TypeScript Cheat Sheet

The typed superset of JavaScript — types are checked, then erased.
Verified against `typescriptlang.org/docs` · `strict` assumed on

INSTALL & RUN

```
npm i -D typescript
npx tsc --init      # write tsconfig.json
npx tsc             # compile
npx tsc --watch     # recompile on save
npx tsc --noEmit    # type-check only (CI)
```

```
npx tsx app.ts      # run, NO type-check
node --experimental-strip-types app.ts
```

Type errors do **not** block emit unless `noEmitOnError` is set. Bundlers (esbuild, SWC, Babel, Vite) strip types but never check them.

PRIMITIVES & SPECIAL TYPES

```
let s: string; let n: number; let b: boolean;
let big: bigint; let sym: symbol;
let u: undefined; let nl: null;
```

`any`	opt out of checking — avoid; keep at narrowest scope
`unknown`	safe top type; must narrow before use
`never`	bottom type / empty set; unreachable, exhaustiveness
`void`	function returns nothing meaningful

Everything is assignable *to* `unknown`; `never` is assignable *to* everything. Avoid wrapper types `String` / `Number`.

ARRAYS & TUPLES

```
let a: number[];          // = Array<number>
let r: readonly number[]; // = ReadonlyArray
let t: [string, number];          // tuple
let o: [number, number?];         // optional
let v: [string, ...number[]];     // rest
let l: [x: number, y: number];    // labelled
const c = [1, 2] as const;  // readonly [1, 2]
```

`arr[i]` is typed `T`, not `T | undefined`, unless `noUncheckedIndexedAccess` is on.

OBJECTS: INTERFACE VS TYPE

```
interface User { id: string; age?: number;
                 readonly created: Date; }
type Point = { x: number; y: number };

interface Admin extends User { level: number }
type Admin2 = User & { level: number };

// index signature — prefer Record/Map
interface Bag { [key: string]: number }
type Bag2 = Record<string, number>;
```

`interface`	declaration merging, `extends`, augment globals
`type`	unions, tuples, mapped & conditional types, primitives

Excess-property check fires only on a fresh object literal, not on a variable.

UNIONS & NARROWING

```
type Id = string | number;   // only common members
```

`typeof x === "string"`	primitives
`x instanceof Cls`	classes
`"key" in obj`	property presence
`if (x)`	truthiness
`x.kind === "a"`	discriminant

```
// discriminated union + exhaustiveness
type S = { kind: "ok"; data: string }
       | { kind: "err"; error: Error };

function f(s: S) {
  switch (s.kind) {
    case "ok":  return s.data;
    case "err": return s.error.message;
    default: { const _x: never = s; return _x; }
  }
}
```

```
// user-defined guards
function isStr(x: unknown): x is string {
  return typeof x === "string";
}
function assertOk(c: unknown): asserts c { }
```

FUNCTIONS

```
function add(a: number, b = 0): number { return a+b }
const g = (x: string): void => {};
function all(...xs: number[]) {}
type Fn = (a: string, b?: number) => boolean;

// overloads — prefer unions/generics where possible
function len(x: string): number;
function len(x: unknown[]): number;
function len(x: any): number { return x.length }

// this param (erased at runtime)
function onClick(this: HTMLElement, e: Event) {}
```

A `void`-returning callback type accepts a function returning anything. Parameters are inferred from context.

CLASSES

```
class Repo extends Base implements Store {
  readonly id: string;
  #secret = 1;      // runtime private
  private p = 2;    // compile-time only
  static version = "1.0";
  late!: number;    // definite assignment

  constructor(private readonly db: Db) { super() }
  // ^ parameter property: EMITS runtime code

  override save(): this { return this }
}
abstract class Shape { abstract area(): number }
```

GENERICS

```
function id<T>(x: T): T { return x }
function len<T extends { length: number }>(x: T) {}
interface Box<T = string> { value: T }
class Q<T> { items: T[] = [] }

// pick a key, safely
function get<T, K extends keyof T>(o: T, k: K): T[K] {
  return o[k];
}

// const type param: infers literals/tuples
function tup<const T extends readonly unknown[]>(x: T) {}
```

Golden rule: a type parameter must appear at least twice — it must *relate* two positions. `NoInfer<T>` excludes a site from inference.

TYPE OPERATORS

```
keyof T              // union of T's keys
typeof value         // value -> type
T["prop"]  T[number] // indexed access
```

```
// conditional + infer
type Ret<T> = T extends (...a: any[]) => infer R ? R :
never;
// distributes over unions; [T] extends [U] disables it
```

```
// mapped types (+ key remapping via `as`)
type Opt<T> = { [K in keyof T]?: T[K] };
type Mut<T> = { -readonly [K in keyof T]-?: T[K] };
type Getters<T> = { [K in keyof T as `g_${string & K}`]:
() => T[K] };
```

```
// template literal types
type Ev = `on${"Click" | "Focus"}`; // onClick|onFocus
```

UTILITY TYPES

`Partial<T>`	all props optional
`Required<T>`	all props required
`Readonly<T>`	all props readonly
`Pick<T,K>`	keep keys K
`Omit<T,K>`	drop keys K
`Record<K,V>`	object with keys K, values V
`Exclude<U,X>`	remove members from union
`Extract<U,X>`	keep members of union
`NonNullable<T>`	drop null / undefined
`ReturnType<F>`	function's return type
`Parameters<F>`	parameter tuple
`Awaited<T>`	unwrap Promise, recursively
`Uppercase<S>`	+ Lowercase / Capitalize / Uncapitalize

ASSERTIONS & SATISFIES

```
x as string          // trust me (related types only)
x as unknown as T    // two-step escape hatch
x!                   // non-null assertion
let y!: number;      // definite assignment
const cfg = {...} as const;  // deep readonly + literal
```

```
// satisfies: CHECK without widening — keeps literals
const routes = {
  home: "/", user: "/u/:id",
} satisfies Record<string, `/${string}`>;
routes.home;  // type "/" — not string
```

`catch (e)` is `unknown` under `useUnknownInCatchVariables`. Assertions only convert between related types.

