# The TypeScript Handbook Reference

## Advanced Types

This page lists some of the more advanced ways in which you can model types, it works in tandem with the [Utility Types](https://www.typescriptlang.org/docs/handbook/utility-types.html) doc which includes types which are included in TypeScript and available globally.

## Type Guards and Differentiating Types

Union types are useful for modeling situations when values can overlap in the types they can take on. What happens when we need to know specifically whether we have a Fish? A common idiom in JavaScript to differentiate between two possible values is to check for the presence of a member. As we mentioned, you can only access members that are guaranteed to be in all the constituents of a union type.

let pet = getSmallPet();

// You can use the 'in' operator to check

if ("swim" in pet) {

pet.swim();

}

// However, you cannot use property access

if (pet.fly) {

Property 'fly' does not exist on type 'Fish | Bird'.

Property 'fly' does not exist on type 'Fish'.Property 'fly' does not exist on type 'Fish | Bird'.

Property 'fly' does not exist on type 'Fish'. pet.fly();

Property 'fly' does not exist on type 'Fish | Bird'.

Property 'fly' does not exist on type 'Fish'.Property 'fly' does not exist on type 'Fish | Bird'.

Property 'fly' does not exist on type 'Fish'.}[Try](https://www.typescriptlang.org/play/#code/PTAEAEFMCdoe2gZwFygEwGYME4BQAXATwAdJQAxAS0QAtQBeUAb1EQHdKBbVACgEoGAPlAA3OJQAmoAL4BuAiTIAhStCmMWAMwA2hXgPrCxkmfImQAxtoCG0MpoCuAOwv5KcJ6ADmkfAGVOa21tAAVfflQqWlAAH1AVNXkQUABaNIsHfDSU3G1fUFJ8Bm9fAKDQ8L55XGSATTgHUAtrTwdEMnwaMgBySidu0DhSaGt8BFAxpq6LAGtcSk1QHgAidi5l0D6C3wEmXFBt-AA6Nc5+eWkasAAJODZIERgAGlBCBqaWpzgitrJieGGRFA1gsFkgiEQ80WPEKRx0hF2+0OcN051w0iAA)

To get the same code working via property accessors, we’ll need to use a type assertion:

let pet = getSmallPet();

let fishPet = pet as Fish;

let birdPet = pet as Bird;

if (fishPet.swim) {

fishPet.swim();

} else if (birdPet.fly) {

birdPet.fly();

}[Try](https://www.typescriptlang.org/play/#code/C4TwDgpgBAYglgZwBZQLxQN5QQdzgWwC4oAKASjQD4oA3AezgBMoBfAbgChRIoAhOAE7N0WAGYAbEMXJVaDZuw6MIAY3EBDAdFEBXAHYrgcOnqgBzCMADK+dePEAFS+WLxkUAD59BjTgHo-KABaEJUdYBCgjnFLKEhgNHNLGztHZzJOGITRRCQnBPR4qHUEWFzM2IAjH3zEopLvIU4OOFFSHOR8gDpcAgoMDigoDrzLHrx8ck4WKAhxBGhW0mqhbokQfsGoFcY1ySmOFiA)

This isn’t the sort of code you would want in your codebase however.

## User-Defined Type Guards

It would be much better if once we performed the check, we could know the type of pet within each branch.

It just so happens that TypeScript has something called a type guard. A type guard is some expression that performs a runtime check that guarantees the type in some scope.

#### Using type predicates

To define a type guard, we simply need to define a function whose return type is a type predicate:

function isFish(pet: Fish | Bird): pet is Fish {

return (pet as Fish).swim !== undefined;

}[Try](https://www.typescriptlang.org/play/#code/C4TwDgpgBAYglgZwBZQLxQN5QQdzgWwC4oAKASjQD4oA3AezgBMoBfAbgChRIoAhOAE7N0WAGYAbEMXJVaDZuw6MIAY3EBDAdFEBXAHYrgcOnqgBzCMADK+dePEAFS+WLxkUAD59BjTgHo-KABaEJUdYBCgjl0DIxMoRDckEkhgV0QUL34hMmJUhIRYDMwOKCgtYB0BUxTLKHVCpLIAOlwCKABCVHR9ZVE4PQhfDhYgA)

pet is Fish is our type predicate in this example. A predicate takes the form parameterName is Type, where parameterName must be the name of a parameter from the current function signature.

Any time isFish is called with some variable, TypeScript will narrow that variable to that specific type if the original type is compatible.

// Both calls to 'swim' and 'fly' are now okay.

let pet = getSmallPet();

if (isFish(pet)) {

pet.swim();

} else {

pet.fly();

}[Try](https://www.typescriptlang.org/play/#code/C4TwDgpgBAYglgZwBZQLxQN5QQdzgWwC4oAKASjQD4oA3AezgBMoBfAbgChRIoAhOAE7N0WAGYAbEMXJVaDZuw6MIAY3EBDAdFEBXAHYrgcOnqgBzCMADK+dePEAFS+WLxkUAD59BjTroNGJlCIbkgkkMCuiChe-EJkxBHBCLDRmBxQUFrAOgKm4ZZQ6imhZAB0uARQAISo6PrKonB6EL4cLBwA9J1QALT9KjrA-b1dPbx0wCgqduIpwHRQAOSV+EtFesxLEiDrmtB6dDhQdADW6iBlHOKFSegW1rb2TsDknBxwoqQh0QXAZBQMBkoBEKnh8G92lAIHNoEDMqCdpCWEA)

Notice that TypeScript not only knows that pet is a Fish in the if branch; it also knows that in the else branch, you don’t have a Fish, so you must have a Bird.

You may use the type guard isFish to filter an array of Fish | Bird and obtain an array of Fish:

const zoo: (Fish | Bird)[] = [getSmallPet(), getSmallPet(), getSmallPet()];

const underWater1: Fish[] = zoo.filter(isFish);

// or, equivalently

const underWater2: Fish[] = zoo.filter<Fish>(isFish);

const underWater3: Fish[] = zoo.filter<Fish>(pet => isFish(pet));

Argument of type '(pet: Fish | Bird) => boolean' is not assignable to parameter of type '(value: Fish | Bird, index: number, array: (Fish | Bird)[]) => value is Fish'.

Signature '(pet: Fish | Bird): boolean' must be a type predicate.Argument of type '(pet: Fish | Bird) => boolean' is not assignable to parameter of type '(value: Fish | Bird, index: number, array: (Fish | Bird)[]) => value is Fish'.

Signature '(pet: Fish | Bird): boolean' must be a type predicate.[Try](https://www.typescriptlang.org/play/#code/PTAEAEFMCdoe2gZwFygEwGYAsBWAUAC4CeADpKAGICWiAFqALygDeoiA7lQLaoAUAlIwB8oAG5wqAE1ABfANyFS5AEJVo0pqwBmAGyJ9BDEeKmyFkyAGMdAQ2jktAVwB2lglTjPQAc0gEAylw2OjoACn4CqNR0oAA+oKrqCk6u7p6gNNG0vGQEUTT08YmS-Ki5GYiUBSx4oKD2BI7QXjl+oDaVWfwAdBzcoACEDEwuFlpUzpCSCjJ4IKAAtEuWjgRLC3iWnogEoABecHB8WXEJaiUA2gC6jKAXvgFBIeEEAgA0Pn6BwWER-B8Pb7PP5XBRbZw7UCjGAAdRsBBgAEZ8nRrrcDnBuuMdAjoLxMgV+Ap5ggPpAAI6OKiiYKQZwEPSbba7aHQOG4tAo2hopgYrFUHEwAA8WSE+MQXTBzKhzgsbPhMAwXJ5+0O-MF0BFBTF5SMFSyrQI-CJQA)

### Using the in operator

The in operator also acts as a narrowing expression for types.

