Tip 28: Create Default Parameters

In this tip, you'll learn how to use default parameters to set values when a parameter is absent.

We'll cover the following

Function parameters

No matter how much planning you do, things change. This is especially true of function parameters. When you write a function, you expect a few parameters. Then, as code grows and edge cases emerge, suddenly the parameters you supplied aren't enough.

In the next several tips, you'll learn different techniques for handling parameters. Nearly all of these techniques can help you cope in some way with changing requirements. But to start, you'll learn the easiest trick: *setting default parameters*.

Handling parameters: Example

Consider a basic helper function. All you want to do is convert pounds to kilograms. That seems simple. You simply need to take the `weight` as an input and divide the `weight` in pounds by `2.2` to get kilograms. (*Apologies to non- Americans who don't have to deal with this silliness. I'm sure you also get stuck converting other measurements.*)

```
function convertWeight(weight) {
    return weight / 2.2;
}
console.log(convertWeight(44));
```

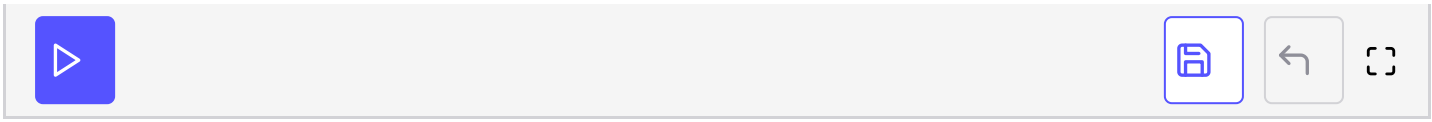
That code seems easy enough. And you use it throughout the app. Before you know it, a ticket comes in because someone needs to be able to pass *ounces*. And because there are *16 ounces* in a pound, you'll need to convert that number to a decimal before adding it to the pounds.

Fine. You add in a parameter for ounces, but now you're in a bind. Do you track down every instance of the function and add in a zero for the ounces? Or do you try to handle cases where a value wasn't provided?

You can try the first approach and update every function, but there's always the chance that you'll miss one. Fortunately, in JavaScript, you don't need to pass all the parameters to a function. They can be *optional*. If you're missing a parameter, the value is set to `undefined`.

Knowing that, you go for the second approach and add a little bit of code to set the value if it doesn't exist.

```
function convertWeight(weight, ounces) {
    const oz = ounces ? ounces / 16 : 0;
    const total = weight + oz;
    return total / 2.2;
}
console.log(convertWeight(44,11));
console.log(convertWeight(44,8));
```



When you run `convertWeight(44,11)`, you get `20.3125`, which isn't bad, but nearly every other conversion returns a long decimal string. `convertWeight(44, 8)` returns `20.22727...`.

Stranger still, when you run `convertWeight(6.6)`, you expect to get `3` and instead you get `2.999999...`. You can thank floating point arithmetic for that.

Great—now you need to round up to handle cases where the floating point arithmetic doesn't match user expectations. And because you're rounding anyway, you should make the number of decimal points an option, too, with a default of two decimal places.

You add some more code to handle the missing parameter. You also add in a helper

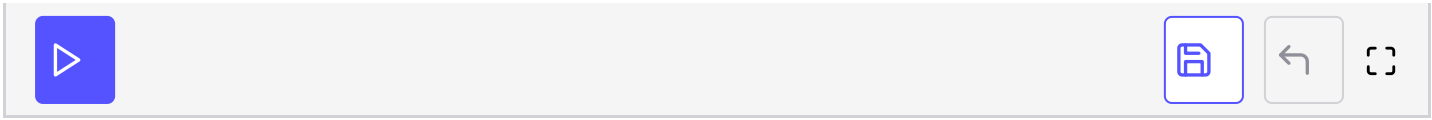
function, `roundTo`, to handle the rounding.

