Tip 31: Passing Variable Number of Arguments with Rest Operator

In this tip, you'll learn to collect an unknown number of parameters with the rest operator.

We'll cover the following

- Handling an unknown number of parameters
- Communicating parameters
  - Using arguments
- Spreading an array into a list
- Rest parameters
  - Why use rest arguments?
  - Recreating array methods using rest

In the previous tips, you saw how object destructuring would let you combine several parameters into a single argument.

Handling an unknown number of parameters

Using objects to hold parameters is a great technique, but it's really only useful in situations where the parameters are different and you know them ahead of time. In other words, it only makes sense in situations with objects.

That may seem obvious, but it raises the question: *How do you handle an unknown number of similar parameters?*

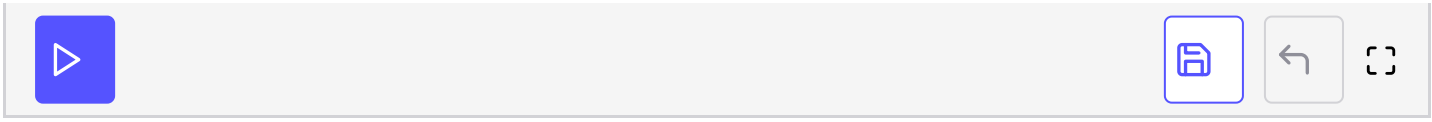
Think back to the photo display application. What if you wanted to allow your users to tag photos but you only wanted the tags to be a certain length? You could easily write a very short validation function that takes a size and an array of tags and returns *true* if all are valid.

```
function validateCharacterCount(max, items) {
    return items.every(item => item.length < max);
}
```

Notice the `every()` method? It's another simple array method you haven't seen before. As with `filter()`, you pass a *callback* that returns a *truthy* or *falsy* value. The `every()` method returns *true* if every item in an array passed to the callback returns truthy. Otherwise, it returns *false*.

Running the function is simple. Just pass in an array of strings.

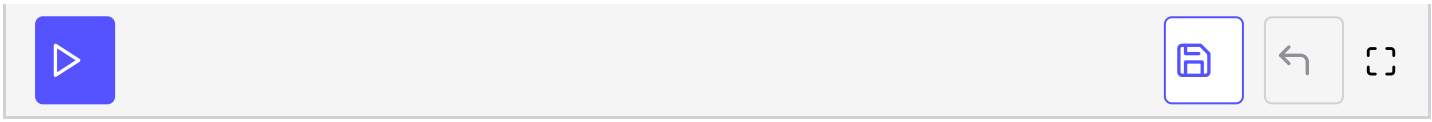
```
function validateCharacterCount(max, items) {
    return items.every(item => item.length < max);
}
console.log(validateCharacterCount(10, ['Hobbs', 'Eagles']));
```



Communicating parameters

The code above is great because it's so generic. You can easily reuse it elsewhere. The only down side to this code is that it locks the users of your function into a particular collection type. Another developer might, for example, want to test that a single username isn't too long. To use the code, they'd have to know that they'd need to pass an array. If they didn't, they would get an *error*.

```
function validateCharacterCount(max, items) {
    return items.every(item => item.length < max);
}
console.log(validateCharacterCount(10, 'wvoquine'));
// TypeError: items.every is not a function
```



Using `arguments`

You could write some documentation to communicate the parameters, but there's a better way. Previously, JavaScript developers solved this problem by using the *built-in* `arguments` object. This handy object gives you an array-like collection of all the arguments that are passed to a function.

```
function getArguments() {
    return arguments;
}
console.log(getArguments('Bloomsday', 'June 16'));
```

You may have noticed the phrase “array-like.” Unfortunately, `arguments` is an object, so you’ll need to do some converting to get it to an array. Specifically, you’ll need to statically call a method on the Array object (as opposed to an array instance) as you see in **line 2** in the code that follows. This line of code takes all arguments after the first one, the character count, and combines them into an array.

```
function validateCharacterCount(max) {
    const items = Array.prototype.slice.call(arguments, 1);
    return items.every(item => item.length < max);
}
console.log(validateCharacterCount(5, 'red', 'blue'));
```

Now you can pass as many arguments as you want knowing you’ll have an array inside the function.

What about situations where you already have an array? Because you are converting the arguments into an array, you’ll need to convert your array into a list of arguments.

