5. Code branching with Conditional Types

After learning about the many kinds of types we can play with in the three previous chapters, it's time to implement **our first type-level algorithms**! We finally get to write some actual code in the language of types. Yay! 🎉🎉🎉

In every programming language, the most basic form of algorithmic logic is **code branching**. The `if` and `else` statements count almost certainly among the first lines of code you have ever written, so I suspect you know how important and widespread they are in programming.

```
if (trafficLight === "green") go();
else stop();
```

Being a Turing Complete **programming language**, the **type system** of TypeScript of course supports code branching! But what are use cases for executing different code paths in our types, and how would we even do that? Let's find out!

Anatomy of a Conditional Type

In Type-level TypeScript, code branching is known as **Conditional Types**. The syntax is very similar to the Ternary Operators we use in JavaScript:

```
type TrueOrFalse = A extends B ? true : false;
/*                 ———————————   ———   ———————
                      ^          /         \
                  condition    branch     branch
                               if true    if false

                   \———————————————————————/
                               ^
                         Conditional Type
*/
```

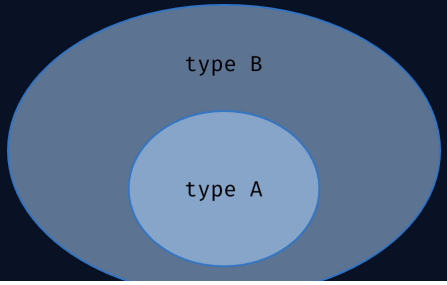
Since the language of types is **functional**, it tends to guide developers towards writing **expressions** rather than statements. There are no `if` or `else` statements at the type level. Instead, we will exclusively write ternary expressions using `?` and `:`.

The `extends` keyword

Before the question mark stands a **condition**. It's **always** of the form `A extends B`, which is how you ask *"Is A assignable to B?"* to the type checker.

> *"A extends B" means "**Is A assignable to B?**"*

We touched on the topic of assignability several times already so you should already know that *"A is assignable to B"* means that the set of values defined by the type `B` *includes* the set of values defined by the type `A`.



type A is assignable to type B

Check out Types Are Just Data if you want a refresher. Assignability is such an important and complex concept that we will dedicate an upcoming chapter to cover the part that isn't so intuitive about it.

Note that `A extends B` **does not return** a boolean literal like `true` or `false`. You **can't** replace it with a hard-coded boolean type:

```
type T = true ? true : false;
//       ^ ❌ Syntax error: TS expects an expression
//                          of the form "A extends B"
```

This also means you can't assign `A extends B` to a variable:

```
type T = 2 extends number;
//         ^ ❌ Syntax error:
//              invalid use of `extends`.
```

To have a valid Conditional Type, you need a full `A extends B ? ... : ...` expression:

```
type T = 2 extends number ? true : false; // ✅
// ⇒ true
```

Of course, conditional types only start to make sense when branching on **type parameters**, so let's implement a very simple **type-level function**:

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

type OK = IsMeaningOfLife<42>; // ⇒ true
type KO = IsMeaningOfLife<41>; // ⇒ false
```

We check if `N` is **assignable** to the literal `42`, return `true` if it is, and `false` if it isn't. With a literal type on the right-hand side, `extends` essentially behaves like an `===`.

▼🤔 **What if we were passing `never` to `isMeaningOfLife`?**

Since `never` is a subtype of every other type, it should be assignable to `42` too, and we should get back the type `true`, right? Let's find out!

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

type wat = IsMeaningOfLife<never>;
//   ?^ never 🤔

type T1 = Expect<Equal<wat, true>>; // ❌
type T2 = Expect<Equal<wat, never>>; // ✅
```

Looks like we get back `never` instead of `true`. That's surprising. `never` isn't even part of the possible outputs of our `IsMeaningOfLife` function! How is this possible?

