Typescript Notes

—> Introduction to Typescript

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.

JS vs TS

- **TypeScript**: Superset of JavaScript with **optional static typing**.
- **JavaScript**: Dynamically typed language used mainly for **web dev** (client/server).

TS/JS Interoperability

- **TS is a superset of JS** → All valid JS is valid TS.
- **Use JS in TS**:
  - Directly import JS files
  - Or use **type definitions** (`@types/library-name`)
- **Use TS in JS**:
  - Compile TS to JS with `tsc`
  - Use compiled JS in any JS environment
- Type-check:
  - The `// @ts-check` comment enables **TypeScript's type-checking** on a plain **JavaScript (.js) file**, without needing to convert it to TypeScript (`.ts`). It's especially useful when you want **type safety in JavaScript code** without rewriting your codebase.

```
// @ts-check

/**
 * Adds two numbers together.
```

```
 * @param {number} a - The first number.
 * @param {number} b - The second number.
 * @returns {number} The sum of the two numbers.
 */
function add(a, b) {
  return a + b;
}
```

Install & Configure TypeScript

1. **Init npm:** `npm init`
2. **Install TypeScript as a project dependency:** `npm install --save-dev typescript`
3. **Create tsconfig.json:**

tsconfig.json is a configuration file in TypeScript that specifies the compiler options for building your project. It helps the TypeScript compiler understand the structure of your project and how it should be compiled to JavaScript. Some common options include:

- `target`: the version of JavaScript to compile to.
- `module`: the module system to use.
- `strict`: enables/disables strict type checking.
- `outDir`: the directory to output the compiled JavaScript files.
- `rootDir`: the root directory of the TypeScript files.
- `include`: an array of file/directory patterns to include in the compilation.
- `exclude`: an array of file/directory patterns to exclude from the compilation.

```
{
  "compilerOptions": {
    "target": "es5",
    "module": "commonjs",
    "strict": true,
    "outDir": "./dist",
    "rootDir": "./src"
```

```
  },
  "exclude": ["node_modules"],
  "include": ["src"]
}
```

4. Compile Typescript

```
npx tsc // Compile full project

npx tsc ./src/index.ts // Compile single file
```

tsc

`tsc` is the command line tool for the TypeScript compiler. It compiles TypeScript code into JavaScript code, making it compatible with the browser or any JavaScript runtime environment.
This command will compile all TypeScript files in your project that are specified in your
`tsconfig.json` file. If you want to compile a specific TypeScript file, you can specify the file name after the tsc command, like this:

```
tsc index.ts
```

TypeScript compiler accepts a number of command line options that allow you to customize the compilation process. These options can be passed to the compiler using the `--` prefix, for example:

`tsc --target ES5 --module commonjs`

You can run `tsc --help` to see a list of all the available options and flags.

Running TypeScript

- Compile & Run:

1. Write code in .ts (e.g. app.ts)
2. Compile: `tsc app.ts`

3. Run: `node app.js`

- Run TypeScript directly (no manual compile step): `npx ts-node app.ts`

—> TypeScript Types

Basic Types

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

Primitive types are string, number, boolean, void, undefined and null.
Object types are Interface, Class, Enum, Array, Tuple and Object.

Key Types:

- `string`, `number`, `boolean`
- `null` / `undefined` - Empty values
- `void` - Functions that return nothing
- `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.
- `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");
```

Array, Tuple and Enum

Array: 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
```

Tuple: 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");
```

Enum: Let’s 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
```

Assertions

The `as` keyword tells the compiler to treat a value as a specific type. This is only used at compile time—it doesn't affect the runtime behavior.

```
let foo = {} // Creating foo as an empty object
foo.bar = 123 // Error: property 'bar' does not exist on `{}`
// Because the inferred type of foo is `{}`, you are not allowed to add bar to it.
// However with type assertion, the following will pass:
interface Foo {
  bar: number;
  baz: string;
}
let foo = {} as Foo; // Type assertion here
foo.bar = 123;
foo.baz = 'hello world'

// It makes the type readonly and array immutable.
const colors = ['red', 'green', 'blue'] as const;
```

```
// Non-Null Assertion operator (!) tells the TypeScript compiler: “This value is
// never null or undefined.” Useful when you're sure a value exists at runtime,
// but TypeScript cannot verify that.
let name: string | null = null;
let nameLength = name!.length; // Asserts that name is not null

// satisfies operator ensures that an expression conforms to a type, without
// changing the type inferred by TypeScript.
// Use satisfies to validate an object matches a type while preserving exact
// literals (unlike type assertions which widen types).
type Config = {
  theme: string;
  layout: string;
};
const settings = {
  theme: "dark",
  layout: "grid"
} satisfies Config;

let input: any = "Hello";
let len: number = (input as string).length;
let len: number = (<string> input).length; // valid TS but syntax not allowed in
JSX
```

—> Type Compatibility

TypeScript uses **structural typing** to determine type compatibility. Two types are considered compatible if they have the same structure. TypeScript checks type compatibility by comparing the actual **shape** of the types (properties and their types), not their names. This approach is known as **structural typing**. For example:

```
interface Point {
  x: number;
  y: number;
```

```
}

let p1: Point = { x: 10, y: 20 };
let p2: { x: number; y: number } = p1;

console.log(p2.x); // Output: 10
```

Here, `p1` has the type `Point`, and `p2` has an inline object type `{ x: number; y: number }`. Even though the types are named differently, they are compatible because their structures match exactly. If a type `A` has all the required properties of type `B`, then `A` is compatible with `B`. This applies to interfaces, inline types, classes, and function types.

—> Combining Types

TypeScript allows combining types using **union (`|`)** and **intersection (`&`)** operators.

Union Types

Union Types: 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 type
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 Types

An intersection type merges **multiple types** into one. The resulting type must satisfy **all the combined types**.

```
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"
};
```

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.

```
// Here, Name and Age are aliases for primitive types, and User is a type alias
// for an object type. Type aliases can represent primitives, objects, functions,
// unions, intersections, and more.
type Name = string;
type Age = number;
type User = { name: Name; age: Age };
const user: User = { name: 'John', age: 30 };

// 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>;
};
```

keyof Operator

The `keyof` operator creates a union of **all property keys** of a given type. In this example, `UserKeys` is a type representing the union of all keys in the `User` interface. It is useful when working with dynamic property names or implementing generic utilities.

```
interface User {
  name: string;
  age: number;
  location: string;
}
```

```
type UserKeys = keyof User; // "name" | "age" | "location"
const key: UserKeys = 'name';
```
