9. Loops with Mapped Types

No doubt about it, **transforming objects** `{}` is at the core of what we do as developers. Building awesome features usually requires getting JSON objects from external APIs into our application and turning them into something that makes sense for our users. In the process, we tend to create a lot of slightly different copies of our data. Wouldn't it be nice if we **didn't have to type** them **manually**?

Here is the good news — we can leverage TypeScript's awesome **type inference** to transform our object types using a super powerful feature called **Mapped Types**.

In this chapter, we will learn how to **transform** and **filter** object types using Mapped Types, and combine them with other features of the type system such as Template Literal Types and Conditional Types to build functions with **super smart type inference**.

For example, we will learn how to type the `deepCamelize` function:

```ts
import { deepCamelize } from "~/helpers/deep-camelize";

// Transforms all keys from snake_case to camelCase:

const apiPayload = {
  chapter_title: "Loops with Mapped Types",
  author: { full_name: "Gabriel Vergnaud" },
};

const camelized = deepCamelize(apiPayload);
//    camelized: { chapterTitle: string; author: { fullName: string } }
```

Or the `update` function from lodash, which transforms a deeply nested value:

```ts
import { update } from "~/helpers/update";

const player = { position: { x: "1", y: 0 } };
//               `x` is a string 👆

const newPlayer = update(player, "position.x", (x) => parseInt(x));
/*    newPlayer: { position: { x: number, y: number } }
                 Now it's a number! 👆                    */

const pkg = { name: "such-wow", releases: [{ version: 1 }] };

const newPkg = update(pkg, "releases[0].version", (v) => `${v}.0.0`);
//    newPkg: { name: string, releases: { version: string }[] }
```

And any other **generic function** traversing nested objects. Lastly, we will see how Mapped Types can be applied to **more** than just object types 🤔

Let's get started!

Deciphering Mapped Types

Mapped Types are often depicted as a syntax to **map over each value** of an object type. The name "Mapped Type" directly comes from there, and I must admit this is probably their most common use case. At their core though, Mapped Types are much simpler. They only **turn unions into objects**!

Let's take a look at the simplest Mapped Type you will ever see:

```ts
type LetterUnion = "a" | "b" | "c";

// Our first Mapped Type
//  👇
type T = {
  [Letter in LetterUnion]: Letter;
};
```

Just like regular object types, we create a Mapped Type by opening curly braces `{ ... }`, but instead

of manually listing properties, we use this interesting `[A in B]` syntax.

The `in` keyword lets us **loop through** each element of a union type by successively assigning them to a **variable**:

```
type T = {
  [ThisIsANewVariable in ThisIsAUnionType]: ...
/*                    👆
  `in` lets us define a variable that will
  successively take each value in a union type.
                                              */
};
```

This variable becomes a new **key** of our object.

What makes mapped types powerful is that we can also use this variable on the **value side**:

```
type A = {
  [Letter in LetterUnion]: Letter;
/*   👆                      👆
   we define             and we use it
  `Letter` here          over there.
                                    */
};
// ⇒ { a: "a"; b: "b"; c: "c" };
```

If we want to, we can of course use `Letter` inside a more complex expression, like a tuple:

```
type B = {
  [Letter in LetterUnion]: [Letter, Letter];
};

// ⇒ { a: ["a", "a"]; b: ["b", "b"]; c: ["c", "c"] };
```

Or interpolate it into a string:

```
type C = {
  [Letter in LetterUnion]: `The letter is ${Letter}`;
};
// ⇒ { a: "The letter is a"; b: "The letter is b"; c: "The letter is c" };
```

We are effectively mapping over each member of a union type and creating an object type out of them!

Cool, but how does that help us with transforming objects? 🤔

Combining `in` with `keyof`

Mapped Types turn **unions** into **objects**, but what if we do not have a union, but an object to start with?

It looks like we need a way to **turn our object into a union first**! We already know one: The `keyof` keyword. As you know `keyof` extracts **keys** from an object as a union type. By placing it right after the `in` keyword, we can loop through each key in an object:

```
type User = {
  name: string;
  age: number;
};

type T = {
  [Key in keyof User]: User[Key] | null;
  /*        👆
       We loop over User's keys!
                                    */
};
```

Naturally, we can use our `Key` variable on the value side to access its corresponding value with `User[Key]`!

In this example, we loop through every value of `User` to put them in a union with `null`, so the resulting object looks like this:

```
type T = {
  name: string | null;
  age: number | null;
};
```

Now, what if we tried replacing `keyof User` by a hardcoded union of keys?

```
type T = {
  //           👇
  [Key in "age" | "name"]: User[Key] | null;
};
// ⇒ { name: string | null; age: number | null }
```

We would get the exact same type back! This demonstrates that there is nothing special about `in keyof`. We are just ***composing*** two very handy features of the type-level language.

Generic Mapped Types

In the previous example, we made every key of our `User` type **nullable**. That was nice, but what if we wanted to apply the same transformation to another type? Being good developers, we do not love to repeat ourselves, and that's why **functions** are our favorite things!

Let's make our code **generic** by building a `NullableValues` type-level function:

```
type NullableValues<Obj> = {
  [Key in keyof Obj]: Obj[Key] | null;
};
```

We only had to replace our `User` type with an `Obj` parameter. **This code structure is extremely common**. We are essentially applying the following type-level function to every value in our object:

```
type OrNull<T> = T | null;
```

And we can now reuse this logic with any object type:

```
type A = NullableValues<{ name: string; age: number }>;
// ⇒ { name: string | null; age: number | null }

type B = NullableValues<{ a: true; b: 42 }>;
// ⇒ { a: true | null; b: 42 | null }
```

I'm always amazed by how **terse** transforming an object is at the type level. In comparison, here is the code you would need to write to perform a similar transformation in JavaScript:

```
const mapOverValues = (obj, fn) => {
  const output = {};
  for (const [key, value] of Object.entries(obj)) {
    output[key] = fn(value); // transform the current value!
  }
  return output;
};
```

This is quite a bit more complex and noisy!

