

## Tip 17: Shorten Conditionals with Falsy Values

In this tip, you'll learn how to use falsy and truthy values to check for information from different types.

### We'll cover the following



- Conditionals
- Equivalency & identity
- Truthy and falsy values
- Using falsy and truthy values
  - Catches
- Mutation problems
  - Solutions

## Conditionals #

Can you remember the first line of code you ever wrote? I can't, but I wouldn't be surprised if it was some sort of *conditional*. Responding one way to some information and a different way to other information is about as basic as programming can get.

I still write a lot of conditionals every day, and I bet you do, too. Fortunately, JavaScript, along with many other languages, gives you many tools for *checking information* and *reassigning* or *standardizing information* very quickly with minimal code.

The secret to being able to check values quickly is to understand the subtle difference between the primitive values `true` and `false` (also called **Boolean** types) and the many *so-called* truthy and falsy values—*values that aren't identical to the Boolean values `true` or `false` but act like they are in most cases*.

## Equivalency & identity #

Give me a moment to review another concept: **equivalency** and **identity**—a *value* that is *identical* to another value for different reasons. This is a bit of a

that's equivalent if it's the same, but of a different type and is checked with `'=='`.

```
console.log(1 == '1');
```



**Identical** values, or values with *strict* equality, mean that they must be of the *same type*.

```
console.log(1 === '1');  
console.log(1 === 1);
```



Objects, including instances of arrays, are checked by their *reference* (remember reference from `Object.assign()`?).

The topic can get much deeper, but for now, we want to identify values that are equivalent to `false` or `true` but not identical.

Okay. Back to truthy and falsy values.

## Truthy and falsy values #

An **empty** string is equal to `false` (but not *identical*). In other words, it's *falsy*.

```
console.log('' == false)  
if ('') {  
  console.log('I am not false!')  
} else {  
  console.log('I am false :( !')  
}
```



Here's a quick list of values that are *falsy* courtesy of [Mozilla Developer Network](#):

- `false`
- `null`
- `0`

- `NaN` (not a number)

- `' '`

- `""`

The ones that are worth memorizing are `0`, `null`, and an **empty string**. Let's hope you can remember that false is a falsy value.

Notice a few things conspicuously absent? If you wondered about the absence of arrays and objects (not to mention the other collection types), good eye. *Arrays and objects, even empty arrays and objects*, are always *truthy*. So you'll have to find another way to check emptiness with either `{}.length` or `Object.keys({}).length`, which will give you either `0` or a nice truthy number.

## Using falsy and truthy values #

Okay, you may be wondering why you should care about falsy values and truthy values (whatever is not falsy is truthy, of course). They're important because you can shorten a lot of otherwise lengthy expressions.

```
const employee = {
  name: 'Eric',
  equipmentTraining: '',
}

if (!employee.equipmentTraining) {
  console.log('Not authorized to operate machinery');
}
```



You don't need the code to know anything about when they received their equipment training. The code doesn't need to know if it's a date or a certificate name. All that the code needs to know is that the value exists and there's something there.

## Catches #

There are a few catches. Here's where things get tricky. It can be easy to create a falsy value unintentionally. The most common problem occurs when you're testing existence in an array by checking the index of a value:

```
console.log(['a', 'b'].indexOf('a')) // 0 which is falsy
```



You already saw this problem when you explored `Array.includes()`, so it should sneak up on you less often than it might have before. A much more subtle problem arises when you look for *key-value* data that's not defined. If you try to pull a value from a key that's not defined, you'll get `undefined`, which may cause a problem if an *object or map* were to change elsewhere in the code.

## Mutation problems #

Let's change the object just a bit to make `equipmentTraining` a Boolean.

```
const employee = {
  name: 'Eric',
  equipmentTraining: true,
};

function listCerts(employee) {
  if (employee.equipmentTraining) {
    employee.certificates = ['Equipment'];
    // Mutation!
    delete employee.equipmentTraining;
  }
  // More code.
}

function checkAuthorization(){
  if (!employee.equipmentTraining) {
    return 'Not authorized to operate machinery';
  }
  return 'Hello, ${employee.name}'
}

listCerts(employee);
console.log(employee)
console.log(checkAuthorization());
```



What happened here? The function `listCerts()` mutated the object and removed the *key-value* data. In the next function, you tried to check a value on the object. On objects, if the key isn't defined, you don't get an *error*—you get `undefined` (the same is true for maps). This would be a puzzling bug because when you inspect the code, it looks like the employee has certifications and should pass the conditional. Once again, be very careful with mutations.

## Solutions #

How can you solve the problem? There are actually two answers. Can you guess both of them?

The first, and far superior, solution is to *not mutate the data*. Falsy statements are way too valuable to give up. If a function is mutating the data, change the function.

If for some reason, you're unable to do that, you can use a *strict equivalency* check to make sure the value is there and it's in the format you want. If you use strict equivalency, you can guard against a situation where someone sets

`employee.equipmentTraining` to `'Not Trained'`, which is truthy.

```
const employee = {
  name: 'Eric',
  equipmentTraining: true,
};

function checkAuthorization() {
  if (employee.equipmentTraining !== true) {
    return 'Not authorized to operate machinery';
  }
  return `Hello, ${employee.name}`
}

employee.equipmentTraining = 'Not Trained';
console.log(checkAuthorization());
```



More code, but that's okay. Things happen. You don't need to chain yourself to falsy values, but you should certainly understand them. They're about to play a big role.

1

What will be the output of the following code?

```
var val1 = "20";
var val2 = 20;

console.log(val1 == val2);
```

2



What will be the output of the following code?

```
console.log(0 === false);
```

3



What will be the output of the following code?

```
var a = "0";  
if(a == false){  
  console.log("Hello");  
}else{  
  console.log("Bye");  
}
```

[Retake Quiz](#)

---

In the next tip, you'll learn how to use falsy and truthy values to make quick data checks.