Review: Functions

(https://github.com/learn-co-curriculum/phase-1-functions) (https://github.com/learn-co-curriculum/phase-1-functions/issues/new)

Learning Goals

- Define abstraction
- Explain that functions are abstractions
- Explain how to *call* a function
- Define "Generalization"
- Demonstrate "Generalization" by using *parameters* and *arguments*
- Demonstrate *return values*

Introduction

Functions are the single most important unit of code in JavaScript. Much like a `<div>` or a `<section>` in HTML, functions serve as ways to group together related bits of JavaScript code. Grouped code is easier to read, debug, and improve.

Define Abstraction

Abstraction comes from Latin roots which mean "to pull away." It's the "take-away" or "impression" of a whole thing. As humans, we often take sets of single actions or things and *abstract them* into another word.

That word that we "pull away" is the "abstraction." Literally it means "the pulled away thing." You might not think about it often, but your brain is full of abstractions.

Single Units	Abstraction
John, Paul, George, Ringo	The Beatles
Get two pieces of bread, put jam on ...	Make a peanut butter and jelly sandwich

Single Units	Abstraction
visit site, make userid, make password...	Sign up for Flatbook
get in the lift, hit "G" button, exit elevator, walk to subway...	Go home

We create abstractions to make it easier to shorten our sentences. We'd never get anything done if we couldn't abstract! We also use abstractions to decide what doesn't fit or what should fit. "Mozart" doesn't belong with The Beatles, but he does fit with "Classical Composers."

Abstractions help us think about complex activities. Humans brought the pattern of "abstracting work" to JavaScript. Abstractions that hold work are called *functions*.

Explain That Functions Are Abstractions

Functions combine a series of steps under a new name. That's why they're *abstractions*. We'll call that the *function name*. More formally:

A function is an object that contains a sequence of JavaScript statements. We can execute or *call* it multiple times.

To *call* a function means to run the independent pieces that make it. Synonyms to *call* that you might see are *execute* and *invoke*.

Let's describe a series of single, non-abstract, tasks:

```
console.log("Wake Byron the poodle");
console.log("Leash Byron the poodle");
console.log("Walk to the park Byron the poodle");
console.log("Throw the frisbee for Byron the poodle");
console.log("Walk home with Byron the poodle");
console.log("Unleash Byron the poodle");
```

To abstract these single actions into a collective name, we do:

```
function exerciseByronThePoodle() {
  console.log("Wake Byron the poodle");
  console.log("Leash Byron the poodle");
  console.log("Walk to the park Byron the poodle");
```

```
    console.log("Throw the frisbee for Byron the poodle");
    console.log("Walk home with Byron the poodle");
    console.log("Unleash Byron the poodle");
  }
```

This code above is a *function declaration*.

Here we have *abstracted* 6 activities into 1 activity: `exerciseByronThePoodle` .

> **ASIDE**: Abstractions themselves can be lumped together *as if* they were single things. The abstraction `dailyDogCareForByron` probably includes `feedByronThePoodle` , `giveWaterToByronThePoodle` , `exerciseByronThePoodle` , etc.

Explain How To *Call* a Function

To "execute" or "call" a function in JavaScript you add `()` after its name. To execute the function we just defined, you run: `exerciseByronThePoodle()` . When we ran `document.querySelector()` , we were *calling* a function. `Math.floor()` is another function. That `()` is also known as the *invocation operator* because it tells JavaScript to...invoke the function.

> **LEARNING TIP**: Try defining a small function in the JavaScript console to test this out. You can copy the syntax provided above.

Of course, calling a function only works if the function has been *declared*.

Note: Later in the course you will learn about different ways of declaring functions and about a concept called *hoisting*. If you declare a function using the `function` keyword, as we have been doing so far, the function call can actually come **before** the function declaration in the code file, due to hoisting. But it needs to exist somewhere; if you try to call a function that hasn't been declared somewhere in the code, you'll get an error.

Define "Generalization"

Looking at our abstraction, `exerciseByronThePoodle()` , it's pretty concrete, the opposite of abstract. It's concrete because it only works for Byron the Poodle. Our function would be more *abstract* if it were written for *all dogs* and it just-so-happened that Byron the Poodle was one of the eligible things to undergo the function's processes. The process of moving from *concrete* to *abstract* is called "generalization" (or "abstraction," by some).

Demonstrate "Generalization" By Using *Parameters* And *Arguments*

Let's make `exerciseByronThePoodle()` more general. Looking at the `console.log()` statements, we repeatedly refer to a dog's name and a dog's breed. Both of these are `Strings` . If we were to write them as JavaScript variables inside the function we might write `dogName` and `dogBreed` .

Let's use `String` interpolation to generalize the *body* of our function

```
function exerciseByronThePoodle() {
  const dogName = "Byron";
  const dogBreed = "poodle";
  console.log(`Wake ${dogName} the ${dogBreed}`);
  console.log(`Leash ${dogName} the ${dogBreed}`);
  console.log(`Walk to the park ${dogName} the ${dogBreed}`);
  console.log(`Throw the frisbee for ${dogName} the ${dogBreed}`);
  console.log(`Walk home with ${dogName} the ${dogBreed}`);
  console.log(`Unleash ${dogName} the ${dogBreed}`);
}
```

If we *call* this function, we'll get the exact *same* result as the original `exerciseByronThePoodle()` .

