From JavaScript to TypeScript: The Type Safety Evolution

JavaScript's Growing Pains:

- Fast development but runtime errors
- No compile-time type checking
- Refactoring is risky in large codebases
- Hard to maintain 5000+ line projects
- 90% of production bugs are type-related

TypeScript's Solution:

- Superset of JavaScript (all JS is valid TS)
- Static type checking at compile time
- Catches errors before code runs
- Better IDE support (autocomplete, refactoring)

What is TypeScript?

TypeScript is a strongly-typed programming language developed by Microsoft (2012) that builds on JavaScript by adding static type definitions.

Key Characteristics:

- **Transpiles to JavaScript:** TypeScript code compiles to plain JavaScript
- **Gradual typing:** Can adopt types incrementally
- **Type inference:** Types often inferred automatically
- **Modern features:** Includes latest ECMAScript features
- **Tooling support:** Excellent IDE integration

Market Reality (2025):

- Used by 78% of professional JavaScript developers
- Powers major projects: VSCode, Slack, Airbnb

TypeScript Basics: Type Annotations

Basic Types:

```typescript
let username: string = "Sarah";
let age: number = 25;
let isActive: boolean = true;
let values: number[] = [1, 2, 3];
let tuple: [string, number] = ["Sarah", 25];

// Type inference (TypeScript guesses the type)
let message = "Hello";  // inferred as string
```

Compare to JavaScript:

```typescript
// JavaScript - no type safety
let age = 25;
age = "twenty-five";  // No error, but likely a bug

// TypeScript - compile error
let age: number = 25;
```

Functions with Types

Function signatures:

```
function greet(name: string): string {
    return `Hello, ${name}`;
}

// Arrow function
const add = (a: number, b: number): number => a + b;

// Optional parameters
function log(message: string, level?: string): void {
    console.log(`[${level || "INFO"}] ${message}`);
}

// Default parameters
function createUser(name: string, role: string = "user") {
    return {name, role};
}

// Rest parameters
function sum(...numbers: number[]): number {
    return numbers.reduce((acc, n) => acc + n, 0);
```

Interfaces: Defining Object Shapes

Interface definition:

```typescript
interface User {
    id: number;
    name: string;
    email: string;
    age?: number;          // Optional property
    readonly created: Date; // Cannot be modified
}

function createUser(data: User): void {
    console.log(`Creating user: ${data.name}`);
    // data.created = new Date(); // ERROR: readonly property
}

const newUser: User = {
    id: 1,
    name: "Sarah",
    email: "sarah@example.com",
    created: new Date()
```

Interfaces vs Type Aliases

Type Aliases:

```
type ID = number | string;
type Status = "pending" | "active" | "inactive";

type User = {
    id: ID;
    name: string;
    status: Status;
};
```

When to use what:

- **Interfaces:** Use for object shapes, especially when extending
- **Type aliases:** Use for unions, primitives, tuples

Extending interfaces:

Union Types and Type Guards

Union Types:

```
type Result = string | number;
type Response = {success: true; data: User} | {success: false; error: string};

function processId(id: string | number): void {
    if (typeof id === "string") {
        console.log(id.toUpperCase());  // TypeScript knows it's string here
    } else {
        console.log(id.toFixed(2));     // TypeScript knows it's number here
    }
}
```

Type Guards:

```
function isUser(obj: any): obj is User {
    return obj && typeof obj.name === "string";
}
```

Generics: Reusable Type-Safe Code

Generic functions:

```
function identity<T>(value: T): T {
    return value;
}

const num = identity<number>(42);     // Returns number
const str = identity<string>("hello"); // Returns string

// Array utilities
function first<T>(arr: T[]): T | undefined {
    return arr[0];
}

const firstNum = first([1, 2, 3]);       // number | undefined
const firstStr = first(["a", "b", "c"]); // string | undefined
```

Generic interfaces:

Classes in TypeScript

Class with types:

```
class User {
    private id: number;
    public name: string;
    protected email: string;

    constructor(id: number, name: string, email: string) {
        this.id = id;
        this.name = name;
        this.email = email;
    }

    public introduce(): string {
        return `Hi, I'm ${this.name}`;
    }

    private validateEmail(): boolean {
        return this.email.includes("@");
    }
}

const user = new User(1, "Sarah", "sarah@example.com");
console.log(user.name);       // OK
```

Practical TypeScript Example

API Response Types:

```
interface ApiResponse<T> {
    status: number;
    data?: T;
    error?: string;
}

interface Post {
    id: number;
    title: string;
    content: string;
    author: string;
    created: Date;
}

async function fetchPost(id: number): Promise<ApiResponse<Post>> {
    try {
        const response = await fetch(`/api/posts/${id}`);
        const post = await response.json();
        return {status: 200, data: post};
    } catch (error) {
        return {status: 500, error: error.message};
    }
}

// Usage with type safety
const result = await fetchPost(1);
if (result.data) {
```

TypeScript Configuration

tsconfig.json:

```json
{
    "compilerOptions": {
        "target": "ES2020",
        "module": "commonjs",
        "strict": true,
        "esModuleInterop": true,
        "skipLibCheck": true,
        "forceConsistentCasingInFileNames": true,
        "outDir": "./dist",
        "rootDir": "./src"
    },
    "include": ["src/**/*"],
    "exclude": ["node_modules"]
}
```

Strict mode benefits:

Introduction to React

What is React?

React is a JavaScript library for building user interfaces, created by Facebook (2013). It focuses on building reusable UI components with declarative syntax.

Core Concepts:

- **Components:** Reusable UI pieces
- **JSX:** HTML-like syntax in JavaScript
- **State:** Component data that changes
- **Props:** Data passed to components
- **Virtual DOM:** Efficient rendering

Why React?

- **Component-based:** Build complex UIs from simple pieces

React Components

Functional Components:

```
import React from 'react';

interface GreetingProps {
    name: string;
    age?: number;
}

function Greeting({name, age}: GreetingProps) {
    return (
        <div>
            <h1>Hello, {name}!</h1>
            {age && <p>Age: {age}</p>}
        </div>
    );
}

// Usage
<Greeting name="Sarah" age={25} />
```

React State with useState

State management:

```
import React, {useState} from 'react';

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

    return (
        <div>
            <p>Count: {count}</p>
            <button onClick={() => setCount(count + 1)}>
                Increment
            </button>
        </div>
    );
}
```

State rules:

React Props: Passing Data

Props flow:

```tsx
interface UserCardProps {
    user: {
        name: string;
        email: string;
        avatar: string;
    };
    onEdit: (id: number) => void;
}

function UserCard({user, onEdit}: UserCardProps) {
    return (
        <div className="card">
            <img src={user.avatar} alt={user.name} />
            <h3>{user.name}</h3>
            <p>{user.email}</p>
            <button onClick={() => onEdit(user.id)}>Edit</button>
        </div>
    );
}

// Parent component
function UserList() {
    const handleEdit = (id: number) => {
        console.log(`Editing user ${id}`);
    };
```

React Effects with useEffect

Side effects:

```
import React, {useState, useEffect} from 'react';

function UserProfile({userId}: {userId: number}) {
    const [user, setUser] = useState<User | null>(null);
    const [loading, setLoading] = useState(true);

    useEffect(() => {
        // Runs after component mounts and when userId changes
        fetch(`/api/users/${userId}`)
            .then(res => res.json())
            .then(data => {
                setUser(data);
                setLoading(false);
            });
    }, [userId]);  // Dependency array

    if (loading) return <div>Loading...</div>;
    if (!user) return <div>User not found</div>;

    return <div>{user.name}</div>;
```

The Virtual DOM

How React Optimizes Performance:

Traditional DOM Updates:

1. User action triggers change
2. Directly manipulate DOM (expensive)
3. Browser reflows/repaints entire page

React's Virtual DOM:

