8. The Union Type Multiverse

Something I love about **JavaScript** is its **flexibility**. It's hard to find a language that feels more productive for quick prototyping. JavaScript is extremely permissive. If we want to pass a new type of value to an existing function, share a callback between several scopes or mutate a shared state, nothing gets in our way. This is both a strength and a weakness because the more complex our code gets and the harder it is to understand it. This is where a **type system** comes in handy!

Designing a type system sufficiently powerful to capture the **dynamic nature** of JavaScript was surely no easy task. TypeScript does an amazing job at embracing even the trickiest JavaScript patterns we came up with, and I would argue this is largely due to its support for **union types**.

Union types are awesome because they let us perfectly model the **finite set** of possible states our applications can be in. Without them, our types would be so imprecise that they would hardly be of any value.

Let's say we are building an application that fetches data. We could represent its state using this type:

```
export type DataState = {
  status: string;
  data?: number;
  error?: Error;
};
```

But this wouldn't help very much. This type doesn't tell us which values the status property can take, or which ***combinations*** of properties are permitted. Can you have an error property if `status` is equal to "success"? Can you have some `data` *and* an `error` at the same time? We just don't know!

```
import { DataState } from "~/data-state";

let state: DataState = {
  status: "success",
  error: new Error("This is fine. 🔥💀☕🔥"), // This type-checks.
};
```

With a **union type**, it's a different story:

```
export type DataState =
  | { status: "loading" }
  | { status: "success"; data: number }
  | { status: "error"; error: Error };
```

This type lists the precise set of cases our code needs to handle and rules out combinations of properties that do not make sense:

```
import { DataState } from "~/data-state-v2";

// Our type doesn't let us create invalid states anymore!
let state2: DataState = {
  status: "success",
  error: new Error("oooooops"), // ❌ does NOT type-check!
};
```

This example is deliberately simple but illustrates perfectly why union types are so common in our day-to-day TypeScript code.

Looking at the type system from the perspective of learning a new **programming language** has been super fruitful so far. Types and values have a lot in common, but if I had to pick a single feature that sets the language of types apart, it would be union types. There are no good JavaScript analogies to fully describe their behavior. Yet, if we want to build **useful type-level algorithms**, it's crucial to have a deep understanding of the way they work.

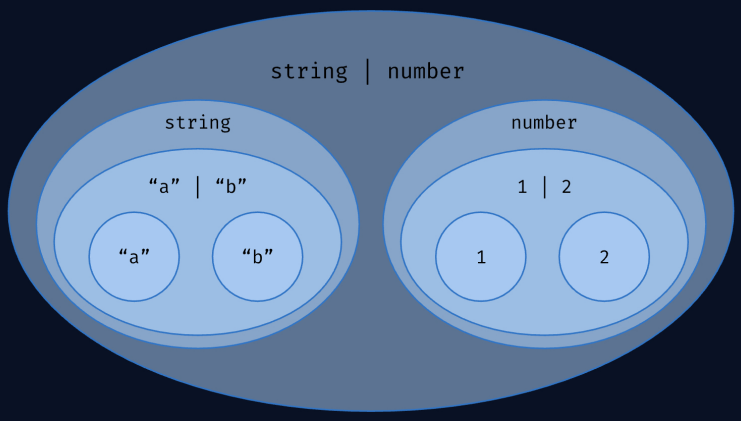
By the end of this chapter, you'll know how to **transform** and **filter** union types, but more

importantly, you'll have an accurate mental model of their behavior. We will see what happens in non-trivial cases such as functions and methods with **advanced type signatures** and understand **why** unions behave this way.

Let's get started!

What do we know about Union Types?

In Types Are Just Data, we discovered that types were really **sets of values**, and that union types were data structures **joining several sets** together to form **larger sets**.



We have also encountered several cases where unions seemed to behave in astonishing ways. For example, when reading several properties from an object using a union of literals, we get a union of values back:

```
type User = { name: string; age: number };
type Value = User["name" | "age"];
// ⇒ string | number
```

Something very similar seems to happen when interpolating a union in a Template Literal Type:

```
type Size = "sm" | "md" | "lg";
type ClassName = `button-${Size}`;
// ⇒ "button-sm" | "button-md" | "button-lg"
```

The union **propagates** to the **outside** of the expression. **Why is that?**

To answer this question, let's take a step back and remember what union types **represent**. At the **type-level**, union types *are* indeed data structures, but unlike object types, they do not *represent* **value-level** data structures. Unions do not have a runtime equivalent!

Let's say I want to read from a `config` object:

```
const config = { env: "dev", port: 3000 };
```

At runtime, I can only access **one** property at a time:

```
const c1 = config["env"]; // string
// later ...
const c2 = config["port"]; // number
```

Now let's define a function that receives a "prop" **parameter** to access a `config` value. I do not know which property I'm accessing anymore, but I can use a union type to represent the different **possibilities** that my code could handle:

```
function logConfig(prop: "env" | "port") {
  const value = config[prop]; // string | number
  console.log(value);
}
```

It doesn't change the fact that I'm accessing one property at a time:

```
logConfig("env");
// later...
logConfig("port");
```

But from the perspective of the type-checker, it's like all possible code paths were happening **simultaneously**! It evaluates the type of `config[prop]` when `prop` is equal to `"env"` and when `prop` is equal to `"port"`, and collects their types as a union representing all possible values I could be reading: `string | number`.

This demonstrates what union types are: **different versions of reality**!

