JavaScript vs TypeScript

Overview

JavaScript

- Dynamic, interpreted programming language
- Foundation of web development
- Runtime type checking

TypeScript

- Superset of JavaScript
- Static type checking
- Compiles down to regular JavaScript

1. Type Annotations

JavaScript - No Type Checking

```
// No type checking - this will run but may cause runtime errors
function greet(name) {
    return "Hello, " + name.toUpperCase();
}

greet("Alice");      // Works fine
greet(123);          // Runtime error: name.toUpperCase is not a function
greet();             // Runtime error: Cannot read property 'toUpperCase' of undefined
```

TypeScript - Compile-time Safety

```typescript
// Explicit type annotations prevent errors at compile time
function greet(name: string): string {
    return "Hello, " + name.toUpperCase();
}

greet("Alice");       // ✅ Works fine
greet(123);           // ❌ Compile error: Argument of type 'number' is not assignable...
greet();              // ❌ Compile error: Expected 1 arguments, but got 0
```

2. Variable Type Checking

JavaScript - Dynamic Typing

```
// Variables can change types freely
let count = 42;
count = "forty-two";        // No problem
count = true;               // Also fine
count = { value: 42 };      // Still works

console.log(count.value); // Works, but could break if count isn't an object
```

TypeScript - Static Typing

```
// Type is inferred from initial assignment
let count = 42;         // TypeScript infers count is a number
count = "forty-two";    // ❌ Error: Type 'string' is not assignable to type 'number'

// Or you can explicitly declare types
let message: string = "Hello";
let isActive: boolean = true;
let score: number = 95;
```

3. Object Structure Validation

JavaScript - No Structure Enforcement

```javascript
// No structure enforcement
function processUser(user) {
    console.log(`Name: ${user.name}, Age: ${user.age}`);
    // What if user doesn't have these properties? Runtime error!
}

processUser({ name: "John" });           // Missing age - undefined
processUser({ firstName: "Jane" });      // Wrong property name - undefined
processUser("not an object");            // Runtime error
```

TypeScript - Interface-based Validation

```typescript
// Define object structure with interfaces
interface User {
    name: string;
    age: number;
    email?: string;  // Optional property
}

function processUser(user: User): void {
    console.log(`Name: ${user.name}, Age: ${user.age}`);
}

processUser({ name: "John", age: 25 });              // ✅ Works
processUser({ name: "Jane" });                       // ❌ Error: Property 'age' is missing
processUser({ firstName: "Bob", age: 30 });          // ❌ Error: Object literal may only specify known properties
```

4. Arrays and Generics

JavaScript - Mixed Types Allowed

```javascript
// Arrays can contain mixed types
const mixedArray = [1, "hello", true, { id: 1 }];
mixedArray.push("another string");
mixedArray.push(999);

// No way to enforce array content types
function getFirstItem(items) {
    return items[0];
}
```

TypeScript - Type-safe Arrays & Generics

```
// Type-safe arrays
const numbers: number[] = [1, 2, 3, 4];
const strings: string[] = ["hello", "world"];

numbers.push(5);        // ✅ Works
numbers.push("text");   // ❌ Error: Argument of type 'string' is not assignable to parameter of type 'number'

// Generic functions for type safety
function getFirstItem<T>(items: T[]): T | undefined {
    return items[0];
}

const firstNumber = getFirstItem([1, 2, 3]);      // TypeScript knows this is number | undefined
const firstString = getFirstItem(["a", "b"]);    // TypeScript knows this is string | undefined
```

5. Class Definitions and Access Modifiers

```
// ES6 classes - no built-in access control
class BankAccount {
    constructor(balance) {
        this.balance = balance;  // Anyone can access this
    }

    deposit(amount) {
        this.balance += amount;
    }

    withdraw(amount) {
        if (amount <= this.balance) {
            this.balance -= amount;
            return true;
        }
        return false;
    }
}

const account = new BankAccount(1000);
account.balance = -999999;  // Oops! Direct manipulation possible
```

JavaScript vs TypeScript

```typescript
// Classes with access modifiers and type safety
class BankAccount {
    private balance: number;           // Private - cannot be accessed from outside
    public readonly accountId: string; // Public and readonly

    constructor(initialBalance: number, accountId: string) {
        this.balance = initialBalance;
        this.accountId = accountId;
    }

    public deposit(amount: number): void {
        if (amount <= 0) {
            throw new Error("Amount must be positive");
        }
        this.balance += amount;
    }

    public withdraw(amount: number): boolean {
        if (amount <= 0 || amount > this.balance) {
            return false;
        }
        this.balance -= amount;
        return true;
    }

    public getBalance(): number {
        return this.balance;
    }
}

const account = new BankAccount(1000, "ACC123");
account.balance = -999;     // ❌ Error: Property 'balance' is private
account.deposit("50");      // ❌ Error: Argument of type 'string' is not assignable to parameter of type 'number'
```

6. Error Catching at Development Time

JavaScript - Runtime Bug Discovery

```javascript
// This code looks fine but has a subtle bug
function calculateTotal(items) {
    let total = 0;
    for (let item of items) {
        total += item.price * item.quantity;
    }
    return total;
}

// Bug only discovered at runtime
const cart = [
    { price: 10, qty: 2 },      // Oops! Should be 'quantity', not 'qty'
    { price: 15, quantity: 1 }
];

console.log(calculateTotal(cart)); // NaN - hard to debug
```

TypeScript - Compile-time Error Detection

```typescript
// Interface catches the error before runtime
interface CartItem {
    price: number;
    quantity: number;
}

function calculateTotal(items: CartItem[]): number {
    let total = 0;
    for (let item of items) {
        total += item.price * item.quantity;
    }
    return total;
}

const cart: CartItem[] = [
    { price: 10, qty: 2 },      // ❌ Error: Object literal may only specify known properties
    { price: 15, quantity: 1 }
];
```

Key Takeaways

Aspect	JavaScript	TypeScript
Type Safety	Runtime errors	Compile-time error catching
Learning Curve	Easier to start	Requires understanding types
Development Speed	Fast prototyping	Slower initial setup, faster debugging

Aspect	JavaScript	TypeScript
Tooling Support	Good	Excellent (autocomplete, refactoring)
File Extension	`.js`	`.ts` (compiles to `.js`)
Browser Support	Native	Requires compilation

Thank You!

Questions?