For a n in x expression, where n is a string literal or string literal type and x is a union type, the “true” branch narrows to types which have an optional or required property n, and the “false” branch narrows to types which have an optional or missing property n.

function move(pet: Fish | Bird) {

if ("swim" in pet) {

return pet.swim();

}

return pet.fly();

}[Try](https://www.typescriptlang.org/play/#code/C4TwDgpgBAYglgZwBZQLxQN5QQdzgWwC4oAKASjQD4oA3AezgBMoBfAbgChRIoAhOAE7N0WAGYAbEMXJVaDZuw4B6JVAC0GgMYBXYBrUdR2gHabgcOsaj46NCCUjBi8ZFAA+fQYwoYOUKHCipABEuATBAVaOPn7+UAIQwNoCUYkAdGH45Jz+LLEJSSlQjmkSINkcLEA)

## typeof type guards

Let’s go back and write the code for a version of padLeft which uses union types. We could write it with type predicates as follows:

function isNumber(x: any): x is number {

return typeof x === "number";

}

function isString(x: any): x is string {

return typeof x === "string";

}

function padLeft(value: string, padding: string | number) {

if (isNumber(padding)) {

return Array(padding + 1).join(" ") + value;

}

if (isString(padding)) {

return padding + value;

}

throw new Error(`Expected string or number, got '${padding}'.`);

}[Try](https://www.typescriptlang.org/play/#code/GYVwdgxgLglg9mABDAzgORAWwEYFMBOAFAB4BciAhmAJ4CU5xyKiYWe+iA3gFCKL64oIfEijUADrjjBEjALwLEAIlY4CSgNzcAvt26hIsBEwDKUfDDABzEuSp0GTRCnOWrXXv0HDREqTPlFJRcLa00dPQNoeCRxCgATABlcYChCADcKABsQXHIQtwAaRDj4+Ld812tEAB8WNgJaDz4YGUJUDDUiUvLrWiaePj4BIRFEAEF8fApqQh63RABqRABGWgA6ACs4S0IlZSblzJzcLT5dFrbUM1Cbeb6Bz2HvMfv3I+zcs8QLxCgAC3wcAA7ixcKCAKJTOBEAAGEOIkmguHiziq7hh9S6xSscCgiAA5AASThvbQE9aw2habRAA)

However, having to define a function to figure out if a type is a primitive is kind of a pain. Luckily, you don’t need to abstract typeof x === "number" into its own function because TypeScript will recognize it as a type guard on its own. That means we could just write these checks inline.

function padLeft(value: string, padding: string | number) {

if (typeof padding === "number") {

return Array(padding + 1).join(" ") + value;

}

if (typeof padding === "string") {

return padding + value;

}

throw new Error(`Expected string or number, got '${padding}'.`);

}[Try](https://www.typescriptlang.org/play/#code/GYVwdgxgLglg9mABABwIYBMAyBTYUAUAbqgDYjYBciAzlAE4xgDmANChuo01bQ84gB9EYEAFsARtjoBKRAG8AUIkQxgifFACeybHDVp0nfgF5TiAEQiJU87MXLldbFBB0kAQTp1Um-AaNMiADUiACM0gB0AFZwjPjmFrIhxGTYANxKiAC+marqWjp67IZciKbGFrxctvKZjs6uSP6lyaTkGco5ylAAFnRwAO7C2EMAol5wdPgABqMAHjrQ2Og09KWTwmKSdGxMcFCIAOQAJHLNzFmHEdPSGVlAA)

These *typeof* type guards are recognized in two different forms: typeof v === "typename" and typeof v !== "typename", where "typename" can be one of [typeof operator’s return values](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/typeof#Description) ("undefined", "number", "string", "boolean", "bigint", "symbol", "object", or "function"). While TypeScript won’t stop you from comparing to other strings, the language won’t recognize those expressions as type guards.

## instanceof type guards

If you’ve read about typeof type guards and are familiar with the instanceof operator in JavaScript, you probably have some idea of what this section is about.

*instanceof* type guards are a way of narrowing types using their constructor function. For instance, let’s borrow our industrial strength string-padder example from earlier:

interface Padder {

getPaddingString(): string;

}

class SpaceRepeatingPadder implements Padder {

constructor(private numSpaces: number) {}

getPaddingString() {

return Array(this.numSpaces + 1).join(" ");

}

}

class StringPadder implements Padder {

constructor(private value: string) {}

getPaddingString() {

return this.value;

}

}

function getRandomPadder() {

return Math.random() < 0.5

? new SpaceRepeatingPadder(4)

: new StringPadder(" ");

}

let padder: Padder = getRandomPadder();

// ^ = let padder: Padder

if (padder instanceof SpaceRepeatingPadder) {

padder;

// ^ = Could not get LSP result: er;>

// < /

}

if (padder instanceof StringPadder) {

padder;

// ^ = Could not get LSP result: er;>

// < /

}[Try](https://www.typescriptlang.org/play/#code/JYOwLgpgTgZghgYwgAgApwCYesg3gKGWQHMIx0tRiBlMKKgCgEoAuZAZzqoG58BffPgQAbOO3bJqAB0QQAShCkQ4YKhWxRkwALZThEbRHAT1OAkQQB7EJygBXBGEtQGU+gDcVKEHe3TZ7Gw+2gBG0Ex4AkSk5JgYVLT0IMTMeIREyFBkdlAgyACCUFBwAJ4MYAAWwOwAdMH+SBIA1MgAjEw1AFaWoAwARMh9TLxEAgJCouKSXMmmmjp6BkZgJnFm6VY2dA5OLm7AnpDInsJ2EGy2VBG4USRk6gkzKdfpRFlgOXmV1TUnZyPIMaCGB2ECOYDWO5gORwEAYSzaOapcyZbK5ZAAWRUFRqxThCNSAB5kAAGGoAVleyAA-MgQBAAO6SGRIBRKFRqNYuAAsTCpQUZ0ySxCRA0Gw34gn0YGQMiw0DYc2QAF4oTD8YiucxeAB6HUZA0APWpgmAMGQri5Wi2sKQlnNDXkimUqlmXJeRDlGgBeoNyGNkrNFq9OFAnFtEHtQs58qgHtlXJ9+qNJr4QA)

The right side of the instanceof needs to be a constructor function, and TypeScript will narrow down to:

1. the type of the function’s prototype property if its type is not any
2. the union of types returned by that type’s construct signatures

in that order.

## Nullable types

TypeScript has two special types, null and undefined, that have the values null and undefined respectively. We mentioned these briefly in [the Basic Types section](https://www.typescriptlang.org/docs/handbook/basic-types.html).

By default, the type checker considers null and undefined assignable to anything. Effectively, null and undefined are valid values of every type. That means it’s not possible to stop them from being assigned to any type, even when you would like to prevent it. The inventor of null, Tony Hoare, calls this his [“billion dollar mistake”](https://wikipedia.org/wiki/Null_pointer#History).

The [--strictNullChecks](https://www.typescriptlang.org/tsconfig#strictNullChecks) flag fixes this: when you declare a variable, it doesn’t automatically include null or undefined. You can include them explicitly using a union type:

let exampleString = "foo";

exampleString = null;

Type 'null' is not assignable to type 'string'.Type 'null' is not assignable to type 'string'.

let stringOrNull: string | null = "bar";

stringOrNull = null;

stringOrNull = undefined;

Type 'undefined' is not assignable to type 'string | null'.Type 'undefined' is not assignable to type 'string | null'.[Try](https://www.typescriptlang.org/play/#code/PTAEAEFMCdoe2gZwFygEwGY1oFABtIAXUSADwEMBbABwIGVDoBLAOwHNQBeUAIgDM4cHgG4cZKrUgNm7LqBYBXPHlH4ioRI1ZsA8tABySvKk0yOAH3lG5PAEbloInKe17DyuYuWqX7N9e4FFgATSD5WSGDhIA)

Note that TypeScript treats null and undefined differently in order to match JavaScript semantics. string | null is a different type than string | undefined and string | undefined | null.

From TypeScript 3.7 and onwards, you can use [optional chaining](https://www.typescriptlang.org/docs/handbook/release-notes/typescript-3-7.html#optional-chaining) to simplify working with nullable types.

