7. Template Literal Types

The previous chapter about Recursive Types was a good demonstration of the **power** of TypeScript's type system. We started applying our programming knowledge to the language of types by creating **type-level algorithms**. This was only the beginning! We've just started exploring the extent of what TypeScript can offer, and what's to come in upcoming chapters is even more mindblowing 🤓

In this chapter, we will learn about **Template Literal Types**, an awesome feature that, to my knowledge, is **unique** to TypeScript's type system.

I'm sure you use **template literals** all the time to concatenate strings together at the value level:

```ts
const firstName = "Albert";
const lastName = "Einstein";
const name = `${firstName} ${lastName}`; // ← template literal
// ⇒ "Albert Einstein"
```

Well, **template literal *types*** let you do the same thing with types:

```ts
type FirstName = "Albert";
type LastName = "Einstein";

type Name = `${FirstName} ${LastName}`; // <- template literal ✨type✨!
//    ^? "Albert Einstein"
```

Easy, right?

More than just string interpolation

Despite their apparent simplicity, Template Literal Types open a **world of possibilities**. They let us build **fully-typed**, string-based **Domain Specific Languages** (DSLs) and enable some pretty cool meta-programming techniques.

A simple example is **inferring** the type of a DOM element based on a CSS selector:

```ts
const p = smartQuerySelector("p:first-child");
//    ^? HTMLParagraphElement | null  instead of `HTMLElement | null` 🙀
```

Or safely accessing a deeply nested object property using an "object path" string:

```ts
declare const obj: { some: { nested?: { property: number }[] } };

const n = get(obj, "some.nested[0].property");
//    ^? number | undefined  instead of `unknown` 🎉
```

💡 This code block is editable. Try updating `obj` to see type-checking in action!

By the end of this chapter, you should not only be able to type these functions but also your very own string-based DSLs!

Let's start with the basics

As their name suggests, Template literal *types* are the type-level equivalents of template literals. You can use **backticks** (`` ` ``) to create one, and interpolate other types inside of it using the `${ ... }` syntax:

```ts
type Hi = `Hello, ${"World"}!`; // "Hello, World!"
```

Just like at the value level, Template Literal Types let you interpolate **other things** than strings, like **numbers** or **booleans**:

```
type Index = 20;

type Accessor = `users[${Index}].isAdmin`;
// ⇒ "users[20].isAdmin"

type EqualsTrue = `${Accessor} === ${true}`;
// ⇒ "users[20].isAdmin === true"
```

That's something to keep in mind, as it's sometimes useful to **stringify** numbers to treat number literal types like `0` as equivalent to string literal types like `"0"`:

```
type Obj = { "0": 100 };
type Index = 0;

type Get<Obj, Key extends keyof Obj> = Obj[Key];

type A = Get<Obj, Index>; // ❌ `0` isn't assignable to `keyof Obj`
type B = Get<Obj, `${Index}`>; // ✅ this works!
```

Templates containing primitive types

You probably feel a little too comfortable right now, so here is a slightly stranger example:

```
type FirstName = "Gabriel";

type Gabriel = `${FirstName} ${string}`; // `Gabriel ${string}`
```

We are interpolating the type `string` in our template, but what does that mean?

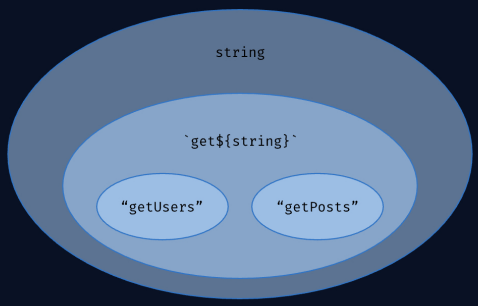
Notice that instead of returning a fully interpolated string, TypeScript returns the type `` `Gabriel ${string}` ``, **without trying to interpolate** the `string` type.