1. User action triggers state change
2. React creates new virtual DOM tree (fast, in memory)
3. React diffs old and new virtual trees
4. React updates only changed parts of real DOM

React Example: Todo List

```
import React, {useState} from 'react';

interface Todo {
    id: number;
    text: string;
    completed: boolean;
}

function TodoList() {
    const [todos, setTodos] = useState<Todo[]>([]);
    const [input, setInput] = useState("");

    const addTodo = () => {
        if (input.trim()) {
            setTodos([...todos, {
                id: Date.now(),
                text: input,
                completed: false
            }]);
            setInput("");
        }
    };

    const toggleTodo = (id: number) => {
        setTodos(todos.map(todo =>
            todo.id === id ? {...todo, completed: !todo.completed} : todo
        ));
    };

    return (
        <div>
            <input
                value={input}
                onChange={(e) => setInput(e.target.value)}
                placeholder="Enter task"
            />
            <button onClick={addTodo}>Add</button>

            <ul>
                {todos.map(todo => (
                    <li
                        key={todo.id}
                        onClick={() => toggleTodo(todo.id)}
                        style={{
                            textDecoration: todo.completed ? 'line-through' : 'none'
                        }}
                    >
                        {todo.text}
                    </li>
                ))}
            </ul>
        </div>
    );
}
```

React Component Lifecycle

Lifecycle with Hooks:

```
function DataFetcher() {
    const [data, setData] = useState(null);

    useEffect(() => {
        // Component mounted
        console.log("Component mounted");

        // Fetch data
        fetchData().then(setData);

        // Cleanup function (component unmounting)
        return () => {
            console.log("Component will unmount");
        };
    }, []);  // Empty array = run once on mount

    useEffect(() => {
        // Runs on every render
        console.log("Component rendered");
    });

    useEffect(() => {
        // Runs when data changes
        console.log("Data updated:", data);
    }, [data]);
```

React Context: Global State

Avoiding prop drilling:

```
import React, {createContext, useContext, useState} from 'react';

interface ThemeContextType {
    theme: string;
    toggleTheme: () => void;
}

const ThemeContext = createContext<ThemeContextType | undefined>(undefined);

function ThemeProvider({children}: {children: React.ReactNode}) {
    const [theme, setTheme] = useState("light");

    const toggleTheme = () => {
        setTheme(prev => prev === "light" ? "dark" : "light");
    };

    return (
        <ThemeContext.Provider value={{theme, toggleTheme}}>
            {children}
        </ThemeContext.Provider>
    );
}

// Use in any component
function ThemedButton() {
    const context = useContext(ThemeContext);
    if (!context) throw new Error("Must be inside ThemeProvider");

    return (
        <button onClick={context.toggleTheme}>
            Current: {context.theme}
        </button>
```

React Custom Hooks

Reusable logic:

```
function useFetch<T>(url: string) {
    const [data, setData] = useState<T | null>(null);
    const [loading, setLoading] = useState(true);
    const [error, setError] = useState<string | null>(null);

    useEffect(() => {
        fetch(url)
            .then(res => res.json())
            .then(data => {
                setData(data);
                setLoading(false);
            })
            .catch(err => {
                setError(err.message);
                setLoading(false);
            });
    }, [url]);

    return {data, loading, error};
}

// Usage
function UserProfile({userId}: {userId: number}) {
    const {data: user, loading, error} = useFetch<User>(`/api/users/${userId}`);

    if (loading) return <div>Loading...</div>;
    if (error) return <div>Error: {error}</div>;
```

React Performance Optimization

React.memo (prevent unnecessary re-renders):

```
interface ExpensiveComponentProps {
    data: string[];
}

const ExpensiveComponent = React.memo(({data}: ExpensiveComponentProps) => {
    console.log("Rendering expensive component");
    return (
        <ul>
            {data.map((item, i) => <li key={i}>{item}</li>)}
        </ul>
    );
});
```

useMemo (cache expensive calculations):

```
function DataProcessor({items}: {items: number[]}) {
    const expensiveResult = useMemo(() => {
```