But there are some advances here. We define the `dogName` and `dogBreed` in only one place. That means we can change things a bit easier now by changing these variables instead of using find-and-replace (`2 * 6 = 12`) twelve times.

Our problem now is that our function has the `dogName` and `dogBreed` locked in. If we could make it possible to tell each *call* of the function "Hey use these `String` s instead" we could get more *general*.

That's the purpose of *parameters*. *Parameters* are locally-scoped variables that are usable ("scoped") to inside the function. In our example, our variables `dogName` and `dogBreed` should become *parameters*. They're defined inside of the *function declaration's* `()` .

```
function exerciseDog(dogName, dogBreed) {
...
```

```
...
```

JavaScript will assign the *arguments* of "Byron" and "poodle" to the *parameters* `dogName` and `dogBreed` when this function is called like so:

```
exerciseDog("Byron", "poodle");
```

The full *function declaration* for `exerciseDog` is:

```
function exerciseDog(dogName, dogBreed) {
  console.log(`Wake ${dogName} the ${dogBreed}`);
  console.log(`Leash ${dogName} the ${dogBreed}`);
  console.log(`Walk to the park ${dogName} the ${dogBreed}`);
  console.log(`Throw the frisbee for ${dogName} the ${dogBreed}`);
  console.log(`Walk home with ${dogName} the ${dogBreed}`);
  console.log(`Unleash ${dogName} the ${dogBreed}`);
}
```

When the function is *called*, it assigns `dogName = "Byron"` and `dogBreed = "poodle"` . The parameters are usable inside the function body *as if* they had been set with `const` inside the function.

Because our function is now more *general*, we can:

```
exerciseDog("Boo", "puggle");
exerciseDog("Jojo", "mutt");
exerciseDog("Emmeline", "bernedoodle");
```

If expected arguments aren't given, the parameters won't be set. The parameters' values will be `undefined` . This is just like non-initialized variables; if you don't assign a value they're `undefined` . **This will not cause an error in JavaScript**. This can lead to humorous bugs like:

```
"Wake undefined the undefined"  // From: console.log(`Wake ${dogName} the ${dogBreed}`);
```

We can assign default arguments to our parameters. While it's not as attention-grabbing as a real error, it's a helpful signal that we've run off the rails.

```
function exerciseDog(dogName="ERROR the Broken Dog", dogBreed="Sick Puppy") {
...
```

In summary, we went from:

- a list of operations
- to a wrapped abstraction called a function
- to a more general version of the function

Demonstrate *Return Values*

Sometimes it's helpful to send something *back* to the place where the function was *called*. It's like a "summary" of what happened in the function. In real life, we expect the function "bake a cake" to return a "cake". Or we expect "Visit the ATM" to return paper money. Functions in JavaScript can also return things. Consider:

```
const weatherToday = "Rainy";

function exerciseDog(dogName, dogBreed) {
  if (weatherToday === "Rainy") {
    return `${dogName} did not exercise due to rain`;
  }
  console.log(`Wake ${dogName} the ${dogBreed}`);
  console.log(`Leash ${dogName} the ${dogBreed}`);
  console.log(`Walk to the park ${dogName} the ${dogBreed}`);
  console.log(`Throw the frisbee for ${dogName} the ${dogBreed}`);
  console.log(`Walk home with ${dogName} the ${dogBreed}`);
  console.log(`Unleash ${dogName} the ${dogBreed}`);
  return `${dogName} is happy and tired!`;
}
```

```
const result = exerciseDog("Byron", "poodle");
console.log(result); // => "Byron did not exercise due to rain"
```

When the JavaScript engine encounters a `return` statement it "returns" the value of the thing that appears to the right of the word. The thing could be a `String`, a `Number` or an *expression* like `1 + 1` (which returns, `2`, sensibly enough).

When a `return` is reached in the code, the `return` will be executed, but if there is any code after that point, that code will **not** be executed. In the example above, the expression `weatherToday === "Rainy"` returns `true`, so **the only thing that happens** is the evaluation and return of the `String` `${dogName} did not exercise due to rain`.

Return values can be saved to variables. Or they can be used as inputs to other functions.

Conclusion

In this lesson we learned about the idea of abstraction, both in real life and in code. Abstractions reduce complexity by allowing us to think in groups of activities or things instead of being fully zoomed-in all the time. JavaScript functions are defined:

```
function functionName(parameter1, parameter2, parameter3) {
  // body code goes here
}
```

Functions are "called" by entering the function's name followed by the *invocation operator*, `()`. "Invoke" or "execute" mean the same thing. Arguments that the function declaration expects should be passed inside of the invocation operator. Functions can, but are not obligated to, return *return values* at the end of their execution. Return values are often results of a process, grand totals, or success / failure data.

Resources

- MDN
  - **Functions — reusable blocks of code** (https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Building_blocks/Functions)
  - **Function return values** (https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Building_blocks/Return_values)

- **Function declaration (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/function)**