At the value level, we could have expected this to result in the infamous `"Gabriel [Object object]"` but our type-level language is much smarter. `` `Gabriel ${string}` `` is the **set** of all string literals that **start with** `"Gabriel "`:

```
const name1: Gabriel = "Gabriel Vergnaud"; // ✅
const name2: Gabriel = "Gabriel Fauré"; // ✅
const name3: Gabriel = "Gabriel "; // ✅ because "" is a string.
const name4: Gabriel = "Who's there?"; // ❌
const name5: Gabriel = "Hello Gabriel"; // ❌
```

If you are into regular expressions, you can think of `` `Gabriel ${string}` `` as the type of all strings matching the `/Gabriel .*/` expression.

Template Literal Types sit right in between primitive types and string literals in our dear subtyping hierarchy:



Template literal types in the subtyping hierarchy.

But what's even cooler is that you can use them to create **patterns** containing `numbers` or `booleans` too.

```ts
type Age = `I was born in ${number}.`;

const age1: Age = "I was born in 1993."; // ✅
const age2: Age = "I was born in 3.141592."; // ✅
const age3: Age = "I was born in a galaxy far, far away ... "; // ❌
// "a galaxy far, far away ... " is not a number!
```

This can be handy to make sure our functions receive strings of the expected format:

```ts
declare function ping(localDomain: `localhost:${number}`): void;

ping("localhost:3000"); // ✅
ping("localhost:8080"); // ✅
ping("localhost:three-thousand"); // ❌
```

Pretty neat!

Almost **all** primitive types can be used in a template:

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

The only exception is the `symbol` type, which makes sense since symbols cannot be stringified in JavaScript either.

As you can see, Template Literal Types are **more than just a way to concatenate strings**. They behave more like **data structures** containing string fragments and primitive types.

Templates and union types

What if we dropped a union type in a template? Let's find out:

```ts
type Size = "sm" | "md" | "lg";

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

Each item of this union is interpolated separately, and we get back **the union of their results**!

Now, what if we try to interpolate **several union types** in the same template?

```ts
type Variant = "primary" | "secondary";
type Size = "sm" | "md" | "lg";

type ButtonStyle = `${Variant}-${Size}`;
// ⇒ | "primary-sm"   | "primary-md"   | "primary-lg"
//   | "secondary-sm" | "secondary-md" | "secondary-lg"
```

We get back the union of **all possible combinations** of our two input union types. This behavior may seem a bit arbitrary, but unions tend to **turn the code inside out** like this in a lot of different contexts. This is an expression of **the distributive nature of union types**. We'll explore this concept in depth in The Union Type Multiverse.

In any case, this behavior is extremely useful when we need to branch on all possible combinations of two union types in a single `switch` statement:

```ts
type X = "left" | "right";
type Y = "top" | "bottom";

function getArrow(x: X, y: Y) {
  const diagonal = `${x}-${y}` as const;
  /*                              👆
     `as const` tells TypeScript to infer this
      type as `${X}-${Y}` instead of `string`.
                                                 */
  switch (diagonal) {
    case "left-top": return "↖";
    case "left-bottom": return "↙";
    case "right-top": return "↗";
```

```
    // case "right-bottom": return "↘";

    default: return exhaustive(diagonal);
    /*                         ~~~~~~~~
       ❌ The "right-bottom" case isn't handled!
                                                */
  }
}

function exhaustive(arg: never): never {
  throw new Error('non exhaustive.');
}
```

Does the `exhaustive` function ring a bell? We've seen it before in Types Are Just Data. It takes a parameter of type `never`, which means it can only type-check in **unreachable** code branches. Pieces of the type-level puzzle start to come together!

▼ 🤔 **What happens if we generate a giant union type?**

When I mentioned that Template Literal Types were similar to **regular expressions**, you may have been tempted to try to use them for some crazy matching, like checking that a phone number string is well-formed:

```
type D = 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9;

type USPhoneNumber = `+1 ${D}${D}${D} ${D}${D}${D} ${D}${D}${D}${D}`;

type T1 = "+1 123 456 7890" extends USPhoneNumber ? true : false;
// ⇒ true?

