**Arguments in Javascript**

Arguments in JS behave differently than they do in other languages. Namely, JavaScript functions will happily take fewer arguments than specified (in which case the unspecified arguments have value undefined), or extra arguments (they will be available in a special arguments array-like object).

**Fewer Arguments**

JS functions can take fewer arguments than expected. In that case, unspecified arguments have the value undefined.

function foo(arg) {

return arg;

}

foo(5); // => 5

foo(); // => undefined

Occasionally this can be annoying to debug if you expect a function to throw an error when it doesn't receive as many arguments as it requires to return the correct output. Always keep in mind that a function will still run even if it has been passed no arguments at all.

**More Arguments**

JS functions will also accept more arguments than are asked for. You have access to all of the arguments through a special array called arguments. arguments is set each time you call a function. It contains the values of all the arguments: ones that were anticipated in the function definition, plus the extras.

function logArguments(arg1, arg2) {

let result = [];

for (let i = 0; i < arguments.length; i++) {

result.push(arguments[i]);

}

return result

}

logArguments("boop", "candle", 3); // ["boop", "candle", 3]

One very annoying thing about arguments is that it is not a true Array object. It is only **Array-like** in that it can be indexed with integers and has a length property. This is infuriating because we can't use any of our favorite Array methods.

For example,

function thisBreaks() {

arguments instanceof Array; //=> false

arguments.forEach((arg => console.log(arg)); // Raises an error

}

thisBreaks(); //=> TypeError: arguments.forEach is not a function

We can, however, use [Array.prototype.slice](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/slice) to create a copy of arguments that is an array by calling it on arguments:

function thisWorks() {

let args = Array.prototype.slice.call(arguments);

args instanceof Array; //=> true

args.forEach((arg)=> console.log(arg)); // This works!

}

Why does this work? arguments is *Array-like* enough for the [slice](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/slice) method to work.

Array.from**(ES6+)**

If you thought the trick above was hacky, [Ecma International](https://en.wikipedia.org/wiki/Ecma_International) would agree. That's why ES6 includes a new method, Array.from, that accomplishes the same thing as our Array.prototype.slice.call trick above.

function thisWorksToo() {

let args = Array.from(arguments);

args instanceof Array; //=> true

args.forEach((arg)=> console.log(arg)); // This works too!

}

**Rest Parameters (ES6+)**

ES6 also introduces another way to handle arguments that deprecates the need to coerce arguments at all: the ... operator (Rest Operator). ... works just like Ruby's splat operator (\*) and can be used to capture all a function's arguments into an actual array.

The differences between arguments and Rest Parameters are:

* Rest Parameters only grab un-named arguments.
* Rest Parameters give us back a real array, so we can use methods like forEach, pop and sort.

Let's write a quick example method that will start by logging the first argument, followed by a list of the remaining arguments.

function argumentsWay(firstArg) {

console.log(`The first arg is ${firstArg}!`);

// We grab the arguments and call slice with 1 to eliminate the firstArg

let otherArgs = Array.prototype.slice.call(arguments, 1);

console.log(`The other args are:`);

otherArgs.forEach((arg) => {

console.log(arg);

});

}

function restWay(firstArg, ...otherArgs) {

console.log(`The first arg is ${firstArg}!`);

console.log(`The other args are:`);

otherArgs.forEach((arg) => {

console.log(arg);

});

}

Rest arguments are often simpler to use than the old arguments keyword and are stylistically preferred by companies that have adopted ES6. However, for the sake of interviews and for understanding JavaScript, it is important to understand both forms of grabbing arguments.

**Spread Syntax (ES6+)**

ES6 also allows us to use Spread Syntax, which is like the Ruby splat for destructuring elements. We can now pass a "spread" array to a function with the ... as shown below:

function madLib(verb, pluralNoun1, pluralNoun2, place) {

return `I like to ${verb} ${pluralNoun1} with ${pluralNoun2} by the ${place}.`;

}

const words = ["eat", "socks", "rabbits", "sea"];

madLib(...words); // equivalent to 'madLib(words[0], words[1], words[2], words[3])'

//=> "I like to eat socks with rabbits by the sea."

We can even destructure multiple arguments in a function call.

const myFunction = (v, w, x, y, z) => { }

const args = [2, 3];

myFunction(1, ...args, 4, ...[5]); // v = 1, w = 2, x = 3, y = 4, z = 5

**Default Values (ES6+)**

Default values are new to ES6. We can now set default values in a way similar to Ruby.

function add(x, y = 17) {

// y is 17 if not passed or passed as `undefined`

return x + y;

}

add(3) === 20; //=> true

add(3, undefined) === 20; //=> true