The Union Type Multiverse

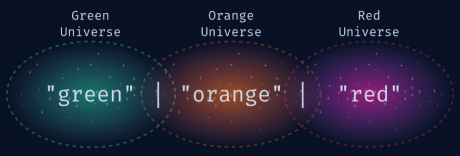
Since unions do not represent **a single runtime *thing*** but rather **different versions** of the runtime **reality**, I like to visualize them as a superposition of states. It's like each member of a union type was a different parallel ***universe***.

For example, if I define a `TrafficLight` union type:

```
type TrafficLight = "green" | "orange" | "red";

const trafficLight: TrafficLight = "green";
```

My runtime `trafficLight` variable will only ever be in one of the three `TrafficLight` states, but from the perspective of the type-checker, all of them exist at the same time:



Each member of a union type is a parallel universe.

We just created a multiverse! 😅

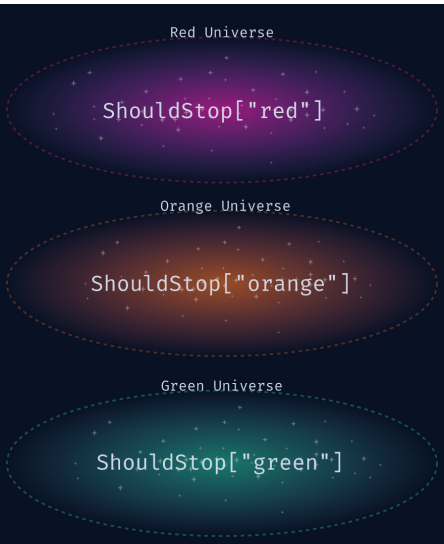
In our Union Type Multiverse, **everything that can happen happens**. Every time we use a union type in a type-level expression:

```
type ShouldStop = { green: "no"; orange: "yes"; red: "yes" };

type T = ShouldStop["green" | "orange" | "red"];
```

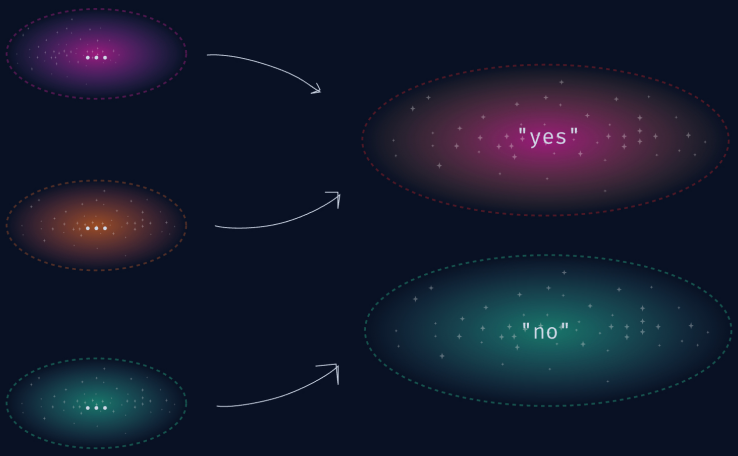
It's like all possible branches were occurring separately:

👇



As a result, we get back another union type that represents this new superposition of states.

Sometimes universes branch out, but sometimes they **merge** back into fewer versions of reality:




```
type T = ShouldStop["green" | "orange" | "red"];
```

Is turned into:

```
type T = ShouldStop["green"] | ShouldStop["orange"] | ShouldStop["red"];
```

Which evaluates to:

```
type T = "no" | "yes" | "yes";
```

Which simplifies into:

```
type T = "no" | "yes";
```

This is what makes union types **special**. At the type level, they're **non-deterministic values**. Every time we use a union inside an expression, it turns it into a non-deterministic one. These expressions can return an arbitrary number of results. This is what gives union types their **distributive nature**.

The Distributive Nature of Union Types

In mathematics, **distributivity** means that an operation produces the **same result** whether you operate it on the whole expression, or on each of its parts before collecting their results.

Multiplication famously *distributes* over addition:

```
x * (a + b) ⟺ (x * a) + (x * b)
--          👆
-- This means "Is equivalent to".
```

And interpolation of union types has the same property:

```
type T1 = `x ${"a" | "b" | "c"}`;
// ⟺
type T2 = `x ${"a"}` | `x ${"b"}` | `x ${"c"}`;
```

Template literal types **distribute over union types**. Our `|` behaves like a `+` and the template literal's `${ ... }` interpolation syntax behaves like a `*`.

Accessing object properties also distributes over union types:

```
type T1 = User["name" | "age"];
// ⟺
type T2 = User["name"] | User["age"];
```

`T1` and `T2` are **equivalent** because they represent the same set of values. TypeScript chooses to turn most expressions that involve union types into their distributed form.

It makes practical sense because TypeScript is very good at analyzing the **control flow** of our code and **eliminating handled cases** from top-level union types to refine them over time:

```
type State =
  | { status: "success"; data: number }
  | { status: "error"; error: Error };

function handler(state: State): string {

  if (state.status === "success") {
    // state is *narrowed*:
    state; // { status: "success", data: number }
    return "🎉";
  }

  if (state.status === "error") {
    state; // { status: "error"; error: Error }
    return "😭";
  }

  return state; // never
  /*       ^
    ✅ Even though `state` isn't a `string`,
    This line type-checks because its type
    is narrowed to `never`.
                                          */
}
```

This mechanism is called **type narrowing**. It's especially effective on *discriminated union types*. These are unions of objects with a discriminant — a property that contains a different value for each of our objects, like the `status` property of the previous example.

