Conditional Types Introduction

This lesson introduces conditional types.

WE'LL COVER THE FOLLOWING

Introduction

Conditional types are *type constructors* that create a type based on some condition. In other words, they are a special kind of generic type where you can inspect the type argument and “return” a different type based on whether or not it satisfies a given type condition.

There is only one kind of *condition* allowed in conditional types and you can check whether the provided type argument is assignable to some type. You can think of type assignability the same way you'd think of value equality in a regular condition. In fact, even the syntax for conditional types is similar to regular ternary expressions.

```
type IsString<T> = T extends string ? true : false;

type A = IsString<string>; // A === true
type B = IsString<"abc">; // B === true
type C = IsString<123>; // C === false
```

In the above example, `IsString` is a conditional type. It examines the type argument `T` by checking if it is assignable to the `string` type. If the condition is true, the `true` Boolean literal type is returned (not a `true` Boolean value). Otherwise, `false` Boolean literal type is returned. Notice how type `B` is equal to `true`. This is because the `"abc"` string literal is also assignable to type `string`.

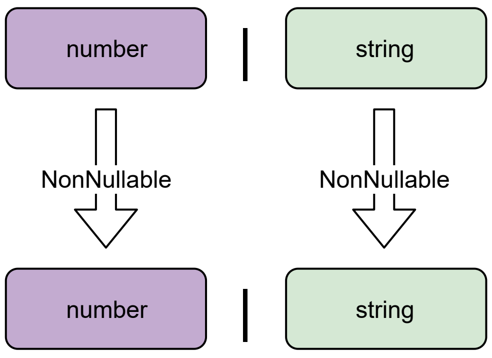
Distributive conditional types

One very important aspect of conditional types is that they're distributive. The easiest way to understand this is by looking at an example.

```
type NonNullable<T> = T extends null | undefined ? never : T;
```

`NonNullable` is an example of a built-in conditional type that comes with TypeScript 2.8. It's a very useful type that takes any type and returns a non-optional or a type with `null` and `undefined` removed from the domain of that type). You can use this type to annotate a generic function that requires a non-nullable type as a parameter (e.g., `function foo<T>(bar: NonNullable<T>) {}`).

To understand how this works, you need to know that conditional types are distributive over union type members. This means that if you apply `NonNullable` to a union type, the result will be a union of the results of `NonNullable` applied to all union members.

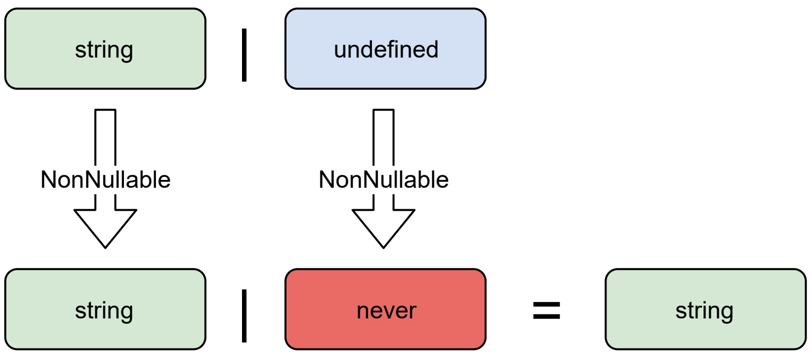


NonNullable distributed over number | string

```
type Foo = NonNullable<number | string>; // Foo is number | string
```

Of course, `NonNullable<number | string>` is still `number | string`. Let's look at a more meaningful example.

```
type Foo = NonNullable<string | undefined>; // Foo is string
```



NonNullable distributed over string | undefined

1. `NonNullable<string | undefined>` distributes over union members, so we need to evaluate `NonNullable<string>` and `NonNullable<undefine>`.
2. `NonNullable<string>` is `string` because `string` doesn't extend `null | undefined`.
3. `NonNullable<undefined>` is `never` because `undefined` extends `null | undefined`.
4. The result is `string | never` which is the same as a `string`.

But wait, what is the `never` type? It's a special type that has no instances. In other words, it's not possible to create an instance of this type. If we look at it from the perspective of set theory, `never` is associated with an empty set. Therefore, `string | never` is just a `string` because a union of any set with an empty set is just that set.

We'll look at more built-in conditional types in the next lesson.

Exercise

Define two types, `Not` and `Or`.

`Not`, given a `true` Boolean literal type, should return `false`.

`Or` should act like the logical OR operator, but on Boolean literal types.

Note that the system can only verify that your code compiles without errors. To check your solution, click on ***Show solution*** *and compare the code.*

```
// Define Not and Or
type Not<T> = never;
type Or<T> = never;
```

```
// The following assignments should compile correctly
const foo: Not<true> = false;

const foo: Not<false> = true;

const bar: Or<[true, false]> = true;
const bar: Or<[false, false]> = false;
```

The next lesson walks you through several examples of built-in conditional types.