There is a satisfying answer to this question that doesn't conflict with our subtyping mental model, but it's a bit subtle and takes time to explain properly. I do think this is important though, that's why we will cover it in the upcoming The Union Type Multiverse chapter!

Until then, you can consider `never` a **special case**. When put on the left side of `extends` in a conditional type, it always evaluates to `never`.

If we push this idea to its limit and parameterize everything, we can even implement an `If` **type-level function** that takes a boolean, two arbitrary types and returns one or the other depending on the boolean value:

```
type If<A extends boolean, B, C> = A extends true ? B : C;
```

```ts
type a = If<true, number, string>; // number
type b = If<false, {}, []>; // []
```

As you can see, we use the `extends` keyword **twice** here. Once after the type parameter `A`, and another time in the `A extends true` condition. Even though we use the same keyword, these two occurrences of `extends` don't exactly have the same **meaning**.

`A extends true` is the **condition** of our code branching expression, but what about `A extends boolean`?

This is called a **type constraint**.

Type constraints

Type Constraints let us make sure that one of our type-level function's parameters **will always be assignable to some type.**

In our example, `If<A extends boolean, ...> = ...` is a way to enforce that our function `If` can only accept types **assignable** to `boolean` as first parameter.

The `boolean` type has **4 subtypes**: `true`, `false`, `boolean` and `never`. Remember `never`, the empty set? It's a subtype of every other type, including `boolean`!

```ts
type T1 = If<true, number, string>; // ✅ ⇒ number
type T2 = If<false, number, string>; // ✅ ⇒ string
type T3 = If<never, number, string>; // ✅ ⇒ never
type T4 = If<"not a boolean", number, string>;
//           ~~~~~~~~~~~~~~~ ❌ not assignable to type `boolean`.
```

Intuitively, you can think of constraints as a way to **type** our type-level function parameters. They are **the types of our types**! 🤓

This intuition is helpful, but I must admit that this isn't *strictly* true. Type constraints only set an **upper-bound** to a type parameter, but they aren't fundamentally different from regular types the way types are different from values.

By default, type parameters have no constraints, which is equivalent to constraining them with the `unknown` type:

The `extends` keyword is always about assignability. In a **conditional type**, we use `extends` to ask the question *"Is A assignable to B?"* to the type-checker. In a **type constraint**, however, `extends` is an affirmation: *"A must be assignable to B."*

Narrowing input types with type constraints

Type constraints can not only prevent invalid inputs, but also make TypeScript **infer precise types** for our parameters. Let's illustrate that with an example.

Imagine we have a function taking a `string` and wrapping it in an object:

```ts
const createUser = <S>(name: S) => ({ name });
```

If we give this function the string `"Gabriel"` we hope to get a value of type `{name: "Gabriel"}` back. Let's try it out:

```ts
const user = createUser("Gabriel");
//     ^? { name: string } 😿
```

We get `{ name: string }` instead! This isn't very helpful. Why would we even use a type parameter if TypeScript always infers it as a `string`? It would have been just as good to hardcode `name` to `string`!

But there is a way to make TypeScript infer `name` as a **string literal type**:

```
const createUser = <S extends string>(name: S) => ({ name });
//                            👆
//             We add a `string` constraint on `S`.

const user1 = createUser("Gabriel");
//      ^? { name: "Gabriel" } 🙀

const user2 = createUser("Alice");
//      ^? { name: "Alice" } 🎈
```

When TypeScript sees a constraint on a type parameter, it will try to infer it to a **subtype** of this constraint! We are going to use this a lot in upcoming chapters to write functions that lift as much information as possible to the type level and improve their **type safety**.