type T2 = "+33 1 23 45 67 89" extends USPhoneNumber ? true : false;
// ⇒ false?
```

This sounds like a good idea. At least until you realize that we just generated a union type of `10^10` elements, 10 **billion** of them!

As you might expect, TypeScript isn't too happy about it. If we try to run this code, we get the following error:

```
Expression produces a union type that is too complex to represent. (2590)
```

The hard limit before TypeScript throws this error is **100 000 items** within a single union type. This is already a huge number, so be careful if you don't want the type-checking of your project to take ages!

Helper functions

TypeScript has four built-in functions to change the casing of string literal types. There's `Uppercase`, which turns every letter in the string into uppercase:

```
type T1 = Uppercase<"hello">;
//   ⇒   "HELLO"
```

`Lowercase` performs the opposite transformation and turns every letter into lowercase:

```
type T2 = Lowercase<"HELLO">;
//   ⇒   "hello"
```

`Capitalize` only puts the first letter of the string in uppercase:

```
type T3 = Capitalize<"hello, world!">;
//   ⇒   "Hello, world!"
```

And `Uncapitalize` puts the first letter of the string in lowercase:

```
type T4 = Uncapitalize<"HelloWorld">;
```

```
// ⇒ "helloWorld"
```

With different languages using **different naming conventions** for variables and object keys (looking at you, Python and JavaScript!) we often need to rename them in our API layer. These type-level helper functions are pretty useful for that, though insufficient if we want to automatically rename, say, `snake_case` names to `camelCase`.

Solving this problem will require building a **Recursive Type**! Luckily, we've become pretty good at this in the previous chapter! 💪

Templates and object properties

Computing object properties is another area where Template Literal Types really shine. Let's say we have an HTTP service that should define a `GET` and `POST` method **for all of our resources**. We can compute the **names** of all the necessary **fetch functions** pretty easily:

```
type Method = "GET" | "POST";

type Resource = "user" | "blogPost";

type PropName = `${Lowercase<Method>}${Capitalize<Resource>}`;
// ⇒ "getUser" | "getBlogPost" | "postUser" | "postBlogPost"
```

Here, we generate the combinations of our `Method`s and `Resource`s, and we use two of the helper functions we've just seen to format them as method names. To create an object type, the only thing left to do is to drop our `PropName` type in a `Record`:

```
type HTTPService = Record<PropName, Function>;
// ⇒ { getUser: Function; getBlogPost: Function; postUser: Function; ... }
```

▼🤔 **Using `HTTPService` as an upper-bound with the new `satisfies` keyword.**

The `HTTPService` type isn't super useful on its own, but using the new `satisfies` keyword that the TypeScript team introduced in version 4.9, we can tell the type-checker that our `httpService` implementation has to be assignable to `HTTPService` while letting it infer a more precise type for each method:

```
const httpService = {
  getUser: () ⇒ Promise.resolve({ name: "Gabriel" }),
  postUser: (user: User) ⇒ Promise.resolve(),
  postBlogPost: (blogbost: BlogPost) ⇒ Promise.resolve(),
} satisfies HTTPService;
/*           ~~~~~~~~~~~
                 ^
   ❌ Doesn't type-check because we forgot `getBlogPost`!
                                                        */

const user = await httpService.getUser();
// ⇒ `{ name: string }` 👌
```

This way, if we ever add a new resource to our `Resource` union type, we are 100% sure that we won't forget to update the `httpService` object accordingly.

Pattern matching on string literals

As often in Type-Level TypeScript, it's **Conditional Types** that reveal the full potential of Template Literal Types. Using the `infer` keyword, we can **pattern-match** on a string literal to **split it** in several parts.

For example, let's write a `GetNameTuple` function that takes a `Name` string and split it into a `FirstName` and a `LastName`:

```
type GetNameTuple<Name> =
  Name extends `${infer FirstName} ${infer LastName}`
    ? [FirstName, LastName] //    ^
    : never;      /*      notice the space 👌   */
```

The `${infer FirstName} ${infer LastName}` pattern uses a **space character** as a **separator** to assign the part of the string that's before it to a `FirstName` variable and the part of the string that's after it to a `LastName` variable. These two variables are then wrapped into a tuple and returned:

```
type T1 = GetNameTuple<"Hermione Granger">;
//   ⇒   ["Hermione", "Granger"]

type T2 = GetNameTuple<"Ron Weasley">;
//   ⇒   ["Ron", "Weasley"]
```

Our template literal patterns can get as complex as we need them to be. In fact, as with every data structure, `infer` can be used **any number of times** in a template:

```
type SplitDomain<Name> =
  Name extends `${infer Sub}.${infer Domain}.${infer Extension}`
    ? [Sub, Domain, Extension]
    : never;

type T1 = SplitDomain<"www.google.com">;
//   ⇒   ["www", "google", "com"]

type T2 = SplitDomain<"en.wikipedia.org">;
//   ⇒   ["en", "wikipedia", "org"]

type T3 = SplitDomain<"github.com">;
//   ⇒   never. The subdomain part is missing.
```

Each `infer` assignment is separated by a dot, which makes it pretty easy for TypeScript to *infer* which part of the string should be assigned to which variable.

Ambiguous patterns

In the two conditional type examples we have seen so far, our template matched the input string in a **single**, **non-ambiguous way**. This is all well and good, but what if there were **several ways** for our template literal to match the input string? For instance, how would TypeScript react if we were passing a string containing **multiple spaces** to our `GetNameTuple` generic?

```
type GetNameTuple<Name> =
  Name extends `${infer FirstName} ${infer LastName}`
    ? [FirstName, LastName]
    : never;

type T = GetNameTuple<"Albus Perceval Wulfric Brian Dumbledore">;
//   ⇒   ["Albus", "Perceval Wulfric Brian Dumbledore"]
```

The string is split at the **first** space character. It looks like the type checker traverses our string **from left to right**! The direction in which TypeScript goes is even more obvious when we **do not** include **any separator** in our pattern:

```
type SplitString<Input> =
  Input extends `${infer A}${infer B}`
    ? [A, B] //           ^ no separator 🤔
    : never;

type T1 = SplitString<"some string">;
//   ⇒   ["s", "ome string"]

type T2 = SplitString<"what">;
//   ⇒   ["w", "hat"]
```

The string is split after the **first character** in this case!

That seems arbitrary. Engineers behind TypeScript could have very well chosen to split the string in the middle, before the last character, or in any other possible way given this template!

Practically speaking though, this choice makes a ton of sense. Being able to get the first character and the end of a string is extremely useful in type-level algorithms where we need to loop through each character.

Doesn't this remind you of something?

```
type SplitString<Str> = Str extends `${infer First}${infer Rest}`
  ? [First, Rest]
```

```
  : never;

type T = SplitString<"Albus">;
// ⇒ ["A", "lbus"]
```

This looks pretty similar to the way we were splitting **tuple types** inside recursive loops in the previous chapter:

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

type T = SplitTuple<["A", "l", "b", "u", "s"]>;
// ⇒ ["A", ["l", "b", "u", "s"]]
```

This is no coincidence. We are going to put this to good use very soon 😉

Accessing the last chunk of a split string

Let's go back to our `GetNameTuple` generic. The last example didn't quite work as expected, did it?

```
type T = GetNameTuple<"Albus Perceval Wulfric Brian Dumbledore">;
//   ⇒   ["Albus", "Perceval Wulfric Brian Dumbledore"]
```

Since "Perceval", "Wulfric" and "Brian" are all middle names of Dumbledore, It would make more sense if our function returned `["Albus", "Dumbledore"]` instead! But since TypeScript always goes from left to right, is there a way to retrieve only **the last word** in a string?

If you are thinking about **recursion**, you are totally right!