▼🤔 **Why don't objects distribute over union types?**

Since TypeScript is so good at narrowing types in their **distributed form**, you might be wondering why it does ***not*** distribute data structures types, like objects and tuples, over union types:

```
// Why doesn't TS turn this type:
type State = {
  status: "success" | "error";
};
// Into this one:
type State =
  | { status: "success" }
  | { status: "error" };
```

These two versions of `State` represent the same set of values, but if TypeScript sticks to the first

version, it's because of type-checking **performance**. There is a **tradeoff** with distributing union types: they get **very big very quickly**.

Imagine you need a type to represent all CSS Properties:

```
type CSSProperties = {
  color: Color;
  display: Display;
  position: Position;
  ...
}
```

There are already around **140 color names** supported by the CSS specification:

```
type Color =
  | "lightseagreen"   | "tomato"           | "chartreuse"
  | "blanchedalmond" | "mediumaquamarine" | "rebeccapurple"  ...
```

Multiply this by the 41 `display` values, the 9 `position` values and the number of possible values of the 259 remaining CSS properties and you get the number of elements your resulting `CSSProperties` union would have if TypeScript was distributing unions over objects:

```
type CSSProperties =
 | { color: "lightseagreen"; display: "flex"; position: "absolute";  ... }
 | { color: "lightseagreen"; display: "inline"; position: "absolute";  ... }
 |  ...  // 1293049102847548102943582O lines later ...
 | { color: "white"; display: "flex"; position: "fixed";  ... }
 | { color: "white"; display: "inline"; position: "fixed";  ... };
```

This union would be ***ginormous***!

This ***combinatorial explosion*** explains why it's not reasonable to distribute data structure types over unions. It's much more valuable to have a way to represent these large collections of properties and the type-narrowing benefit that distributing would bring is just not worth it. That's why **Objects**, **tuples** and **function types** do not distribute over unions.

If you want TypeScript to narrowing your types in `if` or `switch` statements, you'll have to define them as top-level unions manually, or reach out to libraries like TS-Pattern that can narrow any type for you.

Now that we understand why type-level *expressions* tend to distribute over union types, let's talk about the most important instance of distributive union types for type-level programming — **Conditional Types**.

Unions & Conditional Types

Conditional types are the secret weapon of type-level programmers. They are the main building blocks of our smart type-inference logic. The most interesting things happen when combining them with other features of the type system, and union types are no exception.

Let's start with a **simple example**:

```
type IsString<T> = T extends string ? "yes" : "no";
```

This `IsString` generic returns "yes" if the input type is assignable to `string` and "no" otherwise:

```
type T1 = IsString<"is this a string?">; // "yes"

type T2 = IsString<123>; // "no"
```

Now, what if we give a **union type** to `IsString`:

```
type T = IsString<"a" | 2 | "c">; // yes? no? 🤔
```

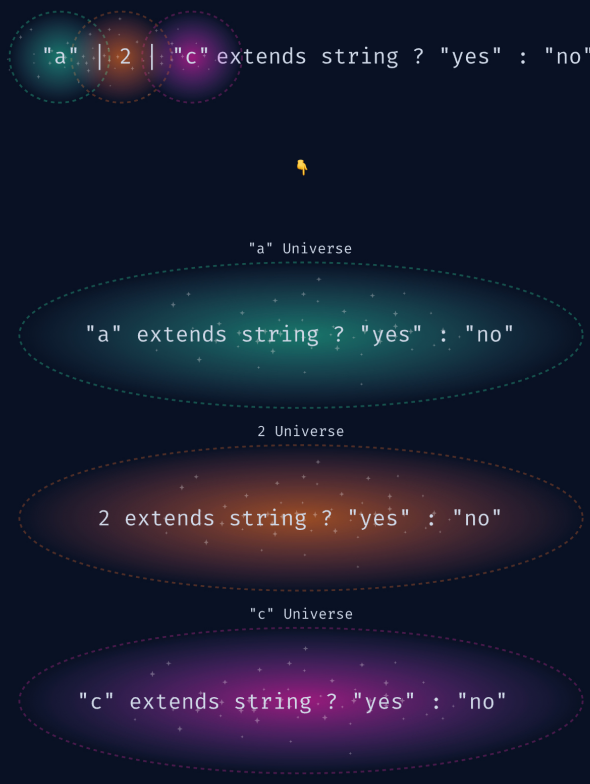
Using the knowledge we've acquired in Types Are Just Data and Code Branching with Conditional Type, we can tell that `"a" | 2 | "c"` isn't assignable to `string`, which means that the expression `IsString<"a" | 2 | "c">` should evaluate to `"no"`, right?

But this is **not** what happens:

```
type T = IsString<"a" | 2 | "c">;
//   ⇒  "yes" | "no"                ✨ 🪶 ✨
```

It returns `"yes" | "no"` instead because **conditional types *distribute* over union types** too. We just entered the multiverse again.

Conditional Types distribute over Union Types.



All possibilities happen separately.

TypeScript turns `IsString<"a" | 2 | "c">` into:

```
type T =
    | ("a" extends string ? "yes" : "no")
    | ( 2  extends string ? "yes" : "no")
    | ("c" extends string ? "yes" : "no");
```

Which simplifies to:

```
type T = "yes" | "no" | "yes";
// ⟺
type T = "yes" | "no";
```

Just like before, the type checker evaluates the expression **for each element** of the union types and collects their results as a union afterward.