### Optional parameters and properties

With [--strictNullChecks](https://www.typescriptlang.org/tsconfig#strictNullChecks), an optional parameter automatically adds | undefined:

function f(x: number, y?: number) {

return x + (y ?? 0);

}

f(1, 2);

f(1);

f(1, undefined);

f(1, null);

Argument of type 'null' is not assignable to parameter of type 'number | undefined'.Argument of type 'null' is not assignable to parameter of type 'number | undefined'.[Try](https://www.typescriptlang.org/play/#code/PTAEAEFMCdoe2gZwFygEwGYAsBWAUAGYCuAdgMYAuAlnCaAQBQAeqJRAtgEYwA0oAngH5WHbtACUoAN55QoaJApFodJqADUoBv1CDBoAAziA3HgC+eQgwCMfNCavWHjW6FIATSASolI75zZ8bAA2wSZAA)

The same is true for optional properties:

class C {

a: number;

b?: number;

}

let c = new C();

c.a = 12;

c.a = undefined;

Type 'undefined' is not assignable to type 'number'.Type 'undefined' is not assignable to type 'number'.c.b = 13;

c.b = undefined;

c.b = null;

Type 'null' is not assignable to type 'number | undefined'.Type 'null' is not assignable to type 'number | undefined'.[Try](https://www.typescriptlang.org/play/#code/PTAEAEGcBcCcEsDG0BcoBmBDANpApgFAgQwLIByArttgMIAWeiA1pGnJYceHrLAPaw2oAEwBmESIKJsmSJFC1QAbwKhQmNADtKAWwBGvANxrQ+gPza9h2CYC+BAtjzRQiUAF5QWvAHdFABQAlCbSAHSYnqAAjCImiBFRlFoAJnjo8D4p8WH6UdFiOXleyWkZWUVROjRGQA)

### Type guards and type assertions

Since nullable types are implemented with a union, you need to use a type guard to get rid of the null. Fortunately, this is the same code you’d write in JavaScript:

function f(stringOrNull: string | null): string {

if (stringOrNull === null) {

return "default";

} else {

return stringOrNull;

}

}[Try](https://www.typescriptlang.org/play/#code/GYVwdgxgLglg9mABMAFAZygJxmA5geUwDkQAbUgLkQ2z0QB9EwzSBKKmnXRAbwChEiGMETosXQiXKIAvHKYtWvAYMSYAplBCYkAIgAm64AEMyUXQG4VAX0TrSadctVrN2pJzySWVwdb7WQA)

The null elimination is pretty obvious here, but you can use terser operators too:

function f(stringOrNull: string | null): string {

return stringOrNull ?? "default";

}[Try](https://www.typescriptlang.org/play/#code/GYVwdgxgLglg9mABMAFAZygJxmA5geUwDkQAbUgLkQ2z0QB9EwzSBKKmnXRAbwChEiTAFMoITEk55CJcogD88xACIAJsOABDMlGUBuPgF8gA)

In cases where the compiler can’t eliminate null or undefined, you can use the type assertion operator to manually remove them. The syntax is postfix !: identifier! removes null and undefined from the type of identifier:

interface UserAccount {

id: number;

email?: string;

}

const user = getUser("admin");

user.id;

Object is possibly 'undefined'.Object is possibly 'undefined'.

if (user) {

user.email.length;

Object is possibly 'undefined'.Object is possibly 'undefined'.}

// Instead if you are sure that these objects or fields exist, the

// postfix ! lets you short circuit the nullability

user!.email!.length;[Try](https://www.typescriptlang.org/play/#code/PTAEAEFMCdoe2gZwFygEwFYDMaBQAzAVwDsBjAFwEs5jQBzScgVURgApKATVRc6S4nQCUqFjACCpUnBLlQAH1AlOkfAMidQAb1yhQ0RoWi0tAX1ABDRJeIBPANy5TuEKAC0H0oXIe3uAeQw+BakkKBi0JLSstq6oFyoxIQAtgBGMI56kMkWlAA2APw8fAJ0js640sS8SqzQoAC89IwRbABEFpzJAm1CjoR1AHRcjv74oGwDMEKxelPQg9m5eYN5kILkABbluC5gAJLVgZ3x47YylgagiEZhWxZyW5CsoHCpAFaQFNYIoGqQeU41kgAA9KLwADSgJ57UAABzgvDUINAAEJQGtyNZzoRrpsEHJSJRoF5KI9NmEknk8hZUvkybZcPNUYscvkWWsNtsgA)

## Type Aliases

Type aliases create a new name for a type. Type aliases are sometimes similar to interfaces, but can name primitives, unions, tuples, and any other types that you’d otherwise have to write by hand.

type Second = number;

let timeInSecond: number = 10;

let time: Second = 10;[Try](https://www.typescriptlang.org/play/#code/C4TwDgpgBAyhDGB7AdgEygXisgrgWwCMIAnAbgChyAbCYKYASzwgElk4k0AubfI4zFACMABgo06jZjw4p0WUaSA)

Aliasing doesn’t actually create a new type - it creates a new name to refer to that type. Aliasing a primitive is not terribly useful, though it can be used as a form of documentation.

Just like interfaces, type aliases can also be generic - we can just add type parameters and use them on the right side of the alias declaration:

type Container<T> = { value: T };

We can also have a type alias refer to itself in a property:

type Tree<T> = {

value: T;

left?: Tree<T>;

right?: Tree<T>;

};

Together with [intersection](https://www.typescriptlang.org/docs/handbook/unions-and-intersections.html) types, we can make some pretty mind-bending types:

type LinkedList<Type> = Type & { next: LinkedList<Type> };

interface Person {

name: string;

}

let people = getDriversLicenseQueue();

people.name;

people.next.name;

people.next.next.name;

people.next.next.next.name;

// ^ = (property) next: LinkedList[Try](https://www.typescriptlang.org/play/#code/CYUwxgNghgTiAEAzArgOzAFwJYHtXwHMQMARGLANxBgGcAZLMEVGkARWRE4AoBKALngNUAaxDAGNDAB4ACtRp4AfAG4AUAHoN8ALR6wyDHp1qMATwAOCYWIlYp0gCqWQS+AF54zq-ABk8AG94VBAADwxBG3FJGW9XeABfdTUsVAxqRCgmeHlaPEC1eGCoAFsQQSlyVAJ1BLU1CGJ4KxwLRo9CYjJKBQYmFnZOHl51FraQADpUUpBRkFbGqbCMKZm5hcmQ8KXt6bL18Z2VrePl1f3NbSLrm9vrgD0AfiA)

## Interfaces vs. Type Aliases

As we mentioned, type aliases can act sort of like interfaces; however, there are some subtle differences.

Almost all features of an interface are available in type, the key distinction is that a type cannot be re-opened to add new properties vs an interface which is always extendable.

|  |  |
| --- | --- |
| **Interface** | **Type** |
| Extending an interface  interface Animal {  name: string  }  interface Bear extends Animal {  honey: boolean  }  const bear = getBear()  bear.name  bear.honey | Extending a type via intersections  type Animal = {  name: string  }  type Bear = Animal & {  honey: Boolean  }  const bear = getBear();  bear.name;  bear.honey; |
| Adding new fields to an existing interface  interface Window {  title: string  }  interface Window {  ts: import("typescript")  }  const src = 'const a = "Hello World"';  window.ts.transpileModule(src, {}); | A type cannot be changed after being created  type Window = {  title: string  }  type Window = {  ts: import("typescript")  }  // Error: Duplicate identifier 'Window'. |

Because an interface more closely maps how JavaScript objects work [by being open to extension](https://wikipedia.org/wiki/Open/closed_principle), we recommend using an interface over a type alias when possible.

On the other hand, if you can’t express some shape with an interface and you need to use a union or tuple type, type aliases are usually the way to go.

## Enum Member Types

As mentioned in [our section on enums](https://www.typescriptlang.org/docs/handbook/enums.html#union-enums-and-enum-member-types), enum members have types when every member is literal-initialized.

Much of the time when we talk about “singleton types”, we’re referring to both enum member types as well as numeric/string literal types, though many users will use “singleton types” and “literal types” interchangeably.