Spreading an array into a list

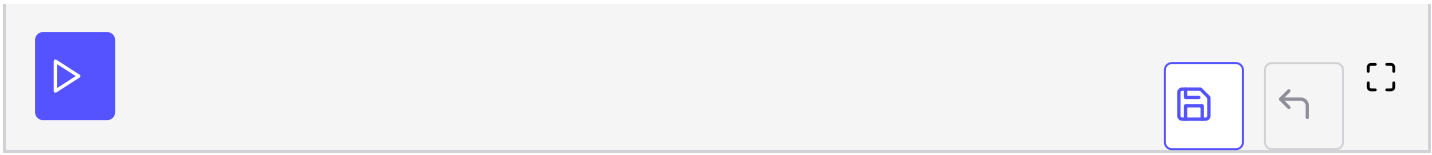
Think back to when you learned about the spread operator. At the time, you learned you always need to spread into something. Up until now, you’ve only spread it into another array. You can also spread it as a list of parameters. In other words, when you collect parameters into a list, you can easily handle cases of strings or arrays.

Using these techniques, you can now use your function with a variety of parameters.

```
function validateCharacterCount(max) {
    const items = Array.prototype.slice.call(arguments, 1);
    return items.every(item => item.length < max);
}

console.log(validateCharacterCount(10, 'wvoquie'));
const tags = ['Hobbs', 'Eagles'];
```

```
console.log(validateCharacterCount(10, ...tags));
```



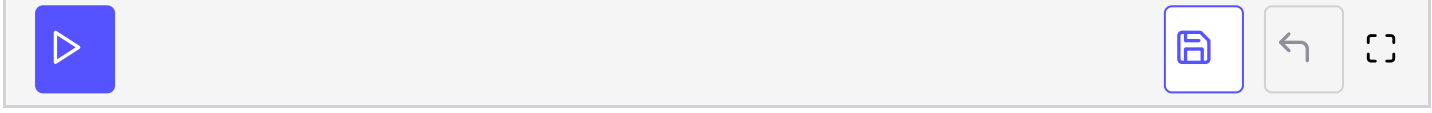
This is more flexible, but it's far from perfect. The biggest problem is that the syntax to work with the arguments object is a little convoluted. As a result, few developers (except the most hard-core JavaScript developers) used it. Not to mention that when you use the arguments object, there are absolutely no clues in the function parameters that you accept a list of arguments. Another developer would have to dig into the function body to understand what they can pass to the function.

Rest parameters

Enter rest parameters. **Rest parameters** enable you to pass a list of arguments and assigns them to a variable.

You declare rest operators using your favorite three dots (`...`) followed by the variable you'd like to assign them to. Any parameters passed beyond that point are collected into the variable as an *array*.

```
function getArguments(...args) {
    return args;
}
console.log(getArguments('Bloomsday', 'June 16'));
```



It's that simple. Try rewriting the `validateCharacterCount()` function using rest arguments.

It probably took you no time at all to come up with this:

```
function validateCharacterCount(max, ...items) {
    return items.every(item => item.length < max);
}
```

In addition to being simpler and cleaner, it is more predictable. Now a developer

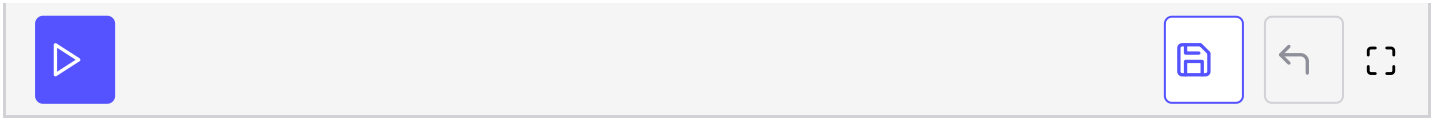
can tell that this function takes at minimum *two* arguments. Even those unfamiliar

with the rest operator have enough clues that a quick Stack Overflow search would fill in the details.

You'd call the function exactly the same way as you did with the previous function, by either passing a list of arguments or spreading an array of arguments into a list. This is no different from the previous code when you used the arguments object.

```
function validateCharacterCount(max, ...items) {
    return items.every(item => item.length < max);
}

console.log(validateCharacterCount(10, 'wvoquie'));
console.log(validateCharacterCount(10, ...['wvoquie']));
const tags = ['Hobbs', 'Eagles'];
console.log(validateCharacterCount(10, ...tags));
console.log(validateCharacterCount(10, 'Hobbs', 'Eagles'));
```



Why use rest arguments?

At this point, you've accounted for a situation where you might get either a list or an array. There are a few other reasons you might use rest arguments.