This is a great feature!

This is probably **the most confusing** part of the type-level language, but it's also an **awesome feature**. Most of the time, the type-level code we write will *Just Work* with union types!

Remember the `SnakeToCamel` challenge from the previous chapter? The goal was to write a generic to turn strings formatted in `snake_case` to `camelCase`. Here is a possible solution:

```
type SnakeToCamel<Str> =
  Str extends `${infer First}_${infer Rest}`
      /*              👆
           Split the string on underscores
                                          */
    ? `${First}${SnakeToCamel<Capitalize<Rest>>}`
    /*                        👆
       Capitalize each word before joining them.
                                               */
    : Str;

type T = SnakeToCamel<'hello_world'> // "helloWorld"
```

It turns out you can pass a union type to this function ***too***!

```
type T1 = SnakeToCamel<"user_name" | "is_admin">;
// => "userName" | "isAdmin" ✨

type T2 = SnakeToCamel<"conditional_types" | "distribute" | "over_unions!">;
// => "conditionalTypes" | "distribute" | "overUnion!" 🤯
```

Our union branches out at the first `extends`:

```
type SnakeToCamel<Str> =
  Str extends `${infer First}_${infer Rest}`
  /*    👆
     The expression distributes
     over `Str` here.            */
    ? ...
    : ... ;
```

After that, TypeScript executes the rest of `SnakeToCamel` for each element of the union separately, in their own parallel universe!

TypeScript just made us fall into The Pit of Success. We did ***not*** think a single time about union types when implementing this function and yet, it handles them **perfectly**! Isn't that cool?

Everything is a union type!

Since union types behave differently from other types inside type-level expressions, it may feel like our *mental model* for how the type-checker interprets our code suddenly got more complex. When writing a generic, we now have to think about two kinds of inputs: **union types** and **single types**.

The good news is that we can **unify** these by realizing that every type can be thought of as a **union** of **zero**, **one** or **many** elements!

We usually think of unions as sets of **several** types:

```
type U = "a" | "b";
// is the type-level equivalent of:
const u = new Set(["a", "b"]);
```

But you can also think about a **single** type as a *union*:

```
type U = "a";
// is the type-level equivalent of:
const u = new Set(["a"]);
```

It just happens to contain a single element!

As you know, the `never` type is the **empty set**:

```
type U = never;
// is the type-level equivalent of:
const u = new Set([]);
```

Which means `never` is an **empty union type**!

Once you realize this, everything we've seen in this course unifies nicely with this new concept of ***distributivity***. Regular types ***also*** "distribute", but since they are unions containing a single element

distributivity. Regular types **also** distribute, but since they are unions containing a single element, our code is executed only once, in a single universe! Isn't it beautiful? 😊

`never` distributes too

In Code Branching with Conditional Types, I mentioned that using `never` on the left-hand side of `extends` would always result in the `never` type:

```
type IsMeaningOfLife<N> = N extends 42 ? true : false;

type T = IsMeaningOfLife<never>; // never
```

I introduced this as a *special case* at the time, but it isn't special at all! `never` being an empty union, how would you distribute a conditional type over it?

```
type IsMeaningOfLife<N> = N extends 42 ? true : false;
//                        ^
//              How to distribute over `never` 🤔
//
```

The solution is to do nothing! It's like **mapping over an empty list**:

```
// Running a conditional type on `never`
type IsMeaningOfLife<N> = N extends 42 ? true : false;
type Result = IsMeaningOfLife<never>; // never

// Is similar to mapping over an empty list:
const isMeaningOfLife = (list: number[]) => list.map((n) => n === 42);
const result = isMeaningOfLife([]); // []
```

In both cases, the **empty element** (`never` or `[]`) is returned unchanged. This similarity between the distribution of union types and mapping over arrays is pretty interesting, so let's explore this idea further.

Mapping Over Union Types

When we use a conditional type on a union, **we apply a transformation to each of its elements**. Conditional types have two branches, but if we use a condition that **always passes**, we can transform all elements **in the same way**:

```
type MapOverUnion<U> = U extends unknown
  /*                   👆           👆
              The expression      This condition
             distributes here.    always passes.
                                                  */
  ? Transform<U>
  /*          👆
    `Transform` is called with every member, one by one.
                                                    */
  : never;
```

Since all TypeScript types are assignable to `unknown`, `U extends unknown` always passes, so all members of `U` will go through the same code branch.

Let's say we have a `Duplicate` generic that puts a type twice in a tuple:

```
type Duplicate<T> = [T, T];
```

If we call `Duplicate` with `1 | 2 | 3` we get a tuple containing this union twice:

```
type T1 = Duplicate<1 | 2 | 3>;
//   ⇒  [1 | 2 | 3, 1 | 2 | 3]
```

But if we use `U extends unknown` beforehand, we get a union of tuples instead:

```
type DistributeDuplicate<U> = U extends unknown ? [U, U] : never;
```

```ts
type T2 = DistributeDuplicate<1 | 2 | 3>;
//   ⇒  [1, 1] | [2, 2] | [3, 3]
```

`T1` and `T2` are different because `T1` can contain mixed arrays, but `T2` cannot:

```ts
const tuple1: T1 = [1, 2]; // ✅ type-checks!
const tuple2: T2 = [1, 2]; // ❌ does not type-check.
```