Here is another example. Inferring an input array as a **tuple type**:

```
const inferAsTuple = <
  T extends [unknown, ...unknown[]]
  //                  👆
  //   We use a non-empty array of unknown values.
>(tuple: T) => tuple;

const t1 = inferAsTuple([1, 2]);
//    ^? [number, number]
// instead of number[]

const t2 = inferAsTuple(["a", 2, true]);
//    ^? [string, number, boolean]
// instead of (string | number | boolean)[]
```

Regular arrays like `number[]` ***aren't*** assignable to a non-empty array but **tuples are**. By **constraining** the set of possible inputs of this function, we make TypeScript **infer** a more **precise** type that keeps track of the length and the type of each index of this array.

That's about everything there is to know about type constraints. Let's get back to conditional types!

Nesting conditions

Just like ternaries, conditional types can be nested to chain several branches of code together:

```
// Nested conditions
type GetColor<I> =
    I extends 0 ? "black"
  : I extends 1 ? "cyan"
  : I extends 2 ? "magenta"
  : "white";
```

Granted, nested ternaries are a little hard to read sometimes. The thing is, the type-level language doesn't provide us with many alternatives for code branching, so we have to roll with them.

That said, There is **an interesting trick** when branching on **unions** of **literals**. We can declare an **object type** with a key for each of our cases, and use our literal as a property **accessor**:

```
type GetColor<I extends 0 | 1 | 2 | 3> = {
  0: "black";
  1: "cyan";
  2: "magenta";
  3: "white";
}[I];
```

Note that it only works thanks to the **type constraint** on `I`. Without it, we would get the following error:

```
Type 'I' cannot be used to index type '{ 0: "black"; 1: "cyan"; 2: "magenta";
3: "white"; }'. (2536)
```

TypeScript doesn't let us do `T[I]` unless it knows that `I` is assignable to `keyof T`. Type constraints are already proving themselves useful!

Pattern Matching with Conditional Types

So far, we have only used `extends` with literal and primitive types, but we can also use it to check if a type is assignable to a **data structure**!

```
type IsUser<T> =
  T extends { name: string; age: number }
    ? true
    : false;

type T1 = IsUser<{ name: "Gabriel" }> // ⇒ false
type T2 = IsUser<{ name: "Alice", age: 32 }> // ⇒ true
```

If you are familiar with functional programming languages, you might recognize this as Pattern Matching. Here, we essentially check if `T` matches the pattern `{name:string, age:number}`.

Pattern matching is powerful because it's **recursive**. We can **simultaneously** check several properties of our types, at **any depth** of nesting:

```
type IsUser<T> = T extends {
  name: string;
  team: { memberCount: number };
}
  ? true
  : false;

type T1 = IsUser<{
  name: "Gabriel";
  team: {
    memberCount: 12;
    name: "dataviz"; // ← extra prop
  };
}>; // ⇒ true

type T2 = IsUser<{ name: "Alice" }>; // ⇒ false
```

Using pattern matching, we can also branch on **several type parameters at once**. We only need to wrap them in a data structure first, like a **Tuple**:

```
type Plan = "basic" | "pro" | "premium";
type Role = "viewer" | "editor" | "admin";

// branching on several types by wrapping
// them in a tuple:
type CanEdit<P extends Plan, R extends Role> =
  [P, R] extends ["pro" | "premium", "editor" | "admin"]
  ? true
  : false;

type T1 = CanEdit<"basic", "editor">; // ⇒ false
type T2 = CanEdit<"premium", "viewer">; // ⇒ false
type T3 = CanEdit<"pro", "editor">; // ⇒ true
type T4 = CanEdit<"premium", "admin">; // ⇒ true
```

Since `extends` checks if the type on the left is **assignable** to the type on the right, we can use it with any type, **however complex**. Neat!

▼🤔 **Is there any real-world application for `CanEdit`?**

You might be wondering how to use a conditional type like `CanEdit` in the real-world, given that the `plan` or `role` of your user objects are most likely unknown at compile time. The answer is to start by **narrowing down** the type of your user objects!

