What is a Conditional Type?

This lesson will teach you how to use and create a conditional type.

WE'LL COVER THE FOLLOWING

- A little background on the conditional type
- Using the conditional type with the dynamic number type
- Using the conditional type with generic
- Nested conditional type
- Conditional type examples in TypeScript

A little background on the conditional type

TypeScript 2.8 brings the possibility of the **conditional type**. The conditional type creates a type by checking if an interface or an existing type extends a type or not. It uses the ternary operator (`?:`) to the final type.

Using the conditional type with the dynamic number type

The following code shows a function at **line 10** that takes a type `T` for a parameter. The generic `T` extends a union of two types defined at **line 1** and **line 5**.

The function returns the object itself and the educational goal put on the front instead. What is important is the return type at **line 10**: it is a type that uses the condition type. This type accepts the two union values and applies a condition that swaps the two types.

Line 16 returns a `TypeB` because `TypeA` was passed in parameter.

```
interface TypeA {
  kind: "TypeA";
  m1: string;
}
interface TypeB {

  kind: "TypeB";
  m2: number;
}

function fct<T extends TypeA | TypeB>(obj: T): T extends TypeA ? TypeB : TypeA {
  return obj as any; // Won't be any
}


let typeA: TypeA = { kind: "TypeA", m1: "abc" };
let returnA: TypeB = fct(typeA);
```

Using the conditional type with generic

For example, in the code below, the dynamic type is a number because `InterfaceChild` inherits `InterfaceBase`, and the result uses the first type.

```
interface InterfaceBase {
    method1(): void;
}
interface InterfaceChild extends InterfaceBase {
    method2(): void;
}

type DynamicTypeFromCond = InterfaceChild extends InterfaceBase ? number : string;

let x: DynamicTypeFromCond = 3;
// let y: DynamicTypeFromCond = "123"; // Does not transpile
```

The example is not a likely scenario because the type is always `InterfaceChild`. However, using the conditional type with generic is pragmatic. The example's idea is to demonstrate the lack of flexibility without generic.

On the other side, you can enforce a generic to not be `undefined` by extending `undefined` and setting it to `never`, which will cause the compiler to reject any `undefined` types but accept all others. The **line 1** defines a generic type that leverages the conditional type to set the type by comparing the generic.

```
type NotUndefined<T> = T extends undefined ? never : T;

let x1: string = "test";

let x2:  undefined = undefined;

function printEverythingExceptUndefined<T>(p:NotUndefined<T>){
  console.log(p);
}

printEverythingExceptUndefined(x1);
// printEverythingExceptUndefined(x2);
```

The code above returns *Argument of type 'undefined' is not assignable to parameter of type 'never'* if we uncomment **line 11**. The reason is that `x2` is `undefined` and at **line 1** we defined that if `undefined` that we return `never.` To fix the transpilation, we need to swap 'p:NotUndefined' for 'T' it transpiles in the function at **line 6**.

Nested conditional type

It is possible to have many nested conditional types. The best way to understand the nested conditional type is with an example. Let's build a type that removes all boolean type from a type.

The first thing we need to consider is that we need to affect the member of a type, not the type directly. Hence, we need to loop the member with `[something in keyof somethingelse]` syntax. The syntax *loops* all the members and following the square brackets we will set a condition. In the following example, at **line 11* you can hover over the variable `NoBoolean1` and notice that `m2` is not present. Exactly what we desired? Not really. The type is each of the strings `m1` and `m2` and not the member.

```
type RemoveBoolean<T> = {
  [Key in keyof T]: boolean extends T[Key] ? never : Key
}[keyof T];

interface Inf1 {
  m1: string;
  m2: boolean;
  m3: number;
}

type NoBoolean1 = RemoveBoolean<Inf1>; // "m1" | "m3"
```

The output makes sense because we are returning the `Key` which is the name of the member. So, we need another conditional type that will loop all keys to get the member signature (name and type).

```
type RemoveBoolean<T> = {
  [Key in keyof T]: boolean extends T[Key] ? never : Key
}[keyof T];

interface Inf1 {
  m1: string;
  m2: boolean;
  m3: number;
}

type RemoveBooleanWithMembers<P> = {
  [Key in RemoveBoolean<P>]: P[Key]
};


type NoBoolean2 = RemoveBooleanWithMembers<Inf1>; // {m1: string, m3: number}
```

The example at **line 11** creates a type that calls the previous type we created: here comes the nested conditional type. This time **line 12** loops with the `in` all member's names and returns each property's member type (`P[Key]`);

Conditional type examples in TypeScript

Many functions you may use already are based on the conditional type. For instance, we saw in the mapped type many functions like `Include` and `Exclude`. These two functions need conditional type.

```
type MyInclude<T, U> = T extends U ? T : never;
type MyExclude<T, U> = T extends U ? never : T;
```

Now that we have covered the conditional type, in the next lesson, we will go on to study TypeScript inference.