#### Literal Type, Narrowing, and Const

#### WE'LL COVER THE FOLLOWING ^

- Literal type
  - A second example
- Literal type with const

## Literal type #

A literal type sets a single value to a variable's type. Initially, TypeScript supported only a string. Nowadays, a literal type can be a Boolean, a number, or an enum.

```
let x : "test";
let y : 123;
let z : true;
```

The concept of having a value that controls the data type flow can be extended beyond type checking in order to narrow down to a single type within a union. A return type can borrow the concept of narrowing by depending on the value's discriminant field. The result is a specific return type.

Imagine the scenario where a method can return a member from the Cat or the Dog interface. At line 11 the function takes one parameter that can be Cat or Dog. The function will return a name if the discriminant is a cat, see line 13. Otherwise, the function returns the nickname value, see line 15. A discriminant is powerful since the consumer can compare the return value and narrow it to a single type automatically. The narrowing affects the scope of the condition. The line 12 narrow p only between the curly bracket at line 12 and line 14.

```
interface Cat {
                                                                                         G
  kind: "cat", // Discriminant because shared with Dog
  name: string
}
interface Dog {
  kind: "dog", // Discriminant because shared with Dog
  nickname: string
}
function callMeBy(p: Cat | Dog): string {
  if (p.kind === "cat") { // In the IF, p is Cat
    return p.name;
  } else if (p.kind === "dog") { // In the IF, p is Dog
    return p.nickname
  return "unknown";
const c: Cat = { kind: "cat", name: "Hello Kitty" }
const d: Dog = { kind: "dog", nickname: "Snoopy" }
console.log(callMeBy(c));
console.log(callMeBy(d));
```







### A second example

Another example could be a response from an Ajax call which returns a success or a failure. In the case of a failure, the returned interface can have HTTP status, and error, and, when it is successful, the payload. Instead of having a single type that contains all these fields, you can return a type that unites a successful request and a failed request.

The consumer can then check the discriminant and leverage the automatic narrowed type to fulfill both scenarios. This will only contain the fields required, depending on the response.

```
interface Success {
                                                                                         C)
    success: true;
    httpCode: string;
    payload: string;
interface Failure {
    success: false;
    errorMessage: string;
}
function ajax(url: string): Success | Failure {
    return { success: false, errorMessage: "Error!" }; // Hardcoded failure
```

```
function ajaxCall(): string {
    const ajaxResult = ajax("http://blablac.com");
    if (ajaxResult.success === true) {
        return ajaxResult.payload; // Access to all Success interface members
    } else {
        return ajaxResult.errorMessage; // Access to all Failure
    }
}
const result = ajaxCall();
```







[]

# Literal type with const #

A literal type declared as a constant without specifying a type is automatically set to the type of the associated string. An implicit declaration of a literal type will make the constant's type to the same as the value.

However, setting a string to a let variable instead of const doesn't set the type to the string value, but to the type of string. The explicit value is the only valid path with let to set a literal type. Another interesting characteristic is that if you create a let variable and you associate a constant literal type, the let variable will be a string and not the literal.

It opens the definition to a broader scope since <a>let</a> allows you to redefine the value at any time. Here are examples that illustrate the difference between <a>const</a> and <a>let</a> with literal type.