## Polymorphic this types

A polymorphic this type represents a type that is the subtype of the containing class or interface. This is called F-bounded polymorphism, a lot of people know it as the [fluent API](https://en.wikipedia.org/wiki/Fluent_interface) pattern. This makes hierarchical fluent interfaces much easier to express, for example. Take a simple calculator that returns this after each operation:

class BasicCalculator {

public constructor(protected value: number = 0) {}

public currentValue(): number {

return this.value;

}

public add(operand: number): this {

this.value += operand;

return this;

}

public multiply(operand: number): this {

this.value \*= operand;

return this;

}

// ... other operations go here ...

}

let v = new BasicCalculator(2).multiply(5).add(1).currentValue();[Try](https://www.typescriptlang.org/play/#code/MYGwhgzhAEBCkEtgGEwmAV3AFwPYCdoBvAKGmgAcMAjEJaYXAOwm3w2D3wAoL9dsAU06CAJtABuaDIIBc0JhgC21QYQC80AAwBKYgF8ylGnWAMM+fIKbYAatMHcd8xSrXEj5K9gtNo2AAsECAA6KRAZAG4jQ3IqWnowUVFuXAo1MCZRF2VVfGd-IJhScnJA4LCHaABqTTSMrOjS6G9fQuCm6FjjBLMlLGwEChAAT1T0-EzshVy1AvLiz3bQ8JloACo6ianOr0EffD8Fzu6AelPoEKvoAQD3esnB5hgAc1xoO6tLq5JDEhB9pJoJomIIAO5wRAoNCYHAEbgAJh0IX6IEGwzGAFZkUkUgBGZGYSzWOwOJyRIA)

Since the class uses this types, you can extend it and the new class can use the old methods with no changes.

class ScientificCalculator extends BasicCalculator {

public constructor(value = 0) {

super(value);

}

public sin() {

this.value = Math.sin(this.value);

return this;

}

// ... other operations go here ...

}

let v = new ScientificCalculator(2).multiply(5).sin().add(1).currentValue();[Try](https://www.typescriptlang.org/play/#code/MYGwhgzhAEBCkEtgGEwmAV3AFwPYCdoBvAKGmgAcMAjEJaYXAOwm3w2D3wAoL9dsAU06CAJtABuaDIIBc0JhgC21QYQC80AAwBKYgF8ylGnWAMM+fIKbYAatMHcd8xSrXEj5K9gtNo2AAsECAA6KRAZAG4jQ3IqWnowUVFuXAo1MCZRF2VVfGd-IJhScnJA4LCHaABqTTSMrOjS6G9fQuCm6FjjBLMlLGwEChAAT1T0-EzshVy1AvLiz3bQ8JloACo6ianOr0EffD8Fzu6AelPoEKvoAQD3esnB5hgAc1xoO6tLq5JDc+gALRAzDYIEAkigSAwADKwAQ1kGADMkKh0FgwFxoIIAB5CLIweAQFFoTA4AgeOImeiMFhsDhcbirQTQTS6CmlCAYCaMhw6E5GeKmaBEphOdllIqVCLMzQAWQxARCIu4CylMj5S1ah2W-PI-yuIRugXu2yeLGgbw+amZBt+JBIIH2khZCkEAHdoLD4TYEMiUCT0QyAEw6EL9ECDYZjACsoeVoaSKQAjKHMJYEfZpU5IkA)

Without this types, ScientificCalculator would not have been able to extend BasicCalculator and keep the fluent interface. multiply would have returned BasicCalculator, which doesn’t have the sin method. However, with this types, multiply returns this, which is ScientificCalculator here.

## Index types

With index types, you can get the compiler to check code that uses dynamic property names. For example, a common JavaScript pattern is to pick a subset of properties from an object:

function pluck(o, propertyNames) {

return propertyNames.map((n) => o[n]);

}

Here’s how you would write and use this function in TypeScript, using the **index type query** and **indexed access** operators:

function pluck<T, K extends keyof T>(o: T, propertyNames: K[]): T[K][] {

return propertyNames.map((n) => o[n]);

}

interface Car {

manufacturer: string;

model: string;

year: number;

}

let taxi: Car = {

manufacturer: "Toyota",

model: "Camry",

year: 2014,

};

// Manufacturer and model are both of type string,

// so we can pluck them both into a typed string array

let makeAndModel: string[] = pluck(taxi, ["manufacturer", "model"]);

// If we try to pluck model and year, we get an

// array of a union type: (string | number)[]

let modelYear = pluck(taxi, ["model", "year"]);[Try](https://www.typescriptlang.org/play/#code/GYVwdgxgLglg9mABABwDYggawDwBUA0iA0ogKYAeUpYAJgM6KakCecwiuAfABRwBcHQsgBOcZKWFRmAOQCGAW1J0BRANoBdAJQDcqouo2IA3gChEiYaSghhSEWIlS5iugDp5s5N25hNiALyciHCqYFoA3CYAviYmMGBUwsCyEKSIAMKywsZmiB5gIMnQNhICdFDC8QDmkebycDSkqGUV1bWIzKRZAgXyAEYSkTEmqFaIULLkMAKZ2f45dbIFRdaWwgIARLhwrBMb+Ln1jc2IG5nywsz7uZ3diABMAAwAjAAsB1GRJgD034gAsktCilVhJEEsaHkGk1wZZEH04FAABbBdhScSIcqVMBVA6-TFwRAAdzSECWKHQWHGSNI8nhiJR8SghNk42Y4khWOqsOEsmYIzGHiYAEFaP9oScuTjDPM0BhMNwJlNCKoNvlgcU1vtTkcmhsIrF8QBJdgk8aXcaEuVU3WocG0DpdYSEM1VMZLH5-LK85io8GIcDwJDo0gCbhSqqIAA+iF6A2Emg0AqgUOOAE0nQEKfLFZMYCq1RLtRtbsJ9ZpwkA)

The compiler checks that manufacturer and model are actually properties on Car. The example introduces a couple of new type operators. First is keyof T, the **index type query operator**. For any type T, keyof T is the union of known, public property names of T. For example:

let carProps: keyof Car;

// ^ = let carProps: keyof Car[Try](https://www.typescriptlang.org/play/#code/JYOwLgpgTgZghgYwgAgMJysg3gKGcgWzhAFd4EwSpoAuZAZzClAHMBuPQgewBMIAbOo2Yh2nAJ4QMdUgQBG0DgF8cAelXIAtNoQkw2zTn4QwyBBgAKULgAd6dANYRxXGGgwd1+bz4B6AfiA)

keyof Car is completely interchangeable with "manufacturer" | "model" | "year". The difference is that if you add another property to Car, say ownersAddress: string, then keyof Car will automatically update to be "manufacturer" | "model" | "year" | "ownersAddress". And you can use keyof in generic contexts like pluck, where you can’t possibly know the property names ahead of time. That means the compiler will check that you pass the right set of property names to pluck:

// error, Type '"unknown"' is not assignable to type '"manufacturer" | "model" | "year"'

pluck(taxi, ["year", "unknown"]);

The second operator is T[K], the **indexed access operator**. Here, the type syntax reflects the expression syntax. That means that taxi["manufacturer"] has the type Car["manufacturer"] — which in our example is just string. However, just like index type queries, you can use T[K] in a generic context, which is where its real power comes to life. You just have to make sure that the type variable K extends keyof T. Here’s another example with a function named getProperty.

function getProperty<T, K extends keyof T>(o: T, propertyName: K): T[K] {

return o[propertyName]; // o[propertyName] is of type T[K]

}

In getProperty, o: T and propertyName: K, so that means o[propertyName]: T[K]. Once you return the T[K] result, the compiler will instantiate the actual type of the key, so the return type of getProperty will vary according to which property you request.

let manufacturer: string = getProperty(taxi, "manufacturer");

let year: number = getProperty(taxi, "year");

let unknown = getProperty(taxi, "unknown");

