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.

1️⃣ 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");
```

2️⃣ 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");
```

3️⃣ 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 complext 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>;
};
```

4️⃣ Enums, Union Types & Intersection Types

Enums: Let you define a set of named constants.

```
enum Color { Red, Green, Blue };
let c: Color = Color.Green;
console.log(c); // 1
console.log(Color[c]); // Green

// String enum with custom values
```

```
enum Status {
  Loading = "LOADING",
  Success = "SUCCESS",
  Error = "ERROR"
}
let current: Status = Status.Success;
console.log(current); // SUCCESS
console.log(Status[current]); // undefined
```

Unions: Allows a variable to accept multiple types.

Tagged (Discriminated) Union: A tagged union is a way to combine multiple types using a "tag" (usually a property) to know which kind of value it is.

```
// Union
function display(id: string | number) {
  console.log(`ID: ${id}`);
}

// Discriminated union
type APIState =
  | { status: "loading" }
  | { status: "success"; data: number[] }
  | { status: "error"; message: string };

function render(state: APIState) {
  switch (state.status) {
    case "success":
      console.log("Data:", state.data);
      break;
    case "error":
      console.error("Error:", state.message);
      break;
  }
}
```

```
// Intersection type (combine multiple types into one)
interface BusinessPartner {
  name: string;
  credit: number;
}

interface Contact {
  email: string;
  phone: string;
}

type Customer = BusinessPartner & Contact;

const customer: Customer = {
  name: "ABC Inc.",
  credit: 1000000,
  email: "contact@abc.com",
  phone: "123-456-7890"
};
```

5️⃣ Functions with Typed Parameters and Return Types

```
// Typed parameters and return value
function add(x: number, y: number): number {
  return x + y;
}

// Arrow function
const multiply = (x: number, y: number): number ⇒ x * y;

// Rest parameters
function sum(...numbers: number[]): number {
  return numbers.reduce((a, b) ⇒ a + b, 0);
}
```

```
// Takes a callback function with a User parameter
getUser(function (user: User) { ... });

// Function type expression
type MathOperation = (a: number, b: number) => number;
const multiply: MathOperation = (a, b) => a * b;

// this inside functions
// If we remove this then TypeScript error: "Cannot find name 'this'."
function object(this: {a: number, b: number}, a: number, b: number) {
  this.a = a;
  this.b = b;
}

// this parameter typing
interface Card {
  suit: string;
  card: number;
}
interface Deck {
  suits: string[];
  cards: number[];
  createCardPicker(this: Deck): () => Card;
}
let deck: Deck = {
  suits: ["hearts", "spades", "clubs", "diamonds"],
  cards: Array(52),
  createCardPicker: function(this: Deck) {
    return () => {
      const pickedCard = Math.floor(Math.random() * 52);
      const pickedSuit = Math.floor(pickedCard / 13);

      return {
        suit: this.suits[pickedSuit],
        card: pickedCard % 13
```

```
    };
   }
  }
}

// Explicit
function xyz(options: Card) { ... }

// Function overloading
function reverse(str: string): string;
function reverse<T>(arr: T[]): T[];
function reverse(value: string | any[]): string | any[] {
  if (typeof value === "string") {
    return value.split("").reverse().join("");
  }
  return [...value].reverse();
}
```