Introduction to Callbacks

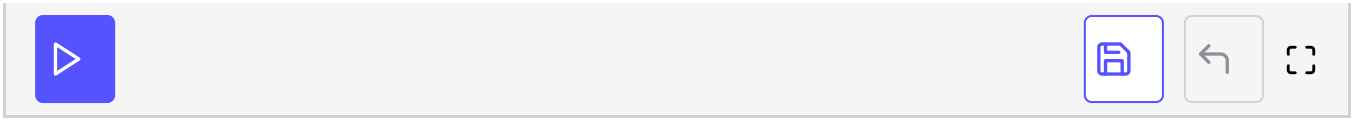
Callbacks are functions that are passed to other functions as parameters. They're necessary for asynchronous code, encapsulation, eliminating code repetition, and so much more. The fact that functions are first-class objects in JavaScript is part of what makes the language so powerful.

One of the most powerful properties of JavaScript is that functions are first-class objects. This means that they are like any other object and have the same properties as standard objects. In fact, we should think of them as nothing more than *callable objects*.

Like an object, we can store properties on a function.

```
function fn() {};

fn.property = 'Hello!';
console.log(fn.property); // -> Hello!
```

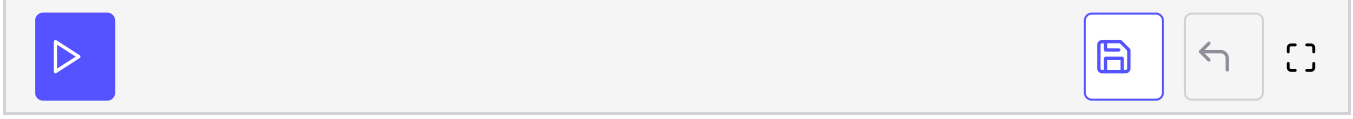


An object can have any data type as a property. A function can as well. In fact, we can even set a function as a property on another function.

```
function fn() {
    console.log('Executing fn');
}

fn.func = function() {
    console.log('Executing func');
}

fn(); // -> Executing fn
fn.func(); // -> Executing func
```

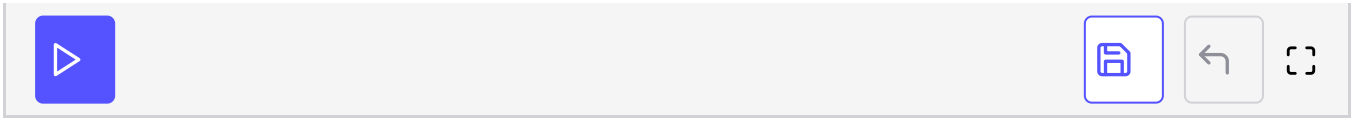


The point is that functions in JavaScript are like any other data type. They can be created at any time, set to a variable, and reassigned. It shouldn't be a

surprise then that they can be passed into another function and returned from another function. The following is perfectly legal.

```
function fnGenerator() {
    return function() {
        console.log('Ran the inner function');
    }
}

const fnReturned = fnGenerator();
console.log(fnReturned); // -> [Function]
fnReturned(); // -> Ran the inner function
```



Running the above block in Chrome yields:

> *ƒ* () {
>
> ```
> console.log('Ran the inner function');
> ```
>
> }
>
> Ran the inner function

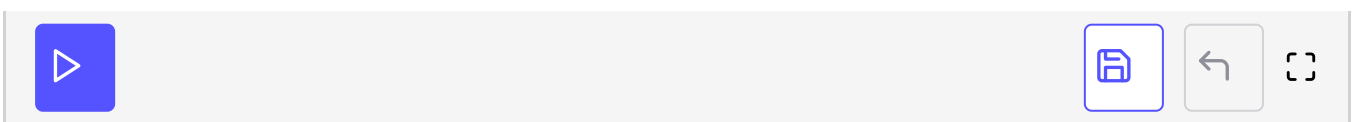
So we can see that `fnReturned` is indeed the function that was returned.

We can also pass functions *in* to other functions.

```
function fnCaller(fn) {
    fn();
}

function log() {
    console.log('Calling log');
}

fnCaller(log); // -> Calling log
```



This gives us the definition of a callback. **A callback is a function that is passed in to another function**.

The purpose of a callback is to be called later after some other code runs first. We pass in a function reference to another function and that recipient function calls it when ready. That can be immediately, or after performing a few tasks.

A function that receives a callback can call that function in different ways as well. It can pass in its own arguments, it can tell it to wait some amount of time before executing, it can call it multiple times, or it can even return a new function based on the callback function.

Callback functions allow us to do interesting things with our code. Passing functions as parameters and returning functions allow us to:

- Eliminate code repetition
- Allow asynchronous execution
- Allow event binding, such as with user interaction
- Hide data and create a pleasant user interface
- Prevent scope pollution

We'll learn more about how callbacks help us achieve each of the above in later lessons. In the next lesson, we'll go over closures, a concept you're guaranteed to be asked in a JavaScript interview.

Example

Let's go through an exercise.

Finish the `callMultiply` function such that when we invoke it by passing in `multiply` and two values, it calls `multiply` with those values and returns that value.

```
function multiply(x, y) {
    return x * y;
}

function callMultiply(fn, val1, val2) {
    // Your code here
}
```