JavaScript the Hard Parts **Callbacks & Higher Order Functions**

But first - In JSHP we start with a set of fundamental mental models

- One of the most misunderstood concepts in JavaScript
- Enables powerful pro-level functions like map, filter, reduce
- Makes our code more declarative and readable
- Forms the backbone of the Codesmith technical interview (and professional mid/senior level engineering interviews)

JavaScript principles

We will diagram each line of code to understand how it works under-the-hood

Functions

Code we save ('define') and can use (call/invoke/execute/run) later on by using the function's name/label with parentheses

Thread of execution

JavaScript goes through the code (globally or in a function) line by line and does whatever the line of code says to do

Memory

A store of data ('Variable environment') where anything defined in the function is stored

```
let num = 3;
function multiplyBy2 (inputNumber){
 const result = inputNumber*2;
 return result;
}

const output = multiplyBy2(num);
const newOutput = multiplyBy2(10);
```

Functional Programming

Some essential features

- Pure functions (no side effects)
- Higher order functions
  - Valuable professional tool (filter, map, reduce)
  - More readable code
  - Part of almost every Codesmith technical interview

Why do we even have functions?

Let’s see why...

Create a function 10 squared

- Takes no input
- Returns 10*10

What is the syntax (the exact code we type)?

Why do we even have functions?

Let's see why...

Create a function 10 squared

- Takes no input
- Returns 10*10

What is the syntax (the exact code we type)?

```
function tenSquared() {
   return 10*10;
}

tenSquared() // 100
```

Now let's create a function that returns 9 squared

Create a function 9 squared

- Takes no input
- Returns 9*9

What is the syntax (the exact code we type)?

```
function tenSquared() {
   return 10*10;
}

tenSquared() // 100
```

Now let's create a function that returns 9 squared

Create a function 9 squared

- Takes no input
- Returns 9*9

What is the syntax (the exact code we type)?

```
function nineSquared() {
    return 9*9;
}

nineSquared() // 81
```

Now 8 squared and so on...

We have a problem - it's getting repetitive, we're breaking our DRY principle

What could we do?

```
function eightSquared() {
    return 8*8;
}

eightSquared() // 64
```

We can generalize the function

‘Parameters’ (placeholders) mean we don’t need to decide what data to run our functionality on until we run the function and provide an actual value (‘argument’)

Higher order functions follow this same principle. We may not want to decide exactly what some of our *functionality* is until we run our function

```
function squareNum(num){
    return num*num;
}
squareNum(10); // 100
squareNum(9); // 81
squareNum(8); // 64
```

Now suppose we have a function:

`copyArrayAndMultiplyBy2`

Let's diagram it out

```
function copyArrayAndMultiplyBy2(array) {
   const output = [];
   for (let i = 0; i < array.length; i++) {
     output.push(array[i] * 2);
   }
   return output;
 }

const myArray = [1,2,3];
const result = copyArrayAndMultiplyBy2(myArray);
```

What if want to copy array and divide by 2?

```
function copyArrayAndDivideBy2(array) {
   const output = [];
   for (let i = 0; i < array.length; i++) {
     output.push(array[i] / 2);
   }
   return output;
 }

const myArray = [1,2,3];
const result = copyArrayAndDivideBy2(myArray);
```

Or add 3?

What principle are we breaking?

```
function copyArrayAndAdd3(array) {
   const output = [];
   for (let i = 0; i < array.length; i++) {
     output.push(array[i] + 3);
   }
   return output;
 }

const myArray = [1,2,3];
const result = copyArrayAndAdd3(myArray);
```

Or add 3?

What principle are we breaking?

DRY - Don’t Repeat Yourself

```
function copyArrayAndAdd3(array) {
   const output = [];
   for (let i = 0; i < array.length; i++) {
     output.push(array[i] + 3);
   }
   return output;
 }

const myArray = [1,2,3];
const result = copyArrayAndAdd3(myArray);
```

```
function copyArrayAndManipulate(array, instructions) {
   const output = [];
   for (let i = 0; i < array.length; i++) {
     output.push(instructions(array[i]));
   }
   return output;
}

function multiplyBy2(input) {
   return input * 2;
}

const result = copyArrayAndManipulate([1, 2, 3], multiplyBy2);
```

We could generalize our function

So that we pass in our specific instruction only when we run the `copyArrayAndManipulate` function!

```
function copyArrayAndManipulate(array, instructions) {
   const output = [];
   for (let i = 0; i < array.length; i++) {
     output.push(instructions(array[i]));
   }
   return output;
}

function multiplyBy2(input) {
   return input * 2;
}

const result = copyArrayAndManipulate([1, 2, 3], multiplyBy2);
```

How was this possible?

Functions in javascript = first class objects

They can co-exist with and can be treated like any other javascript object

1. Assigned to variables and properties of other objects
2. Passed as arguments into functions
3. Returned as values from functions

```
function copyArrayAndManipulate(array, instructions) {
   const output = [];
   for (let i = 0; i < array.length; i++) {
     output.push(instructions(array[i]));
   }
   return output;
}

function multiplyBy2(input) {
   return input * 2;
}

const result = copyArrayAndManipulate([1, 2, 3], multiplyBy2);
```

Which is our Higher Order Function?

Which is our Callback Function?

```
function copyArrayAndManipulate(array, instructions) {
   const output = [];
   for (let i = 0; i < array.length; i++) {
     output.push(instructions(array[i]));
   }
   return output;
}

function multiplyBy2(input) {
   return input * 2;
}

const result = copyArrayAndManipulate([1, 2, 3], multiplyBy2);
```

Which is our Higher Order Function?

The outer function that *takes in* the function (our callback) is a higher-order function

Which is our Callback Function?

The function we *pass in* is a callback function

```javascript
function copyArrayAndManipulate(array, instructions) {
   const output = [];
   for (let i = 0; i < array.length; i++) {
     output.push(instructions(array[i]));
   }
   return output;
}

function multiplyBy2(input) {
   return input * 2;
}

const result = copyArrayAndManipulate([1, 2, 3], multiplyBy2);
```

Higher-order functions

Takes in a function or passes out a function

Just a term to describe these functions - any function that does it we call that - but there's nothing different about them inherently

Callbacks and Higher Order Functions simplify our code and keep it DRY

And they do something even more powerful. They allow us to run asynchronous code

Declarative readable code

Map, filter, reduce - the most readable way to write code to work with data

Codesmith & pro interview prep

One of the most popular topics to test in interview both for Codesmith and mid/senior level job interviews

Asynchronous JavaScript

Callbacks are a core aspect of async JavaScript, and are under-the-hood of promises, async/await

What’s Next

Career Workshops

Whiteboarding for Technical Interviews

How to Prepare for Technical Interviews

JSHP

JavaScript the Hard Parts

Callbacks & Higher Order Functions

Closures

Recursion

Classes & Prototypes

Asynchronous JavaScript

CSX - Online Learning

Comprehensive training on JavaScript fundamentals from beginner to advanced including callbacks, closure, OOP, async and projects

Fast Track Application

Skip the application essays with the Fast Track coding challenge