TYPESCRIPT クイックリファレンス

型、インターフェース、ジェネリクス、ユーティリティ型

基本型

プリミティブ型

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

特殊な型

`any`	型チェックを無効化
`unknown`	型安全な `any`（使用前にナロウイングが必要）
`void`	戻り値なし
`never`	関数が決して返らない（スローまたは無限ループ）
`object`	プリミティブ以外の任意の型

配列とタプル

配列

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

タプル

```
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];
```

インターフェース

定義と使用

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

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

インターフェースの拡張

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

インデックスシグネチャ

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

型エイリアス

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

interface vs type

`interface`	`extends` で拡張可能、宣言マージが可能
`type`	ユニオン、交差、マップド型、タプル

ユニオンと交差

ユニオン型

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

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

交差型

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

判別ユニオン

```
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;
  }
}
```

関数

型付きパラメータと戻り値

```
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 {}
```

関数のオーバーロード

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

レストパラメータ

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

ジェネリクス

ジェネリック関数

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

ジェネリックインターフェースと制約

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

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

列挙型

```
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 }
```

型ガード

組み込みガード

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

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

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

カスタム型ガード

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

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

アサーション関数

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

ユーティリティ型

`Partial<T>`	すべてのプロパティをオプションに
`Required<T>`	すべてのプロパティを必須に
`Readonly<T>`	すべてのプロパティを読み取り専用に
`Pick<T, K>`	T から K のプロパティを選択
`Omit<T, K>`	T から K のプロパティを除外
`Record<K, V>`	キー K と値 V を持つオブジェクト
`Exclude<T, U>`	T の中で U に含まれない型
`Extract<T, U>`	T の中で U に含まれる型
`NonNullable<T>`	T から null と undefined を除外
`ReturnType<T>`	関数 T の戻り値の型
`Parameters<T>`	関数 T のパラメータの型
`Awaited<T>`	Promise 型のアンラップ

ユーティリティ型の例

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

type UserPreview = Pick<User, "name" | "email">;
type UserUpdate = Partial<User>;
type UserMap = Record<string, User>;
type CreateUser = Omit<User, "id">;
```