

TypeScript

Types, interfaces, generics, narrowing and utility types

1

Why TypeScript?

Definition

TypeScript = JavaScript + a **static type system**. Types are checked at compile time and **erased** at runtime — the output is plain JavaScript.

- ✎ Catches typos, wrong arguments and null bugs before running.
- ✎ Better editor autocomplete and safe refactors across a big codebase.
- ✎ Types document APIs (props, responses, DB models).

Common mistake

Types don't validate data at runtime. JSON from an API is whatever the server sent — validate it with a schema library such as Zod at the boundary.

Shapes of data

```
interface User {
  id: number
  name: string
  email?: string // optional
  readonly createdAt: Date
}

type Role = 'admin' | 'doctor' | 'patient' // union of literals
type Staff = User & { role: Role } // intersection

const s: Staff = { id: 1, name: 'Asha', createdAt: new Date(), role: 'doctor' }
const ids: number[] = [1, 2, 3]
const pair: [string, number] = ['age', 31] // tuple

function greet(u: User): string {
  return `Hi ${u.name}${u.email ? ' <' + u.email + '>' : ''}`
}

console.log(greet(s), ids.length, pair[0])
```

interface	type
Objects and classes; can be extended with <code>extends</code>	Anything: unions, tuples, primitives, mapped types
Declaration merging (re-open to add fields)	Cannot be re-opened
Good default for object shapes and public APIs	Needed for unions and computed types

- `any` turns type checking off — avoid it.
- `unknown` is the safe "anything": you must narrow it before use.
- `never` = a value that can't exist (exhaustive checks, functions that throw).

Exhaustive switch

```
type Result =  
  | { status: 'loading' }  
  | { status: 'success'; data: string[] }  
  | { status: 'error'; message: string }  
  
function render(r: Result): string {  
  switch (r.status) {  
    case 'loading': return 'Loading...'  
    case 'success': return r.data.join(', ') // data exists here  
    case 'error':   return 'Error: ' + r.message  
    default: {  
      const unreachable: never = r // compile error if a case is  
missing  
      return unreachable  
    }  
  }  
}  
  
function len(x: string | string[] | null): number {  
  if (x === null) return 0  
  return typeof x === 'string' ? x.length : x.length  
}  
  
console.log(render({ status: 'success', data: ['a', 'b'] }), len('hello'),  
len(null))
```

- ✎ Narrow with `typeof`, `instanceof`, `in`, equality and truthiness checks.
- ✎ A user-defined type guard: `function isUser(x: unknown): x is User { ... }`.

Definition

Generics are type parameters: write one function or type that works for many types while keeping them linked.

Generic helpers with constraints

```
function first<T>(items: T[]): T | undefined {
  return items[0]
}

function pluck<T, K extends keyof T>(items: T[], key: K): T[K][] {
  return items.map((item) => item[key])
}

interface ApiResponse<T> { data: T; error: string | null }

const users = [{ id: 1, name: 'Asha' }, { id: 2, name: 'Ben' }]
const names = pluck(users, 'name') // string[]
const res: ApiResponse<typeof users> = { data: users, error: null }
console.log(first(names), res.data.length)
```

Utility	Result
<code>Partial<T></code>	every property optional (PATCH bodies)
<code>Required<T></code>	every property required
<code>Readonly<T></code>	every property readonly
<code>Pick<T, "a" "b"></code>	only those keys
<code>Omit<T, "password"></code>	all keys except those
<code>Record<K, V></code>	an object with keys K and values V
<code>ReturnType<typeof fn></code>	the return type of a function
<code>Awaited<P></code>	the value inside a promise

Typical API types

```
interface User { id: number; name: string; email: string; password: string }

type PublicUser = Omit<User, 'password'>
type UpdateUser = Partial<Pick<User, 'name' | 'email'>>
type UsersById = Record<number, PublicUser>

// keyof, typeof, and mapped types
type Flags<T> = { [K in keyof T]: boolean }
const config = { port: 3000, host: 'localhost' }
type Config = typeof config // { port: number; host: string }
```

Props, state and events

```
type ButtonProps = {
  label: string
  variant?: 'primary' | 'ghost'
  onClick: (event: React.MouseEvent<HTMLButtonElement>) => void
} & React.ComponentProps<'button'>

function Button({ label, variant = 'primary', ...rest }: ButtonProps) {
  return <button className={variant} {...rest}>{label}</button>
}

const [user, setUser] = useState<User | null>(null)
const inputRef = useRef<HTMLInputElement>(null)
```

Revision

Types are erased at runtime → validate external data. interface for object shapes, type for unions.
Narrow unknown, never use any. Generics keep types linked; Partial / Pick / Omit / Record cover most API types.

Rapid-fire Q&A

Q. interface vs type?

A. interface describes object shapes, can be extended and merged. type can also express unions, tuples and computed types but can't be re-opened.

Q. any vs unknown vs never?

A. any disables checking; unknown accepts anything but must be narrowed before use; never is a value that cannot exist.

Q. What are generics?

A. Type parameters that let one function or type work with many types while keeping input and output types linked, e.g. `first<T>(items: T[]): T`.

Q. What is a discriminated union?

A. A union whose members share a literal field (like status). Switching on that field narrows to the right member.

Q. Does TypeScript check types at runtime?

A. No. Types are erased when compiling to JavaScript, so data from APIs must be validated at runtime (e.g. with Zod).

Q. Partial, Pick and Omit?

A. Partial makes all properties optional; Pick keeps chosen keys; Omit removes chosen keys.

Q. What does keyof do?

A. Produces a union of an object type's keys, e.g. `keyof { a: 1; b: 2 }` is `"a" | "b"`.

Q. How do you make a switch exhaustive?

A. Assign the value to a variable of type never in the default branch; the compiler errors if a case is missing.

Q. What is a type guard?

A. A check that narrows a type — `typeof`, `instanceof`, `in`, or a function returning `"x is Type"`.