

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  |  |  |  | | --- | --- | --- | --- | --- | | **Basic types** | |  | **Literal types** | | | Untyped | any |  | String | let direction: 'left' | 'right'; | | A string | string |  | Numeric | let roll: 1 | 2 | 3 | 4 | 5 | 6; | | Template string | ${string}Suffix |  |  |  | | A number | number |  |  |  | | A true / false value | boolean |  |  |  | | A non-primitive value | object |  |  |  | | Uninitialized value | undefined |  | **Object types** | | | Explicitly empty value | null |  | Object | { requiredStringVal: string;  optionalNum?: number;  readonly readOnlyBool: bool;} | | Null or undefined (usually only used for function returns) | void |  | Object with arbitrary string properties (like a hashmap or dictionary) | { [key: string]: Type; }  { [key: number]: Type; }  { [key: symbol]: Type; }  { [key: `data-${string}`]: Type; } | | A value that can never occur | never |  |  |  | | A value with an unknown type | unknown |  |  |  |  |  |  | | --- | --- | | **Arrays and tuples** | | | Array of strings | string[]  or  Array<string> | | Array of functions that return strings | (() => string)[]  or  { (): string; }[]  or  Array<() => string> | | Basic tuples | let myTuple: [ string, number, boolean? ];  myTuple = [ 'test', 42 ]; | | Variadic tuples | type Numbers = [number, number];  type Strings = [string, string];    type NumbersAndStrings = [...Numbers, ...Strings];  // [number, number, string, string]  type NumberAndRest = [number, ...string[]];  // [number, varying number of string]  type RestAndBoolean = [...any[], boolean];  // [varying number of any, boolean] | | Named tuples | type Vector2D = [x: number, y: number];  function createVector2d(...args: Vector2D) {}  // function createVector2d(x: number, y: number): void |  |  |  | | --- | --- | | **Functions** | | | Function type | (arg1: Type, argN: Type) => Type;  or  { (arg1: Type, argN: Type): Type; } | | Constructor | new () => ConstructedType;  or  { new (): ConstructedType; } | | Function type with optional param | (arg1: Type, optional?: Type) => ReturnType | | Function type with rest param | (arg1: Type, ...allOtherArgs: Type[]) => ReturnType | | Function type with static property | { (): Type; staticProp: Type; } | | Default argument | function fn(arg1 = 'default'): ReturnType {} | | Arrow function | (arg1: Type): ReturnType => { ...; return value; }  or  (arg1: Type): ReturnType => value; | | this typing | function fn(this: Foo, arg1: string) {} | | Overloads | function conv(a: string): number;  function conv(a: number): string;  function conv(a: string | number): string | number { ... } | | **Union and intersection types** | | | Union | let myUnionVariable: number | string; | | Intersection | let myIntersectionType: Foo & Bar; | | **Named types** | | | Interface | interface Child extends Parent, SomeClass {  property: Type;  optionalProp?: Type;  optionalMethod?(arg1: Type): ReturnType;} | | Class | class Child  extends Parent  implements Child, OtherChild {  property: Type;  defaultProperty = 'default value';  private \_privateProperty: Type;  private readonly \_privateReadonlyProperty: Type;  static staticProperty: Type;  static { try {  Child.staticProperty = calcStaticProp();  } catch {  Child.staticProperty = defaultValue;  } }  constructor(arg1: Type) {  super(arg1); }  private \_privateMethod(): Type {}  methodProperty: (arg1: Type) => ReturnType;  overloadedMethod(arg1: Type): ReturnType;  overloadedMethod(arg1: OtherType): ReturnType;  overloadedMethod(arg1: CommonT): CommonReturnT {}  static staticMethod(): ReturnType {}  subclassedMethod(arg1: Type): ReturnType {  super.subclassedMethod(arg1); }} | | Enum | enum Options {  FIRST,  EXPLICIT = 1,  BOOLEAN = Options.FIRST | Options.EXPLICIT,  COMPUTED = getValue()}  enum Colors {  Red = "#FF0000",  Green = "#00FF00",  Blue = "#0000FF"} | | Type alias | type Name = string;  type Direction = 'left' | 'right';  type ElementCreator = (type: string) => Element;  type Point = { x: number, y: number };  type Point3D = Point & { z: number };  type PointProp = keyof Point; // 'x' | 'y'  const point: Point = { x: 1, y: 2 };  type PtValProp = keyof typeof point; // 'x' | 'y' | | **Generics** | | | Function using type parameters | <T>(items: T[], callback: (item: T) => T): T[] | | Interface with multiple types | interface Pair<T1, T2> {  first: T1;  second: T2;} | | Constrained type parameter | <T extends ConstrainedType>(): T | | Default type parameter | <T = DefaultType>(): T | | Constrained and default type parameter | <T extends ConstrainedType = DefaultType>(): T | | Generic tuples | type Arr = readonly any[];    function concat<U extends Arr, V extends Arr>(a: U, b: V):  [...U, ...V] { return [...a, ...b] }    const strictResult = concat([1, 2] as const, ['3', '4'] as const);  const relaxedResult = concat([1, 2], ['3', '4']);  // strictResult is of type [1, 2, '3', '4']  // relaxedResult is of type (string | number)[] | | **Index, mapped, and conditional types** | | | Index type query (keyof) | type Point = { x: number, y: number };  let pointProp: keyof Point = 'x';  function getProp<T, K extends keyof T>(  val: T,  propName: K  ): T[K] { ... } | | Mapped types | type Stringify<T> = { [P in keyof T]: string; }  type Partial<T> = { [P in keyof T]?: T[P]; } | | Conditional types | type Swapper = <T extends number | string>  (value: T) => T extends number ? string : number;  *is equivalent to*  (value: number) => string  *if T is number, or*  (value: string) => number  *if T is string* | |