Now, imagine we wanted to perform the opposite transformation: **filtering out** the `null` type from each of our values. Hopefully, you remember from the previous chapter how to remove elements from a union: we can use the `Exclude` generic!

```
type NotNull<T> = Exclude<T, null>;
```

Let's apply this to all values using the same pattern:

```
type NonNullValues<Obj> = {
  [Key in keyof Obj]: Exclude<Obj[Key], null>;
```

```
};

type A = NonNullValues<{
  name: string | null;
  age: number | null;
}>;
// ⇒ { name: string; age: number }
```

A concrete use case for this generic could be a `withDefault` function, smart enough to come up with default values when keys are `null`:

```
declare function withDefault<T>(obj: T): NonNullValues<T>;

declare const partialUser: { name: string | null };

const user = withDefault(partialUser);
//     ^? { name: string }
```

Filtering object keys

The fact that the `in` keyword accepts **any union type** gives us a lot of **liberty** when transforming objects. Not only can we loop over every key, but also filter some out or add more if we want to!

Let's say we're building a full-stack TypeScript application that reads some environment variables. We want to make sure our frontend can never access our app's secrets, but maintaining two separate type definitions for public and private variables is cumbersome:

```
type BackendEnv = { publicKey: string; secretKey: string };
//                  👇👇 This key is duplicated
type FrontendEnv = { publicKey: string };
```

Given that we use the neat convention of prefixing secret variables with "secret", can you think about a way to generate `FrontendEnv` from `BackendEnv`?

Here is one:

```
type FrontendEnv = {
  [K in Exclude<keyof BackendEnv, `secret${string}`>]: BackendEnv[K];
};
// ⇒ { publicKey: string }
```

Thanks to `Exclude`, we can first remove keys assignable to the `` `secret${string}` `` type. Then, we simply map over the remaining ones and read their values with `BackendEnv[K]`!

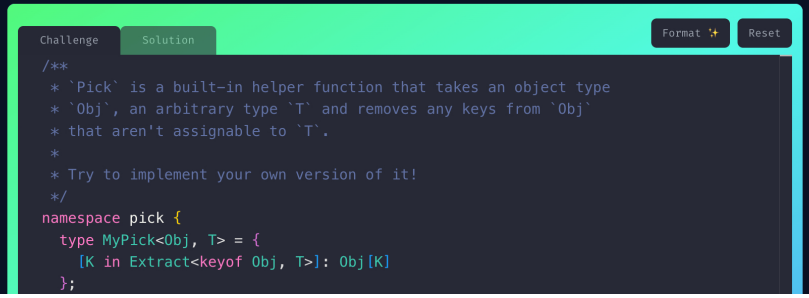
This pattern is so common that TypeScript comes with a built-in `Omit` generic that covers exactly this. Here is how it's implemented:

```
type Omit<Obj, Omitted> = {
  [K in Exclude<keyof Obj, Omitted>]: Obj[K];
};

type FrontendEnv = Omit<BackendEnv, `secret${string}`>;
// ⇒ { publicKey: string }
```

This is the same code, except we replaced `BackendEnv` and `` `secret${string}` `` with two type parameters: `Obj` and `Omitted`!

Now, here is a little exercise for you. How would you reimplement the built-in `Pick` function?



```ts
type res1 = MyPick<{ a: string; b: number; c: boolean }, "a">;
type test1 = Expect<Equal<res1, { a: string }>>;

type res2 = MyPick<{ a: string; b: number; c: boolean }, "a" | "b">;
type test2 = Expect<Equal<res2, { a: string; b: number }>>;

type res3 = MyPick<{
  getName: () => string;
```

Well done! 👍

Arrays, Tuples & Mapped Types

Something many people don't know about Mapped Types is that you can also use them with **arrays** and **tuple types**!

```ts
//   👇 We created this Mapped Type a few sections ago
type NullableValues<Obj> = {
  [Key in keyof Obj]: Obj[Key] | null;
};

type A = NullableValues<string[]>;
// ⇒ (string | null)[]

type B = NullableValues<[null, boolean, string]>;
// ⇒ [null, boolean | null, string | null]
```

It just works! We can simply reuse the code we wrote earlier to **map over** array values.

Wait 🤔 didn't we learn in Arrays & Tuples that `keyof SomeArray` returns method names from the `Array` class, like `"map"`, `"filter"` or `"reduce"` in addition to its indices?

```ts
const key1: keyof [1, 2, 3] = 1; // ✅ `1` is a key of this tuple
const key2: keyof [1, 2, 3] = "map"; // ✅ but "map" is also valid.
```

Does that mean we unintentionally made all of these methods nullable ***too***?

Well, no. TypeScript handles arrays and tuples a little bit differently from other object types. When passed through a Mapped Type, only their **numeric properties** get transformed:

```ts
type NullableArray = NullableValues<string[]>;
// ⇒ (string | null)[]

const map: NullableArray["map"] = null;
//    ~~~ ❌ `null` isn't assignable to `NullableArray["map"]`
```

Even though this may look hacky, this behavior is actually pretty standard. TypeScript's built-in library of types makes extensive use of this feature. For example, here is how `Promise.all` is typed:

```ts
class Promise {
  // This looks a bit complex 🤔
  all<Promises extends unknown[]>(promises: Promises): Promise<{
    [Index in keyof Promises]: Awaited<Promises[Index]>;
  }>;
}
```

Let's decompose this into smaller bits.

