Advanced Function Parameter Syntax: Default / Rest / Spread

(https://github.com/learn-co-curriculum/phase-1-advanced-function-parameter-syntax-default-rest-spread) (https://github.com/learn-co-curriculum/phase-1-advanced-function-parameter-syntax-default-rest-spread/issues/new)

Learning Goals

- Set a default value for a function parameter
- Use JavaScript's `rest` parameter to define parameters in a function
- Use JavaScript's `spread` operator in a function call

Introduction

The typical way to handle function parameters for JavaScript functions is to define them individually and then pass a value for each as an argument when calling the function:

```
function customGreeting(greeting, name, adjective, occasion) {
  return `${greeting}, ${name}! Have a ${adjective} ${occasion}!`;
}

customGreeting("Good morning", "Pouja", "fantastic", "Tuesday");
// => "Good morning, Pouja! Have a fantastic Tuesday!"
```

This always works, and is perfectly adequate a lot of the time. However, for those times when our needs may be more complex, JavaScript provides several helpful tools that can make handling of parameters and arguments more flexible and efficient: default values, the `rest` parameter and the `spread` operator. We will learn how to use all three in this lesson.

Set a Default Value for a Function Parameter

Let's say you work for an e-commerce site, and you're prepping for your post-holiday sales. You're working on some code for your website and you need to set a discount of 25% across the board for everything that you sell on the website.

We can create a function that takes in an `itemPrice` as a price in dollars and returns the discounted price:

```
function discountedPrice(itemPrice){
    return itemPrice - (itemPrice * 0.25);
}
```

But it seems unwise to hard-code the discount to `0.25`. Management's whims on discount rates change almost daily, as the corporate sales office's machine-learning algorithms recommend new discount rates. Because of this, we want to encode the discount rate as a *parameter* of the function.

```
function discountedPrice(itemPrice, discount){
    return itemPrice - (itemPrice * discount);
}
```

So, calls to `discountedPrice()` will look like:

```
function discountedPrice(itemPrice, discount){
    return itemPrice - (itemPrice * discount);
}
discountedPrice(100, 0.25); //=> 75.0
```

But it *also* seems a bit of a burden to **have** to pass the discount amount on every call. We'd like `discount` to *default* to `0.25`. It'll be 25% off *unless* we choose to pass a new discount percentage into `discountedPrice()`. We set the default value by simply assigning the value in the parameter list:

```
function discountedPrice(itemPrice, discount = 0.25){
    return itemPrice - (itemPrice * discount);
}
discountedPrice(100); //=> 75.0
```

Then if we want to use a different discount, we simply pass the new value as the second argument — that value will *override* the default value:

```
discountedPrice(100, 0.5); //=> 50.0
```

Note that we place the parameter with a default value at the **end** of the parameter list. This is because the argument values are assigned to the parameters from left to right, regardless of whether a default value is specified for any of them. If we put the discount with its default value first and `itemPrice` second, when we call `discountedPrice(100)`, 100 will get assigned as the discount and the `itemPrice` will be undefined. The function would then return `NaN`:

```
function discountedPrice(discount = 0.25, itemPrice) {
  return itemPrice - (itemPrice * discount);
}

discountedPrice(100);
// => NaN
```

Now say we want to add a `tax` parameter to `discountedPrice()` so that we can include a tax percentage to be added to the discounted sales price. First we put our function declaration back the way it was, with `discount = .25` last. Then we add `tax` as the second parameter:

```
function discountedAndTaxedPrice(itemPrice, tax, discount = 0.25) {
  let subtotal = itemPrice - (itemPrice * discount);
  return subtotal + (subtotal * tax);
};

discountedAndTaxedPrice(100, 0.10); //=> 82.5
discountedAndTaxedPrice(100, 0.10, 0.20); //=> 88
```

Use JavaScript's `spread` Operator in a Function Call

In previous lessons, we've used JavaScript's `spread` operator to copy an array; it "spreads out" the elements of the original array into a new array. We can use `spread` in the exact same way to pass elements of an array into a function as an argument! Try it out in console with a

simple add function:

```
function add(a, b, c) {
  return a + b + c ;
}
const arr = [1, 2, 3];

add(...arr); // returns 6
```

Just as it does when we use `spread` to copy an array, the JavaScript engine sees the `spread` operator and knows to "spread out" the elements in the array into the parentheses. In other words, it converts this: `add(...arr)` into this: `add(1, 2, 3)` .

Play around with it using a bigger array and see what happens when the array has more numbers than our function has parameters.

Use JavaScript's `rest` Parameter to Define Parameters in a Function

We've learned how to use the `spread` operator in our function *calls*, but we can also use the `...` syntax in our function's parameter list. In this context, it's called the `rest` parameter because it allows us to capture the `rest` of the arguments that are passed into the function and store them in an array.

Sometimes, we might not know exactly how many arguments we want to pass into a function, but we might know that we only want to do something with the first two arguments. In JavaScript, it's possible to pass in any number of arguments into a function when we call it, regardless of how many parameters are defined:

```
function muppetLab(a,b){
  console.log(a,b); // LOG: Dr. Bunson Beaker
}

muppetLab("Dr. Bunson", "Beaker", "Miss Piggy", "Kermit", "Animal");
```

Here we have two parameters defined, so the first two arguments are stored into those variables. But what happens if we want to capture the left over arguments? The `rest` parameter allows us to take the rest of the arguments that we pass in to the function, regardless of how many there are, and gather them into an array. Here's how this works:

```
function muppetLab(a, b, ...muppets) {
  console.log(a, ' ', b); // LOG: Dr. Bunson Beaker

  console.log(muppets); // LOG: ["Miss Piggy", "Kermit", "Animal"]
  console.log(muppets[0]); // LOG: Miss Piggy
  console.log(muppets.length); // LOG: 3
}

muppetLab("Dr. Bunson", "Beaker", "Miss Piggy", "Kermit", "Animal");
```

The first two argument values are stored in `a` and `b`, respectively, and the remaining values are stored in the `muppets` array. If we call `muppetLab()` and only pass two arguments, the value of `muppets` will be an empty array.

Since the `rest` parameter gathers the "rest" of the arguments given to a function, it should always come at the end of the list of parameters.

Conclusion

In this lesson, we've learned about three tools we can use with arguments and parameters: `default`, `spread`, and `rest`. We use `default` any time we have a value we want to use by default for a parameter but still want to be able to override that value easily. We can use the `spread` operator to efficiently pass values in an array as arguments to a function. Finally, we use the `rest` parameter when we want to capture arguments in an array. This can be particularly helpful when we aren't sure how many arguments will be passed in but still want access to all of them inside our function.

Given that the syntax for the `spread` operator and the `rest` parameter is the same, how do we know which is being used? It's all about context. If the three dots occur when you are *calling* the function, then it's the `spread` operator. If they happen when you're *defining* the function, it's the `rest` parameter.

Resources

- **Default parameters** (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/Default_parameters)
- **Rest parameters** (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/rest_parameters)
- **Spread operators** (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Spread_syntax)