Typescript Notes

1️⃣ 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`

2️⃣ 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
```

3️⃣ 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.

4️⃣ 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';
```

5️⃣ Type Guards/ Narrowing

Type Guards help you narrow down the type of a variable at runtime so you can perform type-safe operations.

instanceof Operator

Used to check if an object is an instance of a class.

```
class Bird {
  fly() {
    console.log('flying...');
  }
}

const pet = new Bird();

if (pet instanceof Bird) {
  pet.fly(); // ✅ Safe to call Bird methods
} else {
  console.log('pet is not a bird');
}
```

typeof Operator

Used to check the primitive type (`'string'`, `'number'`, `'boolean'`, etc.) of a variable.

```
let value: string | number = 'hello';

if (typeof value === 'string') {
  console.log('value is a string');
} else {
```

```
  console.log('value is a number');
}
```

Equality Checks (`===` , `!==` , etc.)

Can be used to narrow types when comparing values.

```
function example(x: string | number, y: string | boolean) {
  if (x === y) {
    // TS infers x and y must be string
    x.toUpperCase();
    y.toLowerCase();
  } else {
    console.log(x);
    console.log(y);
  }
}
```

If `x` and `y` are equal, TypeScript assumes their types must also be equal — so the only possible shared type (like `string`) is inferred.

Truthiness Checks

JavaScript allows any value in conditionals. TypeScript uses this to narrow types based on whether a value is "truthy" or "falsy".

```
function getUsersOnlineMessage(numUsersOnline: number) {
  if (numUsersOnline) {
    return `There are ${numUsersOnline} online now!`;
  }
  return "Nobody's here. :(";
}
```

Falsy values include: `0` , `''` , `null` , `undefined` , `false` , `NaN` .

Type Predicates

Type predicates are functions that return a boolean value. Format: `value is Type`. Helpful for checking complex conditions or working with `unknown` and `any`.

```
// This function checks if value is a string. But the key part is the
// return type: value is string (this is type predicate)
function isString(value: unknown): value is string {
  return typeof value === 'string';
}

function example(x: unknown) {
  if (isString(x)) {
    // Now x is treated as a string
    x.toUpperCase();
  } else {
    console.log(x);
  }
}
```

Type predicate tells TypeScript the type of a variable.
“If the function (isString(x)) returns `true`, then `value` should be treated as a `string` from that point onward.”

Without the type predicate, TypeScript wouldn't narrow the type — it doesn’t learn anything from this check `return typeof value === 'string'`. You'd still get an error if you tried `x.toUpperCase()` without a manual cast.

6️⃣ Functions with Typed Parameters and Return Types

```
// Function declaration syntax
function name(param1: type1, param2: type2, ...): returnType {
  return value;
}

// Function expression syntax
let name = function(param1: type1, param2: type2, ...): returnType {
  return value;
```

```
};

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

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

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

// Function Type Variable. Define the shape of the function first, then assign it.
let divide: (a: number, b: number) ⇒ number;
divide = (a, b) ⇒ {
  return a / b;
};

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

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

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

// 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
      };
    }
  }
}

// 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();
}
```

7️⃣ Interfaces

An **interface** in TypeScript defines a contract that an object, class, or function must follow. It ensures the object structure conforms to a specific shape.

Types vs Interfaces - In TypeScript, both types and interfaces can be used to define the structure of objects and enforce type checks. However, there are some differences between the two.

Types are used to create a new named type based on an existing type or to combine existing types into a new type.

Interfaces, on the other hand, are used to describe the structure of objects and classes.

```
// 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;
}
```

8️⃣ Classes

```
// Class
class Animal {
  // members are public by default
  name: string;

  // Private - only accessible within class
  private secret: string;

  // Protected - accessible in subclasses
```

```
  protected age: number;

  // Readonly - can only be set at initialization
  readonly species: string;

  // Static property
  static totalAnimals: number = 0;

  constructor(name: string, species: string) {
    this.name = name;
    this.species = species;
    this.secret = "shh!";
    this.age = 0;
    Animal.totalAnimals++;
  }

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

// Inheritance
class Bird extends Animal {
  // Shortcut for declaring and initializing properties
  constructor(name: string, public wingSpan: number) {
    super(name, "Bird");
  }

  // Override method
  move(distance = 5) {
    console.log("Flying...");
    super.move(distance);
  }

  // Getter/setter
  private _tag: string = "";
```

```
  get tag(): string {
    return this._tag;
  }

  set tag(value: string) {
    if (value.length > 5) {
      this._tag = value;
    }
  }
}
const eagle = new Bird("Eagle", 2.1);
eagle.tag = "avish_";
console.log(eagle.tag)
eagle.move();

// Abstract class
abstract class Shape {
  abstract area(): number;
  abstract perimeter(): number;
}

// Fields which do not require initialisation
class Point {
  public someUselessValue!: number;
  ...
}

// Constructor Overloading
// TS allows multiple constructor signatures, but only one actual implementati
on
class Point {
  constructor(x: number, y: string);
  constructor(s: string);
  constructor(xs: any, y?: any) {
    // One actual implementation
```

```
    console.log(xs, y);
  }
}
```

