CODESMITH

Software Engineering & Machine Learning Residency

nuary 2017

ill Sentance

ademic work:
ford, Harvard

rrently:
O & Cofounder
desmith

eviously:
creator @Icecomm
ftware Engineer @Gem

ch softare engineering at Codesmith

ter of Software eering Excellence

ent Codesmith student ects have been featured at gle I/O and as Facebook's developer tool

st mentors include Brian t at LinkedIn, Gavin Doughtie oogle, Tom Occhino at ebook

tered in LA, NYC and at ord University

2. Selective and tight-knit community

- Each selected student has shown enormous potential in five capacities that make an excellent engineer
- Students come from an exceptional and eclectic range of backgrounds from PhDs to software engineers, from Stanford graduates to self-teachers

3. Community of alumni for life

- 25% of graduates receive offers for Senior Engineer position and above, 70% receive offers for Mid-level Engineer (Codesmith graduate salaries range from $95k to $190k)
- Postgraduate education in advanced software architecture and Machine Learning

nt Codesmith graduates are building at

LinkedIn NIKE amazon Microsoft WEVE

The team that makes it all possible

Will Sentance

Ryan Smith

Shanda McCune

Schno Mozingo

Eric Kirsten

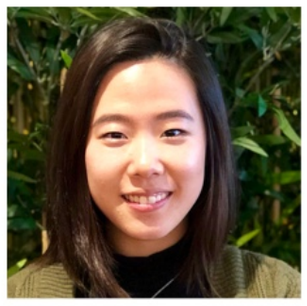
Haley Godtfredsen

Olivia Leitner

Mircea Ilie

Aurora Silva

Jon Perera



Jac Chang

Sam Salley

Kaylee Anderson

Jenny Mith

Pete Fasula

Erik Larsen

Jon Coe

Mejin Leechor

Xavyr Moss

Phil Troutman

Luis Ramirez

Pauline Chang

The 5 capacities we look for in candidates

1. Analytical problem solving with code
2. Technical communication (can I implement your approach just from your explanation)
3. Engineering best practices and approach (Debugging, code structure, patience and reference to documentation)
4. Non-technical communication (empathetic and thoughtful communication)
5. Language and computer science experience

Our expectations

- Support each other - engineering empathy is the critical value at Codesmith
- Work hard, Work smart
- Thoughtful communication

Frontend Masters - JavaScript the Hard Parts

1. **Foundations of JavaScript**
2. Asynchronous JavaScript (callbacks, promises)
3. Iterators
4. Generators & Async/await

Principles of JavaScript

In JSHP we start with a set of fundamental principles

These tools will enable us to problem solve and communicate almost any scenario in JavaScript

— We'll start with an essential approach to get ourselves up to a shared level of understanding
— This approach will help us with the hard parts to come

What happens when javascript executes (runs) my code?

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

As soon as we start running our code, we create a *global execution context*

— Thread of execution (parsing and executing the code line after line)
— Live memory of variables with data (known as a Global Variable Environment)

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(4);
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

We keep track of the functions being called in JavaScript with a Call stack

Tracks which execution context we are in - that is, what function is currently being run and where to return to after an execution context is popped off the stack

One global execution context, a new function execution context for every time we run a function

Frontend Masters - JavaScript the Hard Parts

1. Foundations of JavaScript
2. **Asynchronous JavaScript (callbacks, promises)**
3. Iterators
4. Generators

Asynchronicity is the backbone of modern web development in JavaScript

JavaScript is single threaded (one command executing at a time) and has a synchronous execution model (each line is executed in order the code appears)

So what if we need to **wait some time before we can execute certain bits of code**? Perhaps we need to wait on fresh data from an API/server request or for a timer to complete and then execute our code

We have a conundrum - a tension between wanting to **delay some code execution** but **not wanting to block the thread** from any further code running while we wait

Solution 1

```
function display(data){
    console.log(data)
}

const dataFromAPI = fetchAndWait('https://twitter.com/will/tweets/1')

//... user can do NOTHING here 😭
//... could be 300ms, could be half a second
// they're just clicking and getting nothing

display(dataFromAPI)

console.log(“Me later!”);
```

Problems

— Fundamentally untenable - blocks our single javascript thread from running any further code while the task completes

Benefits

— It’s easy to reason about

Goals

1. Be able to do tasks that take a long time to complete e.g. getting data from the server
2. Continue running our JavaScript code line by line without one long task blocking further JavaScript executing
3. When our slow task completes, we should be able to run functionality knowing that task is done and data is ready!

