Utility Types

TypeScript provides several utility types to facilitate common type transformations. These utilities are available globally.

Awaited<Type>

This type is meant to model operations like `await` in `async` functions, or the `.then()` method on `Promise` s - specifically, the way that they recursively unwrap `Promise` s.

Released: 4.5

Example

```
type A = Awaited<Promise<string>>;

    type A = string


type B = Awaited<Promise<Promise<number>>>;
```

```
     type B = number

type C = Awaited<boolean | Promise<number>>;

     type C = number | boolean
```

Partial<Type>

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

Released:
[2.1]

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",
});
```

Required<Type>

Constructs a type consisting of all properties of `Type` set to required. The opposite of [`Partial`].

Released:
[2.8]

Example

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

const obj: Props = { a: 5 };

const obj2: Required<Props> = { a: 5 };
```

```
Property 'b' is missing in type '{ a: number; }' but required in type
'Required<Props>'.
```

Readonly<Type>

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

Released:
2.1

Example

```
interface Todo {
  title: string;
}

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

todo.title = "Hello";
Cannot assign to 'title' because it is a read-only 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).

`Object.freeze`

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

Record<Keys, Type>

Constructs an object type whose property keys are `Keys` and whose property values are `Type`. This utility can be used to map the properties of a type to another type.

Released:
2.1

Example

```
type CatName = "miffy" | "boris" | "mordred";

interface CatInfo {
  age: number;
  breed: string;
}

const cats: Record<CatName, CatInfo> = {
  miffy: { age: 10, breed: "Persian" },
```

On this page

Intrinsic String Manipulation Types

Uppercase<StringType>

Lowercase<StringType>

Capitalize<StringType>

Uncapitalize<StringType>

Is this page helpful?

Yes No

```
  boris: { age: 5, breed: "Maine Coon" },
  mordred: { age: 16, breed: "British Shorthair" },
};

cats.boris;

  const cats: Record<CatName, CatInfo>
```

Pick<Type, Keys>

Constructs a type by picking the set of properties Keys (string literal or union of string literals) from Type .

Released:
2.1

Example

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

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

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

todo;

  const todo: TodoPreview
```

Omit<Type, Keys>

Constructs a type by picking all properties from Type and then removing Keys (string literal or union of string literals). The opposite of Pick.

Released:
3.5

Example

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

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

```
const todo: TodoPreview = {
  title: "Clean room",
  completed: false,
  createdAt: 1615544252770,
};

todo;

  const todo: TodoPreview


type TodoInfo = Omit<Todo, "completed" | "createdAt">;

const todoInfo: TodoInfo = {
  title: "Pick up kids",
  description: "Kindergarten closes at 5pm",
};

todoInfo;

    const todoInfo: TodoInfo
```

`Exclude<UnionType, ExcludedMembers>`

Constructs a type by excluding from `UnionType` all union members that are assignable to `ExcludedMembers`.

Released:
2.8

Example

```
type T0 = Exclude<"a" | "b" | "c", "a">;

      type T0 = "b" | "c"

type T1 = Exclude<"a" | "b" | "c", "a" | "b">;

      type T1 = "c"

type T2 = Exclude<string | number | (() => void), Function>;

      type T2 = string | number


type Shape =
  | { kind: "circle"; radius: number }
  | { kind: "square"; x: number }
  | { kind: "triangle"; x: number; y: number };

type T3 = Exclude<Shape, { kind: "circle" }>
```

```
      type T3 = {
          kind: "square";
          x: number;
      } | {
          kind: "triangle";
          x: number;
          y: number;
      }
```

Extract<Type, Union>

Constructs a type by extracting from `Type` all union members that are assignable to `Union`.

Released:
2.8

Example

```
type T0 = Extract<"a" | "b" | "c", "a" | "f">;

      type T0 = "a"

type T1 = Extract<string | number | (() => void), Function>;

      type T1 = () => void


type Shape =
  | { kind: "circle"; radius: number }
  | { kind: "square"; x: number }
  | { kind: "triangle"; x: number; y: number };

type T2 = Extract<Shape, { kind: "circle" }>

      type T2 = {
          kind: "circle";
          radius: number;
      }
```

NonNullable<Type>

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

Released:
2.8

Example

```
type T0 = NonNullable<string | number | undefined>;

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

      type T1 = string[]
```

Parameters<Type>

Constructs a tuple type from the types used in the parameters of a function type `Type`.

For overloaded functions, this will be the parameters of the *last* signature; see Inferring Within Conditional Types.