Let's say we are building a note-taking app. We have a `User` type that looks like this:

```
type User = {
  plan: Plan;
  role: Role;
  // ... some other props
};
```

We also have "notes" that can be either read-only or editable:

```ts
type ReadonlyNote = {
  content: string;
};

type EditableNote = {
  content: string;
  edit: (newContent: string) => void;
};
```

Only users who `CanEdit` should have access to the `edit` method, and to express this constraint, we wrote a `getNote` function with the following signature:

```ts
declare function getNote<U extends User>(
  user: U,
): CanEdit<U["plan"], U["role"]> extends true
    ? EditableNote
    : ReadonlyNote;
/*     👆
   If CanEdit returns true, we return an EditableNote,
   otherwise a ReadonlyNote.                          */
```

When we get the current user from our database, it initially is typed as `User`, which means we don't know if they should be able to edit or not. When we pass the object to `getNote`, we get a `ReadonlyNote`:

```ts
const currentUser: User = getCurrentUser();
const note = getNote(currentUser);
// => ReadonlyNote
```

We can't call `edit` on it:

```ts
note.edit("new content");
//   ^ ❌ Property 'edit' does not exist on type 'ReadonlyNote'.
```

That's good! We can't mistakenly edit a note if we don't make sure our user is allowed to first.

Lets add a few type guards to narrow the type of our `currentUser` down:

```ts
function isAdmin(user: User): user is { role: "admin" } {
  return user.role === "admin";
}

function isPremium(user: User): user is { plan: "premium" } {
  return user.plan === "premium";
}
```

When we use these type guards in a `if` block, TypeScript will **narrow down** the type of our `currentUser`:

```ts
if (isAdmin(currentUser) && isPremium(currentUser)) {
  currentUser;
  // ^? User & { plan: "premium" } & { role: "admin" }
}
```

We now know that our user is an admin and has a premium plan! This means calling `getNote` returns an `EditableNote`:

```ts
if (isAdmin(currentUser) && isPremium(currentUser)) {
  const res1 = getNote(currentUser); // => EditableNote

  // We can edit the note!
  res1.edit("hello");
}
```

In summary, combining type guards and conditional types is a powerful way to **enforce business rules** in your types, and make sure your code is working as intended before it even starts running!

The `infer` keyword

The `infer` keyword is the real **superpower** of Conditional Types. It enables us to **declare a type variable** by **destructuring** the type on the left-hand side of `extends`:

```
type GetRole<User> =
  User extends { name: string; role: infer Role }
    ? Role //                          ^ new! 🤔
    : never;

type T1 = GetRole<{ name: "Gabriel"; role: "admin" }>;
// ⇒ 'admin'

type T2 = GetRole<{ role: "user" }>;
// ⇒ `never` because the input type doesn't have a `name` property.
```

Here, the function `GetRole` checks if `User` is assignable to `{ name: string; role: any }`. If it is, we declare a variable `Role` that contains the type of the `role` property and we return it. If `User` isn't assignable to this pattern, we return the `never` type.

Just like with the `let` and `const` keywords, you can give **any name** to your variable. You can only use it in the truthy code branch:

```
type GetRole<User> =
  User extends { name: string; role: infer lol }
    /*                                  👆
                              We can give it any name!     */
    ? lol // ✅ we can use `lol` here
    : lol // ❌ but not here.
```

I always felt like `infer` was similar in spirit to JavaScript's Destructuring Assignments:

```
type GetRole<User> =
  User extends { role: infer Role } ? Role : never;

// `GetRole` is the type-level equivalent of:
const getRole = ({ role }) ⇒ role;
```

But since `infer` can **only** be used on the **right-hand side** of `extends`, it makes Conditional Types even more similar to Pattern Matching:

```
type Fn<A> = A extends { a: { deeply: { nested: { prop: infer P } } } }
  ? P
  : never; // We have to handle the "else" case too!
```

Notice that we don't even need to add a type constraint on our type parameter to be able to pattern-match on it. It's ok because we have to handle the falsy case anyway.