But there's a complication. To make the default two decimal places, you can't just check to see if the parameter `roundTo` is truthy. You can't, for example, write `const round = roundTo || 2;` because if the user were to pass in `0` as the number of decimal places they wanted, it would default to falsy and go back to two places.

Instead, you'd have to explicitly check that the value was `undefined`, which means that no value was submitted.

```
function roundToDecimalPlace(number, decimalPlaces) {
    const round = 10 ** decimalPlaces;
    return Math.round(number * round) / round;
}

function convertWeight(weight, ounces, roundTo) {
    const oz = ounces / 16 || 0;
    const total = weight + oz;
    const conversion = total / 2.2;
    const round = roundTo === undefined ? 2 : roundTo;
    return roundToDecimalPlace(conversion, round);
}
console.log(convertWeight(44, 8, 2));
```



Setting default parameters

Every time, the function becomes a little more complex. That's unavoidable in a world with changing requirements. What you don't want to do is create problems by having *undefined variables*. That means every time you add a new parameter, you end up adding a new ternary or short circuiting to create a default value.

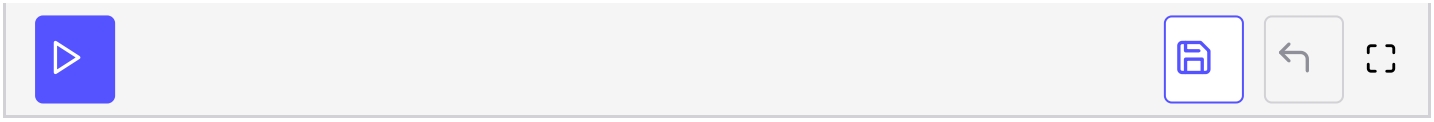
Changing requirements are part of life. There's nothing any syntax can do about that. But you can minimize a bunch of variable checks with **default parameters**.

All this means is that if the value isn't passed, it takes the placeholder value. It's that simple. You've probably seen it in countless other languages. You define the *default* parameter by putting an equal sign (`=`) after the parameter name along with the value. If there's no value for that parameter, it falls back to the default.

The updated function still has the additional logic to handle the new requirements (adding ounces, rounding decimals), but at least you can be confident you'll get something.

```
function roundToDecimalPlace(number, decimalPlaces) {
    const round = 10 ** decimalPlaces;
    return Math.round(number * round) / round;
}

function convertWeight(weight, ounces = 0, roundTo = 2) {
    const total = weight + (ounces / 16);
    const conversion = total / 2.2;
    return roundToDecimalPlace(conversion, roundTo);
}
console.log(convertWeight(44, undefined, 2));
console.log(convertWeight(44, 11, 0));
```



As a bonus, you give a clue to other developers that you're looking for a particular data type. They'd know, for example, that ounces is an integer. This isn't a substitute for a proper type system, but it's a nice little extra.

JavaScript and Type Checking

Default parameters aren't a perfect solution. Parameter order still matters. If you didn't want to include ounces but you did want to specify the number of decimal points, you would still need to clarify the number—in this case, it would be `0`.

```
convertWeight(4, 0, 2);
```

Setting `undefined` as the default

If you absolutely don't want to pass in a value, you can pass in `undefined` and the function would use the default parameter, but use this approach with caution. It's too easy to make mistakes when you pass in `undefined`. If you passed in `null`, for example, you wouldn't get the default value. Besides, if you really don't care what the default is, you should just use the value set as the default parameter. It's more clear to others reading the code and it's less likely to break later if the function changes slightly.

```
convertWeight(4, undefined, 2);
```

A common way around this problem is to pass an *object* as a *second* parameter. Because an object can have multiple key-value pairs, you won't need to change the

function parameters every time a new option is added. You will, however, need to pull the information from the object.

What will be the output of the following code?

```
function sumValues(val1, val2 = 10, val3 = 15) {
    return val1 + val2 + val3;
}
console.log(sumValues(20,undefined,3));
```

Retake Quiz

In the next tip, you'll see how it's easier to use objects in parameters by pulling out data with destructuring.