9️⃣ Generics

Generics allow you to write flexible, reusable code that works with multiple data types. You define type **placeholders/variable** (like `<T>`) that get replaced with actual types when the function or class is used.

```
// Generic function
function identity<T>(arg: T): T {
  return arg;
}
let output = identity<string>('Hello'); // Output type: string

// Generic interface
interface GenericIdentityFn<T> {
  (arg: T): T;
}
let myIdentity: GenericIdentityFn<number> = identity;

// Generic defaults
interface Box<T = string> {
  contents: T;
}
let stringBox: Box = { contents: "hello" };
let numberBox: Box<number> = { contents: 42 };

// Generic class
class GenericNumber<T> {
  zeroValue: T;
  add: (x: T, y: T) => T;

  constructor(zeroValue: T, add: (x: T, y: T) => T) {
```

```
    this.zeroValue = zeroValue;
    this.add = add;
  }
}
let myGenericNumber = new GenericNumber<number>(1, (x, y) => x + y);
console.log(myGenericNumber.zeroValue);
console.log(myGenericNumber.add(5,10));

// Generic constraints
interface Lengthwise {
  length: number;
}
function loggingIdentity<T extends Lengthwise>(arg: T): T {
  console.log(arg.length);
  return arg;
}

// Using type parameters in generic constraints
function getProperty<T, K extends keyof T>(obj: T, key: K) {
  return obj[key];
}
```

🔟 Decorators

Decorators are special functions in TypeScript that allow you to modify the behavior of a class, property, method, or parameter. They are a way to add additional functionality to existing code, and they can be used for tasks like logging, performance optimization, and validation.

Decorators are applied using the `@` symbol.

```
function log(
  target: Object,
  propertyKey: string | symbol,
  descriptor: PropertyDescriptor
) {
```

```
  const originalMethod = descriptor.value;

  descriptor.value = function (...args: any[]) {
    console.log(`Calling ${String(propertyKey)} with arguments: ${args}`);
    return originalMethod.apply(this, args);
  };

  return descriptor;
}

class Calculator {
  @log
  add(a: number, b: number): number {
    return a + b;
  }
}

const calculator = new Calculator();
calculator.add(1, 2);
// Output: Calling add with arguments: 1,2
// Output: 3
```

In this example, we use the `@log` decorator to modify the behavior of the `add` method in the `Calculator` class. This allows us to see what arguments are being passed to the method, without having to modify the method's code.

1️⃣1️⃣ Utility Types

TypeScript provides **utility types** that help transform or construct new types based on existing ones.
They are listed below:

Partial

Makes all properties optional.

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

function createUser(user: Partial<User>): User {
  return { name: 'John Doe', age: 30, email: 'john@example.com', ...user };
}

const newUser = createUser({ name: 'Jane' });
console.log(newUser);
// Output: { name: 'Jane Doe', age: 30, email: 'john.doe@example.com' }
```

Pick<T, K>

Creates a type by **picking specific keys** from a type.

```
interface Todo {
  title: string;
  description: string;
  completed: boolean;
}

type TodoPreview = Pick<Todo, 'title' | 'completed'>;

const todo: TodoPreview = {
  title: 'Clean room',
  completed: false,
};
```

Omit<T, K>

Creates a type by **omitting specific keys** from a type.

```
type TodoInfo = Omit<Todo, 'completed' | 'createdAt'>;

const todoInfo: TodoInfo = {
  title: 'Pick up kids',
  description: 'Kindergarten closes at 5pm',
};
```

Readonly<T>

Makes **all properties read-only** (cannot be reassigned).

```
interface Todo {
  title: string;
}

const todo: Readonly<Todo> = { title: 'Clean inbox' };
todo.title = 'New Title'; // ❌ Error
```

Record<K, T>

Record constructs an object type whose property keys are Keys and whose property values are Type. Simply we can say, it constructs an object type with keys `K` and values of type `T`.