Conditional types are super powerful. I wish we had value-level syntax for code branching that was as flexible as this![1]

`infer` with Tuples

`infer` can be used inside any kind of type-level data structure, including tuples!

You can use it to retrieve the **first element** of the list:

```
type Head<Tuple> = Tuple extends [infer First, ...any] ? First : never;

// `Head` is the type-level equivalent of:
const head = ([first]) ⇒ first;

type T1 = Head<["alpha", "beta", "gamma"]>; // ⇒ "alpha"
type T2 = Head<[]>; // ⇒ never
```

Or the **rest** of the list:

```
type Tail<Tuple> = Tuple extends [any, ...infer Rest] ? Rest : [];

// `Tail` is the type-level equivalent of:
const tail = ([first, ...rest]) => rest;

type T1 = Tail<["alpha", "beta", "gamma"]>; // => ["beta", "gamma"];
type T2 = Tail<["alpha"]>; // => []
type T3 = Tail<[]>; // => []
```

You're also allowed to use `infer` **several times** in a pattern:

```
type FirstAndLast<Tuple> = Tuple extends [infer First, ...any[], infer Last]
  ? [First, Last]
  : [];

type T1 = FirstAndLast<[1]>; // => []
type T2 = FirstAndLast<[1, 2]>; // => [1, 2]
type T3 = FirstAndLast<[1, 2, 3]>; // => [1, 3]
type T4 = FirstAndLast<[1, 2, 3, 4]>; // => [1, 4]
```

`infer` with function types

Even though they are so common in our day-to-day code, we haven't talked too much about function types so far, so let's start with a simple example:

```
type IsEqual = (a: number, b: number) => boolean;
```

Nothing surprising yet, but **here is the catch**: at the type level, **function types are data structures too**!

They are essentially wrappers around a Tuple representing the list of arguments bundled with an arbitrary return type:

```
// this type
type IsEqual = (a: number, b: number) => boolean;
// contains the same type-level information as this one
type IsEqual = {
  inputs: [a: number, b: number];
  output: boolean;
};
```

With that in mind, it makes sense that we can use `infer` inside function types too!

For example, we can retrieve the **list of parameters** as a **tuple** using `infer`:

```
type Parameters<F> = F extends (...params: infer P) => any ? P : never;

type Fn = (name: string, id: number) => boolean;

type T1 = Parameters<Fn>; // => [name: string, id: number]
```

And here is how to retrieve the return type of any function:

```
type ReturnType<F> = F extends (...params: any[]) => infer R ? R : never;

type Fn = (name: string, id: number) => boolean;

type T1 = ReturnType<Fn>; // => boolean
```

`Parameters<F>` and `ReturnType<F>` are actually built into TypeScript's standard library. Now you know how they are implemented!

`infer` with custom generics

the `infer` keyword can also be used in place of a generic type parameter. For example, you can extract the type of values contained in a `Set` with it:

```ts
type SetValue<S> = S extends Set<infer V> ? V : never;

type T1 = SetValue<Set<number>>; // ⇒ number
type T2 = SetValue<Set<string>>; // ⇒ string
```

This is cool, but what's even cooler is that you can do the same thing with your very own generic types:

```ts
type MyOwnGeneric<A, B> = { content: A; children: B[] };

type ExtractParams<S> =
  S extends MyOwnGeneric<infer A, infer B> ? [A, B] : never;

type T1 = ExtractParams<MyOwnGeneric<number, boolean>>;
// ⇒ [number, boolean]
type T2 = ExtractParams<MyOwnGeneric<string[], unknown>>;
// ⇒ [string[], unknown]
```

`infer` works with any type you can make up!

Variable assignment

Before we wrap up this chapter, I want to show you one more thing: how to assign a **local variable** in a type-level **expression**!

There is no standard syntax for declaring block-scoped variables at the type level. This is a shame because we know how much more readable the code can be by declaring intermediary variables with sensible names. It can also help performance — we can store the result of an expensive computation to reuse it multiple times later without having to recompute it every time.