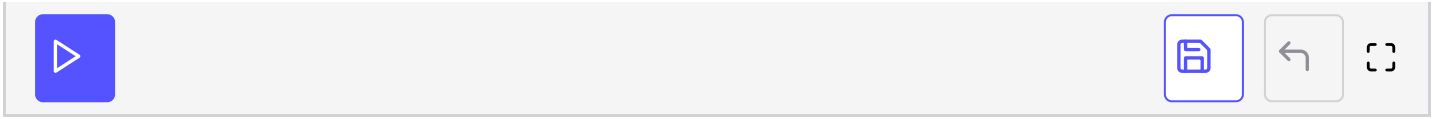
First, you want to signal to other developers that you'll be working with arguments as an array. In the absence of type checking, this is another little clue that will help future developers. A lot of developers will use the rest operator even though the data they're passing in will be in the form of an array. Even though they always spread in the information when calling the function, it's a clear marker of the expected parameter type.

Second, the rest operator can give you a nice way to debug code. For example, it can help you decode library functions that you suspect may be getting additional parameters, and you can use a rest argument to collect any lingering arguments.

You've worked with the `map()` method several times, and you know the callback function takes the item being checked as an argument. It turns out that the callback function passes a few more arguments after the individual items. If you collect the rest of the parameters and log them, you'll see the `map()` operator also takes the index of the item being checked and the full collection.

```
['Spirited Away', 'Princess Mononoke'].map((film, ...other) => {
    console.log(other);

    return film.toLowerCase();
});
```



This isn't a big deal on the `map()` operator, which is well-documented, but the rest operator can help you see parameters that you might not have otherwise known about. The rest operator is a great way to debug.

Third, rest arguments are a great way to pass props through functions if you have no plans to alter them. This is nice when you want to wrap a couple of functions and pass the arguments through. For example, you may have a modal, and when changes are saved, you'll want to close a modal while simultaneously updating some information with another function.

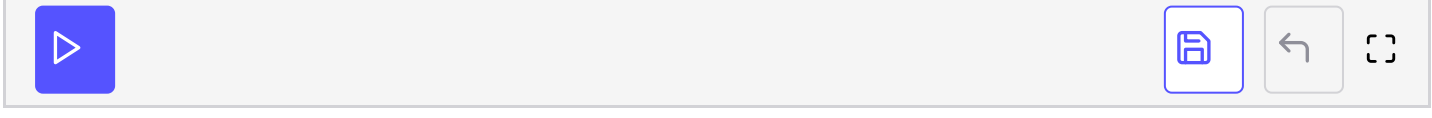
```
function applyChanges(...args) {
    updateAccount(...args);
    closeModal();
}
```

Finally, don't forget the rest operator isn't just for parameters. As you've seen, it works for pulling the remaining key-values from objects or the remaining values from arrays.

Recreating array methods using rest

Much like the spread operator, you can recreate a common array method while removing side effects. If you wanted to recreate the `shift()` method, which returns the first item of an array and removes that item from the original array, simply combine the rest operator and destructuring.

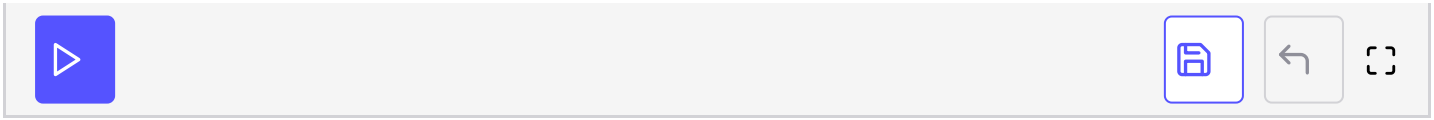
```
const queue = ['stop', 'collaborate', 'listen'];
const [first, ...remaining] = queue;
console.log(first);
console.log(remaining);
```



You get the first value and an array of the remaining values. As a bonus, the original array is still intact.

The only downside to using the rest operator as an argument is that it must be the last argument in all situations. It must be the last parameter for a function. It *must* be the *last* value when destructuring. This means that although you can recreate the `shift()` method—*return the first item*—you can't recreate the `pop()` method, which *returns the last item of an array*.

```
const queue = ['stop', 'collaborate', 'listen'];
const [...beginning, last] = queue;
// SyntaxError: Rest element must be last element
```



Still, the rest operator is very useful, and you'll find lots of opportunities to work it into your code.

What will be the output of the code below?

```
const func = (val1, val2, val3, ...args) => {
  const res1 = args.map(val => val*2);
  const res2 = res1.reduce((number, acc) => number += acc);
  const res3 =  val1 + val2 + val3 + res2;
  return res3;

}

console.log(func(1,2,3,4,5));
```

Retake Quiz

Look at how much more you can do with functions already. And you've only just begun. In the next chapter, you'll move beyond parameters and return statements and explore how to construct more powerful and flexible functions.