The first thing to understand is that the `Promises` type represents the tuple of promises given to `Promise.all`:

```ts
const p1 = Promise.resolve("a string");
const p2 = Promise.resolve(42);
const p3 = Promise.resolve(true);

Promise.all([p1, p2, p3]);
/*       👆
     Here, the `Promises` type parameter is inferred as:
     [Promise<string>, Promise<number>, Promise<boolean>]
                                                      */
```

We then loop through `Promises` using a mapped type. Thanks to the `Awaited` generic, we can **unwrap** each promise in the list:

```
type UnwrapAll<Promises> = {
  /*
     `Awaited` is a built-in helper that
      extracts the value from a promise.
                                 👇         */
  [Index in keyof Promises]: Awaited<Promises[Index]>;
};

// We could implement it this way:
type Awaited<P> = P extends Promise<infer V> ? V : P;

type T = UnwrapAll<[Promise<string>, Promise<number>, Promise<boolean>]>;
//   ^? [string, number, boolean]
```

Once we have our list of values, we need to wrap it in a new `Promise< ... >` type to make the signature of `Promise.all` correct:

```
class Promise {
  all< ... >( ... ): Promise<{  ...  }>;
  //                 👆
}
```

And that's how the type of `Promise.all` works!

```
const result = Promise.all([
  Promise.resolve("a string"),
  Promise.resolve(42),
  Promise.resolve(false),
]);
// result: Promise<[string, number, boolean]>
```

▼🤔 **Couldn't we have used recursion instead?**

In Loops with Recursive Types, we learned about another way to map over a tuple: **recursive map loops**! We could have solved the same problem using this concept instead of a Mapped Type:

```
type PromiseAll<Promises> = Promise<UnwrapAll<Promises>>;

//     👇 this is a recursive "map" loop
type UnwrapAll<Promises> =
  Promises extends [infer First, ...infer Rest]
    ? [Awaited<First>, ...UnwrapAll<Rest>]
    : [];

type T = PromiseAll<[Promise<string>, Promise<number>, Promise<boolean>]>;
//   ^? Promise<[string, number, boolean]>
```

This works great too! Recursion is super **powerful** and enables much more than mapping over tuples, but for simple mapping use cases, I like how terse the Mapped Type syntax looks.

Using curly braces to define an array can look a little confusing at first, but remember that arrays and tuples ***are*** object types. They are assignable to `{}`, so it makes sense that you can map over them like any other object.

The optional property modifier

By now, you're probably familiar with the built-in `Partial` generic that can turn all values in an object into optional properties. It is implemented as a Mapped Type:

```
type Partial<Obj> = {
  [Key in keyof Obj]?: Obj[Key];
  //                 👆
};
```

This question mark character is the **optional property modifier**. Without it, the output of `Partial` would be identical to its input, but add a single `?` and all properties become optional:

```
type T = Partial<{ name: string; age: number }>;
//   ^? { name?: string, age?: number }
```

You shouldn't be surprised if I tell you that this works with **arrays** and **tuples** too:

```
type A = Partial<string[]>;
//   ^? (string | undefined)[]

type B = Partial<[string, number]>;
//   ^? [string?, number?]
```

The `?:` syntax is pretty intuitive, but here is a slightly more disturbing piece of code:

```
type Required<Obj> = {
  [Key in keyof Obj]-?: Obj[Key];
  //                ☝️
};
```

`-?` is the **required property modifier**. It's literally the opposite of `?` because it removes the optional modifier from object properties — hence the minus sign:

```
type T = Required<{ name?: string; age?: number }>;
//   ^? { name: string, age: number }
```

`Required` is also part of TypeScript's standard library. At least we know how it works now 😄

The `readonly` modifiers

We haven't talked much about **read-only objects and arrays** so far, but in a nutshell, it makes TypeScript check that we aren't mutating any property of one of our objects:

```
function doSomething(obj: { readonly prop: string }) {
  //                           ☝️
  obj.prop = "Hello";
  //  ~~~~
  //  ❌ Cannot re-assign 'prop' because it is a read-only property.
}
```

It often comes up when using the `as const` keywords, which turn all objects and arrays into `readonly` and all values into **literal** types:

```
const units = {
  bytes: ["B", "kB", "MB", "GB"],
} as const;
// units: { readonly bytes: readonly ["B", "kB", "MB", "GB"] }

type Byte = (typeof units)["bytes"][number];
//    ^? "B" | "kB" | "MB" | "GB"
```

We will talk more about `readonly` objects in the next chapter covering assignability, but I can't talk about mapped types without a word about the `readonly` modifier.

Here is how the built-in `Readonly` generic is defined:

```
type Readonly<Obj> = {
  readonly [Key in keyof Obj]: Obj[Key];
  // ☝️
};

type T = Readonly<{ name: string; age: number }>;
//   ^? { readonly name: string, readonly age: number }
```

It loops through each key of an object and makes them read-only using the `readonly` keyword.

There is no `Mutable` generic in the standard library, but can you guess how to create one?

Challenge | Solution — Format ✨ — Reset

```ts
/**
 * `Mutable` should take an object with read-only properties
 * and make them mutable.
 *
 * Hint: the syntax to remove the `readonly` modifier
 * is pretty similar to the syntax to remove the `?` optional
 * modifier...
 */
namespace mutable {
  type Mutable<Obj> = {
    -readonly [Key in keyof Obj] : Obj[Key]
  };

  type res1 = Mutable<{ readonly name: string; readonly age: number }>;
  type test1 = Expect<Equal<res1, { name: string; age: number }>>

  type res2 = Mutable<{ readonly a: string; b: 'not readonly' }>;
  type test2 = Expect<Equal<res2, { a: string; b: 'not readonly' }>>
}
```

🙌

Preserving property modifiers

Now that we know about property modifiers, we should make sure that we don't inadvertently erase them when writing type-level functions!

Imagine we need to build a **form**. It contains several input fields, which all have a `value`, and an `isValid` boolean. We use the following type to represent the state of our form:

```ts
type UserForm = {
  firstName: InputState;
  lastName?: InputState;
};

type InputState = {
  value: string;
  isValid: boolean;
};
```

Notice that `lastName` is **optional** — we do not require our users to fill it in.