We essentially just mapped `Duplicate` over our union type!

```ts
[1, 2, 3].map((x) => [x, x]);
// ⇒ [[1, 1], [2, 2], [3, 3]]
```

Distributivity & Unions Of Objects

Suppose you want to retrieve **every key** from a **union** of **objects**. You might initially think that using `keyof` would do the trick:

```ts
type AllKeys<T> = keyof T;
```

But this isn't going to return every key:

```ts
type T = AllKeys<{ a: 1; b: 1 } | { a: 1; c: 1 }>;
// This only returns "a".
```

As we've seen in Objects & Records, `keyof` only returns keys that exist on **all objects** in the **union**:

```ts
type T = keyof ({ a: 1; b: 1 } | { a: 1; c: 1 });
// ⇒ "a"
```

If you want to retrieve **all existing keys**, you will need to **distribute** over your union first!

```ts
type AllKeys<T> = T extends unknown ? keyof T : never;
//                  👆
//        We distribute over `T` first!

type T = AllKeys<{ a: 1; b: 1 } | { a: 1; c: 1 }>;
// ⇒ "a" | "b" | "c" 🎉
```

Now, `keyof` is applied to each member of the union separately, which means we get back the union of all of their keys!

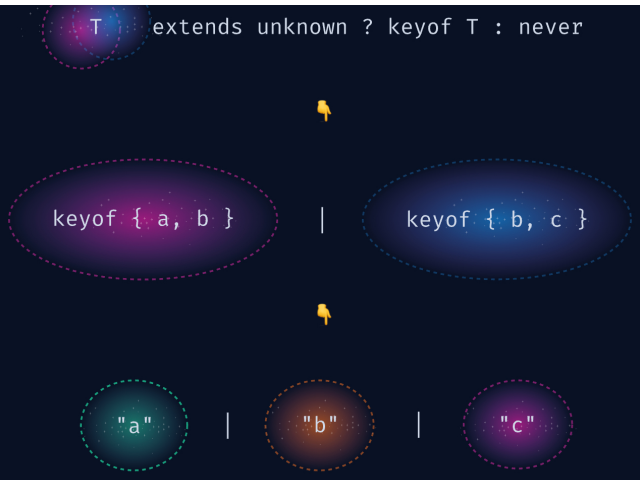
```ts
type T = keyof { a: 1; b: 1 } | keyof { a: 1; c: 1 };
// ⇒ "a" | "b" | "c"
```

This `AllKeys` generic can be very handy to **merge** several object types into a single one. We will explore this in the next chapter.

This also demonstrates something that took me a while to understand. Even though we use **the same** `T` variable name **inside** and **outside** of the conditional type, they do ***not*** mean the same thing!

```ts
type AllKeys<T> = T extends unknown ? keyof T : never;
//                👆                    👆
//      This is our union type         This is each item
//      BEFORE distribution.           one by one.
```

Note that all unions returned by individual instances of `keyof T` are **flattened** into a **single union type**:



It makes sense since unions can't be nested, but this is also where the analogy between distributive union types and `array.map` breaks down. Unlike mapping on arrays, the number of elements in our input and output unions may be different.

We can use conditional types to generate larger unions, but can we also use them to create union types with **fewer** elements?

Filtering Union Types

The purpose of most of our type-level programs is to help TypeScript **narrow** one of our input types down, so knowing how to filter elements from a union type is invaluable.

Imagine we are building the newsfeed of a social network. We need to display many different types of posts on a single page. We can represent this data as a union of objects:

```
type FeedItem =
  | { type: "post"; content: string }
  | { type: "likedBy"; user: string; content: string }
  | { type: "followSuggestions"; users: string[] }
  | { type: "image"; src: string }
  | { type: "video"; src: string };
```

We would like to add a "media" section to our page that only lists `video` and `image` items. Let's also suppose that filtering a list of objects based on their `"type"` property comes up over and over in our codebase. We would like to build an `only` generic function to ease this process:

```
declare const items: FeedItem[];

// How should we type this function?
const only = (items, types) =>
  items.filter((item) => types.include(item.type));

const mediaItems = only(items, ["image", "video"]);
// we would like `mediaItems` to be inferred as:
//  ( | { type: "image"; src: string }
//    | { type: "video"; src: string } )[]
```

So how do we **type** our `only` function?

First, we need to define which input types TypeScript should **infer**.

If we want `only` to be truly generic, it should work with all possible arrays, so we want the type of item to be inferred. It should also work with any list of "type" strings, so we'll need a second type parameter:

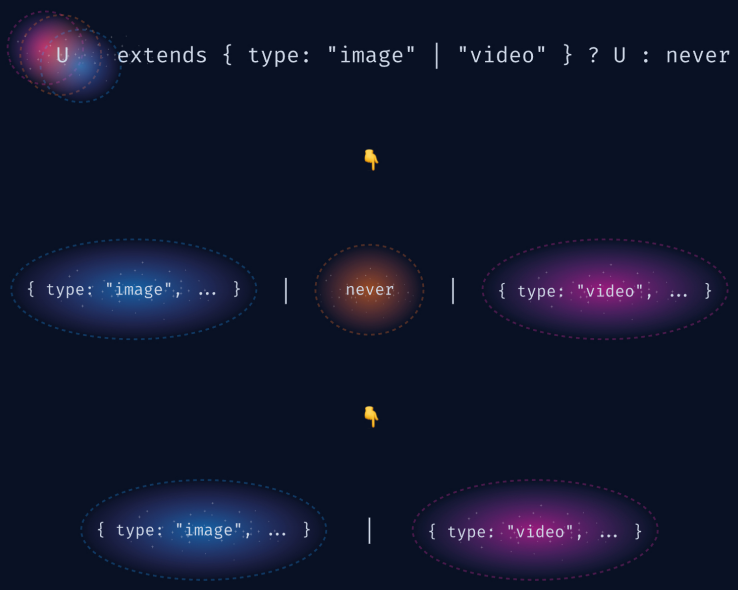
```
// Both `Item` and `Type` are union types!
declare function only<
  // Items should have a `type` property
  Item extends { type: string },
  // `Type` strings should be valid type properties.
  Type extends Item["type"]
>(list: Item[], types: Type[]): FilterUnion<Item, Type>[];
//                               ~~~~~~~~~~
//                                  ^ 🤔
```