Let's create a `GetLastWord` generic that returns the last word of a string literal:

```
type GetLastWord<Str> =
  // 1. Split the first word from the rest of the string:
  Str extends `${string} ${infer Rest}`
    // 2. Keep doing this until there are no more spaces:
    ? GetLastWord<Rest>
    // 3. Return the input string if there are no spaces:
    : Str;
```

`GetLastWord` tries to match the input string with the pattern `${string} ${infer Rest}`. If it succeeds, it means there *was* a space in it, and the `Rest` part *may also contain* some other space characters. That's why we need to pass it to `GetLastWord` again **until it no longer contains any spaces**. At this point, we can simply return the input string.

Now we can update our `GetNameTuple` generic to use `GetLastWord` under the hood:

```
type GetNameTuple<Name> =
  Name extends `${infer FirstName} ${infer Rest}`
    ? [FirstName, GetLastWord<Rest>]
    : never;

type T1 = GetNameTuple<"Albus Perceval Wulfric Brian Dumbledore">;
//   ⇒   ["Albus", "Dumbledore"] 🧙 ✨
```

And we finally get the result we expect!

We've just built our first string parser. This one is very simple, but we can push this idea further. What are other use cases for type-level string parsing?

Transforming strings with Recursive Types

Let's try building something else: A function that takes a **"Title Case"** string and turns it into **"snake_case"**:

```
type T1 = TitleToSnake<"Hello, Type Level TypeScript Member!">;
//   ⇒   "hello_type_level_typescript_member"
```

How would you approach this problem if you were building this function **at the *value* level**?

How would you approach this problem if you were building this function **at the value level**?

I personally would decompose it into smaller sub-problems. Here, I see three things we need to do:

1. Replace spaces with underscores.
2. Remove punctuation characters.
3. Turn the string to lowercase.

In **JavaScript**, I'd probably start with something like this:

```
const titleToSnake = (str) => {
  const snakeStr = spacesToUnderscores(str);
  const noPunctStr = removePunctuation(snakeStr);
  return noPunctStr.toLowercase();
};
```

And then implement missing functions. ***Composing*** several functions together is a powerful way of dividing the complexity of our code into smaller units, and we can apply this technique to the type level too:

```
type TitleToSnake<Str extends string> =
    Lowercase<RemovePunctuation<SpacesToUnderscores<Str>>>;
//   Step 3  ←    Step 2      ←      Step 1
```

1. Mapping spaces to underscores

First, let's implement this `SpacesToUnderscores` generic. One thing is for sure, we need to split our string using a space **separator**:

```
type SpacesToUnderscores<Str> =
    Str extends `${infer Start} ${infer Rest}`
      ? ...
      : ...
```

If `Str` is ***not*** assignable to `` `${infer Start} ${infer Rest}` ``, it means it **does not** contain spaces. In this case, we can return it *as is*:

```
type SpacesToUnderscores<Str> =
    Str extends `${infer Start} ${infer Rest}`
      ? ...
      : Str; // ← str doesn't contain spaces!
```

Now, it might be tempting to fill the remaining blank by concatenating `Start` and `Rest` with a `_` character and call the job done:

```
type SpacesToUnderscores<Str> =
    Str extends `${infer Start} ${infer Rest}`
      ? `${Start}_${Rest}` // ← 🤔
      : Str;
```

But this wouldn't quite work:

```
type T1 = SpacesToUnderscores<"so much space in the universe">;
//   ⇒   "so_much space in the universe"
//                ^     ^  ^   ^
//            What about those spaces?
```

Before we stitch it to `Start`, We have to transform `Rest` by *recursively* calling our `SpacesToUnderscores` function on it:

```
type SpacesToUnderscores<Str> =
    Str extends `${infer Start} ${infer Rest}`
      ? `${Start}_${SpacesToUnderscores<Rest>}` // ← recursion
      : Str;

type Yay = SpacesToUnderscores<"yay it works now">;
//    ⇒   "yay_it_works_now"
```

Awesome! We can cross the first item from our to-do list:

Awesome! We can cross the first item from our to-do list:

```
type TitleToSnake<Str extends string> =
    Lowercase<RemovePunctuation<SpacesToUnderscores<Str>>>;
//   Step 3  ←    Step 2      ←   Step1: ✅ Done
```

Let's switch to step 2!

