // Could be undefined
}

function sendEmail(user: User) {
    // TypeScript error: Object is possibly 'undefined'
    // console.log(user.email.toLowerCase());

    // Correct way - check first
    if (user.email) {
        console.log(user.email.toLowerCase());
    }

    // Or use optional chaining
    console.log(user.email?.toLowerCase());
}

// Non-null assertion (use carefully!)
function processUser(user: User) {
    // Tell TypeScript you know email exists
    const email = user.email!;  // Use only when you're certain
    console.log(email.toLowerCase());
}
```

DOM Manipulation with Types

Type-Safe DOM Selection

```
// TypeScript knows the element types
const button = document.getElementById('submit-btn') as HTMLButtonElement;
const input = document.querySelector('#name-input') as HTMLInputElement;
const form = document.querySelector('form') as HTMLFormElement;

// Now TypeScript provides proper autocomplete
button.disabled = true;
input.value = "TypeScript";
form.submit();

// Generic querySelector with type
function querySelector<T extends Element>(selector: string): T | null {
    return document.querySelector(selector) as T | null;
}

const myDiv = querySelector<HTMLDivElement>('#my-div');
if (myDiv) {
    myDiv.style.backgroundColor = 'blue';
}
```

Type-Safe Event Handling

```
// Typed event listeners
const button = document.querySelector('#click-me') as HTMLButtonElement;

button.addEventListener('click', (event: MouseEvent) => {
    console.log(`Clicked at: ${event.clientX}, ${event.clientY}`);

    // TypeScript knows event.target might be null
    const target = event.target as HTMLButtonElement;
    target.disabled = true;
});
```

```typescript
// Form events
const form = document.querySelector('#user-form') as HTMLFormElement;

form.addEventListener('submit', (event: Event) => {
    event.preventDefault();

    const formData = new FormData(form);
    const name = formData.get('name') as string;
    const age = parseInt(formData.get('age') as string);

    console.log({ name, age });
});

// Input events with proper types
const input = document.querySelector('#search') as HTMLInputElement;

input.addEventListener('input', (event: Event) => {
    const target = event.target as HTMLInputElement;
    const value: string = target.value;

    if (value.length < 3) {
        target.classList.add('error');
    } else {
        target.classList.remove('error');
    }
});
```

Creating Elements with Types

```
// Type-safe element creation
function createTaskElement(task: Task): HTMLLIElement {
    const li = document.createElement('li');
    li.className = task.completed ? 'completed' : 'pending';

    const checkbox = document.createElement('input');
    checkbox.type = 'checkbox';
    checkbox.checked = task.completed;

    const label = document.createElement('label');
    label.textContent = task.title;

    li.appendChild(checkbox);
    li.appendChild(label);

    return li;
}

// Using the function
const taskList = document.querySelector('#task-list') as HTMLUListElement;
const newTask: Task = { id: 1, title: "Learn TypeScript", completed: false };
const taskElement = createTaskElement(newTask);
taskList.appendChild(taskElement);
```

Advanced TypeScript Features

Type Assertions and Casting

```
// Type assertion - telling TypeScript what type something is
let someValue: any = "this is a string";
let strLength: number = (someValue as string).length;

// Alternative (old) syntax (not recommended in .tsx files)
let strLength2: number = (<string>someValue).length;

// Const assertions
const config = {
    endpoint: "https://api.example.com",
    timeout: 5000
} as const;  // Makes all properties readonly

// config.endpoint = "new";  // Error!
```

Literal Types

```
// String literal types
type Direction = "north" | "south" | "east" | "west";

function move(direction: Direction) {
    console.log(`Moving ${direction}`);
}

move("north");    // OK
// move("up");    // Error!

// Numeric literal types
type DiceRoll = 1 | 2 | 3 | 4 | 5 | 6;

function rollDice(): DiceRoll {
    return Math.floor(Math.random() * 6 + 1) as DiceRoll;
}
```

Mapped Types

We may have all properties can be an optional.

```
{ name?: string; age?: number; email?: string; }
```

In this case, we can use Mapped Types

```
// Make all properties optional
type Partial<T> = {
    [P in keyof T]?: T[P];
};
```

1. `keyof T` : gets all the property names of T.
2. `P in keyof T` : means "for each property P inside T".
3. `?:` : makes each property optional.
4. `T[P]` : means "use the same type as the original property".

```typescript
interface Student {
    name: string;
    age: number;
    email: string;
}

type PartialStudent = Partial<Student>;
// Same as: { name?: string; age?: number; email?: string; }

// Make all properties readonly
type Readonly<T> = {
    readonly [P in keyof T]: T[P];
};
```

We can create type with specific keys:

For example, when we want to only select "name" and "email" from Student interface, we can do

```
type Pick<T, K extends keyof T> = {
    [P in K]: T[P];
};

type StudentName = Pick<Student, "name" | "email">;
// Same as: { name: string; email: string; }
```

Utility Types

```
// Record - create object type with specific key/value types
// An object whose keys are strings and whose values are numbers
type Grades = Record<string, number>;
const courseGrades: Grades = {
    "Math": 85,
    "Science": 92,
    "English": 88
};

// Exclude - remove types from union
type Grade = "A" | "B" | "C" | "D" | "F";
type PassingGrade = Exclude<Grade, "F">;  // "A" | "B" | "C" | "D"

// Extract - keep only specified types
type HighGrade = Extract<Grade, "A" | "B">;  // "A" | "B"

// NonNullable - remove null and undefined
type MaybeString = string | null | undefined;
type DefiniteString = NonNullable<MaybeString>;  // string
```

Decorators

A JavaScript **decorator** is a special function that can **wrap or modify**:

- It's like adding "extra behavior" to a method
- *without changing the method's original code*.

Example use:

- Logging
- Validation
- Timing
- Security checks

