Will Sentance

Academic work:
Oxford, Harvard

Currently:
- CEO & Cofounder Codesmith
- Frontend Masters

Previously:
- Cocreator & Engineer @Icecomm
- Software Engineer @Gem

JS The Hard Parts

Execution context and Closure

Closure gives us access to:

— The most esoteric of JavaScript concepts 🔮
— Powerful pro-level functions like ‘once’ and ‘memoize’
— Vital JavaScript design patterns including the module pattern
— Allows us to build iterators and maintain in an asynchronous world

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

Running/calling/invoking a function

This is not the same as defining a function

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

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

When you execute a function you create a new execution context comprising:

1. The thread of execution (we go through the code **in the function** line by line)
2. A local memory ('Variable environment') where anything defined in the function is stored

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 starts with us returning a function from another function

Functions can be returned from other functions in JavaScript

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

const generatedFunc = instructionGenerator()
```

How can we run/call multiplyBy2 now?

Let's call (run) our generated function with the input 3

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

const generatedFunc = instructionGenerator()

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

Calling a function in the same scope as it was defined

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

outer();
```

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

But what if we call our function outside of where it was defined?

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

outer()

incrementCounter();
```

What happens here?

There is a way to run a function outside where it was defined with out an error - we return the inner function and assign it to a new variable

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

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

Now we can run incrementCounter in the global context through its new label myNewFunction

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

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

What happens if we execute myNewFunction again?

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

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

The ‘Backpack’

1. When `incrementCounter` is defined, it gets a bond to the surrounding Local Memory of live data in `outer` in which it has been defined
2. We then return `incrementCounter` out of `outer` into global and store it in `myNewFunction`
3. BUT we maintain the bond to the surrounding live local memory from inside of `outer` - this live memory gets ‘returned out’ attached to the `incrementCounter` function definition and is therefore 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 its ‘backpack’** before it looks in global memory

What’s the official name for the ‘backpack’?

The Closed over Variable Environment (COVE) or ‘Closure’

This ‘backpack’ of live data that gets returned out with `incrementCounter` is known as the ‘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

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

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

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

const anotherFunction = outer(); // myNewFunction = incrementCounter
anotherFunction();
anotherFunction();
```

The power of Closure

Now: Our functions get 'memories' - once, memoize

Advanced: We can implement the module pattern in JavaScript

Allows us to build iterators and maintain in an asynchronous world