Now, let's say we want to turn our `UserForm` type into the object we send to our backend. This object should only contain the values for `firstName` and `lastName`. How do we do?

We are, of course, going to use a mapped type!

```ts
type FormToPayload<Form> = {
  [InputName in keyof Form]: GetValue<Form[InputName]>;
};

type GetValue<T> =
  T extends { value: infer V } ? V : never;
/*                    👆
        We use a conditional type
    to *destructure* each `InputState`
                                      */
```

Wait a second. does that mean that our `?` optional modifier on the `lastName` property **has been lost**? Let's find out:

```ts
type T = FormToPayload<UserForm>;
//  ^? { firstName: string, lastName?: string }
```

Looks like the `?` is still there! 🎉 but why?

Again, TypeScript's got our back. The type-checker is somehow hiding additional **metadata** in types created with `keyof` to keep track of modifiers. If we reuse the keys of an object `T` to create a new object `U`, `U` will **inherit** property modifiers of `T`:

```ts
type T = { readonly a: string; b?: number };
```

```ts
type U = { [K in keyof T]: 42 };
// { readonly a: 42; b?: 42 };
```

In compiler lingo, `T` and `U` are called **homomorphic objects** — they share the same shape. As a strange side effect, as soon as we put something different from `keyof T` after `in`, modifiers are gone:

```ts
type U2 = { [K in (keyof T | 'c')]: 42 };
// { a: 42; b: 42; c: 42 };
//   👆       👆
// no more modifiers!
```

Looks like `... in keyof ...` was a little special after all! 😅

Key remapping with the `as` keyword

The `as` keyword probably scares you a little bit. In TypeScript `as` often means unsafe code. It's a way to force TypeScript to "cast" a value to a different type:

```ts
const unsafeFunction = () => {
  return {} as { name: string };
  /*        👆
    We "cast" our empty object
        to a different type.
                              */
};

unsafeFunction().name.toLowerCase();
//                ^ ✅ This type-checks ...
//                     but throws at runtime!
```

Even though this code is **guaranteed to throw** at runtime, the type-checker accepts it. It's obviously not ideal and it's a good practice to avoid `as` as much as possible.

A different kind of `as`

In the context of a Mapped Type, however, the `as` keyword has a **totally different meaning**. We can put it after `Item in Union` to use something else than `Item` as our object key:

```ts
type RenameKeys<Obj> = {
  [K in keyof Obj as `new_${K}`]: Obj[K];
  //              👆
};

type T = RenameKeys<{ id: number; name: string }>;
// => { new_id: number, new_name: string }
```

This is called **key remapping**. It's very often used in combination with template literal types to interpolate keys inside a different string, like in the previous example.

Here is how you can turn every **value** in an object into a **getter function**:

```ts
type ToGetters<Obj> = {
  [K in keyof Obj as `get${Capitalize<K>}`]: () => Obj[K];
};

type T = ToGetters<{ id: number; name: string }>;
// => { getId: () => number, getName: () => string }
```

Key remapping makes mapped types even more powerful. We can not only map over object values, but also over their keys!

A very common real-world use case for key remapping is **converting object keys from `snake_case` to `camelCase`**. We've learned how to convert string literals to different cases in chapter 7, but, as a reminder, here is the definition of a `SnakeToCamel` function:

```ts
type SnakeToCamel<Str> =
  Str extends `${infer First}_${infer Rest}`
  //    Split on underscores 👆
```

```
    ? `${First}${SnakeToCamel<Capitalize<Rest>>}`
    //      Capitalize each word 👆
    : Str;
```

Let's apply this function to each of our keys using the `as` keyword!

```
type Camelize<T> = {
  [K in keyof T as SnakeToCamel<K>]: T[K];
  //                 👆
};

type T = Camelize<{ some_long_key: number; another_one: string }>;
// { someLongKey: number; anotherOne: string }
```

You should note that `as ...` does **not** change the value of our `K` variable. When we use it on the value side of our object, it still contains a `snake_case` string literal:

```
type T = {
  [K in "some_long_key" | "another_one" as SnakeToCamel<K>]: K;
};
// ⇒ { someLongKey: "some_long_key"; anotherOne: "another_one" }
```

All `as` does is assign our value to a different key!

Advanced key remapping

Something I find amazing about key remapping is how it enables turning **any union type** into an object — not only unions of strings or numbers but also **unions of other objects**!

Let's make this more concrete.

Imagine we want to write a `groupByType` function to group a list of objects by their `type` property:

```
const groups = groupByType([
  { type: "img" as const, src: "a.png" },
  { type: "img" as const, src: "b.png" },
  { type: "paragraph" as const, content: "..." },
]);
```

Given this input, we expect our function to return this object:

```
const groups = {
  img: [
    { type: "img", src: "a.png" },
    { type: "img", src: "b.png" },
  ],
  paragraph: [
    { type: "paragraph", content: "..." }
  ],
};
```

So in this case, the output type of `groupByType` should be:

```
type GroupByTypeOutput = {
  img: { type: "img"; src: string }[];
  paragraph: { type: "paragraph"; content: string }[];
};
```

But how can we type our function to make it work with any list of objects?