```
function LogMethod(target: any,
  propertyKey: string, // name of the method
  descriptor: PropertyDescriptor) {
  const original = descriptor.value;

  // decorator code here
  descriptor.value = function(...args: any[]) {
    console.log(`Calling ${propertyKey} with args:`, args);

    // original method used here
    const result = original.apply(this, args);
    console.log(`Result:`, result);
    return result;
  };

  return descriptor;
}

class Calculator {
    @LogMethod
    add(a: number, b: number): number {
        return a + b;
    }
}
```

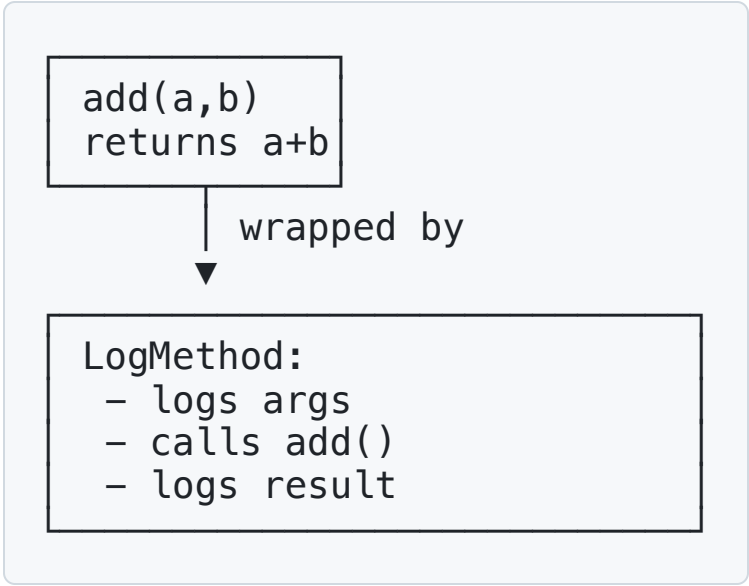
Usage

```
const calc = new Calculator();
calc.add(2, 3);

> Calling add with args: [2, 3]
> Result: 5
```

How it works?

1. @LogMethod is attached to add()
2. When add() is called:
    - Logs arguments
    - Calls original method
    - Logs result



Building Your First TypeScript App

Let's rebuild our Grade Calculator with full type safety:

Run the run.sh script to build

Make sure tsc is installed: `npm install -g typescript` .

```
tsc src/app.ts --outDir . --target ES2018
```

The generated js file is accessed in the HTML.

```
  <script src="../app.js"></script>
```

Best Practices

1. Use Strict Mode

```
// tsconfig.json
{
  "compilerOptions": {
    "strict": true  // Enables all strict type checking options
  }
}
```

2. Avoid `any` Type

```
// Bad - loses all type safety
function processData(data: any) {
    return data.value;  // No type checking!
}

// Good - use specific types or generics
function processData<T extends { value: string }>(data: T) {
    return data.value;  // Type safe!
}
```

3. Use Interfaces for Object Shapes

```
// Bad - inline type annotations
function createUser(data: { name: string; email: string; age: number }) {
    // ...
}

// Good - reusable interface
interface UserData {
    name: string;
    email: string;
    age: number;
}

function createUser(data: UserData) {
    // ...
}
```

4. Prefer `const` Assertions for Literals

```
// Without const assertion
const config = {
    api: "https://api.example.com",
    timeout: 5000
};
// config.api is string, can be changed

// With const assertion
const config = {
    api: "https://api.example.com",
    timeout: 5000
} as const;
// config.api is "https://api.example.com" literal type, readonly
```

5. Use Union Types Instead of Enums

```
// Enum (okay)
enum Status {
    Pending = "PENDING",
    Active = "ACTIVE",
    Completed = "COMPLETED"
}

// Union type (often better)
type Status = "PENDING" | "ACTIVE" | "COMPLETED";

// Benefits: Smaller bundle size, works better with type narrowing
```

6. Leverage Type Inference

```
// Over-annotated
const numbers: number[] = [1, 2, 3];
const doubled: number[] = numbers.map((n: number): number => n * 2);

// Let TypeScript infer
const numbers = [1, 2, 3];  // TypeScript knows it's number[]
const doubled = numbers.map(n => n * 2);  // TypeScript infers everything
```

7. Handle Null/Undefined Explicitly

```
// Bad - might crash
function getLength(str: string | null) {
    return str.length;  // Error: Object is possibly 'null'
}

// Good - explicit handling
function getLength(str: string | null): number {
    return str ? str.length : 0;
}

// Or use nullish coalescing
function getLength(str: string | null): number {
    return (str ?? '').length;
}
```

Summary

TypeScript transforms JavaScript development by adding a powerful type system that catches errors early, improves code documentation, and enables better tooling support. Key takeaways:

1. **Type Safety** - Catch errors at compile time, not runtime
2. **Better IDE Support** - Autocomplete, refactoring, and navigation
3. **Self-Documenting** - Types serve as inline documentation
4. **Scalability** - Easier to maintain large codebases
5. **Gradual Adoption** - Start with JavaScript, add types incrementally

Remember Sarah's journey:

- **JavaScript**: Fast to write, but runtime errors in production
- **TypeScript**: Slightly more code, but 90% fewer bugs!

TypeScript is not just about types—it's about building more reliable, maintainable software. As Sarah discovered, the initial learning investment pays off tremendously in reduced debugging time and increased confidence in your code!

The Grade Calculator example shows how TypeScript:

- Prevents type-related bugs
- Makes code more maintainable
- Provides confidence when refactoring
- Improves the development experience