In our previous example, the `Item` type parameter will be inferred as `FeedItem` and `Type` as `"image" | "video"`. Now, we need to narrow `Item` based on `Type`. It turns out we can simply use a **conditional type**:

```
type FilterUnion<Item, Type> =
  Item extends { type: Type }
    ? Item
    : never;
```

This looks simple, but how does it work?

Once again we rely on **distributivity**. Our condition will run **for each element** in the union. If the element is assignable to `{type: Type}`, we return it, otherwise, we return `never`:



`never` simplifies out, which means we only get elements passing our condition as a result:

```
type T = FilterUnion<FeedItem, "image" | "video">;
// ⇒ | { type: "image"; src: string }
//   | { type: "video"; src: string }
```

We successfully filtered our union type!

▼🤔 Couldn't we have used an intersection too?

We could have!

Intersections are another valid way of narrowing union types in TypeScript. When intersecting a type `A` with a union, we filter it to only include members assignable `A`:

```
type T = FeedItem & { type: "image" | "video" };
// ⇒ | { type: "image"; src: string } & { type: "image" | "video" }
//   | { type: "video"; src: string } & { type: "image" | "video" }
```

But as you can see, the displayed type isn't as clean as with a conditional type because the intersection remains. This is semantically equivalent to:

```
type T = | { type: "image"; src: string }
         | { type: "video"; src: string };
```

But our type errors will be a bit more confusing with an intersection.

Conditional types also allow more complex filtering logic so I tend to prefer them over intersections.

Filtering union types is such a common operation that it could be worth making our `FilterUnion` generic even more ***generic***! 😅

The only "business logic" assumption it currently makes is that every member of the input union has a `type` property. Let's make it fully configurable:

```
type FilterUnion<U, C> = U extends C ? U : never;

type T = FilterUnion<FeedItem, { type: "image" | "video" }>;
// ⇒ | { type: "image"; src: string }
//   | { type: "video"; src: string }
```

The only change we've made is to define the `{ type: ... }` object outside of `FilterUnion`, and now we can use it to filter **any possible union type**:

```
type T1 = FilterUnion<true | "maybe", boolean>;
// ⇒ true

type T2 = FilterUnion<1 | "👋" | 2 | "🤓", string>;
// ⇒ "👋" | "🤓"
```

This is great! The only downside is that we just reinvented the wheel a tiny bit. `FilterUnion` **already exists** under a different name in TypeScript's standard library. **It is called** `Extract`.

Helper functions

TypeScript comes with two built-in generics to deal with union types: `Extract` and `Exclude`.

Extract

`Extract` takes two types, and **removes** all elements of the first type that **aren't assignable** to the second one. It works exactly like `FilterUnion` from the previous section:

```
type T1 = Extract<"a" | "b" | "c" | null, string>;
// ⇒ "a" | "b" | "c"

type T2 = Extract<"value" | "onChange" | "onSubmit", `on${string}`>;
// ⇒ "onChange" | "onSubmit"
```

Here is its definition:

```
type Extract<A, B> = A extends B ? A : never;
```

▼ 🤔 **Aren't `Extract` and `As` the same thing?**

If the definition of `Extract` feels familiar, it's because we've seen it in Loops with Recursive Types under yet another name. We were calling it `As` at the time, and we used it to **infer a type constraint** on our type parameters and "force" TypeScript to accept our code:

```
type Push<List, Item> = [...List, Item];
//                          ^ ❌ List isn't an array

type As<A, B> = A extends B ? A : never;

type Push<List, Item> = [...As<List, any[]>, Item];
//                          ^ ✅ type-checks!
```

It turns out we could just have used `Extract`!

```
type Push<List, Item> = [...Extract<List, any[]>, Item];
//                         ^ ✅ this works too!
```

Exclude

`Exclude` takes two types, and **filters out** all elements of the first type that **are assignable** to the second one.

It's the opposite of `Extract`:

```
type T1 = Exclude<"a" | "b" | "c" | null, null>;
// ⇒ "a" | "b" | "c"

type T2 = Exclude<"value" | "onChange" | "onSubmit", `on${string}`>;
// ⇒ "value"
```

Here is how `Exclude` is defined in TypeScript's standard library:

```
type Exclude<A, B> = A extends B ? never : A;
```

Distributive conditional types rules

I have to confess something. To make the mind-bending concept of *distributivity* more accessible, I omitted an important detail when presenting how union types and conditional types interact. There's one more rule you should know: **Conditional types *only* distribute over *"naked" type variables***.