Conundrum 🤔

Solution 2 - Introducing Web Browser APIs/ Node background threads

```
function printHello(){
    console.log(“Hello”);
}

setTimeout(printHello,1000);

console.log(“Me first!”);
```

We're interacting with a world outside of JavaScript now - so we need rules

```
function printHello(){
    console.log("Hello");
}

function blockFor1Sec(){
    //blocks in the JavaScript thread for 1 second
}

setTimeout(printHello,0);

blockFor1Sec()

console.log("Me first!");
```

Problems

— No problems!
— Our response data is only available in the callback function - Callback hell
— Maybe it feels a little odd to think of passing a function *into* another function only for it to run much later

Benefits

— Super explicit once you understand how it works under-the-hood

Pair Programming

Answer these:

— I know what a variable is
— I've created a function before
— I've added a CSS style before
— I have implemented a sort algorithm (bubble, merge etc)
— I can add a method to an object’s prototype
— I understand the event loop in JavaScript
— I understand 'callback functions'
— I’ve implemented `filter` from scratch
— I can handle collisions in hash tables

Challenges

Asynchronicity & Promises: csbin.io/promises

Iterators, Generators & Async/await: csbin.io/iterators

Introducing the readability enhancer - Promises

- Special objects built into JavaScript that get returned immediately when we make a call to a web browser API/feature (e.g. `fetch`) that's set up to return promises (not all are)
- Promises act as a placeholder for the data we hope to get back from the web browser feature's background work
- We also attach the functionality we want to defer running until that background work is done (using the built in .then method)
- Promise objects will automatically trigger that functionality to run
  - The value returned from the web browser feature's work (e.g. the returned data from the server using `fetch`) will be that function's input/argument

Solution 3 - Using two-pronged ‘facade’ functions that both initiate background web browser work *and* return a placeholder object (promise) immediately in JavaScript

```
function display(data){
    console.log(data)
}

const futureData = fetch('https://twitter.com/will/tweets/1')

futureData.then(display); // Attaches display functionality

console.log(“Me first!”);
```

But we need to know how our promise-deferred functionality gets back into JavaScript to be run

```
function display(data){console.log(data)}
function printHello(){console.log(“Hello”);}
function blockFor300ms(){/* blocks js thread for 300ms with long for loop */}

setTimeout(printHello, 0);
const futureData = fetch('https://twitter.com/will/tweets/1')
futureData.then(display)

blockFor300ms()

// Which will run first?

console.log(“Me first!”);
```

We need a way of queuing up all this deferred functionality

Problems

— 99% of developers have no idea how they're working under the hood
— Debugging becomes super-hard

Benefits

— Cleaner readable style with pseudo-synchronous style code
— Nice error handling process

We have rules for the execution of our asynchronously delayed code

1. Hold each promise-deferred functions in a microtask queue and each non-promise deferred function in a task queue (callback queue) when the API ‘completes’
2. Add the function to the Call stack (i.e. execute the function) ONLY when the call stack is totally empty (Have the Event Loop check this condition)
3. Prioritize tasks (callbacks) in the microtask queue over the regular task queue

Promises, Web APIs, the Callback & Microtask Queues and Event loop allow us to defer our actions until the ‘work’ (an API request, timer etc) is completed and continue running our code line by line in the meantime

---

Asynchronous JavaScript is the backbone of the modern web - letting us build fast ‘non-blocking’

Frontend Masters - JavaScript the Hard Parts

1. Foundations of JavaScript
2. Asynchronous JavaScript (callbacks, promises)
3. **Iterators**
4. Generators & Async/await

Iterators

We regularly have lists or collections or data where we want to go through each item and do something to each element

```
const numbers = [4,5,6]

for (let i = 0; i < numbers.length; i++){
  console.log(numbers[i])
}
```

We're going to discover there's a new beautiful way of thinking about using each element one-by-one

Programs store data and apply functionality to it. But there are two parts to applying functions to collections of data

1. The process of accessing each element
2. What we want to do to each element

Iterators automate the accessing of each element - so we can focus on what to do to each element - and make it available to us in a smooth way

Imagine if we could create a function that stored `numbers` and each time we ran the function it would return out an element (the next one) from `numbers`. NOTE: It'd have to remember which element was next up somehow

But this would let us think of our array/list as a 'stream'/flow of data with our function returning the next element from our 'stream' - this makes our code more readable and more functional

But it starts with us returning a function from another function

Functions can be returned from other functions in JavaScript!

```
function createNewFunction() {
  function add2 (num){
    return num+2;
  }
  return add2;
}

const newFunction = createNewFunction()

const result = newFunction(3)
```

How can we run/call add2 now? Outside of createNewFunction?

We want to create a function that holds both our array, the position we are currently at in our 'stream' of elements and has the ability to return the next element

```
function createFunction(array){
  let i = 0
  function inner(){
    const element = array[i]
    i++
    return element
  }
  return inner
}

const returnNextElement = createFunction([4,5,6])
```

How can we access the first element of our list?