Let's start by declaring the type TypeScript should ***infer***:

```
declare function groupByType<U extends { type: string }>(
  objects: U[]
  //       👆 We let TypeScript infer the type of our objects
): GroupByType<U>;
// ~~~~~~~~~~ 🤔 not implemented yet.
```

With `objects: U[]`, we make the type checker extract the content of the list as a **union type** and

assign it to the variable `U`. We now need to turn `U` into a single object with a key for each unique `"type"` property. Let's use a mapped type!

```
type GroupByType<UnionOfObj extends { type: string }> = {
  [Obj in UnionOfObj as Obj["type"]]: Obj[];
  //        👆                  👆
  //    We loop over        We use their "type"
  // a union of objects!      as our keys.
};
```

Thanks to key remapping, we can directly loop over the union of objects, and use their `"type"` prop as our keys!

```
type Img = { type: "img"; src: string }
type Paragraph = { type: "paragraph"; content: string }

type T = GroupByType<Img | Paragraph>
//   ^? { img: Img[]; paragraph: Paragraph[] }
```

Our `groupByType` function is great because we can reuse it easily:

```
type Input = { type: "input"; value: string };
type Button = { type: "button"; onClick: () => void };
type Select = { type: "select"; options: string[] };

declare const formElements: (Input | Button | Select)[];

const grouped = groupByType(formElements);
//     ^? { input: Input[], button: Button[], select: Select[] } ✨
```

🤯

The ability to generate objects from arbitrary union types opens a world of possibilities. Let's put this idea to work in a pattern I've been using quite frequently when transforming object types — The **entries pattern**!

The Entries Pattern

The concept of object "entries" comes directly from JavaScript. `Object.entries` and `Object.fromEntries` allow us to convert objects into lists of "entry" tuples, and vice-versa:

```
const entries = Object.entries({ name: "Gabriel", age: 29 });
// => [["name", "Gabriel"], ["age", 29]]

const user = Object.fromEntries(entries);
// => { name: "Gabriel", age: 29 }
```

It turns out we can easily create a **type-level version** of these functions, and it enables all kinds of interesting algorithms:

```
type E = Entries<{ name: "Gabriel"; age: 29 }>;
// => ["name", "Gabriel"] | ["age", 29]

type User = FromEntries<E>;
// => { name: "Gabriel", age: 29 }
```

The only difference between our type-level "entries" and their JavaScript counterparts is that they are **unions** of `[key, value]` pairs instead of arrays.

Before we dive into a practical application, let's see how to implement `Entries` and `FromEntries`. If you remember the `ValueOf` function we wrote in Objects & Records, `Entries` should be pretty straightforward:

```
type Entries<Obj> = ValueOf<{
  [Key in keyof Obj]: [Key, Obj[Key]];
}>;

type ValueOf<T> = T[keyof T];
```

We create an object containing `[key, value]` tuples, and we extract its values as a union type using `ValueOf`.

Now, using what we've seen in the previous section, **can you figure out how to implement `FromEntries`?**

This involves some key remapping 🤔

Challenge | Solution — Format ✨ — Reset

```ts
/**
 * Implement a `FromEntries` generic, transforming
 * a union of [key, value] entries into an object type.
 */
namespace fromEntries {
  type FromEntries<Entries extends [any, any]> = {
    [K in Entries as  K[0]]: K[1]
  }

  type res1 = FromEntries<["a", string]>;
  type test1 = Expect<Equal<res1, { a: string }>>;

  type res2 = FromEntries<["a", string] | ["b", number]>;
  type test2 = Expect<Equal<res2, { a: string; b: number }>>;

  type res3 = FromEntries<never>;
  type test3 = Expect<Equal<res3, {}>>;
}
```

You rock 🤘

With these shiny new tools in our toolbelt, let's head to a new challenge: Filtering objects based on their values!

Filtering Objects By Values

`Omit` and `Pick` make filtering object keys pretty easy, but what if you want your filtering logic to depend on the type of the **value**?

Let's say we want to exclude functions from an object to later serialize it into JSON. The first thing that might come to mind is to use a **conditional type** and return `never` if the value is assignable to a function:

```ts
type AnyFunction = (...args: any) => any;

type NoFunctions<T> = {
  [K in keyof T]: T[K] extends AnyFunction ? never : T[K];
  //                        Does this work? 🤔 ^
};
```

This seems sensible, but if you try this out, you will notice that it does not actually remove any key:

```ts
type T = NoFunctions<{ someFn: () => string; something: string }>;
//   ^? { someFn: never; something: string }
```

`someFn` is still there, but it contains `never` now. That's not what we want!

We could maybe use `Exclude` then?

```ts
type NoFunctions<T> = {
  [K in Exclude<keyof T, ...>]: T[K]
  //                    ^
  // What should we put here? 🤔
};
```

Hmm. Looks like we can't use `Exclude` either because we don't know which keys should be excluded yet...

But what if we were **turning our object into *entries* first**?

```ts
type NoFunctions<T> = FromEntries<ExcludeFunctions<Entries<T>>>
//                      Step 3  ←    Step 2    ← Step 1
//                        ✅          TODO         ✅
```

Converting objects into entries and vice-versa isn't a problem. The only remaining thing we need is a function to **exclude entries containing functions**. Let's build it!

```ts
type ExcludeFunctions<Entry> =
  // If the value is a function 👇
  Entry extends [any, AnyFunction]
    // filter the entry out
    ? never
    // otherwise, keep it!
    : Entry;
```

> 💡 If you are not sure why this `ExcludeFunctions` works, I greatly encourage you to take another look at The Union Type Multiverse.