2. Removing punctuation

Removing punctuation seems very similar to the problem we've just solved. Again, we need to **replace some characters by other characters**, but instead of replacing `" "` by `"_"`, we need to replace punctuation characters with the empty string `""`. The first question we need to answer is "What's a punctuation character?"

Let's use a union type to list the characters we care about:

```
type Punctuation = "." | "!" | "?" | ",";
```

Now, could we reuse the code of `SpacesToUnderscores` with **different separators** to split and join the string? Let's give it a try:

```
type RemovePunctuation<Str> =
  Str extends `${infer Start}${Punctuation}${infer Rest}`
    //                        ^
    //     We use the `Punctuation` union as a separator
    ? `${Start}${RemovePunctuation<Rest>}`
    //         ^ No separator here.
    : Str;
```

This looks sensible. Let's feed this function some input:

```
type T1 = RemovePunctuation<"Hello, world!">;
/*   ⇒ | "Hello, world"
       | "Hello"
       | "Hello, world world"
       | "Hello world"            */
```

Oh no 😱 This is not at all what we were expecting! Why do we get this weird-looking union instead of `"Hello world"`?

The issue comes from the fact that `Punctuation` is a **union type**. As we have seen earlier, union types turn Template Literal Types into unions of templates, so this line:

```
Str extends `${infer Start}${Punctuation}${infer Rest}` ? ... : ...
```

actually becomes:

```
Str extends ( | `${infer Start}.${infer Rest}`
              | `${infer Start}!${infer Rest}`
              | `${infer Start}?${infer Rest}`
              | `${infer Start},${infer Rest}` ) ? ... : ...
```

Because `"Hello, world!"` contains **two** different punctuation characters, `','` and `'!'`, TypeScript has to make a **choice** between:

- Using the pattern `` `${infer Start},${infer Rest}` ``, and assign `Start` to `"Hello"` and `Rest` to `" world!"`.
- Or using the pattern `` `${infer Start}!${infer Rest}` ``, and assign `Start` to `"Hello, world"` and `Rest` to `""`.

TypeScript decides not to decide. Instead, it evaluates *all possible code branches* and assigns *the union of all possible values* to our variables.

```
type SplitTest<Str> =
  Str extends ( | `${infer Start},${infer Rest}`
                | `${infer Start}!${infer Rest}` )
    ? [Start, Rest]
    : never;
```

```
    : never;

type T = SplitTest<"Hello, world!">
// ⇒ ["Hello" | "Hello, world", " world!" | ""]
```

This behavior leads to super **unpredictable** outputs! My advice is to **never** use template literals containing unions on the right-hand side of `extends`. If you want to split the string using a union, the simplest is to loop through **each character**, and check if they are assignable to your union **one by one**.

Let's re-implement our `RemovePunctuation` generic with this technique:

```
type RemovePunctuation<Str, Output extends string = ''> =
  // Split the string after the first char:
  Str extends `${infer First}${infer Rest}`
    ? First extends Punctuation
      // If `First` is a punctuation char, drop it:
      ? RemovePunctuation<Rest, Output>
      // Otherwise, keep it in the output:
      : RemovePunctuation<Rest, `${Output}${First}`>
    : Output;
```

Recognize this pattern? This is one of the "filter" loops we've seen in the previous chapter, except we applied it to a **template literal type** instead of a **tuple**!