```
interface CatInfo {
  age: number;
  breed: string;
}

type CatName = 'miffy' | 'boris' | 'mordred';

const cats: Record<CatName, CatInfo> = {
  miffy: { age: 10, breed: 'Persian' },
  boris: { age: 5, breed: 'Maine Coon' },
```

```
  mordred: { age: 16, breed: 'British Shorthair' }
};
```

Exclude<T, U>

Removes from union type `T` those types that are assignable to `U`.

```
type T0 = Exclude<'a' | 'b' | 'c', 'a'>; // "b" | "c"
type T1 = Exclude<'a' | 'b' | 'c', 'a' | 'b'>; // "c"
type T2 = Exclude<string | number | (() => void), Function>; // string | number
```

Extract<T, U>

Extracts from `T` those types that are **assignable to** `U`.

```
type T0 = Extract<'a' | 'b' | 'c', 'a' | 'f'>; // "a"
```

Awaited<T>

This type is meant to model operations like await in async functions, or the `.then()` method on Promises - specifically, the way that they recursively unwrap Promises.

```
type A = Awaited<Promise<string>>;
// type A = string

type B = Awaited<Promise<Promise<number>>>;
// type B = number

type C = Awaited<boolean | Promise<number>>;
// type C = number | boolean
```

Parameters<T>

Extracts the **parameter types** of a function as a tuple.

```
type T0 = Parameters<(x: number, y: string) => void>; // [x: number, y: string]
```

NonNullable<T>

Non-Nullable constructs a type by excluding `null` and `undefined` from Type. Simply, it removes `null` and `undefined` from `T`.

```
type T0 = NonNullable<string | null | undefined>; // string
```

ReturnType<T>

Extracts the **return type** of a function type.

```
type T0 = ReturnType<() => number>; // number
```

InstanceType<T>

Constructs the **instance type** from a constructor type.

```
class MyClass {
  x = 10;
}

type T0 = InstanceType<typeof MyClass>; // MyClass
```

1️⃣2️⃣ Advanced Types

Mapped Types

Mapped Types let you transform properties of an existing type into a new form.

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

```
const obj = { x: 10, y: 20 };
const readonlyObj: Readonly<typeof obj> = obj;

// readonlyObj.x = 30; ❌Error: Cannot assign to 'x' because it's a read-only p
roperty
```

In this example, the `Readonly` mapped type takes an object type `T` and creates a new type with all properties of `T` but with their type changed to `readonly`. The keyof `T` operator is used to extract the names of the properties of `T`, and the `T[P]` syntax is used to access the type of each property of `T`.
The `readonly` keyword is used to make the properties of the new type `readonly`.

Conditional Types

A conditional type selects one type or another depending on a condition. Useful for type transformations or filtering.

```
// "Whatever type you give me, I'll turn it into an array of that type."
type ToArray<T> = T extends any ? T[] : never;
type StrArrOrNumArr = ToArray<string | number>; // string[] | number[]


type Extends<T, U> = T extends U ? T : U;
type A = Extends<string, any>; // type A is 'string'
type B = Extends<any, string>; // type B is 'string'
```

Literal Types

Literal Types enforce exact values, not just types.

```
type Age = 42;

let age: Age = 42; // ✅
let wrongAge: Age = 43; // ❌ Error
```

Template Literal Types

Used to create complex string types

```
type HttpMethod = "GET" | "POST" | "PUT" | "DELETE";
type ApiRoute = `/api/${string}`;
type Endpoint = `${HttpMethod} ${ApiRoute}`;

const route: Endpoint = "GET /api/products"; // Valid
const invalid: Endpoint = "PATCH /api/users"; // Error
```

Recursive Types

Recursive types in TypeScript are a way to define a type that references itself. Recursive types are used to define complex data structures, such as trees or linked lists, where a value can contain one or more values of the same type.

```
type LinkedList<T> = {
  value: T;
  next: LinkedList<T> | null;
};

let list: LinkedList<number> = {
  value: 1,
  next: { value: 2, next: { value: 3, next: null } },
};
```

Inferring and Type extraction

```
// Inferring Return Types
- Use `infer` to pull out subtypes like return values.
- Example:
  type GetReturnType<T> = T extends (...args: unknown[]) ⇒ infer R ? R : neve
r
  type Num = GetReturnType<() ⇒ number> // number

// Tuple Inference
```

- Destructure tuples with `infer` to access specific positions.
- Example:
  type First<T extends any[]> = T extends [infer F, ...infer Rest] ? F : never
  type Str = First<['hello', 1, false]> // 'hello'

```
// Type extraction
interface Building {
  room: {
    door: string;
    walls: string[];
  };
}
type Walls = Building['room']['walls']; // string[]

// Indexed access types
type Person = { age: number; name: string; alive: boolean };
type Age = Person["age"]; // number
```