But as often with TypeScript, there is a trick:

```ts
type Fn<I> = SuperHeavyComputation<I> extends infer Result
  //                                   ^  🤓  ^
  ? [Result, Result, Result]
  : never; // this branch can never be reached
```

That's right! We can use `infer` directly after `extends`. This will have the effect of assigning the result of `SuperHeavyComputation<I>` to a variable. The downside is that we need to declare an "else" code branch even though we know it will never be reached because any type is assignable to `infer Result`.

Summary 📝

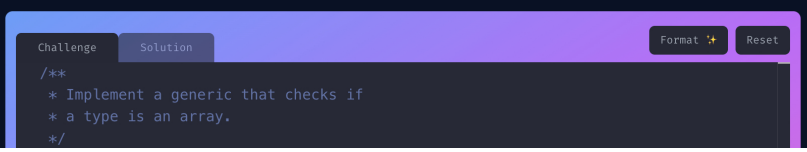
In this chapter, we've learned about the most fundamental building block of any algorithm — **code branching**. We covered a lot of notions, so let me summarize what we saw:

- At the type level, we use **Conditional Types** for code branching.
- Conditional types **must be** of the form `A extends B ? T : F`.
- `A extends B` means *"A is assignable to B"*.
- We can also use `extends` to declare a type constraint on a type parameter.
- The `infer` keyword lets us **assign a variable** by **extracting** a piece from a data-structure type.
- `infer` can only be used within a conditional type, on the right-hand side of `extends`.
- Even though conditional types look like value-level ternaries, they are much more powerful because they let us **pattern-match** on types.

Awesome. Now let's solve a few challenges!

Challenges 🎉

As always, you can refer to How Challenges Work if you are not sure how to take these challenges.



```ts
namespace isArray {
  type IsArray<T> = TODO

  type res1 = IsArray<number[]>;
  type test1 = Expect<Equal<res1, true>>;

  type res2 = IsArray<["a", "b", "c"]>;
  type test2 = Expect<Equal<res2, true>>;

  type res3 = IsArray<"Not an array">;
  type test3 = Expect<Equal<res3, false>>;

  type res4 = IsArray<string | null | undefined>;
  type test4 = Expect<Equal<res4, false>>;
}
```

Challenge Solution Format ✨ Reset

```ts
/**
 * Implement a generic returning:
 *  - the 2nd parameter if the 1st one is `true`
 *  - the 3rd parameter if the 1st one is `false`
 */
namespace ifElse {
  type If<Condition, Branch1, Branch2> = TODO

  type res1 = If<true, string, number>;
  type test1 = Expect<Equal<res1, string>>;

  type res2 = If<false, string, number>;
  type test2 = Expect<Equal<res2, number>>;

  type res3 = If<boolean, string, number>;
  type test3 = Expect<Equal<res3, string | number>>;
  /**                              ^ 😎
   * Don't worry if this doesn't fully make sense yet.
   * We will cover why this works in the advanced union
   * types chapter!
```

Challenge Solution Format ✨ Reset

```ts
/**
 * Type the `getWithDefault` function. It takes a key, an object
 * and a default value.
 *  - If the key exists on the object, it returns the corresponding
 *    value with the right type.
 *  - Otherwise, it returns the default value.
 *
 * Note: The type of the property and the type of the default value
 * can be different.
 */
namespace getWithDefault {
  type GetWithDefault<Key, Obj, Default> = TODO

  function getWithDefault<K extends string, O extends {}, D>(
    key: K,
    obj: O,
    defaultValue: D
  ): GetWithDefault<K, O, D> {
    return (obj as any)[key] ?? defaultValue;
  }
```