Argument of type '"unknown"' is not assignable to parameter of type 'keyof Car'.Argument of type '"unknown"' is not assignable to parameter of type 'keyof Car'.[Try](https://www.typescriptlang.org/play/#code/PTAEAEFMCdoe2gZwFygEwGYAsBWAUAGYCuAdgMYAuAlnCaAOaQUAK8ADjBQJ4A8AKgBpQAaVCQAHhUgkAJolABrSFzgFQfAHwAKOKkGg27TlwByAQwC2kVMICUegNrCAuqADeeUKGhMi0OnAOhnAc0NzmVs4A3KAgoIHBoeGWkK5U8qqg3BzqTs54AL54VCRS0ARmZJCgAMJm0O6eoBZmJEQVlH4wqIgU0CX0UU0WcDKQADY9fQNDXlyQ9ahtFgBGMENF40xZZuJUqHUNALyNXi1tHRRd0KgARHxwKhRmtwLDoxN3dRbQXK9N80W6AADABGLBvApDOIAWjhZCIFDhMLwWwozVa7UqVx8N1AvX6JHooBOjBYRjCXC0zz2Qlu5yxnVxt1sQzRoEBeOWa2ODCYrBCxmpuyodM5LKGqO2pAUJDgAHc6KT+RTuMLaaBbjK5YqJUA)

## Index types and index signatures

keyof and T[K] interact with index signatures. An index signature parameter type must be ‘string’ or ‘number’. If you have a type with a string index signature, keyof T will be string | number (and not just string, since in JavaScript you can access an object property either by using strings (object["42"]) or numbers (object[42])). And T[string] is just the type of the index signature:

interface Dictionary<T> {

[key: string]: T;

}

let keys: keyof Dictionary<number>;

// ^ = let keys: string | number

let value: Dictionary<number>["foo"];

// ^ = let value: number[Try](https://www.typescriptlang.org/play/#code/JYOwLgpgTgZghgYwgAgCLAWYB7EcoCeAPACoB8yA3gFDLIDaA1hAQFzIDOYUoA5gLrsSAbmoBfagBsIYZMwId287DDQYsufMRABXALYAjaGVEB6U3UsA9APxSZyAG5xJOiO3SYceQkV2HjegAiGGxsIP4zC0s6WyA)

If you have a type with a number index signature, keyof T will just be number.

interface Dictionary<T> {

[key: number]: T;

}

let keys: keyof Dictionary<number>;

// ^ = let keys: number

let numberValue: Dictionary<number>[42];

// ^ = let numberValue: number

let value: Dictionary<number>["foo"];

Property 'foo' does not exist on type 'Dictionary<number>'.Property 'foo' does not exist on type 'Dictionary<number>'.[Try](https://www.typescriptlang.org/play/#code/PTAEAEFMCdoe2gZwFygEwGYME4BQBLAOwBcYAzAQwGNJQARfK4-OQi6ATwB4AVAPlABvXKFABtANaQOqQgFcAtgCMYAXVQ8A3LgC+uXABtIxUFI4pT0uGXqNmrdt3nKYfbSFGeAegH5Dx0GcVaAA1CgM5SFQGJhY2Ti4g1zEAFjRVdzBPUF9-EwA3cMjouzjHRMVgvjEAIjI4OBqMoA)

## Mapped types

A common task is to take an existing type and make each of its properties optional:

interface PersonSubset {

name?: string;

age?: number;

}

Or we might want a readonly version:

interface PersonReadonly {

readonly name: string;

readonly age: number;

}

This happens often enough in JavaScript that TypeScript provides a way to create new types based on old types — **mapped types**. In a mapped type, the new type transforms each property in the old type in the same way. For example, you can make all properties optional or of a type readonly. Here are a couple of examples:

type Partial<T> = {

[P in keyof T]?: T[P];

};

type Readonly<T> = {

readonly [P in keyof T]: T[P];

};[Try](https://www.typescriptlang.org/play/#code/C4TwDgpgBACghgJ2ASzgGwDwBUB8UC8UA3gFBRQDaMUyAdlANYQgD2AZlFgLoD8AXJypcA3CQC+okgHopUAAK0WAUQQIWCAM4lQkKACUIcACYtaaENjyFS5BIZNmQlanUbN2nLgKxDREoA)

And to use it:

type PersonPartial = Partial<Person>;

// ^ = type PersonPartial = {

// name?: string | undefined;

// age?: number | undefined;

// }

type ReadonlyPerson = Readonly<Person>;

// ^ = type ReadonlyPerson = {

// readonly name: string;

// readonly age: number;

// }[Try](https://www.typescriptlang.org/play/#code/C4TwDgpgBAChBOBnA9gOygXigbwFBSlQEMBbCALikWHgEtUBzAbnyiIYsIFcSAjBFgF8WAehFQAtFIDGXYFIm5QkWAhSoYReMFpEANplhad+gDxwkaAHyjxBAHoB+JeGgAlCEQAmaPSAvqhh7eviDmata2BFBOQA)

Note that this syntax describes a type rather than a member. If you want to add members, you can use an intersection type:

// Use this:

type PartialWithNewMember<T> = {

[P in keyof T]?: T[P];

} & { newMember: boolean }

// This is an error!

type WrongPartialWithNewMember<T> = {

[P in keyof T]?: T[P];

newMember: boolean;

'boolean' only refers to a type, but is being used as a value here.  
'}' expected.'boolean' only refers to a type, but is being used as a value here.  
'}' expected.}

Declaration or statement expected.Declaration or statement expected.[Try](https://www.typescriptlang.org/play/#code/PTAEAEFMCdoe2gZwFygEwDYCcBmUBGABkIFYD80AOAKBFAFVFJQAXACwEsVqWBPAB2YAFAIbQWHEQBsA6h3YA5SAHcAspAC2AIxgAeACoA+UAF5QAb2qhQAbSGgOAO1ABrSLzgAzUPoC6AflR9O18AbmoAX1AAMgtQRxV1bRhULTg4KUgRZwjqWjB9TkQHYuzQGHhoAEIeAWYZeEcAc1FxSVl5NiU1TR1oA2MzS2s7B2c3D28-QJ8Q8OsEnuToVPTM7PCIoA)

Let’s take a look at the simplest mapped type and its parts:

type Keys = "option1" | "option2";

type Flags = { [K in Keys]: boolean };[Try](https://www.typescriptlang.org/play/#code/C4TwDgpgBA0hIGcoF4oCID2ZgEsMDsBGNKAH3S1wICY0BuAKFEigDEAbAQwHMlUBvKAG0YUHPljwEAXQBcUAEYYM7CJwkBfOkA)

The syntax resembles the syntax for index signatures with a for .. in inside. There are three parts:

1. The type variable K, which gets bound to each property in turn.
2. The string literal union Keys, which contains the names of properties to iterate over.
3. The resulting type of the property.

In this simple example, Keys is a hard-coded list of property names and the property type is always boolean, so this mapped type is equivalent to writing:

type Flags = {

option1: boolean;

option2: boolean;

};[Try](https://www.typescriptlang.org/play/#code/C4TwDgpgBAYgNgQwOYGcoF4oG8BQUoD2YwAlgQHYCMAXFAEYEFwQLkDcehxZ5ATLQyYt2OAL5sgA)

Real applications, however, look like Readonly or Partial above. They’re based on some existing type, and they transform the properties in some way. That’s where keyof and indexed access types come in:

type NullablePerson = { [P in keyof Person]: Person[P] | null };

// ^ = type NullablePerson = {

// name: string | null;

// age: number | null;

// }

type PartialPerson = { [P in keyof Person]?: Person[P] };

// ^ = type PartialPerson = {

// name?: string | undefined;

// age?: number | undefined;

// }[Try](https://www.typescriptlang.org/play/#code/C4TwDgpgBAChBOBnA9gOygXigbwFBSlQEMBbCALikWHgEtUBzAbnyiIYsIFcSAjBFgF8WAehFQAtFIDGXYFIm5QkKADkuAGw1FeGiHCRpMOKAG0YUelADWEEMgBmsBClQBdSgdfm3UAD7cWlDCuGIEUAB6APxK4NAwRPDAtEQaXkZY2GYWVrb2TunuUZ4uaD7BouIE0UA)

But it’s more useful to have a general version.

type Nullable<T> = { [P in keyof T]: T[P] | null };

type Partial<T> = { [P in keyof T]?: T[P] };

In these examples, the properties list is keyof T and the resulting type is some variant of T[P]. This is a good template for any general use of mapped types. That’s because this kind of transformation is [homomorphic](https://wikipedia.org/wiki/Homomorphism), which means that the mapping applies only to properties of T and no others. The compiler knows that it can copy all the existing property modifiers before adding any new ones. For example, if Person.name was readonly, Partial<Person>.name would be readonly and optional.

