Built-in Conditional Types

This lesson walks through several of TypeScript's built-in conditional types and explains them in detail.

`Exclude`

`Exclude` takes a union type, `T`, and removes the members that are assignable to `U`. It's another example of the distributiveness of conditional types. For all union members that extend `U`, the conditional expression will return `never`. The result will be a union of the remaining members.

`Exclude` is particularly useful when combined with the `keyof` operator. If you're not familiar with it, `keyof` takes an interface and returns a union type of literal types representing names of properties of this interface.

As you can see in *Example 3*, `Exclude` can be used to allow passing all object properties to a function *except* some specific property (or properties). In this example, `safeSetProp` is a generic function with two type parameters, `T` and `K`. We require that `K` extends `Exclude<keyof T, 'id'>` as it must be a literal type representing one of the properties except for `'id'`.

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

// Example 1
type AB = Exclude<'A' | 'B' | 'C', 'C'>; // 'A' | 'B'

// Example 2
type SomeNumbers = MyExclude<'A' | 'B' | 1 | 2, string>; // 1 | 2

// Example 3
interface Person {
    id: number;
    name: string;
    age: number;

}

function safeSetProp<T, K extends MyExclude<keyof T, 'id'>>(obj: T, key: K, value: T[K]) {
    obj[key] = value;
}

declare const obj: Person;
safeSetProp(obj, 'name', 'Miłosz');
safeSetProp(obj, 'id', 100); // ⛔ Error!
```

Run the code to see the error.

`Extract`

`Extract` is the exact opposite of `Except`. It will only pick those union members of `T` that **are** assignable to `U`.

Example 3 shows how `Extract` can be used to limit property keys passed to a function to only string keys. In the example `keyof StrangeObj` would return a union of `'foo' | 'bar' | 1 | 42`. `Extract<keyof StrangeObj, string>` eliminates non-string literal types and returns `'foo' | 'bar'`.

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

// Example 1
type AB = MyExtract<'A' | 'B' | 'C', 'C'>; // 'C'

// Example 2
type SomeNumbers = MyExtract<'A' | 'B' | 1 | 2, string>; // 'A' | 'B'

// Example 3
interface StrangeObj {
    foo: string;
    bar: number;
    1: string;
    42: number;
}

function setStringProp<T, K extends MyExtract<keyof T, string>>(obj: T, key: K, value: T[K])
    obj[key] = value;
}

declare const obj: StrangeObj
setStringProp(obj, 'bar', 1);
setStringProp(obj, 42, 1); // ⛔ Error!
```

Run the code to see the error.

Omit

`Omit` is a useful type that lets you take an object type and remove some properties from it.

It's defined using `Pick` which we can't discuss yet because it is a mapped type. For now, just assume that `Pick<T, U>` selects some properties from type `T` based on union members of `U`. By feeding `Exclude<keyof T, K>` as `U`, we will end up with all properties from `T` except those in `K`.

Let's break down the below example:

1. `Omit<Person, 'id'>` will resolve to `Pick<Person, Exclude<keyof Person, 'id'>>`.
2. `keyof Person` returns `'id' | 'name' | 'age'`.
3. `Exclude<keyof Person, 'id'>` returns `'name' | 'age'`.
4. `Pick<Person, 'name' | 'age'>` returns an object type `{ name: string; age: number; }`.

```
interface Person {
    id: number;
    name: string;
    age: number;
}

type AnonymousPerson = Omit<Person, 'id'>;

const anonymousPerson: AnonymousPerson = { id: 1, name: 'John', age: 33 }; // ▨ Error!
```

Hover over `AnonymousPerson` to see the inferred type. Run the code to see the error when trying to assign an object with omitted property.

The next lesson talks about the `infer` keyword which makes conditional types even more powerful.