REFERÊNCIA RÁPIDA DE TYPESCRIPT

Tipos, interfaces, genéricos, tipos utilitários

Tipos Básicos

Primitivos

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

Tipos Especiais

`any`	Desabilitar verificação de tipo
`unknown`	`any` com segurança de tipo (deve ser refinado antes de usar)
`void`	Sem valor de retorno
`never`	Função nunca retorna (lança / infinita)
`object`	Qualquer não-primitivo

Arrays e Tuplas

Arrays

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

Tuplas

```
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

Definir e Usar

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

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

Estender Interfaces

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

Assinaturas de Índice

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

Aliases de Tipo

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

Interface vs Type

`interface`	Extensível com `extends`, mesclagem de declaração
`type`	Uniões, interseções, tipos mapeados, tuplas

Uniões e Interseções

Tipos União

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

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

Tipos Interseção

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

Uniões Discriminadas

```
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;
  }
}
```

Funções

Parâmetros e Retorno Tipados

```
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 {}
```

Sobrecargas de Função

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

Parâmetros Rest

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

Genéricos

Funções Genéricas

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

Interfaces Genéricas e Restrições

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

// Constraints
function getLen<T extends { length: number }>(
  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

Guards Integrados

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

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

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

Type Guard Personalizado

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

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

Funções de Asserção

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

Tipos Utilitários

`Partial<T>`	Todas as propriedades opcionais
`Required<T>`	Todas as propriedades obrigatórias
`Readonly<T>`	Todas as propriedades somente leitura
`Pick<T, K>`	Selecionar propriedades K de T
`Omit<T, K>`	Remover propriedades K de T
`Record<K, V>`	Objeto com chaves K e valores V
`Exclude<T, U>`	Tipos em T que não estão em U
`Extract<T, U>`	Tipos em T que também estão em U
`NonNullable<T>`	Excluir null e undefined de T
`ReturnType<T>`	Tipo de retorno da função T
`Parameters<T>`	Tipos de parâmetros da função T
`Awaited<T>`	Desempacotar tipo Promise

Exemplos de Tipos Utilitários

```
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">;
```