Here’s one more example, in which T[P] is wrapped in a Proxy<T> class:

type Proxy<T> = {

get(): T;

set(value: T): void;

};

type Proxify<T> = {

[P in keyof T]: Proxy<T[P]>;

};

function proxify<T>(o: T): Proxify<T> {

// ... wrap proxies ...

}

let props = { rooms: 4 };

let proxyProps = proxify(props);

// ^ = let proxyProps: Proxify<{

// rooms: number;

// }>[Try](https://www.typescriptlang.org/play/#code/PTAEAEDsHsFECd7XgZwFABcCeAHApqAApIAeWAPACoB8oAvKAN5qigDmeGAFAJQBcoSgG4WoFJy4A3AIYAbAK54BlfqEnQAlgBMRAXxGZcBYtBIaAZhRr0mogNqFQGyKADWeLNHOCAugJNkVA4+1HoG5vKQAMYYGtAuOKQWVtRc0MqqAclUtMysIKAAdMWgAO7w0jigiaYaeChFxWi6aGiynNVIOA0MjKBI0AC2KAIALKD6bR01ZCbdNjPJXDXdPCIFoAB6APxAA)

Note that Readonly<T> and Partial<T> are so useful, they are included in TypeScript’s standard library along with Pick and Record:

type Pick<T, K extends keyof T> = {

[P in K]: T[P];

};

type Record<K extends keyof any, T> = {

[P in K]: T;

};

Readonly, Partial and Pick are homomorphic whereas Record is not. One clue that Record is not homomorphic is that it doesn’t take an input type to copy properties from:

type ThreeStringProps = Record<"prop1" | "prop2" | "prop3", string>;[Try](https://www.typescriptlang.org/play/#code/C4TwDgpgBAKgFgJwhAysBBLAdgcwAoID2YAzlALxQBKEAxoQgCYA8ARGEWAIytQA+UdpwBMvAUOIBmVgBooJdNhwA+ANxA)

Non-homomorphic types are essentially creating new properties, so they can’t copy property modifiers from anywhere.

Note that keyof any represents the type of any value that can be used as an index to an object. In otherwords, keyof any is currently equal to string | number | symbol.

## Inference from mapped types

Now that you know how to wrap the properties of a type, the next thing you’ll want to do is unwrap them. Fortunately, that’s pretty easy:

function unproxify<T>(t: Proxify<T>): T {

let result = {} as T;

for (const k in t) {

result[k] = t[k].get();

}

return result;

}

let originalProps = unproxify(proxyProps);

// ^ = let originalProps: {

// rooms: number;

// }[Try](https://www.typescriptlang.org/play/#code/C4TwDgpgBACgTgewB4gDwBUB8UC8UDeAUFFAOYTAAUAlAFxToDcxUAzhZQG4CGANgK4R66OlE4IAlgBNmAX2aFQkWIiQSAZmiy4CLANowoEgHZQA1hBAJ1DALr14yLQduY5C9f2MBjYBISmYKoaWpiUCMKijmqaGNhEJHAU-HCm+LJQ3KyZxiByhIS8FFBBCGDZePhQiAgAtqz0ACxQ8oXFpSiO5TodIZSl5dTMAPTDUAC0k978wJPjhJ4+fgFQXr2xWJTADsEbmKLouiRFwNUQrPy8p5UZWQzMJOoIcFCU3gGsp2ZGpsDURyQzhcrnozLYdMBQbYAHTkKhDFiyFhJYApUxJYHAfJtU7PCSkEx8LoVVbGdYgfqqEDEhGjEgAPQA-EA)

Note that this unwrapping inference only works on homomorphic mapped types. If the mapped type is not homomorphic you’ll have to give an explicit type parameter to your unwrapping function.

## Conditional Types

A conditional type selects one of two possible types based on a condition expressed as a type relationship test:

T extends U ? X : Y

The type above means when T is assignable to U the type is X, otherwise the type is Y.

A conditional type T extends U ? X : Y is either resolved to X or Y, or deferred because the condition depends on one or more type variables. When T or U contains type variables, whether to resolve to X or Y, or to defer, is determined by whether or not the type system has enough information to conclude that T is always assignable to U.

As an example of some types that are immediately resolved, we can take a look at the following example:

declare function f<T extends boolean>(x: T): T extends true ? string : number;

// Type is 'string | number'

let x = f(Math.random() < 0.5);

// ^ = let x: string | number[Try](https://www.typescriptlang.org/play/#code/CYUwxgNghgTiAEAzArgOzAFwJYHtVIB4AVeEADwxFWAGd4AjHHCEKVAPgAoyAueIgJR8S5StToYYyBAH54NSVlQBzeH1TIAtvRAwA3ACgDAemP8AngAcEWOgHIFMJaoA+8Ddt12DLDPDLwALxInACyUBgAFgB0MGzAOJqcAvAE8AAM0QCsAoam8PAAejJAA)

Another example would be the TypeName type alias, which uses nested conditional types:

type TypeName<T> = T extends string

? "string"

: T extends number

? "number"

: T extends boolean

? "boolean"

: T extends undefined

? "undefined"

: T extends Function

? "function"

: "object";

type T0 = TypeName<string>;

// ^ = type T0 = "string"

type T1 = TypeName<"a">;

// ^ = type T1 = "string"

type T2 = TypeName<true>;

// ^ = type T2 = "boolean"

type T3 = TypeName<() => void>;

// ^ = type T3 = "function"

type T4 = TypeName<string[]>;

// ^ = type T4 = "object"[Try](https://www.typescriptlang.org/play/#code/C4TwDgpgBAKuEDkCGBbCAeGA+KBeWUEAHsBAHYAmAzlFcAE4CWZA5gFBRQD8UARHU1a8OUAFwFipSjTIBXFACMI9ET15zFy4Z3ExCJctSgKA9iYA2EJGVV9TFq2W1iJB6VFmUIAM2YQKtryeFD5+FM66+lJGAGKeAMbAjCY2nGreCUkpEXwmCgBWEIm8ANxsbKCQsAAMeLDwyGjoAswsWGUA9B2cUAB6XBXwsACMdXCQjRi8SLztbF09-YNVMABMYw2oGAyyEHMLnEuV0DAAzBsTW+gAFACUeDgAbiaMFPvdhwPHsAAsF4hXFqsADaAF13osuEA)

But as an example of a place where conditional types are deferred - where they stick around instead of picking a branch - would be in the following:

interface Foo {

propA: boolean;

propB: boolean;

}

declare function f<T>(x: T): T extends Foo ? string : number;

function foo<U>(x: U) {

// Has type 'U extends Foo ? string : number'

let a = f(x);

// This assignment is allowed though!

let b: string | number = a;

}[Try](https://www.typescriptlang.org/play/#code/JYOwLgpgTgZghgYwgAgGIHt3IN4ChnIAOU6hAggFzIBGmANhHCANz5EmEBCVt6DTrAL65cAEwgI6cKChgBXEAjDB0IZDAA8AFQB8ACgAeVLQEpjyCAcghRAZzSZkAfmS2wUUAHNkVEHIC21NCsuPKKyqrqmBoAqvpGyDEmOGwA9KnIABJw9mAAnoQoAOQxFlYQNvYYWC5uHiDevgFBUEVsDGDIcMgAvOqGJiEE6chaABbA9jm2wJ4g-hWdk110dOgA7hCiyGBj6HKeYwCE7RCd1FR1XsgAPsh+gdC9XUJAA)

In the above, the variable a has a conditional type that hasn’t yet chosen a branch. When another piece of code ends up calling foo, it will substitute in U with some other type, and TypeScript will re-evaluate the conditional type, deciding whether it can actually pick a branch.

