# TYPESCRIPT QUICK REFERENCE

## Basic Types

### Primitives

```
let name: string = "Alice";
let age: number = 25;
let active: boolean = true;
let data: null = null;
let x: undefined = undefined;
```

### Special Types

| | |
|---|---|
| `any` | Opt out of type checking |
| `unknown` | Type-safe any (must narrow before use) |
| `void` | No return value |
| `never` | Function never returns (throws / infinite) |
| `object` | Any non-primitive |

## Arrays & Tuples

### Arrays

```
let nums: number[] = [1, 2, 3];
let names: Array = ["a", "b"];
let matrix: number[][] = [[1, 2], [3, 4]];
```

### Tuples

```
let pair: [string, number] = ["age", 25];
let rgb: [number, number, number] = [255, 0, 0];

// Named tuples (labels for readability)
type Point = [x: number, y: number];
```

## Interfaces

### Defining & Using

```
interface User {
  name: string;
  age: number;
  email?: string;        // optional
  readonly id: number;   // immutable
}

const user: User = { name: "Alice", age: 25, id: 1 };
```

### Extending Interfaces

```
interface Employee extends User {
  role: string;
  department: string;
}
```

### Index Signatures

```
interface StringMap {
  [key: string]: string;
}
const env: StringMap = { NODE_ENV: "prod" };
```

## Type Aliases

```
type ID = string | number;
type Point = { x: number; y: number };
type Callback = (data: string) => void;
```

### Interface vs Type

| | |
|---|---|
| `interface` | Extendable with `extends`, declaration merging |
| `type` | Unions, intersections, mapped types, tuples |

# TYPESCRIPT QUICK REFERENCE (continued)

## Unions & Intersections

### Union Types

```
type Status = "loading" | "success" | "error";
type ID = string | number;

function print(val: string | number) {
  if (typeof val === "string") {
    console.log(val.toUpperCase());
  }
}
```

### Intersection Types

```
type Named = { name: string };
type Aged = { age: number };
type Person = Named & Aged;
// Person has both name and age
```

### Discriminated Unions

```
type Shape =
  | { kind: "circle"; radius: number }
  | { kind: "rect"; w: number; h: number };

function area(s: Shape): number {
  switch (s.kind) {
    case "circle": return Math.PI * s.radius ** 2;
    case "rect":   return s.w * s.h;
  }
}
```

## Functions

### Typed Parameters & Return

```
function add(a: number, b: number): number {
  return a + b;
}

// Arrow function
const greet = (name: string): string =>
  `Hello, ${name}!`;

// Optional & default params
function log(msg: string, level?: string): void {}
function log(msg: string, level = "info"): void {}
```

### Function Overloads

```
function parse(input: string): number;
function parse(input: number): string;
function parse(input: string | number) {
  return typeof input === "string"
    ? parseInt(input)
    : input.toString();
}
```

### Rest Parameters

```
function sum(...nums: number[]): number {
  return nums.reduce((a, b) => a + b, 0);
}
```

## Generics

### Generic Functions

```
function identity(value: T): T {
  return value;
}
identity("hello"); // explicit
identity(42);              // inferred: number
```

### Generic Interfaces & Constraints

```
interface Box {
  value: T;
}
const box: Box = { value: 42 };

// Constraints
function getLen(
  item: T
): number {
  return item.length;
}
```

## Enums

```
enum Direction { Up, Down, Left, Right }
let d: Direction = Direction.Up; // 0

enum Status {
  Active = "ACTIVE",
  Inactive = "INACTIVE",
}
let s: Status = Status.Active; // "ACTIVE"

// const enum (inlined at compile time)
const enum Color { Red, Green, Blue }
```

## Type Guards

### Built-in Guards

```
// typeof
if (typeof x === "string") { /* x: string */ }

// instanceof
if (err instanceof Error) { /* err: Error */ }

// in
if ("name" in obj) { /* obj has name */ }
```

### Custom Type Guard

```
function isString(val: unknown): val is string {
  return typeof val === "string";
}

if (isString(input)) {
  input.toUpperCase(); // narrowed to string
}
```

### Assertion Functions

```
function assertDefined(
  val: T | null
): asserts val is T {
  if (val === null) throw new Error("null");
}
```

## Utility Types

| | |
|---|---|
| `Partial<T>` | All properties optional |
| `Required<T>` | All properties required |
| `Readonly<T>` | All properties readonly |
| `Pick<T, K>` | Select properties K from T |
| `Omit<T, K>` | Remove properties K from T |
| `Record<K, V>` | Object with keys K and values V |
| `Exclude<T, U>` | Types in T not in U |
| `Extract<T, U>` | Types in T also in U |
| `NonNullable<T>` | Exclude null and undefined from T |
| `ReturnType<T>` | Return type of function T |
| `Parameters<T>` | Parameter types of function T |
| `Awaited<T>` | Unwrap Promise type |

### Utility Type Examples

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

type UserPreview = Pick;
type UserUpdate = Partial;
type UserMap = Record;
type CreateUser = Omit;
```