Built-in Mapped Types

This lesson walks through some of the built-in mapped types and explains them in detail.

WE'LL COVER THE FOLLOWING

`Partial`

`Partial<T>` returns a type that has the same properties as `T` but all of them are optional. This is most useful when the `strictNullChecks` flag is enabled. `Partial` works on a single level, meaning it doesn't affect nested objects. A good use case for `Partial` is when you need to type a function that lets you override default values of properties of some object.

As you can see, `Partial` is defined in a similar way to `Readonly`. Instead of the `readonly` modifier, the optionality modifier `?` is appended to each property of `T`.

```
interface Settings {
  width: number;
  autoHeight: boolean;
}

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

const defaultSettings: Settings = {};

function getSettings(custom: Partial<Settings>): Partial<Settings> {
  return { ...defaultSettings, ...custom };
}
```

Try code completion on `custom` to see the types of the properties.

Required

`Required<T>` removes optionality from `T`'s properties. It demonstrates how mapped types can not only add modifiers to properties but also remove them with the `-` syntax.

```
type Required<T> = {
    [P in keyof T]-?: T[P];
};

type Foo = Required<{ name?: string }>; // { name: string; }
```

Hover over `Foo` to see the inferred type.

Pick

`Pick` lets you create an interface that contains only a subset of properties of `T`.

Contrary to previous examples of mapped types, `Pick` doesn't use `[P in keyof T]` to iterate over properties. Instead, `[P in K]` is used, where `K` is a type argument to `Pick`. In fact, any union type can be placed on the right side of `in`. Here, `K` is constrained to extend `keyof T` to make sure that only existing properties of `T` are present in the result type.

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

type Pick<T, K extends keyof T> = {
    [P in K]: T[P];
};

type NameAndAge = Pick<Person, 'name' | 'age'>; // { name: string; age: number; }
```

Hover over `NameAndAge` to see the inferred type.

Record

Finally, `Record` is an interesting example of a mapped type that doesn't actually map over any source type. Instead, it creates a completely new object type out of *thin air*.

`Record` takes two type arguments: `K` representing property names in the resulting type, and `T` representing the type of values of these properties.

```
type Record<K extends keyof any, T> = {
    [P in K]: T;
};

type StringDictionary = Record<string, string>; // { [x: string]: string; }
type ABCNumbers = Record<'a' | 'b' | 'c', number>; // { a: number; b: number; c: number; }
```

Hover over `ABCNumbers` to see the inferred type.

Exercise

Define a type called `ReturnTypes` that takes an interface with methods and returns an interface with properties typed using return types of these methods.

*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.*

```
type Test = ReturnTypes<{
    a: string;
    b: (a: number) => number;
}>; // { a: never; b: number; }
```

The next lesson introduces another advanced type mechanism, type guards.