🎉 Introduction to TypeScript

A friendly guide for JavaScript developers

?? What is TypeScript?

- A **superset of JavaScript**
- Adds **static types** ✅
- Developed by **Microsoft**
- Compiles to **plain JavaScript**
- Catches **errors early**

❓❓ Why Use TypeScript?

- **Type safety**: Find bugs before runtime
- **Better IDE support**: Autocomplete & refactoring
- **Readable & maintainable code**
- Popular in **large projects**

?? Basic Types

```
let isDone: boolean = false;
let age: number = 25;
let name: string = "Alice";
let numbers: number[] = [1, 2, 3];
let tuple: [string, number] = ["Alice", 30];
let anyValue: any = "can be anything";
let unknownValue: unknown = "safer than any";
let neverValue: never; // function that never returns
```

- `boolean` → true/false
- `number` → integers & floats
- `string` → text
- `array` → list of values
- `tuple` → fixed-length array
- `any` → opt out of type checking

⚙️ Functions with Types

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

// Arrow functions
const multiply = (a: number, b: number): number => a * b;

// Function types
type MathFunc = (x: number, y: number) => number;
const add: MathFunc = (a, b) => a + b;

console.log(greet("Bob"));
console.log(multiply(5, 3));
console.log(add(10, 20));
```

- Parameters have **types**
- Functions can have **return types**

?? Interfaces

```typescript
interface Person {
  name: string;
  age: number;
  email?: string; // optional property
  readonly id: number; // read-only property
}

interface Employee extends Person {
  department: string;
  salary: number;
}

let user: Person = { name: "Alice", age: 30, id: 1 };
let employee: Employee = {
  name: "Bob",
  age: 25,
  id: 2,
  department: "IT",
  salary: 50000,
};
```

?? Type Aliases

```
type Point = {
  x: number;
  y: number;
};

type Status = "loading" | "success" | "error"; // union type
type ID = string | number; // union type

type Coordinates = Point & { z: number }; // intersection type

let position: Point = { x: 10, y: 20 };
let status: Status = "loading";
let id: ID = "user123";
let coords: Coordinates = { x: 10, y: 20, z: 30 };
```

- **Type aliases** for complex types
- **Union types** with `|`

✨ Optional & Default Parameters

```
function greet(name: string, age?: number, greeting: string = "Hello") {
  console.log(`${greeting}, ${name}, age ${age ?? "unknown"}`);
}

// Rest parameters
function sum(...numbers: number[]): number {
  return numbers.reduce((total, num) => total + num, 0);
}

greet("Alice"); // Hello, Alice, age unknown
greet("Bob", 25); // Hello, Bob, age 25
greet("Charlie", 30, "Hi"); // Hi, Charlie, age 30
console.log(sum(1, 2, 3, 4, 5)); // 15
```

- `?` → optional parameter
- `=` → default value
- **Rest parameters** with `...`

🔍 Type Inference

```
let message = "Hello TypeScript"; // inferred as string
let numbers = [1, 2, 3]; // inferred as number[]
let mixed = [1, "hello", true]; // inferred as (number | string | boolean)[]

// Contextual typing
const names = ["Alice", "Bob", "Charlie"];
names.forEach((name) => console.log(name.toUpperCase())); // name inferred as string
```

- TypeScript can **guess types**
- **Contextual typing** in callbacks
- Explicit typing is optional

🥪 Generics

```typescript
// Generic function
function identity<T>(arg: T): T {
  return arg;
}

// Generic interface
interface Container<T> {
  value: T;
  getValue(): T;
}

// Generic class
class Stack<T> {
  private items: T[] = [];

  push(item: T): void {
    this.items.push(item);
  }

  pop(): T | undefined {
    return this.items.pop();
  }
}

let stringStack = new Stack<string>();
stringStack.push("hello");
stringStack.push("world");
console.log(stringStack.pop()); // "world"
```

🔒 Type Guards & Narrowing

