

# Typescript Notes

TypeScript is a **typed superset of JavaScript**. It brings optional static typing, enabling **better tooling, code safety, and maintainability** — especially useful when scaling JavaScript codebases like MERN apps.

---

## 1 Basic Types

Basic types ensure that variables hold specific data types, catching type-related bugs during development.

### Key Types:

- `string`, `number`, `boolean` - Primitive types
- `any` - Opts out of type checking (avoid when possible)
- `unknown` - Safer alternative to `any` (requires type checking)
- `never` - For functions that never return —like ones that always throw an error or run forever.
- `void` - Functions that return nothing
- `null` / `undefined` - Empty values
- `bigint` / `symbol` - Specialized types

```
// Explicit typing
let username: string = "Alice";
let age: number = 30;
let isDone: boolean = false;

// TypeScript can infer types without explicit annotations.
let isAdmin = false; // type inference

let anything: any = "disable type checking";

let userInput: unknown = getUserInput();
```

```

if (typeof userInput === "string") {
  console.log(userInput.toUpperCase());
}

let neverHappens: never;
function throwError(): never {
  throw new Error("Something went wrong!");
}

let nothing: void = undefined;

let bigNum: bigint = 12345678901234567890n;
let uniqueKey: symbol = Symbol("key");

```

## 2 Arrays and Tuples

**Arrays:** Collections of same-type elements

```

let scores: number[] = [95, 87, 92];
let names: Array<string> = ["Alice", "Bob"];

const nums: ReadonlyArray<number> = [1, 2, 3];
// nums[0] = 5; // Error

```

**Tuples:** Fixed-length arrays with specific types per position

```

let point2D: [number, number] = [10, 20];
let httpStatus: [number, string] = [200, "OK"];

// Named tuples (TypeScript 4.0+)
let user: [name: string, age: number] = ["Alice", 30];

// React's useState returns a tuple
const [state, setState]: [string, (val: string) => void] = useState("value");

```

### 3 Interfaces & Type Aliases

**Interfaces:** Define the structure of an object.

```
// Interface
interface User {
  readonly id: string; // Cannot be modified after creation (Immutable)
  name: string;
  age?: number; // Optional property

  // Index signature (Dynamic Keys) for additional properties
  [key: string]: any; // It means this object can have any number of extra
  // properties, as long as: the key is a string, the value is any. If in place
  // of any, string[] was there then error bcz of name & greet properties as they
  // are of type string and not string[].

  greet(): string;
}

// user1 is a valid object that follows the User interface
let user1: User = {
  id: "1",
  name: "Bob",
  greet() {
    return `Hello, my name is ${this.name}`;
  }
};

// Implementing interface
class Admin implements User {
  readonly id: string;
  name: string;
  permissions: string[]; // Additional property

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

```

    this.name = name;
    this.permissions = [];
}

greet() {
    return `Hello, I'm ${this.name}`;
}
}
const admin1 = new Admin("1", "Alice");
console.log(admin1.greet()); // Hello, I'm Alice

// Interface for a callable object
interface CallableUser {
    (source: string): boolean; // It says the object itself can be used as a
    // function — it takes a string and returns a boolean
}
const callUser: CallableUser = function (source: string): boolean {
    return source.length > 0;
};
console.log(callUser("test")); // true

// Interface for a constructable object
interface UserConstructor {
    new (name: string): User; // It means the interface can be used with the new
    // keyword like a class, taking a name and returning a User.
}

// Function interface
interface SearchFunc {
    (source: string, subString: string): boolean;
}
const mySearch: SearchFunc = (src, sub) => src.includes(sub);

// Hybrid interface (function and object)
interface Counter {
    (start: number): string;

```

```

    interval: number;
    reset(): void;
  }
  function getCounter(): Counter {
    let counter = function(start: number) {} as Counter;
    counter.interval = 123;
    counter.reset = function() {};
    return counter;
  }

```

**Type Aliases:** A **type alias** lets you give a name to any type (even complex ones). It can describe an object, a union, or an intersection. It's like give this complex shape or rule a short label I can reuse.

```

// Type alias for complex types
type ID = string | number;
type ColorfulCircle = { color: string } & { radius: number };
type Coordinates = [number, number];
type Point = { x: number; y: number; };
type Tree<T> = {
  value: T;
  left?: Tree<T>;
  right?: Tree<T>;
};

```