In the meantime, we can assign a conditional type to any other target type as long as each branch of the conditional is assignable to that target. So in our example above we were able to assign U extends Foo ? string : number to string | number since no matter what the conditional evaluates to, it’s known to be either string or number.

## Distributive conditional types

Conditional types in which the checked type is a naked type parameter are called distributive conditional types. Distributive conditional types are automatically distributed over union types during instantiation. For example, an instantiation of T extends U ? X : Y with the type argument A | B | C for T is resolved as (A extends U ? X : Y) | (B extends U ? X : Y) | (C extends U ? X : Y).

#### Example

type T5 = TypeName<string | (() => void)>;

// ^ = type T5 = "string" | "function"

type T6 = TypeName<string | string[] | undefined>;

// ^ = type T6 = "string" | "undefined" | "object"

type T7 = TypeName<string[] | number[]>;

// ^ = type T7 = "object"[Try](https://www.typescriptlang.org/play/#code/C4TwDgpgBAKuEDkCGBbCAeGA+KBeWUEAHsBAHYAmAzlFcAE4CWZA5gFBRQD8UARHU1a8OUAFwFipSjTIBXFACMI9ET15zFy4Z3ExCJctSgKA9iYA2EJGVV9TFq2W1iJB6VFmUIAM2YQKtryeFD5+FM66+lJGAGKeAMbAjCY2nGreCUkpEXwmCgBWEIm8ANxsAPTlUAC0tfGywLXVbKCQsACseLDwyGjoAswsUAA+UAAUYwCUeDgAbiaMFJNYZZWcUAB6XC3wsABsXXCQvRgDrCO0DIMA2gC6F8GhZP4rFVWcWzttMADshz2oU5XVh3C4aJT0O6vNYfLhAA)

In instantiations of a distributive conditional type T extends U ? X : Y, references to T within the conditional type are resolved to individual constituents of the union type (i.e. T refers to the individual constituents after the conditional type is distributed over the union type). Furthermore, references to T within X have an additional type parameter constraint U (i.e. T is considered assignable to U within X).

#### Example

type BoxedValue<T> = { value: T };

type BoxedArray<T> = { array: T[] };

type Boxed<T> = T extends any[] ? BoxedArray<T[number]> : BoxedValue<T>;

type T1 = Boxed<string>;

// ^ = type T1 = {

// value: string;

// }

type T2 = Boxed<number[]>;

// ^ = type T2 = {

// array: number[];

// }

type T3 = Boxed<string | number[]>;

// ^ = type T3 = BoxedValue | BoxedArray[Try](https://www.typescriptlang.org/play/#code/C4TwDgpgBAQg9gDwgEwGoEMA2BXCAeAFQD4oBeKAbygDctcAuKAqAXwG4AoUSWRFAQQBOg9CEIlyVdMNGMCAbQC6rTt2jwkycWSZQICYBAB2yAM5R0RkEqgB+XpqEixCo9gC2AIwiDFJRhooGDj4xJxc4NAEAIw6gVqmwIIAlkYA5kScAPRZUHkAerYRPAQATHF8Wm5ePkqZHDl5UIXFUQDMFZp4iSnpUAA+UNXegnXZuQW2QA)

Notice that T has the additional constraint any[] within the true branch of Boxed<T> and it is therefore possible to refer to the element type of the array as T[number]. Also, notice how the conditional type is distributed over the union type in the last example.

The distributive property of conditional types can conveniently be used to filter union types:

// Remove types from T that are assignable to U

type Diff<T, U> = T extends U ? never : T;

// Remove types from T that are not assignable to U

type Filter<T, U> = T extends U ? T : never;

type T1 = Diff<"a" | "b" | "c" | "d", "a" | "c" | "f">;

// ^ = type T1 = "b" | "d"

type T2 = Filter<"a" | "b" | "c" | "d", "a" | "c" | "f">; // "a" | "c"

// ^ = type T2 = "a" | "c"

type T3 = Diff<string | number | (() => void), Function>; // string | number

// ^ = type T3 = string | number

type T4 = Filter<string | number | (() => void), Function>; // () => void

// ^ = type T4 = () => void

// Remove null and undefined from T

type NotNullable<T> = Diff<T, null | undefined>;

type T5 = NotNullable<string | number | undefined>;

// ^ = type T5 = string | number

type T6 = NotNullable<string | string[] | null | undefined>;

// ^ = type T6 = string | string[]

function f1<T>(x: T, y: NotNullable<T>) {

x = y;

y = x;

Type 'T' is not assignable to type 'Diff<T, null | undefined>'.Type 'T' is not assignable to type 'Diff<T, null | undefined>'.}

function f2<T extends string | undefined>(x: T, y: NotNullable<T>) {

x = y;

y = x;

Type 'T' is not assignable to type 'Diff<T, null | undefined>'.

Type 'string | undefined' is not assignable to type 'Diff<T, null | undefined>'.

Type 'undefined' is not assignable to type 'Diff<T, null | undefined>'.Type 'T' is not assignable to type 'Diff<T, null | undefined>'.

Type 'string | undefined' is not assignable to type 'Diff<T, null | undefined>'.

Type 'undefined' is not assignable to type 'Diff<T, null | undefined>'. let s1: string = x;

Type 'T' is not assignable to type 'string'.

Type 'string | undefined' is not assignable to type 'string'.

Type 'undefined' is not assignable to type 'string'.Type 'T' is not assignable to type 'string'.

Type 'string | undefined' is not assignable to type 'string'.

Type 'undefined' is not assignable to type 'string'. let s2: string = y;

}[Try](https://www.typescriptlang.org/play/#code/PTAEAEFMCdoe2gZwFygEwGYAMX0bWgFAigBKkAtnAG6SgAuAngA6SKgBm8FoAKgwAsAhvVBDodIYkQBLAOYA7IQCMANnXpxQAVUJNWoACIyOHADy8ANDoB8oALx9QkAB71ICgCbttoAPygCpC00KCovADcxGDkVLQMLGyc3E70wqLidApwGdLySmoaWrr6dABiMqru0BbW2naO-K7uXj7+TqhBIVF6iXwAjA5GJuYAREKjoAA+oKPKkzOjAMYLs56j1uOry9scozZRJKCgAHp+vQa8aEMVVTBmW9Oz8087r+ubE68rr3sHoCRHosVtFjqdzqU+BghsZTGZEPRoDIFHIngoAK4UZQwJ4AClxAEoHHZqHAZJ4CdYyuiFEt6DI4Ap-iQEUiUWjMdjoKDjmcLnReAAWG6VarwxHI1EzDFYnEzfFE+wkskUqk0ukMpkRAFgQnE0Ck8k88GEUGxGhZdGqVRiLygGmeSAcZGQTzJOA8Xj80AAORyPqtqhU6gsDWGcKsgUDTwdTpdngOpshvAArEM-fQA9bg5BxWypVHZaEZrHnUEE4cwLyIX1eAA2dP+wM5vOSp6syUAbQAuhzrTGvHHy4mjibCBx1fTGZx+qHcS5wtZGKgM1mg4VQ0SAN6EY4uIaMKLHRhDFxRAC+pontKnCk4aAszjcHm8oA77JLg7Lrps88XoGXX0m2zDdeBsbdd1AfdHEPSCT0cM9IPUUREH6VB31RBCj1AZC3zQdCJXZGCLyAA)

Conditional types are particularly useful when combined with mapped types:

type FunctionPropertyNames<T> = {

[K in keyof T]: T[K] extends Function ? K : never;

}[keyof T];

type FunctionProperties<T> = Pick<T, FunctionPropertyNames<T>>;

type NonFunctionPropertyNames<T> = {

[K in keyof T]: T[K] extends Function ? never : K;

}[keyof T];

type NonFunctionProperties<T> = Pick<T, NonFunctionPropertyNames<T>>;

interface Part {

id: number;

name: string;

subparts: Part[];

updatePart(newName: string): void;

}

type T1 = FunctionPropertyNames<Part>;

// ^ = type T1 = "updatePart"

type T2 = NonFunctionPropertyNames<Part>;