Let's try this again to see if this works:

```
type T1 = RemovePunctuation<"Hello, world!">;
//   ⇒   "Hello world"
```

Perfect 👌

3. Assembling the pieces

We can cross the `RemovePunctuation` item from our to-do list, and since `Lowercase` is built into TypeScript, we can cross that one as well:

```
type TitleToSnake<Str extends string> =
    Lowercase<RemovePunctuation<SpacesToUnderscores<Str>>>;
//  Step 3 ✅   ←  Step 2 ✅    ←    Step1 ✅
```

Let's finally test our `TitleToSnake` function and enjoy the fruits of our labor:

```
type T1 = TitleToSnake<"Hello, Type Level TypeScript Member!">;
//   ⇒   "hello_type_level_typescript_member" 🙌

type T2 = TitleToSnake<"Do you enjoy this chapter?">;
//   ⇒   "do_you_enjoy_this_chapter" 💪
```

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

We are done! `TitleToSnake` behaves exactly as we wanted.

This was a heck of a ride, so let's recap what we've learned.

First, *function composition* is a very nice way to turn a complex problem into several simpler ones.

Second, splitting strings using a union type as a separator doesn't quite work as expected. There is sometimes no other way than looping over every single letter in the string.

Third, the different kinds of loops we have seen in the previous chapter (find, filter, map and reduce) can also be used with template literal types. In the end, strings are just lists of characters!

Summary 📚

In JavaScript, we use **strings** to encode a **wide range of information**. Our functions often assume that input strings have **a specific structure** that they can interpret. This structure can sometimes be relatively simple (e.g. an object path) but it can also get very complex (e.g. a SQL[1] or a GraphQL query).

Template Literal Types allow us to fully **unleash** this ability to give strings special meanings. They

Template Literal Types allow us to fully **embrace** this ability to give strings special meanings. They not only let us enforce that our input strings have the correct structure but also enable us to parse them to infer new types.

We've covered many important notions in this chapter, but here are the key takeaways:

1. Template Literal Types can not only be used to **concatenate** literal types together but also to **represent** sets of string literal types with a **specific structure**.
2. **Interpolating a union type** in a template turns it into a **union of templates**.
3. **Conditional types** let us **split** template literals into several parts.
4. **In ambiguous cases**, TypeScript will use the **first occurrence** of your separator to split the string.
5. **Recursive types** let us build type-level **parser** functions.

Now, let's move on to some challenges! 🚁

Challenges 🎉

Time to practice! These challenges get more and more complex. Give them a good try, but remember that it's ok to fail. It means you are learning new things! You can always check the solution if you need to, or ask questions on Discord 🙂

Let's go! 👇

Challenge | Solution — Format ✨ — Reset

```ts
/**
 * Type the HTTPHeaders object so that it has an `Authentication`
 * property that starts with `Bearer ` and ends with a JWT token.
 *
 * Note: JWT tokens contain 3 parts, separated by dots.
 * More info on https://jwt.io
 *
 * Hint: You shouldn't need a conditional type.
 */
namespace headers {
  type HTTPHeaders = {
    Authentication: TODO
  };

  const test1: HTTPHeaders = {
    // ✅ This is a correct authentication header:
    Authentication:
      "Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJtIjoiWW91J3JlIGEgbmVyZCA7KSJ9.gf
  };
```

Challenge | Solution — Format ✨ — Reset

```ts
/**
 * Implement a `IsYelling` generic that returns
 * true if the input string is in all caps and
 * false otherwise.
 *
 * You shouldn't need recursion to solve this one.
 */
namespace isYelling {
  type IsYelling<Str extends string> = TODO

  type res1 = IsYelling<"HELLO">;
  type test1 = Expect<Equal<res1, true>>;

  type res2 = IsYelling<"Hello">;
  type test2 = Expect<Equal<res2, false>>;

  type res3 = IsYelling<"I am Groot">;
  type test3 = Expect<Equal<res3, false>>;

  type res4 = IsYelling<"YEAAAH">;
```

Challenge | Solution — Format ✨ — Reset

```ts
/**
 * Implement a `StartsWith` generic that takes
 * 2 string literals, and returns true if the
 * first string starts with the second one.
 */
namespace startsWith {
  type StartsWith<Str, Start> = TODO

  type res1 = StartsWith<"getUsers", "get">;
  type test1 = Expect<Equal<res1, true>>;
```

```
  type res2 = StartsWith<"getArticles", "post">;
  type test2 = Expect<Equal<res2, false>>;

  type res3 = StartsWith<"Type-Level Programming!", "Type">;
  type test3 = Expect<Equal<res3, true>>;
}
```