They do *not* distribute over union types defined inline:

```
// This union type is defined inline. Not distributive.
//            👇
type X = string | number extends string ? true : false;
// ⇒ `false` instead of `true | false`
```

Nor union types resulting from a generic:

```
// This isn't a naked type variable. Not distributive.
//                    👇
type IsString<T> = OrNumber<T> extends string ? true : false;
type OrNumber<T> = T | number;
type X = IsString<string>;
// ⇒ `false` instead of `true | false`
```

A conditional type will only distribute over a single type variable:

```
// This is a naked type variable. It distributes.
//                 👇
type IsString<T> = T extends string ? true : false;
type X = IsString<string | number>;
// ⇒ true | false
```

This is an important gotcha if you want a perfect mental model of union types, even though in practice the latter case is the most common.

Avoiding distribution

Before we wrap up, I want to teach you one more trick: **preventing distribution from happening**. This can be useful if, for example, you want to check if **a union** is **assignable** to another one:

```
type Extends<A, B> = A extends B ? true : false;
//                   👆
//           A is distributed here.

type T = Extends<"a" | "b" | "c", "a" | "b">; // ⇒ `boolean`
// `Extends` returns boolean because "a" and "b" are
//  assignable to "a" | "b", but "c" isn't.
```

It would make more sense for `Extends` to return `false` instead of `boolean` in this case, but how can we fix it?

As we've seen in the previous section, we have several ways to prevent distribution from happening. The simplest is to **wrap** your union in a **data structure** before using `extends`:

```
type Extends<A, B> = [A] extends [B] ? true : false;
//                    👆
//      Wrapping `A` in a tuple prevents
//       the union from distributing.

type T = Extends<"a" | "b" | "c", "a" | "b">;
// ⇒ false 🎉
```

In this example, `[A]` evaluates to `["a" | "b" | "c"]`. The union that used to be at the top level is now nested. Since TypeScript only distributes expressions over top-level union type variables, wrapping a union in a **tuple** prevents this behavior.

This means we can access our ***undistributed*** union in one of our branches if we want to:

```
type CreateEdge<Id, Condition> =
  [Id] extends [Condition]
    ? { source: Id; target: Id }
    /*          👆          👆
              Here, `Id` is the
            undistributed union type.
                                        */
    : never;

type Edge1 = CreateEdge<"1" | "2" | "3", `${number}`>;
// ⇒ { source: "1" | "2" | "3"; target: "1" | "2" | "3" }

type Edge2 = CreateEdge<"a" | "b" | "c", `${number}`>;
// ⇒ never

const edge: Edge1 = { source: "1", target: "2" };
//     ^ ✅ type-checks!
```

If we had written `Id extends Condition` instead, The last line wouldn't have type-checked because the conditional type would have been distributed over `Id` and only edges where `source` and `target` are the same would have been permitted. Try it yourself!

💡 Press the "Edit" button at the top-right corner of any code block to see type-checking in action.

Summary 📚

Union types are a powerful feature that allows you to specify **multiple possible types** for a **single value**. While they may seem straightforward at first, they can exhibit **complex behavior** when used in type-level expressions. This gets particularly counterintuitive with conditional types, but with the right **mental model**, everything starts to make sense!

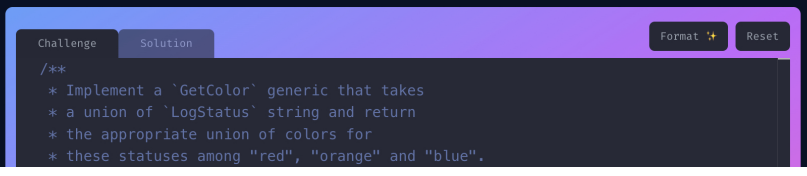
Let's summarize what we've learned:

- Union types represent **different versions of reality**!
- At the type level, they are **non-deterministic values**. When used in an expression, every possible code path is executed and results are returned as a union type.
- That's what gives union types their **distributive nature**.
- We can use this principle to **transform** and **filter** union types with conditional types!
- **Every type** is a union type. Single types can be thought of as unions containing a **single element**, and `never` can be thought of as an **empty union type**.

With this in mind, let's try solving a few challenges!

Challenges 🔥

As always, it's time to put what we have learned into **practice**! Remember that failing means learning new things! Being open to it is the key to making progress. Feel free to check the "solution" tab and ask any questions on Discord 😬



```ts
 */
namespace getColor {
  type LogStatus = "error" | "warning" | "info";

  type GetColor<Status extends LogStatus> = TODO

  type res1 = GetColor<"error">;
  type test1 = Expect<Equal<res1, "red">>;

  type res2 = GetColor<"error" | "warning">;
  type test2 = Expect<Equal<res2, "red" | "orange">>;

  type res3 = GetColor<"warning" | "info">;
  type test3 = Expect<Equal<res3, "orange" | "blue">>;
```

Challenge Solution Format ✨ Reset

```ts
/**
 * Implement an `AllValues` generic that
 * takes a union of objects, and returns
 * every possible value these objects can
 * hold inside any of their properties.
 */
namespace allValues {

  type AllValues<T> = TODO

  type res1 = AllValues<{ a: "value a" }>;
  type test1 = Expect<Equal<res1, "value a">>;

  type res2 = AllValues<{ a: "value a" } | { b: "value b" }>;
  type test2 = Expect<Equal<res2, "value a" | "value b">>;

  type res3 = AllValues<{ a: string; b: number } | { b: boolean; c: bigint }>;
  type test3 = Expect<Equal<res3, string | number | boolean | bigint>>;
}
```