```ts
function isString(value: unknown): value is string {
  return typeof value === "string";
}

function processValue(value: string | number) {
  if (typeof value === "string") {
    console.log(value.toUpperCase()); // TypeScript knows it's a string
  } else {
    console.log(value.toFixed(2)); // TypeScript knows it's a number
  }
}

// Custom type guard
interface User {
  name: string;
  age: number;
}

function isUser(obj: unknown): obj is User {
  return (
    typeof obj === "object" && obj !== null && "name" in obj && "age" in obj
  );
}
```

🎭 Enums

```
enum Color {
  Red = "RED",
  Green = "GREEN",
  Blue = "BLUE",
}

enum Status {
  Pending,
  Approved,
  Rejected,
}

let favoriteColor: Color = Color.Blue;
let currentStatus: Status = Status.Pending;

console.log(Color.Red); // "RED"
console.log(Status.Approved); // 1 (auto-incremented)
```

- **Enums** for named constants

 Classes

```
class Animal {
  protected name: string;

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

  move(distance: number = 0): void {
    console.log(`${this.name} moved ${distance}m.`);
  }
}

class Dog extends Animal {
  bark(): void {
    console.log(`${this.name} barks!`);
  }

  move(distance: number = 5): void {
    console.log(`${this.name} runs ${distance}m.`);
    super.move(distance);
  }
}

let dog = new Dog("Rex");
dog.bark(); // Rex barks!
dog.move(10); // Rex runs 10m.
```

?? Utility Types

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

// Make all properties optional
type PartialUser = Partial<User>;

// Make all properties required
type RequiredUser = Required<User>;

// Pick specific properties
type UserName = Pick<User, "name" | "email">;

// Omit specific properties
type UserWithoutId = Omit<User, "id">;

// Record for object with specific key/value types
type UserMap = Record<string, User>;

// ReturnType for function return type
type GreetReturn = ReturnType<typeof greet>;
```

🎨 Advanced Types

```typescript
// Conditional types
type NonNullable<T> = T extends null | undefined ? never : T;

// Mapped types
type Readonly<T> = {
  readonly [P in keyof T]: T[P];
};

// Template literal types
type EmailLocale = "en" | "es" | "fr";
type EmailTemplate = `welcome_${EmailLocale}`;

// Index access types
type UserName = User["name"];
type UserKeys = keyof User;

// Conditional types with inference
type ArrayElement<T> = T extends (infer U)[] ? U : never;
type StringArrayElement = ArrayElement<string[]>; // string
```

⚔️ TypeScript vs JavaScript

Feature	JavaScript	TypeScript
Static Types	❌	✅
Compile-time checks	❌	✅
Object-oriented features	❌	✅
IDE Autocomplete	Limited	Excellent
Generics	❌	✅
Interfaces	❌	✅
Enums	❌	✅
Utility Types	❌	✅

?? Getting Started

1. Install Node.js (includes npm)
2. Install TypeScript:

```
npm install -g typescript
```

3. Create tsconfig.json:

```
tsc --init
```

4. Compile a file:

```
tsc app.ts
```

5. Run JavaScript:

```
node app.js
```

📁 tsconfig.json Essentials

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

- **target**: JavaScript version to compile to
- **strict**: Enable all strict type checking options

🚀 Best Practices

- **Use strict mode** for better type safety
- **Prefer interfaces** over type aliases for objects
- **Use generics** for reusable components
- **Leverage type inference** when types are obvious
- **Use utility types** for type transformations
- **Write custom type guards** for complex validation
- **Use enums** for related constants
- **Document complex types** with comments

?? Resources

- TypeScript Docs
- TypeScript Handbook
- TypeScript Playground
- TypeScript Utility Types
- TypeScript Design Patterns

🎉 Summary

- TypeScript = JavaScript + Types ✅
- **Interfaces & Type Aliases** for object shapes
- **Generics** for reusable components
- **Type Guards** for runtime safety
- **Utility Types** for type transformations
- **Classes & Enums** for OOP features
- **Advanced Types** for complex scenarios
- **Strict mode** for maximum type safety

Happy typing! 🎯