That's it! We just solved our problem using the entries pattern:

```ts
type A = NoFunctions<{
  someFn: () => string;
  something: string;
}>;
// => { something: string }   🐱

type B = NoFunctions<{
  a: () => null;
  b: () => string;
  c: string;
  d: number;
}>;
// => { c: string, d: number }  🎊
```

Note that we didn't have to handcraft `ExcludeFunctions`. We could have used `Exclude` instead:

```ts
// This would also work:
type ExcludeFunctions<Entry> =
  Exclude<Entry, [any, AnyFunction]>;
  /*                    👆
          Remove any type assignable to
     [any, AnyFunction] from the `Entry` union.
                                              */

type A = ExcludeFunctions<
  ["getId", () => string] | ["version", string]
>
// => ["version", string]
```

We can now **abstract over this code** pretty easily, and solve the problem of omitting object entries by their value type **once and for all**!

```ts
type OmitByValue<T, Omitted> = FromEntries<
  Exclude<
    Entries<T>,
    [any, Omitted]
  >
>;
```

This new generic is ***super reusable*** because it doesn't make assumptions about the type of values to exclude:

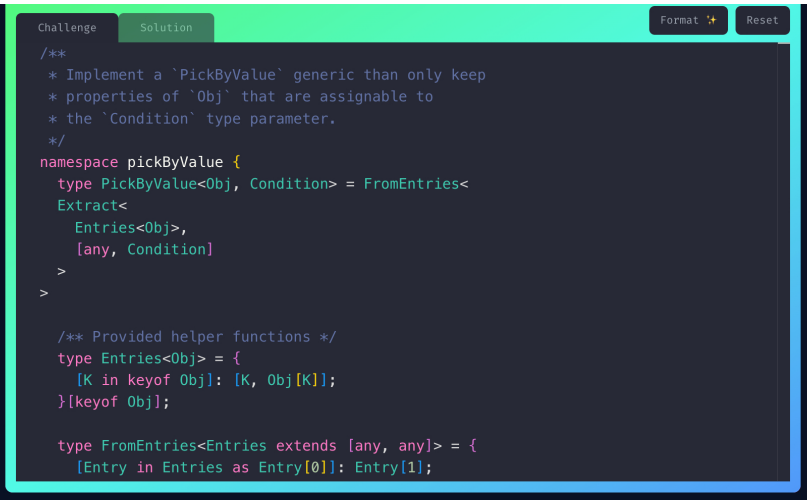
```ts
type A = OmitByValue<{ a: () => string; b: string }, AnyFunction>;
//   ^? { b: string }

type B = OmitByValue<{ a: "a"; b: 1; c: boolean }, string>;
//   ^? { b: 1; c: boolean }

type C = OmitByValue<{ a: null; b: undefined; c: boolean }, null | undefined>;
//   ^? { c: boolean }
```

Now, let's see if you can use the entries pattern to build a `PickByValue` generic, that only keeps properties if their values are assignable to some arbitrary type.

Let's go! 🎉



👈

As you can see, the concept of **entries** is super relevant at the type level too! Keep them in mind, they might come in handy for some challenges at the end of this chapter 😇

Mixing property modifiers

Defining a regular object type with **both optional** and **required** properties is extremely easy, but what about making a few properties of an ***existing*** type optional? How would you build a `MakeOptional` generic that takes an object, a union of keys, and makes these keys optional while leaving other keys untouched?

```ts
type A = MakeOptional<{ a: 1; b: 2; c: 3 }, "a">;
//   ^? { a?: 1, b: 2, c: 3 }

type B = MakeOptional<{ a: 1; b: 2; c: 3 }, "a" | "b">;
//   ^? { a?: 1, b?: 2, c: 3 }
```

It turns out it's unexpectedly hard.

Since the `?` modifier comes after the `[K in ...]` loop, it is applied to all keys in the object:

```ts
type MakeOptional<T, Keys> = {
  [K in keyof T]?: T[K];
  //             👆
  //  This applies to all keys
};
```

Here is an interesting idea. What if we didn't use one, but **two** index signatures using the `in` keyword on the same object?

```ts
// 🦄 Imaginary syntax
type MakeOptional<T, Keys> = {
  // optional props
  [K in Extract<keyof T, Keys>]?: T[K];
  // other props
  [K in Exclude<keyof T, Keys>]: T[K];
// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
// ^ ❌ Invalid syntax error
}
```

This code unfortunately does not compile because TypeScript doesn't allow using `in` more than once in an object type 😿

But what about an **intersection** of Mapped Types instead?

```ts
type MakeOptional<T, Keys> =
  // optional props
  { [K in Extract<keyof T, Keys>]?: T[K] } &
  // other props                          👆
  { [K in Exclude<keyof T, Keys>]: T[K] }
```

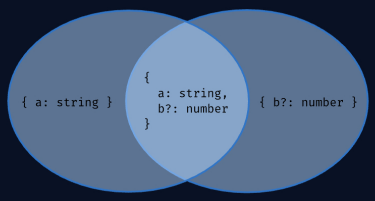
This code does compile, but does it work?

```
type A = MakeOptional<{ a: 1; b: 2; c: 3 }, "a">;
//   ^? { a?: 1 } & { b: 2, c: 3 } 🙌

type B = MakeOptional<{ a: 1; b: 2; c: 3 }, "a" | "b">;
//   ^? { a?: 1, b?: 2 } & { c: 3 } 😎
```

It does 🎉 Even though we are technically creating an **intersection** rather than a single object with different modifiers, this solution perfectly meets our needs!

As you probably know by now, intersections of objects create object types containing **all properties** of **all intersected objects**, so the return type of `MakeOptional` is equivalent to the type we were initially expecting.



intersecting objects

The bottom line is that whenever you want to compute an object type containing different property modifiers, you will have to **intersect** a mapped type **for each modifier**. This might not be intuitive at first, but you'll quickly get used to it!

Debugging Mapped Types

Before we end this chapter, I'd like to tell you about a trick that helped me many times while debugging mapped types: Getting TypeScript to **display** our **fully evaluated** object type on hover instead of something like `SomeGeneric<Input>`!