Challenge Solution Format ✨ Reset

```ts
/**
 * implement a generic to extract the type of the
 * `name` property from an object type.
 *
 * Note: This generic must also accept objects that
 *       Don't have a name property, and return
 *       `undefined` in this case.
 */
namespace getName {
  type GetName<Input> = TODO

  type res1 = GetName<{ name: "Gabriel" }>;
  type test1 = Expect<Equal<res1, "Gabriel">>;

  type res2 = GetName<{ name: string; age: number }>;
  type test2 = Expect<Equal<res2, string>>;

  type res3 = GetName<{ age: number }>;
  type test3 = Expect<Equal<res3, undefined>>;
```

Challenge Solution Format ✨ Reset

```ts
/**
 * implement a generic to extract the
 * type parameter of a Promise.
 */
namespace unwrapPromise {
  type UnwrapPromise<Input> = TODO

  type res1 = UnwrapPromise<Promise<"Hello">>;
  type test1 = Expect<Equal<res1, "Hello">>;

  type res2 = UnwrapPromise<Promise<{ name: string; age: number }>>;
  type test2 = Expect<Equal<res2, { name: string; age: number }>>;

  type res3 = UnwrapPromise<"NOT A PROMISE">;
  type test3 = Expect<Equal<res3, "NOT A PROMISE">>;
}
```

Challenge Solution Format ✨ Reset

```ts
/**
 * Implement a generic that drops the first
 * element of a tuple and returns all other
 * elements.
 */
namespace dropFirst {
  type DropFirst<Tuple extends any[]> = TODO

  type res1 = DropFirst<[1, 2, 3]>;
  type test1 = Expect<Equal<res1, [2, 3]>>;

  type res2 = DropFirst<[1]>;
  type test2 = Expect<Equal<res2, []>>;

  type res3 = DropFirst<[]>;
  type test3 = Expect<Equal<res3, []>>;
}
```

Challenge Solution Format ✨ Reset

```ts
/**
 * Implement a generic that extracts
 * the last element of a tuple.
 */
namespace last {
  type Last<Tuple extends any[]> = TODO

  type res1 = Last<[1, 2, 3]>;
  type test1 = Expect<Equal<res1, 3>>;

  type res2 = Last<[1]>;
  type test2 = Expect<Equal<res2, 1>>;

  type res3 = Last<[]>;
  type test3 = Expect<Equal<res3, never>>;
}
```

Challenge Solution Format ✨ Reset

```ts
/**
 * Implement the AND logical door:
 * AND<true, true> => true
 * AND<false, false> => false
 * AND<true, false> => false
 * AND<false, true> => false
 *
 * Hint: you can check several values at once by wrapping
 *       them in a tuple type (pattern matching).
 */
namespace and {
  type AND<A, B> = TODO

  type res1 = AND<true, true>;
  type test1 = Expect<Equal<res1, true>>;

  type res2 = AND<false, false>;
  type test2 = Expect<Equal<res2, false>>;

  type res3 = AND<true, false>;
```

Challenge Solution Format ✨ Reset

```ts
/**
 * The `flatten` function can take arrays of values,
 * Or arrays containing other arrays.
 * When given a nested array, `flatten` removes
 * one level of nesting. Make it generic!
 */
namespace flatten {

  type Flatten<Arr extends any[]> = TODO

  function flatten<A extends any[]>(arrayOfArrays: A): Flatten<A> {
    return arrayOfArrays.reduce(
      (acc, item) => [...acc, ...(Array.isArray(item) ? item : [item])],
      []
    );
  }

  // Zero levels of nesting
  let res1 = flatten(['a', 'b', 'c', 'd']);
  type test1 = Expect<Equal<typeof res1, string[]>>;
```

Footnotes

1. I wanted this feature so badly that I wrote an open source library implementing Pattern Matching for TypeScript. If you're interested in this topic, there's a TC-39 proposal to add syntax for pattern matching to JavaScript. It's promising, but still pretty far from being implemented. ↩

⟸ Previous
4. Arrays & Tuples FREE

Next ⟹
6. Loops with Recursive Types