JavaScript the Hard Parts
Closure

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

- Closure is the most esoteric of JavaScript concepts
- Enables powerful pro-level functions like ‘once’ and ‘memoize’
- Many JavaScript design patterns including the module pattern use closure
- Build iterators and maintain state in an asynchronous world

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);
```

Functions can have multiple names

Functions are just a set of instructions/block of code that has been stored/saved to memory. We can give that set of instructions multiple labels

Functions Two parts to defining a function:

1. The function's name (label)
2. The function's code (definition)

Behind the scenes:

- Functions store a link/reference/address pointing to a location in the 'heap'

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

const output = multiplyBy2(9);

const newLabelForMultiplyBy2 = multiplyBy2;
const newOutput = newLabelForMultiplyBy2(10);
```

Closure

When our functions get called, we create a live store of data (local memory/variable environment/state) for that function’s execution context.

When the function finishes executing, its local memory is deleted (except the returned value)

But what if our functions could hold on to live data/state between executions? 😲

This would let our function definitions have an associated cache/persistent memory

But it all starts with us returning a function from another function

Functions can be returned from other functions in JavaScript

Two parts to defining a function:

1. The function’s name (label)
2. The function’s code (definition)

```
function createFunction() {
   function multiplyBy2 (num){
       return num*2;
   }
   return multiplyBy2;
}

const generatedFunc = createFunction();

const result = generatedFunc(3); // 6
```

Calling a function in the same function call as it was defined in

Where you define your functions determines what variables your function have access to when you call the function

```
function outer (){
   let counter = 0;
   function incrementCounter (){
       counter ++;
   }
   incrementCounter();
}

outer();
```

Calling a function outside of the function call in which it was defined

Where you define your functions determines what variables your function have access to when you call the function

```
function outer (){
   let counter = 0;
   function incrementCounter (){
       counter ++;
   }
   return incrementCounter;
}

const myNewFunction = outer();
myNewFunction();
myNewFunction();
```

The bond

When a function is defined, it gets a bond to the surrounding Local Memory (“Variable Environment”) in which it has been defined

```
function outer (){
   let counter = 0;
   function incrementCounter (){
       counter ++;
   }
   return incrementCounter;
}

const myNewFunction = outer();
myNewFunction();
myNewFunction();
```

The ‘backpack’

1. When *incrementCounter* is defined inside *outer*, it gets a bond to the surrounding Local Memory in *outer*

2. We then return *incrementCounter’s* code (function definition) out of *outer* into global and give it a new name - *myNewFunction*

3. BUT we maintain the bond to *outer’s* live local memory - it gets ‘returned out’ attached **on the back of** *incrementCounter’s* function definition. So *outer’s* local memory is now stored attached to *myNewFunction* - even though *outer’s* execution context is long gone

4. When we run *myNewFunction* in the global execution context, it will first look in its own local memory for any data it needs (as we’d expect), but then in *myNewFunction’s* ‘backpack’

```
function outer (){
   let counter = 0;
   function incrementCounter (){
      counter ++;
   }
   return incrementCounter;
}

const myNewFunction = outer();
myNewFunction();
myNewFunction();
```

What can we call this ‘backpack’?

1. Closed over ‘Variable Environment’ (C.O.V.E.)
2. Persistent Lexical Scope Referenced Data (P.L.S.R.D.)
3. ‘Backpack’
4. ‘Closure’

The ‘backpack’ (or ‘closure’) of live data is attached incrementCounter (then to myNewFunction) through a hidden property known as [[scope]] which persists when the inner function is returned out

```
function outer (){
   let counter = 0;
   function incrementCounter (){
       counter ++;
   }
   return incrementCounter;
}

const myNewFunction = outer();
myNewFunction();
myNewFunction();
```

```
function outer (){
   let counter = 0;
   function incrementCounter (){
       counter ++;
   }
   return incrementCounter;
}

const myNewFunction = outer();
myNewFunction();
myNewFunction();

const anotherFunction = outer();
anotherFunction();
anotherFunction();
```

Individual backpacks

What if we run '*outer*' again and store the returned '*incrementCounter*' function definition in '*anotherFunction*'

Closure gives our functions persistent memories

If we understand closure under-the-hood, we get access to an entirely new toolkit for writing quality code

Helper functions

Everyday professional helper functions like ‘once’ and ‘memoize’

Iterators and generators

Which use lexical scoping and closure under-the-hood to achieve the most contemporary patterns for handling data in JavaScript

Module pattern

Preserve state for the life of an application without polluting the global namespace

Asynchronous JavaScript

Callbacks and Promises rely on closure to persist state in an asynchronous environment