React + TypeScript Best Practices

1. Type all props:

```
interface Props {
    title: string;
    count: number;
    onUpdate: (value: number) => void;
}

function Component({title, count, onUpdate}: Props) { }
```

2. Use discriminated unions for state:

```
type State =
    | {status: "loading"}
    | {status: "error"; error: string}
    | {status: "success"; data: User[]};
```

3. Type event handlers:

React Ecosystem

State Management:

- **Redux** - Centralized state management
- **MobX** - Observable state
- **Zustand** - Lightweight alternative
- **React Query** - Server state management

Routing:

- **React Router** - Client-side routing for SPAs

UI Libraries:

- **Material-UI (MUI)** - Google's Material Design
- **Chakra UI** - Accessible components
- **Tailwind CSS** - Utility-first CSS

React Development Setup

Create a new project:

```
# With Vite (recommended)
npm create vite@latest my-app -- --template react-ts
cd my-app
npm install
npm run dev

# With Create React App
npx create-react-app my-app --template typescript
cd my-app
npm start
```

Project structure:

```
my-app/
├── src/
│   ├── components/
│   │   ├── Button.tsx
```

Testing React Components

Using React Testing Library:

```
import {render, screen, fireEvent} from '@testing-library/react';
import Counter from './Counter';

test('increments counter', () => {
    render(<Counter />);

    const button = screen.getByText('Increment');
    const count = screen.getByText(/Count: 0/);

    expect(count).toBeInTheDocument();

    fireEvent.click(button);

    expect(screen.getByText(/Count: 1/)).toBeInTheDocument();
});
```

Test best practices:

Declarative vs Imperative UI

Imperative (jQuery style):

```
// Tell the browser HOW to do it
const button = document.getElementById("myButton");
button.addEventListener("click", () => {
    const counter = document.getElementById("counter");
    const count = parseInt(counter.textContent);
    counter.textContent = count + 1;
    if (count + 1 >= 10) {
        button.disabled = true;
    }
});
```

Declarative (React style):

```
// Tell React WHAT you want
function Counter() {
    const [count, setCount] = useState(0);
```

The Three Pillars of Modern Web Development

1. **High-Level Language (JavaScript)**
    - Productivity: 100x faster than C
    - Rich ecosystem and tooling
    - But: No type safety
2. **Type System (TypeScript)**
    - Static type checking
    - 90% fewer bugs
    - Better IDE support
    - But: Still needs better UI management
3. **Framework (React)**

Real-World Impact

Development Speed:

- JavaScript: 100x faster than C for web tasks
- TypeScript: Prevents 90% of production bugs
- React: 3x faster UI development

Code Maintainability:

- JavaScript: Easy to write, hard to maintain at scale
- TypeScript: Scales to millions of lines
- React: Component isolation simplifies changes

Team Collaboration:

- TypeScript: Self-documenting with types

Common Pitfalls and Solutions

TypeScript:

- **Pitfall:** Using `any` everywhere
- **Solution:** Enable strict mode, use proper types

React:

- **Pitfall:** Mutating state directly
- **Solution:** Always use setter functions, spread syntax

Both:

- **Pitfall:** Ignoring warnings
- **Solution:** Fix warnings immediately, they prevent bugs

Performance:

From Week 5 to Week 6: The Evolution

Week 5 (JavaScript):

- Solved complexity problem from C
- Fast development, flexible
- But: Type errors, hard to maintain

Week 6 (TypeScript + React):

- TypeScript: Added type safety to JavaScript
- React: Solved UI state management
- Result: Fast, safe, maintainable

The Journey:

1. C (500 lines, 1 week) → JavaScript (20 lines, 2 hours)

Key Takeaways

TypeScript:

- Static typing prevents runtime errors
- Gradual adoption - use as much or as little as needed
- Essential for large codebases
- Better tooling and refactoring support

React:

- Component-based architecture promotes reusability
- Declarative syntax simplifies complex UIs
- Virtual DOM provides automatic optimization
- Large ecosystem and community

Together: