Tip 19: Maximize Efficiency with Short Circuiting

In this tip, you'll learn to reduce conditionals to the smallest possible expression with short circuiting.

We'll cover the following

Short circuiting

You've been simplifying conditional expressions a lot in the last few tips. But there's one more level of simplification you can use: *short circuiting*.

As the name implies, the goal of **short circuiting** is to *bypass information checks by placing the most relevant information first*.

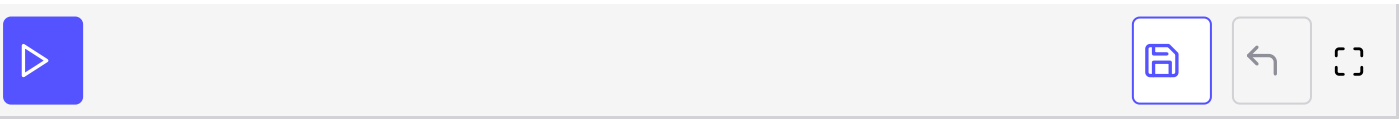
Example 1: Cleaning ternary code using short circuiting

Consider the following ternary, which would fit in well with the discussion from the previous chapter.

```
const image = {
    path: 'foo/bar.png',
};

function getIconPath(icon) {
    const path = icon.path ? icon.path : 'uploads/default.png';
    return `https://assets.foo.com/${path}`;
}

console.log(getIconPath(image));
```

The goal here is fairly clear. If an icon has a *truthy* path (in this case, that means it's defined and isn't an *empty string*), then you want to use the path. If it's *falsy, undefined, or* `''`, then you want to use the default.

```
const icon = {
    path: 'acme/bar.png'
}

function getIconPath(icon) {
    const path = icon.path ? icon.path : 'uploads/default.png';
    return `https://assets.foo.com/${path}`;
}

console.log(getIconPath(icon));
```



Did you see any clues that suggest you can clean up this code a bit?

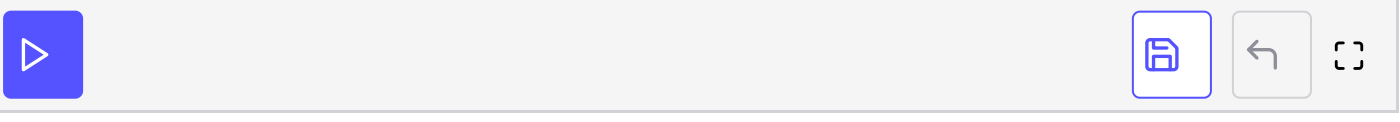
You probably noticed that you're writing the information check, `icon.path`, twice. Let's assume that data is always going to be valid, which means there's no difference between the information we're checking and the information we want. If it's truthy, we're going to use it.

Before updating the code, take a moment to think about how logical operators work. The or operator, symbolized as `||`, will return true if any of the possible values are true. That means that as soon as one thing—anything—returns true, you don't care what the other values might be.

Now here's where it gets exciting. Because you can use truthy values to test a **Boolean** expression, `true` or `false`, there's no incentive for the language to change the value from something truthy to true. So if one value in an `||` check returns true, you get that truthy value and not true.

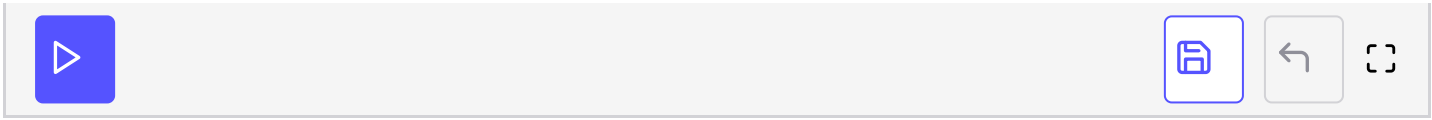
Lost? Don't worry. That's a long way of saying you can assign values directly from a Boolean check.

```
const name = 'joe' || 'I have no name';
console.log(name);
```



Now you have all of the tools you need to rewrite the ternary to something concise.

```
const image = {
    path: 'foo/bar.png',
};

function getIconPath(icon) {
    const path = icon.path || 'uploads/default.png';
    return `https://assets.foo.com/${path}`;
}

console.log(getIconPath(image));
```



As you may have noticed, the best part is that you can append a default value to the end of the expression. This means that you never have to worry about a variable being falsy because you know there's a truthy value waiting at the end.

There you have it. You can use short circuiting to bypass information once something truthy occurs. How about the other way around? How can you halt an expression once something false occurs? That's possible as well.

Another popular usage of short circuiting is to prevent errors, particularly when you plan to use a method or action on a particular collection.

Example 2: Preventing errors using short circuiting

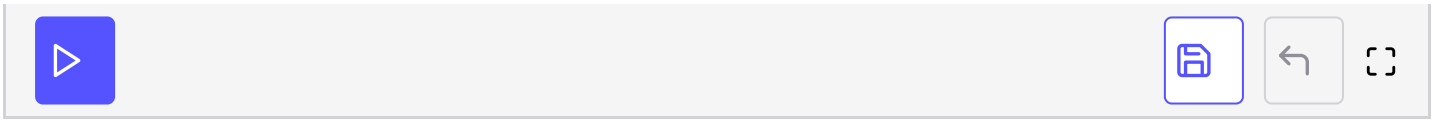
Consider a slight change to the problem of getting an icon. Instead of finding an icon set, you need to get a set of images from a user. The first image will be used as a thumbnail.

Because there are many images, the `images` collection will be an array. And your code needs to be able to handle the following representations:

```
// No array specified
const userConfig1 = {
}
// An array with no elements
const userConfig2 = {
    images: []
}
// An array with elements
const userConfig3 = {
    images: [
        'me.png',
        'work.png'
    ]
}
```

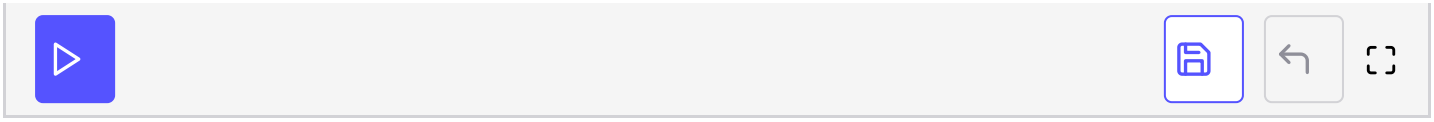
You may start off by thinking you could use short circuiting with the `||` operator to get the value you want. But that won't work for instances where the property isn't defined.

```
const userConfig1 = {
}
const img = userConfig1.images[0] || 'default.png';
//TypeError: Cannot read property '0' of undefined
```



The next step might be to use a series of nested conditionals.

```
const userConfig = {
}

function getFirstImage(userConfig) {
    let img = 'default.png';
    if (userConfig.images) {
        img = userConfig.images[0];
    }
    return img;
}

const img = getFirstImage(userConfig);
console.log(img);
```



At least in that example, you won't get an error if the `images` array isn't defined. But it will create a problem if there are no elements of the array.

```
const userConfig = {
    images: []
}

function getFirstImage(userConfig) {
    let img = 'default.png';
    if (userConfig.images) {
        img = userConfig.images[0];
    }
    return img;
}

const img = getFirstImage(userConfig);
console.log(img);
```

Now to solve that problem, you might add another nested conditional.

```
const userConfig = {
    images: []
}

function getImage(userConfig) {
    let img = 'default.png';
    if (userConfig.images) {
        if (userConfig.images.length) {
            img = userConfig.images[0];
        }
    }
    return img;
}

console.log(getImage(userConfig));
```

Things are already starting to get a little ugly and unreadable.

Fortunately, short circuiting can help. Combining conditionals with the `&&` operator will allow you to avoid the `TypeError` you saw earlier. A logical string built with an `&&` operator will cease as soon as a false value occurs. This means that you don't have worry about a `TypeError` when you try to call a method that doesn't exist. You can safely check for the existence of a collection and then call a method on it.

```
const userConfig = {
}

function getImage(userConfig) {
    if (userConfig.images && userConfig.images.length > 0) {
        return userConfig.images[0];
    }
    return 'default.png';
}

console.log(getImage(userConfig));
```

Now, this isn't perfect because you're just checking for a truthy value, which means that if there's bad data and images is set to a string, you'll get a weird result (the first letter of the string). But I'd leave it the way it is. At some point, you have

(the first letter of the string). But I'd leave it the way it is. At some point, you have

to have a little trust in your data or you need to find a way to normalize the data higher up the stream.

Combining short circuiting & ternary operators

Finally, you can combine your short circuiting back with a ternary to get this check down to a one liner. Start by pulling the `images` property into its own variable. Remember that if it's not there, the variable will merely be `undefined`.

```
const userConfig = {
    images: [
        'me.png',
        'work.png'
    ]
}

function getImage(userConfig) {
    const images = userConfig.images;
    return images && images.length ? images[0] : 'default.png';
}

console.log(getImage(userConfig));
```

Be careful when combining ternaries and short circuiting. Things can get out of hand very quickly. Say, for example, that you want to make sure the image didn't have a GIF extension. You'd still have to make sure there are elements in the array, or else you'd get another `TypeError` by checking for an index value on undefined. The resulting code is getting crazy.

```
const userConfig = {
    images: ['logo.gif'],
};

function getImage(userConfig) {
    const images = userConfig.images;
    return images &&
        images.length &&
        images[0].indexOf('gif') < 0
        ? images[0] : 'default.png';
}

console.log(getImage(userConfig));
```

You could refactor your code to check for an image. Or you could check the extension with a regular expression instead of an index. There are lots of ways around the problem. At some point, you need to make sure your conditionals are making code more clear and not just shorter for the sake of being short.

There's no explicit rule about how many conditionals are too many. It's more a matter of taste and team agreement. But when things get long (usually around three conditional checks), it's better to make it a standalone function.

Simplicity is great. And it's fun to try and find clever ways to reduce things to one line. But the goal is always communication and readability. Use short circuiting to make things readable—not to make code artificially small. Now that you can make simple conditionals, it's time to put that knowledge into action.

In the next chapter, you'll explore loops and how you can create simplified loops that avoid mutations, return predictable results, and can be as short as a single line of code.