Released:
3.1

Example

```
declare function f1(arg: { a: number; b: string }): void;

type T0 = Parameters<() => string>;

      type T0 = []

type T1 = Parameters<(s: string) => void>;

      type T1 = [s: string]

type T2 = Parameters<<T>(arg: T) => T>;

      type T2 = [arg: unknown]

type T3 = Parameters<typeof f1>;

      type T3 = [arg: {
          a: number;
          b: string;
      }]

type T4 = Parameters<any>;

      type T4 = unknown[]

type T5 = Parameters<never>;

      type T5 = never

type T6 = Parameters<string>;
Type 'string' does not satisfy the constraint '(...args: any) => any'.

      type T6 = never

type T7 = Parameters<Function>;
Type 'Function' does not satisfy the constraint '(...args: any) => any'.
  Type 'Function' provides no match for the signature '(...args: any): any'.
```

```
      type T7 = never
```

ConstructorParameters<Type>

Constructs a tuple or array type from the types of a constructor function type. It produces a tuple type with all the parameter types (or the type `never` if `Type` is not a function).

Released:
3.1

Example

```
type T0 = ConstructorParameters<ErrorConstructor>;

      type T0 = [message?: string]

type T1 = ConstructorParameters<FunctionConstructor>;

      type T1 = string[]

type T2 = ConstructorParameters<RegExpConstructor>;

      type T2 = [pattern: string | RegExp, flags?: string]

class C {
  constructor(a: number, b: string) {}
}
type T3 = ConstructorParameters<typeof C>;

      type T3 = [a: number, b: string]

type T4 = ConstructorParameters<any>;

      type T4 = unknown[]


type T5 = ConstructorParameters<Function>;

 Type 'Function' does not satisfy the constraint 'abstract new (...args: any) =>
 any'.
   Type 'Function' provides no match for the signature 'new (...args: any): any'.
any'.

      type T5 = never
```

ReturnType<Type>

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

Released:
2.8

For overloaded functions, this will be the return type of the *last* signature; see Inferring Within Conditional Types.

Example

```
declare function f1(): { a: number; b: string };

type T0 = ReturnType<() => string>;

      type T0 = string

type T1 = ReturnType<(s: string) => void>;

      type T1 = void

type T2 = ReturnType<<T>() => T>;

      type T2 = unknown

type T3 = ReturnType<<T extends U, U extends number[]>() => T>;

      type T3 = number[]

type T4 = ReturnType<typeof f1>;

      type T4 = {
          a: number;
          b: string;
      }

type T5 = ReturnType<any>;

      type T5 = any

type T6 = ReturnType<never>;

      type T6 = never

type T7 = ReturnType<string>;
Type 'string' does not satisfy the constraint '(...args: any) => any'.

      type T7 = any

type T8 = ReturnType<Function>;
Type 'Function' does not satisfy the constraint '(...args: any) => any'.
  Type 'Function' provides no match for the signature '(...args: any): any'.

      type T8 = any
```

InstanceType<Type>

Constructs a type consisting of the instance type of a constructor function in `Type`.

Released:
2.8

Example

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

type T0 = InstanceType<typeof C>;

      type T0 = C

type T1 = InstanceType<any>;

      type T1 = any

type T2 = InstanceType<never>;

      type T2 = never

type T3 = InstanceType<string>;
Type 'string' does not satisfy the constraint 'abstract new (...args: any) =>
any'.

      type T3 = any

type T4 = InstanceType<Function>;
Type 'Function' does not satisfy the constraint 'abstract new (...args: any) =>
any'.
  Type 'Function' provides no match for the signature 'new (...args: any): any'.
any'.

      type T4 = any
```

NoInfer<Type>

Blocks inferences to the contained type. Other than blocking inferences, `NoInfer<Type>` is identical to `Type`.

Released:
5.4

Example

```
function createStreetLight<C extends string>(
  colors: C[],
  defaultColor?: NoInfer<C>,
) {
  // ...
}
```

```
createStreetLight(["red", "yellow", "green"], "red");  // OK
createStreetLight(["red", "yellow", "green"], "blue");  // Error
```

ThisParameterType<Type>

Extracts the type of the this parameter for a function type, or unknown if the function type has no `this` parameter.

Released:
3.3

Example

```
function toHex(this: Number) {
  return this.toString(16);
}

function numberToString(n: ThisParameterType<typeof toHex>) {
  return toHex.apply(n);
}
```

OmitThisParameter<Type>

Removes the this parameter from `Type`. If `Type` has no explicitly declared `this` parameter, the result is simply `Type`. Otherwise, a new function type with no `this` parameter is created from `Type`. Generics are erased and only the last overload signature is propagated into the new function type.

Released:
3.3

Example

```
function toHex(this: Number) {
  return this.toString(16);
}

const fiveToHex: OmitThisParameter<typeof toHex> = toHex.bind(5);

console.log(fiveToHex());
```

ThisType<Type>

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.

Released:
2.3

Example

```
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): void }`. 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.

Intrinsic String Manipulation Types

`Uppercase<StringType>`

`Lowercase<StringType>`

`Capitalize<StringType>`

`Uncapitalize<StringType>`

To help with string manipulation around template string literals, TypeScript includes a set of types which can be used in string manipulation within the type system. You can find those in the Template Literal Types documentation.