TypeScript

Types for everyday JavaScript.

01 Setup & checking

Add to a Node.js project; save code in .ts files.

```
npm install -D typescript
npx tsc --init
npx tsc --noEmit
```

Enable strict in tsconfig.json. --noEmit checks without writing JavaScript. Type annotations disappear at runtime.

02 Types & inference

```
let count = 0;          // inferred number
const ready: boolean = true;
let label: string = "Draft";
let id: string | number = 42;
const total = (n: number) => n + 1;
```

Use lowercase string, number, and boolean. Let inference handle obvious values; annotate function parameters and public APIs.

03 Arrays & tuples

```
const names: string[] = ["Maya"];
const ids: Array<number> = [1, 2];
const point: [number, number] = [10, 20];
const pair: [string, number?] = ["x"];
const fixed: readonly number[] = [1, 2];
// fixed.push(3); // type error
```

Tuples specify each position. readonly blocks writes through this reference; it does not freeze the array at runtime.

04 Object types & interfaces

```
interface User {
  readonly id: number;
  name: string;
  email?: string;
}
const user: User = { id: 1, name: "Maya" };
```

? marks an optional property. Objects are checked structurally: they must have the required properties and compatible types.

05 Unions & literal types

```
type Status = "draft" | "published";
let status: Status = "draft";
status = "published";
// status = "archived"; // type error
type Id = string | number;
const id: Id = 42;
```

A union accepts either type. Before narrowing, you can only use operations safe for every member.

06 Optional & nullable values

```
function upper(value?: string) {
  return value?.toUpperCase() ?? "EMPTY";
}
const name: string | null = null;
const display = name ?? "Guest";
upper(); // "EMPTY"
```

With strictNullChecks, handle null and undefined explicitly. ?. checks for absence; ?? supplies a fallback without replacing 0 or false.

07 Functions & callbacks

```
function greet(name: string): string {
  return `Hi, ${name}!`;
}
type OnSave = (id: number) => void;
const save: OnSave = id => {
  console.log(id);
};
```

Parameters can use ? or defaults. A callback typed as returning void has a return value that the caller ignores.

08 Aliases & intersections

```
type Id = string | number;
interface Named { name: string }
interface Admin extends Named {
  role: "admin";
}
type Entity = Named & { id: Id };
const me: Entity = { id: 1, name: "Maya" };
```

type can name any type; interface describes an object shape. A & B must satisfy both types.

09 unknown & narrowing

```
function format(value: unknown): string {
  if (typeof value === "string") {
    return value.trim();
  }
  return "Unknown";
}
format(" hi "); // "hi"
```

unknown requires checks before use; any skips them. typeof, instanceof, and in can narrow types.

TypeScript

Practical patterns, ready to use.

10 Discriminated unions

```
type Result =
  | { ok: true; value: string }
  | { ok: false; error: string };
function read(result: Result): string {
  if (result.ok) return result.value;
  return result.error;
}
```

Checking a shared literal field selects the matching shape. value is available only in the success branch.

11 Generics

```
function first<T>(
  items: readonly T[]
): T | undefined {
  return items[0];
}
first([10, 20]);   // number | undefined
first(["a", "b"]); // string | undefined
```

T connects input and output types without losing their relationship. The empty array case returns undefined.

12 keyof & indexed access

```
interface User {
  name: string; age: number;
}
type Key = keyof User; // "name" | "age"
type Age = User["age"]; // number
const key: Key = "name";
const age: Age = 28;
```

keyof gets property keys as a union. T[K] gets the type at a key, keeping related types in sync.

13 Utility types

`Partial<T>`	Make properties optional
`Required<T>`	Make properties required
`Readonly<T>`	Make properties readonly
`Pick<T, K>`	Keep selected keys
`Omit<T, K>`	Drop selected keys
`Record<K, V>`	Map keys K to values V
`ReturnType<F>`	Get a function return type

These property modifiers are shallow.

14 as const & satisfies

```
type Method = "GET" | "POST";
const config = {
  method: "GET",
} satisfies { method: Method };
config.method; // type: "GET"
const point = [10, 20] as const;
```

satisfies checks types and retains inference. as const keeps literals and adds readonly properties.

15 Classes

```
class User {
  constructor(public name: string) {}
  greet(): string {
    return `Hi, ${this.name}!`;
  }
}
const user = new User("Maya");
```

public name both declares and assigns the field. Use #fields for runtime privacy.

16 Assertions & runtime guards

Browser example: check the actual DOM type.

```
const el = document.querySelector("#name");
if (el instanceof HTMLInputElement) {
  el.value = "Maya";
}
const value: unknown = "hello";
const text = value as string;
```

as and postfix ! affect checking only. Runtime guards such as instanceof check actual values.

17 Type imports & async

```
// types.ts
export interface User { name: string }
// app.ts (a separate file)
import type { User } from "./types.js";
async function loadUser(): Promise<User> {
  return { name: "Maya" };
}
```

Node ESM: use "type": "module" and module: "NodeNext". .js paths refer to emitted files; import type is erased.

18 Common pitfalls

`any`	Skips type checking
`as / value!`	No runtime validation
`readonly`	No runtime freeze
`T | undefined`	Handle absent values
`never`	No possible value

Validate external JSON at runtime. An annotation or assertion does not check network or storage data.