`Partial<T>`

Constructs a type with all properties of `T` set to optional. This utility will return a type that represents all subsets of a given type.

Example

```
interface Todo {
  title: string;
  description: string;
}

function updateTodo(todo: Todo, fieldsToUpdate: Partial<Todo>) {
  return { ...todo, ...fieldsToUpdate };
}

const todo1 = {
  title: "organize desk",
  description: "clear clutter"
};

const todo2 = updateTodo(todo1, {
  description: "throw out trash"
});
```

`Readonly<T>`

Constructs a type with all properties of `T` set to `readonly`, meaning the properties of the constructed type cannot be reassigned.

Example

```
interface Todo {
  title: string;
}

const todo: Readonly<Todo> = {
  title: "Delete inactive users"
};

todo.title = "Hello"; // Error: cannot reassign a readonly property
```

This utility is useful for representing assignment expressions that will fail at runtime (i.e. when attempting to reassign properties of a frozen object).

```
function freeze<T>(obj: T): Readonly<T>;
```

`Record<K,T>`

Constructs a type with a set of properties `K` of type `T`. This utility can be used to map the properties of a type to another type.

Example

```
interface PageInfo {
  title: string;
}

type Page = "home" | "about" | "contact";

const x: Record<Page, PageInfo> = {
  about: { title: "about" },
  contact: { title: "contact" },
  home: { title: "home" }
};
```

`Pick<T,K>`

Constructs a type by picking the set of properties `K` from `T`.

Example

```
interface Todo {
  title: string;
  description: string;
  completed: boolean;
}

type TodoPreview = Pick<Todo, "title" | "completed">;

const todo: TodoPreview = {
  title: "Clean room",
  completed: false
};
```

`Omit<T,K>`

Constructs a type by picking all properties from `T` and then removing `K`.

Example

```
interface Todo {
  title: string;
  description: string;
  completed: boolean;
}

type TodoPreview = Omit<Todo, "description">;

const todo: TodoPreview = {
  title: "Clean room",
  completed: false
};
```

`Exclude<T,U>`

Constructs a type by excluding from `T` all properties that are assignable to `U`.

Example

```
type T0 = Exclude<"a" | "b" | "c", "a">; // "b" | "c"
type T1 = Exclude<"a" | "b" | "c", "a" | "b">; // "c"
type T2 = Exclude<string | number | (() => void), Function>; // string | number
```

`Extract<T,U>`

Constructs a type by extracting from `T` all properties that are assignable to `U`.

Example

```
type T0 = Extract<"a" | "b" | "c", "a" | "f">; // "a"
type T1 = Extract<string | number | (() => void), Function>; // () => void
```

`NonNullable<T>`

Constructs a type by excluding `null` and `undefined` from `T`.

Example

```
type T0 = NonNullable<string | number | undefined>; // string | number
type T1 = NonNullable<string[] | null | undefined>; // string[]
```

`ReturnType<T>`

Constructs a type consisting of the return type of function `T`.

Example

```
type T0 = ReturnType<() => string>; // string
type T1 = ReturnType<(s: string) => void>; // void
type T2 = ReturnType<<T>() => T>; // {}
type T3 = ReturnType<<T extends U, U extends number[]>() => T>; // number[]
type T4 = ReturnType<typeof f1>; // { a: number, b: string }
type T5 = ReturnType<any>; // any
type T6 = ReturnType<never>; // any
type T7 = ReturnType<string>; // Error
type T8 = ReturnType<Function>; // Error
```

`InstanceType<T>`

Constructs a type consisting of the instance type of a constructor function type `T`.

Example

```
class C {
  x = 0;
  y = 0;
}

type T0 = InstanceType<typeof C>; // C
type T1 = InstanceType<any>; // any
type T2 = InstanceType<never>; // any
type T3 = InstanceType<string>; // Error
type T4 = InstanceType<Function>; // Error
```

`Required<T>`

Constructs a type consisting of all properties of `T` set to required.

Example

```ts
interface Props {
  a?: number;
  b?: string;
}

const obj: Props = { a: 5 }; // OK

const obj2: Required<Props> = { a: 5 }; // Error: property 'b' missing
```

`ThisType<T>`

This utility does not return a transformed type. Instead, it serves as a marker for a contextual `this` type. Note that the `--noImplicitThis` flag must be enabled to use this utility.

Example

```ts
// Compile with --noImplicitThis

type ObjectDescriptor<D, M> = {
  data?: D;
  methods?: M & ThisType<D & M>; // Type of 'this' in methods is D & M
};

function makeObject<D, M>(desc: ObjectDescriptor<D, M>): D & M {
  let data: object = desc.data || {};
  let methods: object = desc.methods || {};
  return { ...data, ...methods } as D & M;
}

let obj = makeObject({
  data: { x: 0, y: 0 },
  methods: {
    moveBy(dx: number, dy: number) {
      this.x += dx; // Strongly typed this
      this.y += dy; // Strongly typed this
    }
  }
});

obj.x = 10;
obj.y = 20;
obj.moveBy(5, 5);
```

In the example above, the `methods` object in the argument to `makeObject` has a contextual type that includes `ThisType<D & M>` and therefore the type of `this` in methods within the `methods` object is `{ x: number, y: number } & { moveBy(dx: number, dy: number): number }`. Notice how the type of the `methods` property simultaneously is an inference target and a source for the `this` type in methods.

The `ThisType<T>` marker interface is simply an empty interface declared in `lib.d.ts`. Beyond being recognized in the contextual type of an object literal, the interface acts like any empty interface.

`Parameters<T>`

The `Parameters<T>` type lets us extract all parameter types of a function type. It produces a tuple type with all the parameter types (or the type `never` if `T` is not a function).

Example

```
type A = Parameters<() => void>; // []
type B = Parameters<typeof Array.isArray>; // [any]
type C = Parameters<typeof parseInt>; // [string, (number | undefined)?]
type D = Parameters<typeof Math.max>; // number[]
```

The `Array.isArray()` method expects exactly one argument of an arbitrary type; this is why type `B` is resolved as `[any]`, a tuple with exactly one element. The `Math.max()` method, on the other hand, expects arbitrarily many numeric arguments (not a single array argument); therefore, type D is resolved as `number[]` (and not `[number[]]`).

`ConstructorParameters<T>`

The `ConstructorParameters<T>` type lets us extract all parameter types of a constructor function type. It produces a tuple type with all the parameter types (or the type `never` if `T` is not a function).

Example

```
type A = ConstructorParameters<ErrorConstructor>; // [(string | undefined)?]
type B = ConstructorParameters<FunctionConstructor>; // string[]
type C = ConstructorParameters<RegExpConstructor>; // [string, (string | undefined)?]
```