Challenge Solution Format ✨ Reset

```ts
/**
 * Implement a `ToState` generic that takes a
 * union of `Status` strings, and returns a union
 * of data fetching state objects that look like
 * `{ status, isLoading, data, error }`.
 *
 * Make your solution as terse as possible!
 */
namespace toState {
  type Status = "loading" | "success" | "error";

  type ToStates<S> = TODO

  /**
   * Test helpers, do not use in your solution!
   */
  type LoadingState = {
    status: "loading";
    isLoading: true;
    data: undefined;
```

Challenge Solution Format ✨ Reset

```ts
/**
 * Lodash's `compact` function takes an array
 * that can contain "falsy values", and returns
 * an array with these values filtered out.
 * Write its generic type signature!
 *
 * Note: The list of "falsy" values is
 * false, null, 0, "", and undefined.
 */
namespace compact {

  declare function compact(
    list: TODO
  ): TODO;

  let res1 = compact([1, 2, null, 3, undefined, 4]);
  type test1 = Expect<Equal<typeof res1, number[]>>;

  let res2 = compact([undefined, 0 as const, "a", "", "b", "c", "d"]);
  type test2 = Expect<Equal<typeof res2, string[]>>;
```

Challenge Solution Format ✨ Reset

```ts
/**
 * Lodash's `partition` function takes an array, a *type guard*
 * function, and returns a *tuple* containg *two arrays*.
 * - The left array contains elements passing the predicate
 * - the right array contains elements that didn't pass
 *
 * Type `partition` so that output arrays are narrowed
 * using the return type of the guard function.
 *
 * Note: Type guards are predicate functions that TypeScript
 * can use for type narrowing. their type signatures look like this:
 * `(param) => param is SomeType`.
 * Learn more: https://www.typescriptlang.org/docs/handbook/advanced-types.html#user-
 */
namespace partition {
  declare function partition(
    list: TODO,
    predicate: (value: TODO) => value is TODO
  ): TODO;
```

Challenge | Solution — Format ✨ — Reset

```ts
/**
 * The `flatten` function takes a heterogeneous array
 * that can contain arrays, but also regular values.
 * It should flatten arrays but should leave other
 * values unchanged.
 *
 * Note: This is *not* a recursive flatten!
 * Only one level of nesting should be removed.
 */
namespace heterogeneousFlatten {

  type Flatten<Arr extends any[]> = TODO

  declare function flatten<A extends any[]>(arr: A): Flatten<A>;

  let res1 = flatten([1, 2, [3, 4]]);
  type test1 = Expect<Equal<typeof res1, number[]>>;

  // if the array is already flat, leave it unchanged
  let res2 = flatten(["a", "b", "c", "d"]);
```

Bonus challenges 🧠

Challenge | Solution — Format ✨ — Reset

```ts
/**
 * Implement a generic that checks if a
 * type is the `never` type.
 *
 * You are not allowed to use `Equal`!
 */
namespace isNever {
  type IsNever<T> = TODO;

  type res1 = IsNever<never>;
  type test1 = Expect<Equal<res1, true>>;

  type res2 = IsNever<"not never">;
  type test2 = Expect<Equal<res2, false>>;

  type res3 = IsNever<1 | 2 | never>;
  type test3 = Expect<Equal<res3, false>>;
}
```

Challenge | Solution — Format ✨ — Reset

```ts
/**
 * Implement a `MyEqual` generic that checks if
 * 2 types are equal. It should support union types,
 * but it doesn't need to support `any`.
 *
 * Note: `Equal` is already in scope, but you
 * aren't allowed to use it! 😅
 */
namespace equal {
  type MyEqual<A, B> = TODO

  type res1 = MyEqual<"a", "a">;
  type test1 = Expect<Equal<res1, true>>;
```

```ts
type res2 = MyEqual<"a", "b">;
type test2 = Expect<Equal<res2, false>>;

type res3 = MyEqual<1 | 2, 1 | 2>;
type test3 = Expect<Equal<res3, true>>;
```

Challenge Solution Format ✨ Reset

```ts
/**
 * A `Table` is a tuple containing `Column`s.
 * A `Column` is an object with a `name` and a `values`
 * property.
 *
 * Using what we've learned, type the `findColumn`
 * function to take a generic `Table`, a column name and
 * return the type of its `values` property.
 *
 * Try to find a solution that doesn't involve
 * a recursive type!
 */
namespace findColumn {

  type Column = { name: string, values: unknown[] }

  declare function findColumn(table: TODO, columnNames: TODO): TODO;

  declare const userTable: [
    { name: 'firstName', values: string[] },
```

Challenge Solution Format ✨ Reset

```ts
/**
 * Create an `AllPaths` generic that returns the
 * union of all possible paths in an object.
 * A path is a succession of properties separated by dots.
 *
 * Try not to use Mapped Types!
 *
 * @examples
 * { a: unknown } => 'a'
 * { a: { b: unknown } } => 'a.b'
 */
namespace allObjectPaths {

  type AllPaths<T> = TODO

  type res1 = AllPaths<{ name: string; age: number }>;
  type test1 = Expect<Equal<res1, "name" | "age">>;

  type res2 = AllPaths<{ user: { name: { first: string } } }>;
  type test2 = Expect<Equal<res2, "user" | "user.name" | "user.name.first">>;
```

⟸ Previous
7. Template Literal Types

Next ⟹
9. Loops with Mapped Types