ENUMS VS LITERAL UNIONS

```
// enum — emits real runtime code
enum Dir { Up, Down }        // 0, 1 + reverse map
enum Cmd { Go = "GO" }       // string, no reverse map
```

```
// preferred: erasable, tree-shakeable
const Dir2 = { Up: "up", Down: "down" } as const;
type Dir2 = typeof Dir2[keyof typeof Dir2];
```

`const enum` is discouraged — it breaks under `isolatedModules` and file-by-file bundlers.

MODULES

```
export const a = 1;  export default fn;
export type { User };          // type-only export
import type { User } from "./u";
import { type User, fn } from "./u";  // inline
const m = await import("./lazy");     // dynamic
```

`bundler`	Vite/webpack/esbuild — no extensions needed
`node16 / nodenext`	real Node ESM/CJS rules; `.js` specifiers
`classic`	legacy, do not use

```
declare module "*.css";     // ambient wildcard
declare global { interface Window { x: number } }
```

`verbatimModuleSyntax`: emit imports exactly as written. Prefer ES modules over `namespace`. `tsc` never rewrites `paths`.

TSCONFIG: STRICT FAMILY

```
{ "compilerOptions": { "strict": true } }
```

`noImplicitAny`	no silent any
`strictNullChecks`	null/undefined not in every type
`strictFunctionTypes`	contravariant parameter check
`strictPropertyInitialization`	fields must be assigned
`useUnknownInCatchVariables`	`catch (e: unknown)`
`noImplicitThis`	no untyped `this`

Beyond strict: `noUncheckedIndexedAccess`, `exactOptionalPropertyTypes`, `noImplicitOverride`, `noFallthroughCasesInSwitch`.

TSCONFIG: EMIT & BUILD

`target / lib`	output JS level / available APIs
`module`	emitted module format
`outDir / rootDir`	output & source roots
`declaration`	emit `.d.ts` (+ declarationMap)
`sourceMap`	emit `.js.map`
`noEmit`	type-check only
`skipLibCheck`	skip `.d.ts` checking (faster)
`allowJs / checkJs`	compile / check `.js`

```
// project references (monorepo)
{ "compilerOptions": { "composite": true },
  "references": [{ "path": "../core" }] }
```

```
tsc -b --verbose   # build graph incrementally
```

ASYNC, ITERATORS, RESOURCES

```
async function f(): Promise<string> { return "x" }
type R = Awaited<ReturnType<typeof f>>;  // string
const [a, b] = await Promise.all([p1, p2]); // tuple

function* gen(): Generator<number, void, unknown> {
  yield 1;
}
async function* agen(): AsyncGenerator<number> {}

// explicit resource management
using handle = open();        // Disposable
await using conn = connect(); // AsyncDisposable
```

DECORATORS

```
// TC39 standard (default, no flag)
function log(fn: any, ctx: ClassMethodDecoratorContext) {
  return function (this: any, ...a: any[]) {
    console.log(String(ctx.name));
    return fn.call(this, ...a);
  };
}
class S { @log run() {} }
// ctx: { kind, name, static, private, addInitializer }
```

```
// legacy (Angular, NestJS, TypeORM)
{ "experimentalDecorators": true,
  "emitDecoratorMetadata": true }
```

The two systems cannot be mixed. `emitDecoratorMetadata` works only with the legacy path.

DECLARATION FILES & JS INTEROP

```
npm i -D @types/lodash    # DefinitelyTyped
// bundled types: "types" field in package.json
```

```
// app.d.ts — ambient declarations
declare function greet(n: string): void;
declare module "untyped-lib" { export const x: number }
```

```
// JSDoc types in plain .js (// @ts-check)
/** @param {string} n @returns {number} */
function size(n) { return n.length }
```

Put TS and `@types/*` in `devDependencies`. Prefer `@ts-expect-error` over `@ts-ignore`.

TYPE DESIGN

- Make illegal states unrepresentable — a union of shapes beats one shape of optionals.
- Be liberal in what you accept, strict in what you produce.
- Keep `null` at the perimeter, not deep in structures.
- Prefer inference; annotate signatures and empty containers.

```
// branding: nominal typing on a structural system
type UserId = string & { readonly __brand: unique symbol
};
```

ERASED VS. EMITTED

Erased — zero runtime cost
type aliases · interfaces · generics · type annotations · `as` / `satisfies` · `import type` · `declare` · `private` / `protected` · overload signatures

Emits runtime code
`enum` · `namespace` · parameter properties · legacy decorators + metadata · class fields · `#private`

`--erasableSyntaxOnly` rejects the right-hand column — needed for Node's native type stripping.