// ^ = type T2 = "id" | "name" | "subparts"

type T3 = FunctionProperties<Part>;

// ^ = type T3 = {

// updatePart: (newName: string) => void;

// }

type T4 = NonFunctionProperties<Part>;

// ^ = type T4 = {

// id: number;

// name: string;

// subparts: Part[];

// }[Try](https://www.typescriptlang.org/play/#code/C4TwDgpgBAYgrgOwMbAJYHsEAUBO7I6gByAhgLYQDOAPACoB8UAvFAN4BQUUA2gNJSoEUANYQQ6AGZRaAXQBc0vjKgQAHsAgIAJpViIUGIQH4o-BQggA3CDgDc7AL7dR4qbPuhIe5Gky58NmhUdIwsWKhIwnQANN4GfngExOTBDPT27J7QRJjwPob+SSCkFDQMzGycPPyCImKS0vKKvMpqGtq6efHGUBbWOFAKvPZOLg3umeDZufq+2ImBqKmhUOGRMVA5CF1zhYHFKWX06ezsgho4EiRI0FgkhJVcqFrmcGQARjb2XAgpCpTAHCCADm3yglDg7zA92AlAUd0I3BkYLgYC0JA0COAAAoLAB3EoQf6AkEASgUlnQzxGpyy0gAjBUdgUFoQDqVqFiTgB6blcKAAPSMky8tAATBUtsyEgE2YSaFz7Lz+UKRdBaABmJmzFmyoIKmE8vlcVV02gAFklM3yMqSSwNhCNKqMQA)

Similar to union and intersection types, conditional types are not permitted to reference themselves recursively. For example the following is an error.

#### Example

type ElementType<T> = T extends any[] ? ElementType<T[number]> : T; // Error[Try](https://www.typescriptlang.org/play/#code/PTAEAEFMCdoe2gZwFygEwBYCsA2dBmARiwCgAXATwAdJQBRAG0gFtIA7MgFWsgB5OAfKAC8oTqEgAPMuwAmiUAEM2FANoBdUAH56TVh241+qtgFdmAIxjqhqTgG5QIerARA)

## Type inference in conditional types

Within the extends clause of a conditional type, it is now possible to have infer declarations that introduce a type variable to be inferred. Such inferred type variables may be referenced in the true branch of the conditional type. It is possible to have multiple infer locations for the same type variable.

For example, the following extracts the return type of a function type:

type ReturnType<T> = T extends (...args: any[]) => infer R ? R : any;[Try](https://www.typescriptlang.org/play/#code/PTAEAEDsHsFECd7XgZwFABcCeAHApqAEp4YCu8kAKrngDyUB8oAvKJaHgB4Z6QAmKUAAoAdGICG8AOYoAXKHGQsAbQC6AShZMAlpABmeeEVAB+Y-MVYA3EA)

Conditional types can be nested to form a sequence of pattern matches that are evaluated in order:

type Unpacked<T> = T extends (infer U)[]

? U

: T extends (...args: any[]) => infer U

? U

: T extends Promise<infer U>

? U

: T;

type T0 = Unpacked<string>;

// ^ = type T0 = string

type T1 = Unpacked<string[]>;

// ^ = type T1 = string

type T2 = Unpacked<() => string>;

// ^ = type T2 = string

type T3 = Unpacked<Promise<string>>;

// ^ = type T3 = string

type T4 = Unpacked<Promise<string>[]>;

// ^ = type T4 = Promise

type T5 = Unpacked<Unpacked<Promise<string>[]>>;

// ^ = type T5 = string[Try](https://www.typescriptlang.org/play/#code/C4TwDgpgBAqgdmAhgYwNYQCYB4AqA+KAXihyggA9gI4MBnKACgEs4AzCAJ1gEoBtAXQBQUKAH5YwqAC4SZStTqMAdCsQcA5rRmI4IAdyIEW7LjEnizImaQpUa9AAocA9gFsmtCFmOdYecxJWJADcgoKgkCQADESwCCjo2LTAHCzqeKEA9JkiUAB6ouHg0DgAjLHwSGiYWMmpcOoCGYLZuQVFkTgATBXx1dgMBoQEdWnNrSLtESUAzL1ViVhObh5eow144zmThdMkACzzCTXL7p61KWNNWdv5u8UkAKxH-ViVx9inqxf16U1bbVEQA)

The following example demonstrates how multiple candidates for the same type variable in co-variant positions causes a union type to be inferred:

type Foo<T> = T extends { a: infer U; b: infer U } ? U : never;

type T1 = Foo<{ a: string; b: string }>;

// ^ = type T1 = string

type T2 = Foo<{ a: string; b: number }>;

// ^ = type T2 = string | number[Try](https://www.typescriptlang.org/play/#code/C4TwDgpgBAYg9nAPAFQHxQLxWVCAPYCAOwBMBnKAbygEMAuKASyIDMIAnKAVQG4oAjBszacuUAL5QA-NygMiEAG4ceAKFWhI2AIyZYCRNXpQywdswDmfQSbOWJqNQHonUNwD0pG8NGQAmPXgkIwZTcyIrAXkAVwBbfg4HZ1cPKSA)

Likewise, multiple candidates for the same type variable in contra-variant positions causes an intersection type to be inferred:

type Bar<T> = T extends { a: (x: infer U) => void; b: (x: infer U) => void }

? U

: never;

type T1 = Bar<{ a: (x: string) => void; b: (x: string) => void }>;

// ^ = type T1 = string

type T2 = Bar<{ a: (x: string) => void; b: (x: number) => void }>;

// ^ = type T2 = never[Try](https://www.typescriptlang.org/play/#code/C4TwDgpgBAQghgJwDwBUB8UC8UVQgD2AgDsATAZygG8o4AuKACnwYEtiAzCBKAVQEosGAG4B7VqQDcUAEYNmbTtz6DMI8aSgBfAFBQoAfj56oDYhGHdJOnaEg4AjFliIkNekxZRywBOwDmquoS0nKeDD5+xIFCUGIS2mjWAPTJ+lAAega24NAoAEzO8Mju8l6RAUFxGqFlZgCuALYy3FXxmlpJOqnpWUA)

When inferring from a type with multiple call signatures (such as the type of an overloaded function), inferences are made from the last signature (which, presumably, is the most permissive catch-all case). It is not possible to perform overload resolution based on a list of argument types.

declare function foo(x: string): number;

declare function foo(x: number): string;

declare function foo(x: string | number): string | number;

type T1 = ReturnType<typeof foo>;

// ^ = type T1 = string | number[Try](https://www.typescriptlang.org/play/#code/CYUwxgNghgTiAEAzArgOzAFwJYHtVJxwAoAPALngGcMYtUBzASgtWQFsAjEGAbgChQkWAhTpseAsXLxWnbsyo069foOhwkaTLnyJCpCtVoN4AHxnsuMBUeVmLc3nz4YAngAcEAFQCM8ALzwAEogGMgwqF4eIAA8bp44iJIAfPwA9GnwWQB6APxAA)

It is not possible to use infer declarations in constraint clauses for regular type parameters:

type ReturnedType<T extends (...args: any[]) => infer R> = R;

'infer' declarations are only permitted in the 'extends' clause of a conditional type.  
Cannot find name 'R'.'infer' declarations are only permitted in the 'extends' clause of a conditional type.  
Cannot find name 'R'.[Try](https://www.typescriptlang.org/play/#code/PTAEAEFMCdoe2gZwFygIwGYMA5QCYMAGAFgCgAXATwAdJQAlScgV2gDtIATAFRsgB5uoSAA9ykNp0SgAFADoFAQ2gBzFKEVtKAbQC6ASlABeAHygAlmwBmMBmaMMA3EA)

However, much the same effect can be obtained by erasing the type variables in the constraint and instead specifying a conditional type:

type AnyFunction = (...args: any[]) => any;

type ReturnType<T extends AnyFunction> = T extends (...args: any[]) => infer R

? R

: any;

## Predefined conditional types

TypeScript adds several predefined conditional types, you can find the full list and examples in [Utility Types](https://www.typescriptlang.org/docs/handbook/utility-types.html).