This is especially relevant when building **recursive mapped types**. Let's say you want to build a `DeepPartial` that will recurse on every property of an object to make all of their keys **optional**. This is pretty easy to implement:

```
type DeepPartial<T> = {
  [K in keyof T]?: DeepPartial<T[K]>;
  //             👆         👆
  //     Make keys       Recurse
  //     optional.       on values.
};

type T = DeepPartial<{ a: string; b: { c: string; d: number } }>;
```

Here is the issue: if you hover `T` here, TypeScript will **not** display the nice-looking:

```
type T = { a?: string; b?: { c?: string; d?: number } };
```

But this instead:

```
type T = {
  a?: DeepPartial<string>;
  b?: DeepPartial<{ c: string; d: number }>;
};
```

Note that these two things are completely equivalent. Our `DeepPartial` generic works perfectly, but TypeScript's heuristics decide not to display its full result.

Luckily, there is a way to circumvent this limitation and force TypeScript to **show all levels of nesting**: returning a **union** or an **intersection** type instead of an object from our function.

For example, appending `| never` to our definition of `DeepPartial` will fix its display:

```ts
type DeepPartial<T> = {
  [K in keyof T]?: DeepPartial<T[K]>;
} | never; // 👈

type T = DeepPartial<{ a: string; b: { c: string; d: number } }>;
//   ^? { a?: string; b?: { c?: string; d?: number } }  🎉
```

Since `A | never` is always equivalent to `A`, using `| never` doesn't change the output in any way. The only difference is that we have a cleaner display on hover! Using `& {}` would also work.

When debugging Mapped Types from **third-party libraries**, you might not have the option of editing their source code. In this case, I use the very handy `Compute` type:

```ts
type Compute<T> = { [K in keyof T]: Compute<T[K]> } | never;

type DeepPartial<T> = { [K in keyof T]?: DeepPartial<T[K]> };

type A = Compute<DeepPartial<{ a: string; b: { c: string; d: number } }>>;
//   ^? { a?: string; b?: { c?: string; d?: number } }  🎉
```

`Compute` recursively maps over each property in a structure of objects and puts `| never` in front of all of them to force TypeScript to ***evaluate*** and display them.

The only downside of `Compute` is that it can be expensive when used on large objects. Try not to use it too frequently in your production code to avoid slowing type-checking down!

Recursion without a base case

You may have noticed something a little surprising about our `DeepPartial` implementation. We did **not** use a conditional type **to stop** this type from **recursing indefinitely**:

```ts
// How does this stops?
type DeepPartial<T> = {
  [K in keyof T]?: DeepPartial<T[K]>;
};
```

Did we just create an infinite recursive loop? 😱 It would probably be much safer to add a base case when the input is a primitive type, right?

```ts
type Primitive = string | number | boolean | bigint | symbol | null | undefined;

type DeepPartial<T> = T extends Primitive
  ? T // 👈 this is our base case
  : { [K in keyof T]?: DeepPartial<T[K]> };
```

It turns out that a conditional type is **unnecessary** here.

Indeed, Mapped Types already **leave primitive types unchanged**:

```ts
type A = DeepPartial<string>; // string
type B = DeepPartial<number | boolean>; // number | boolean
type C = DeepPartial<null>; // null
type D = DeepPartial<undefined>; // undefined
```

They are also **no-ops** on **literal types**:

```ts
type A = DeepPartial<"hello">; // "hello"
type B = DeepPartial<123>; // 123
type C = DeepPartial<true>; // true
```

Thanks to this behavior, we **don't need to worry about a whole class of problems**, from infinitely deep recursions, to unknowingly transforming primitive types 🙌

Summary 📚

Mapped Types might be the most widespread **advanced TypeScript feature** in **real-world projects** and yet, they are often misunderstood. In this chapter, we went beyond the surface and built an

exhaustive repertoire of their features and edge cases. Here is what we've learned:

- Mapped Types turn **unions** into **objects**.
- We can easily use them to transform objects by combining them with the `keyof` keyword.
- There are four property modifiers: `?`, `-?`, `readonly` and `-readonly`.
- When using `[K in keyof Obj]`, the new object will **inherit** modifiers of `Obj`.
- Mapped Types can also transform **arrays** and **tuples**.
- Thanks to the `as` keyword, also known as **key remapping**, Mapped Types acquire a **whole new level of depth**. We can use them to transform keys, but also to turn unions of data structures into a single object.
- The **entries pattern** enables complex transformation logic by giving us access to the key and the value at the same time!

Challenges 🍜

Congrats! You made it to the end of this long chapter. Let's skip to the fun part: solving some type-level challenges! 🎉

Challenge Solution Format ✨ Reset

```ts
/**
 * Implement a `MakeEnum` generic taking a union of strings
 * and returning an object with a `key: value` pair for
 * each element in the union.
 * Keys and values should be the same.
 */
namespace makeEnum {
  type MakeEnum<Props> = TODO

  type res1 = MakeEnum<"red" | "green" | "blue">;
  type tes1 = Expect<Equal<res1, { red: "red"; green: "green"; blue: "blue" }>>;

  type res2 = MakeEnum<"cat">;
  type tes2 = Expect<Equal<res2, { cat: "cat" }>>;

  type res3 = MakeEnum<never>;
  type tes3 = Expect<Equal<res3, {}>>;
}
```

Challenge Solution Format ✨ Reset

```ts
/**
 * The `groupProperties` function takes a homogenous
 * list of objects, and returns a single object
 * where all properties contain lists of values.
 * Make it generic!
 */
namespace groupProperties {
  declare function groupProperties<T>(args: T[]): ValuesToArrays<T>;

  type ValuesToArrays<T> = TODO;

  const input1 = [{ x: 0, y: 0 }, { x: 12, y: 2 }, { x: 7, y: -2 }];
  const res1 = groupProperties(input1);
  type test1 = Expect<Equal<typeof res1, { x: number[]; y: number[] }>>;

  const input2 = [
    { title: "🍪", createdAt: 1678282179 },
    { title: "🔥", createdAt: 1678283456 }
  ];
  const res2 = groupProperties(input2);
```