Challenge Solution Format ✨ Reset

```
/**
 * Write a type-level function that transforms
 * snake_case strings into camelCase.
 */

namespace snakeToCamel {
  type SnakeToCamel<Str> = TODO

  // it should let strings with no underscore in them unchanged
  type res1 = SnakeToCamel<"hello">;
  type test1 = Expect<Equal<res1, "hello">>;

  // one underscore
  type res2 = SnakeToCamel<"hello_world">;
  type test2 = Expect<Equal<res2, "helloWorld">>;

  // many underscores
  type res3 = SnakeToCamel<"hello_type_level_type_script">;
  type test3 = Expect<Equal<res3, "helloTypeLevelTypeScript">>;
}
```

Challenge Solution Format ✨ Reset

```
/**
 * Re-implement `SpacesToUnderscores` by defining a `Split`
 * and a `Join` generic that behaves like the value-level
 * `.split(separator)` and `.join(separator)` methods.
 */
namespace splitAndJoin {
  type SpacesToUnderscores<Str> = Join<Split<Str, " ">, "_">;

  type Split<Str, Separator extends string> = TODO

  type Join<List, Separator extends string> = TODO

  type res1 = Split<"a.b.c", ".">;
  type test1 = Expect<Equal<res1, ["a", "b", "c"]>>;

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

  type res3 = SpacesToUnderscores<"hey">;
  type test3 = Expect<Equal<res3, "hey">>;
```

Bonus challenges 😨

These last few challenges are pretty hard! I'm sure you will manage to solve them 🙂 Good luck!

Challenge Solution Format ✨ Reset

```
/**
 * The `navigate` function takes a `Path` string that
 * can contain parameter names using the `users/:username`
 * syntax. Type it so that the `params` object provided
 * as the second argument always contains a key for each variable
 * in the `Path`.
 */
namespace parseUrlParams {

  declare function navigate<U extends string>(
    path: U,
    params: ParseUrlParams<U>
  ): void;

  type ParseUrlParams<Url> = TODO

  type res1 = ParseUrlParams<"user/:userId">;
  type test1 = Expect<Equal<res1, { userId: string }>>;

  type res2 = ParseUrlParams<"user/:userId/dashboard">;
```

Format ✨ Reset

Challenge Solution

```ts
/**
 * Type the `querySelector` function to return an HTMLElement of the
 * right type!
 *
 * Hint: TypeScript standard library exposes a `HTMLElementTagNameMap` type
 * containing all element types stored by their HTML tag name. Use it
 * to turn tag names into element types!
 */
namespace querySelector {
  type P = HTMLElementTagNameMap["p"];
  //   👆 For example, here is how to get the element type for "p".

  function querySelector<S extends string>(
    selector: S
  ): SelectorToElement<S> | null {
    return document.querySelector(selector) as any;
  }

  type SelectorToElement<Selector> = TODO
```

Challenge Solution Format ✨ Reset

```ts
/**
 * Type the "get" function to infer its return type
 * from the object's type and the "path" string.
 */
namespace smartGet {

  declare function get<T, S extends string>(
    obj: T,
    path: S
  ): TODO

  // several object keys
  declare const obj1: { a: { b: { c: string } } };
  const res1 = get(obj1, "a.b.c");
  type test1 = Expect<Equal<typeof res1, string>>;

  // objects and arrays
  declare const obj2: { author: { friends: [{ age: 29 }] } };
  const res2 = get(obj2, "author.friends[0].age");
  type test2 = Expect<Equal<typeof res2, 29>>;
```

Footnotes

1. Just to demonstrate how powerful Template Literal Types can be, I greatly encourage taking a look at the TS-SQL repository on GitHub. It's a full-fledged SQL database implemented purely with types! ↩

⟸ Previous
6. Loops with Recursive Types

Next ⟹
8. The Union Type Multiverse