Literal Type to Narrow Primitive Type

In this lesson, you will see how string and number literals are different from conventional strings and numbers.

WE'LL COVER THE FOLLOWING

Literal type

A literal type means that the value is an **exact**. A literal type means that the value is an exact value. For example, a string literal of “test” would be that the value of the variable can only be “test”.

A literal type can be made up of multiple types or values from primitive JavaScript types.

String literals

A string literal is a way to define a string that limits the potential values to be used. It’s used mostly with a union, which allows specifying more than one string value. Imagine that you allow several strings’ values but want to limit the choice to specific ones. You could use an enumeration, but a string may be more clear or compatible with existing libraries. For example, you may want to limit the value to “north,” “south,” “east,” “west.”

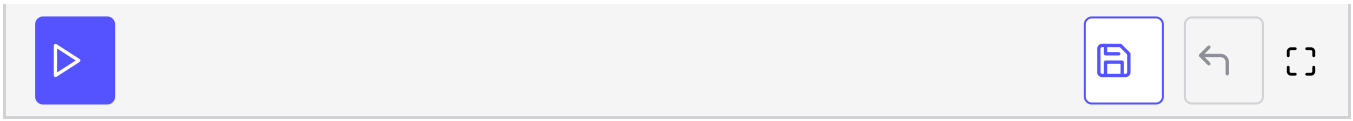
```
let direction: string = "no-where" // We desired to be "north", "south", "east", "west".
```

To create a string literal, define each value separated by the pipe symbol `|` (union of values). TypeScript will be smart enough to not compile if it goes

outside the defined range. The code below does not compile because

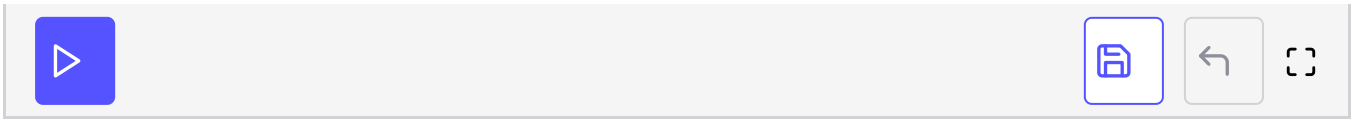
`yourDirection` is declared to be of the `Direction` string literal type. Changing it to `string` allows us to assign any string or change the value to one of the four defined types, and will fix the transpilation.

```
type Direction = "north" | "south" | "east" | "west";
let myDirection:Direction = "north";
// let yourDirection:Direction = "no-where"; // Does not compile
```



A string literal can be assigned without `type` by assigning a string value to with the `const` declaration or `as const`.

```
const stringLit1 = "oneValueOnly";
let stringLit2 = "oneValueOnly" as const
```



Number literals

Similarly, it is possible to use numbers as a set of values. Using multiple defined numbers is convenient if you have a set of values that you accept but that are not all numbers. For example, if you create a framework where you want to create a view grid system that works on a grid of 12 columns, you may restrict the choices from 1 to 12.

```
type Column = 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12;
let menuSize: Column = 4;
let mainContent: Column = 100; // Does not compile because only accept 1 to 12
```

Literal mixed type

It is possible to also create a mixed type with union that causes the literal to be of multiple types. In the following code, you have a line that does not compile because one of the literals is not covered, causing the `never` type to be assigned even though it cannot be compiled.

Un-commenting the `false` option would result in a compilable code since all values of the type are handled.

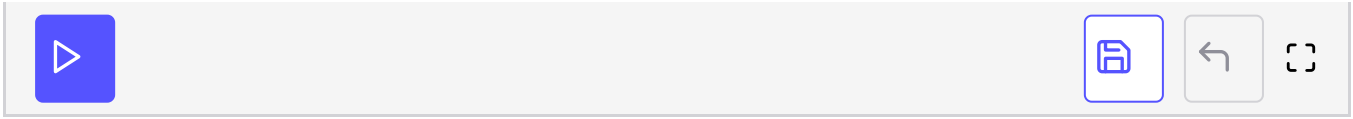
> 📜**Note:** The below code is expected to throw an error as explained ❌

```
type OptionOpen = true | false | "true" | "false"; // Actuall : boolean | "true" | "false"

function openWindow(option: OptionOpen): void {
   if(option === "true" || option === true){

   } else if (option === "false" /* || option === false*/){

   } else{
     const c: never = option;
   }
}
```



We will see later that with object-oriented *overload functions*, that string literal can become handy to distinguish between overload when only the value of the string changes and that literal type can be used to discriminate between objects.

In this lesson, you saw that you can circumscribe the potential string values of a variable to a smaller subset of strings. You also saw that it is possible to extend this concept to `number` with the number literal type.

Literal types are the perfect mix between good design and runtime validation since the condition is executed at runtime. They can be used to narrow type, verify that all possibilities are covered, select the proper overloaded methods, and handle cases where a type evolves (versioning).