By calling the `returnNextElement`

```
function createFunction(array){
  let i = 0
  function inner(){
    const element = array[i];
    i++;
    return element;
  }
  return inner
}

const returnNextElement = createFunction([4,5,6])
const element1 = returnNextElement()
const element2 = returnNextElement()
```

The bond

- When the function `inner` is defined, it gets a bond to the surrounding Local Memory in which it has been defined
- When we return out `inner`, that surrounding live data is returned out too - attached on the 'back' of the function definition itself (which we now give a new global label `returnNextElement`)
- When we call `returnNextElement` and don't find `array` or `i` in the immediate execution context, we look into the function definition's 'backpack' of persistent live data
- The 'backpack' is officially known as the C.O.V.E. or 'closure'

`returnNextElement` has everything we need all bundled up in it

1. Our underlying array itself
2. The position we are currently at in our ‘stream’ of elements
3. The ability to return the next element

This relies completely on the special property of functions in javascript that when they are born inside other functions and returned - they get a backpack (closure)

What is the posh name for `returnNextElement`?

So iterators turn our data into 'streams' of actual values we can access one after another.

Now we have functions that hold our underlying array, the position we're currently at in the array, *and* return out the next item in the 'stream' of elements from our array when run

This lets us have for loops that show us the element itself in the body on each loop *and more deeply* allows us to rethink arrays as flows of elements themselves which we can interact with by calling a function that switches that flow on to give us our next element

We have truly 'decoupled' the process of accessing each element from what we want to do to each element

Frontend Masters - JavaScript the Hard Parts

1. Foundations of JavaScript
2. Asynchronous JavaScript (callbacks, promises)
3. Iterators
4. **Generators & Async/await**

JavaScript's built in iterators are actually objects with a `next` method that when called returns the next element from the 'stream'/ flow - so let's restructure slightly

```
function createFlow(array){
  let i = 0
  const inner = {next :
      function(){
        const element = array[i]
        i++
        return element
      }
  }
  return inner
}

const returnNextElement = createFlow([4,5,6])
const element1 = returnNextElement.next()
const element2 = returnNextElement.next()
```

And the built in iterators actually produce the next element in the format: `{value: 4}` 😭

Once we start thinking of our data as flows (where we can pick of an element one-by-one) we can rethink how we produce those flows - JavaScript now lets us produce the flows using a function 😲

```
function *createFlow(){
  yield 4
  yield 5
  yield 6
}

const returnNextElement = createFlow()
const element1 = returnNextElement.next()
const element2 = returnNextElement.next()
```

What do we hope `returnNextElement.next()` will return? But how?

This allows us to dynamically set what data flows to us (when we run `returnNextElement`'s function)

```
function *createFlow(){
  const num = 10
  const newNum = yield num
  yield 5 + newNum
  yield 6
}

const returnNextElement = createFlow()
const element1 = returnNextElement.next() // 10
const element2 = returnNextElement.next(2) // 7
```

`returnNextElement` is a special object (a generator object) that when its `next` method is run starts (or continues) running createFlow until it hits `yield` and returns out the value being 'yielded'

```
function *createFlow(){
  const num = 10
  const newNum = yield num
  yield 5 + newNum
  yield 6
}

const returnNextElement = createFlow()
const element1 = returnNextElement.next() // 10
const element2 = returnNextElement.next(2) // 7
```

We end up with a 'stream'/flow of values that we can get one-by-one by running `returnNextElement.next()`

And most importantly, for the first time we get to pause ('suspend') a function being run and then return to it by calling `returnNextElement.next()`

In asynchronous javascript we want to

1. Initiate a task that takes a long time (e.g. requesting data from the server)
2. Move on to more synchronous regular code in the meantime
3. Run some functionality once the requested data has come back

What if we were to `yield` out of the function at the moment of sending off the long-time task and return to the function only when the task is complete

We can use the ability to pause createFlow's running and then restart it only when our data returns

```
function doWhenDataReceived (value){
  returnNextElement.next(value)
}

function* createFlow(){
  const data = yield fetch('http://twitter.com/will/tweets/1')
  console.log(data)
}

const returnNextElement = createFlow()
const futureData = returnNextElement.next()

futureData.then(doWhenDataReceived)
```

We get to control when we return back to createFlow and continue executing - by setting up the trigger to do so (`returnNextElement.next()`) to be run by our function that was triggered by the promise resolution (when the value returned from twitter)

Async/await simplifies all this and finally fixes the inversion of control problem of callbacks

```
async function createFlow(){
  console.log("Me first")
  const data = await fetch('https://twitter.com/will/tweets/1')
  console.log(data)
}

createFlow()
console.log("Me second")
```

No need for a triggered function on the promise resolution, instead we auto trigger the resumption of the `createFlow` execution (this functionality is still added to the microtask queue though)

The Hard Parts Challenge Code

— Guarantees interview for the Codesmith Residency
— 90% of accepted students attend JSHP
— We created the Hard Parts Challenge code to guarantee an interview for the Hard Parts community members
— It builds upon the iterators content you worked on today
— Drinks now 🍹🍸🍻

Appendix

JavaScript gives us `for...of` which runs `returnNextElement()` until it runs out of elements

`for...of` automatically

— creates a `returnNextElement` function (all arrays have a createFunction built in to produce `returnNextElement` function)

— calls the `returnNextElement` function and stores the returned element in `element` to be used by us *directly* in the body of the for loop

```
const numbers = [4,5,6]

for (let element of numbers){
  console.log(element)
}
```