Challenge Solution Format ✨ Reset

```ts
/**
 * Build a `DeepRequired` generic that turns every
 * key of a nested object structure into a required property.
 */
namespace deepRequired {
  type DeepRequired<T> = TODO

  type res1 = DeepRequired<{ a?: string; b?: string }>;
  type test1 = Expect<Equal<res1, { a: string; b: string }>>;

  type res2 = DeepRequired<{ a?: { b?: string; c?: { d?: string } } }>;
  type test2 = Expect<Equal<res2, { a: { b: string; c: { d: string } } }>>;

  type res3 = DeepRequired<{ a?: string; b?: { c?: string; d?: number }[] }>;
  type test3 = Expect<Equal<res3, { a: string; b: { c: string; d: number }[] }>>;
}
```

Challenge Solution Format ✨ Reset

```ts
/**
 * When managing application state, a common pattern
 * is to represent user events using "action" object.
 * Action always have a `type` discriminant property,
 * and a "payload" property containing data.
 *
 * Build a `MakeDispatchers` generic that takes a union
 * of Action objects, and turn them into an object with
 * a dispatcher method for every action. a Dispatcher takes
 * the payload and returns void.
 */
namespace dispatchers {
  type UnknownAction = { type: string; payload: unknown };

  type MakeDispatchers<ActionUnion extends UnknownAction> = TODO

  type CreateAction = {
    type: "create";
    payload: { id: number; data: string[] };
  };
```

Challenge Solution Format ✨ Reset

```ts
/**
 * Implement a `SettersAngGetters` generic that takes an object
 * type, and returns a new object with a getter and a setter method
 * for each of the keys present on this object
 */
namespace settersAndGetters {
  type SettersAngGetters<Obj> = TODO

  type res1 = SettersAngGetters<{ theme: "light" | "dark" }>;
  type tes1 = Expect<
    Equal<
      res1,
      {
        getTheme: () => "light" | "dark";
        setTheme: (value: "light" | "dark") => void;
      }
    >
  >;

  type res2 = SettersAngGetters<{ name: string; age: number }>;
```

Challenge Solution Format ✨ Reset

```ts
/**
 * Using what we've learned in chapter 7, implement
 * a `DeepCamelize` generic to turn all keys in a deeply
 * nested structure of objects, arrays and tuples from snake_case
 * to camelCase.
 *
 * Hint: Be careful not to apply key remapping to arrays and tuples.
 */
namespace deepCamelize {
  type DeepCamelize<T> = TODO

  // single layer
  type res1 = DeepCamelize<{ public_key: string; private_key: string }>;
  type test1 = Expect<Equal<res1, { publicKey: string; privateKey: string }>>;

  // nested object
  type res2 = DeepCamelize<{
    url: { path: string; search_params?: { user_id: string } };
  }>;
  type test2 = Expect<
```

Bonus Challenges 🤭

Challenge Solution Format ✨ Reset

```ts
/**
 * Let's improve the type-safety of lodash's `omitBy` function.
 * `omitBy` takes an object, a type predicate, and removes
 * keys that do not pass this predicate.
 *
 * The output type should:
 * - Keep keys that are *not* assignable to the predicate's return type.
 * - Remove keys that are *equal* to the predicate's return type.
 * - Make keys that aren't equal but *assignable* to the predicate's
 *   return type *optional*.
 *
```

```ts
 * @examples:
 *
 * let res1 = omitBy({ name: 'a', age: 1 }, isNumber) // => { name: 'a' }
 * //    ^? { name: string }
 * let res2 = omitBy({ name: 'a', age: 1 }, isString) // => { age: 1 }
 * //    ^? { age: number }
 * let res3 = omitBy({ key: Math.random() > .5 ? 'a' : 1 }, isString)
 * //    ^? { key?: number }
 */
namespace omitBy {
  declare function omitBy(
    obj: TODO,
    predicate: (value: TODO) => value is TODO
  ): OmitBy<TODO, TODO>;

  type OmitBy<Obj, OmittedValue> = TODO
```

Challenge | Solution — Format ✨ — Reset

```ts
/**
 * You fetch data from an API that returns data as a nested JSON structure.
 * The shape of the data isn't very ergonomic, and you would like to
 * write a `deserialize` function that transforms the response into
 * a flatter structure.
 *
 * Your API returns entities with this shape:
 * `{ id, attributes, relationships }`.
 *
 * - `id` is the unique identifier of the current entity.
 * - `attributes` is an object containing all properties of the
 *   current entity.
 * - `relationships` is a record of other related entities.
 *    They can be either objects, or array of objects of
 *    shape `{ id, attributes, relationships }`.
 *
 * Type the generic `deserialize` function to take one of these objects,
 * and recursively flattens the `id`, `attributes` and `relationships` keys
 * to the same level of nesting.
 */
namespace jsonResponse {
  declare function deserialize<T>(response: T): TODO;

  // simple object
  const res1 = deserialize({
    id: 1,
    type: "user",
```

Challenge | Solution — Format ✨ — Reset

```ts
/**
 * Let's type lodash's `update` function.
 *
 * `update` takes an object, a path to one of the
 * value it contains, and an `updater` function that
 * takes the value under this path and can turn it
 * into a value of a different type.
 *
 * @examples
 * ```ts
 * const player = { position: { x: "1", y: 0 } };
 * const res1 = update(player, "position.x", toNumber);
 * //    ^? { position: { x: number, y: number } }
 *
 * const pkg = { name: "such-wow", releases: [{ version: 1 }] };
 * const res2 = update(pkg, "releases[0].version", (v) => `${v}.0.0`);
 * //    ^? { name: string, releases: { version: string }[] }
 */
namespace deepUpdate {
  declare function update<Obj, Path extends string, T>(
    obj: Obj,
    path: Path,
    updater: (value: GetDeep<Obj, Path>) => T
  ): SetDeep<Obj, Path, T>;

  // Hint: It's easier if you start by parsing the
  // `Path` string into a list of properties!
```