**Introduction**

We are often unaware of the number of assumptions we make when we communicate with other people in our native languages. If we told you to "count to three," we would expect you to say or think the numbers one, two and three. We assumed you would know to start with "one" and end with "three". With programming, we’re faced with needing to be more explicit with our directions to the computer. Here's how we might tell the computer to "count to three":

for (let i = 1; i<=3; i++) { console.log(i) }

When we speak to other humans, we share a vocabulary that gives us quick ways to communicate complicated concepts. When we say "bake", it calls to mind a familiar subroutine— preheating an oven, putting something into an oven for a set amount of time, and finally removing it. This allows us to *abstract* away a lot of the details and communicate key concepts more concisely. Instead of listing all those details, we can say, "We baked a cake," and still impart all that meaning to you.

In programming, we can accomplish “abstraction” by writing functions. In addition to allowing us to reuse our code, functions help to make clear, readable programs. If you encountered countToThree() in a program, you might be able to quickly guess what the function did without having to stop and read the function's body.

We're also going to learn about another way to add a level of abstraction to our programming: *higher-order functions*. *Higher-order functions*are functions that accept other functions as arguments and/or return functions as output. This enables us to build abstractions on other abstractions, just like "We hosted a birthday party" is an abstraction that may build on the abstraction "We made a cake."

In summary, using more abstraction in our code allows us to write more modular code which is easier to read and debug.

# Functions as Data

JavaScript functions behave like any other data type in the language; we can assign functions to variables, and we can reassign them to new variables.

Below, we have an annoyingly long function name that hurts the readability of any code in which it's used. Let's pretend this function does important work and needs to be called repeatedly!

const announceThatIAmDoingImportantWork = () => { console.log("I’m doing very important work!"); };

What if we wanted to rename this function without sacrificing the source code? We can re-assign the function to a variable with a suitably short name:

const busy = announceThatIAmDoingImportantWork; busy(); // This function call barely takes any space!

busy is a variable that holds a reference to our original function. If we could look up the address in memory of busy and the address in memory of announceThatIAmDoingImportantWork they would point to the same place. Our new busy() function can be invoked with parentheses as if that was the name we originally gave our function.

Notice how we assign announceThatIAmDoingImportantWork without parentheses as the value to the busy variable. We want to assign the value of the function itself, not the value it returns when invoked.

In JavaScript, functions are first class objects, this means that like other objects you've encountered, JavaScript functions can have properties and methods.

Since functions are a type of object, they have properties such as .length and .name and methods such as .toString(). You can see more about the methods and properties of functions [in the documentation](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Function).

Functions are special because we can invoke them, but we can still treat them like any other type of data. Let's get some practice doing that!

# Functions as Parameters

Since functions can behave like any other type of data in JavaScript, it might not surprise you to learn that we can also pass functions (into other functions) as parameters. A higher-order function is a function that either accepts functions as parameters, returns a function, or both! We call the functions that get passed in as parameters and invoked callback functions because they get called during the execution of the higher-order function.

When we pass a function in as an argument to another function, we don't invoke it. Invoking the function would evaluate to the return value of that function call. With callbacks, we pass in the function itself by typing the function name without the parentheses (that would evaluate to the result of calling the function):

const timeFuncRuntime = funcParameter => { let t1 = Date.now(); funcParameter(); let t2 = Date.now(); return t2 - t1; } const addOneToOne = () => 1 + 1; timeFuncRuntime(addOneToOne);

We wrote a higher-order function, timeFuncRuntime(). It takes in a function as an argument, saves a starting time, invokes the callback function, records the time after the function was called, and returns the time the function took to run by subtracting the starting time from the ending time.

This higher-order function could be used with any callback function which makes it a potentially powerful piece of code.

We then invoked timeFuncRuntime() first with the addOneToOne() function - note how we passed in addOneToOne and did not invoke it.

timeFuncRuntime(() => { for (let i = 10; i>0; i--){ console.log(i); } });

In this example, we invoked timeFuncRuntime() with an anonymous function that counts backwards from 10. Anonymous functions can be arguments too!

Let’s get some practice using functions and writing higher-order functions.

# Review

Great job! By thinking about functions as data and learning about higher-order functions, you've taken important steps in being able to write clean, modular code and take advantage of JavaScript's flexibility.

Let's review what we learned in this lesson:

* Abstraction allows us to write complicated code in a way that's easy to reuse, debug, and understand for human readers
* We can work with functions the same way we would any other type of data including reassigning them to new variables
* JavaScript functions are first-class objects, so they have properties and methods like any object
* Functions can be passed into other functions as parameters
* A higher-order function is a function that either accepts functions as parameters, returns a function, or both