
CODESMITH
Software Engineering Residency
Cohort 10 January 2017

Will Sentance

Academic work:
Oxford, Harvard

Currently:
- CEO & Cofounder Codesmith
- Frontend Masters

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

The team that makes it all possible

Will Sentance

Ryan Smith

Shanda McCune

Schno Mozingo

Eric Kirsten

Haley Godtfredsen

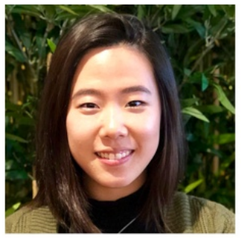
Olivia Leitner

Phil Troutman

Mircea Ilie

Aurora Silva

Jon Perera



Jac Chang

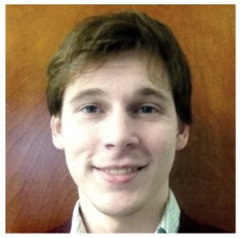
Sam Salley

Kaylee Anderson

Jenny Mith

Alesi Ladas

Shane Yao



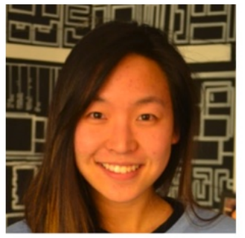
Alex Egorenkov

Jon Coe

Mejin Leechor

Xavyr Moss

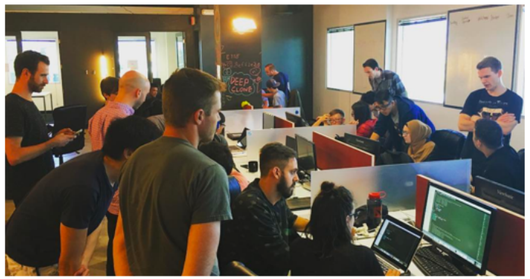
Luis Ramirez



Pauline Chang

What is Codesmith?

1. Center of Software Engineering Excellence



- Codesmith students (known as Residents) have built some of the most prominent tools in the developer community - React Monocle, Reactide, Overvue, WebDSP
- Advisory Board and Guest mentors are leaders in their fields: Brian Holt at Netflix, Gavin Doughtie at Google, Tom Occhino at Facebook
- Centered in Los Angeles and New York with Summer Academy of Code at Oxford University

2. Selective and tight-knit community

Cohort 13 Camping trip to Joshua Tree

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

What is Codesmith?

3. Community of leaders for life

- 25% of graduates receive offers for Senior Engineer position and above, 70% receive offers for Mid-level Engineer
- Graduates are transforming healthcare at Heal, Impact Health, mental health at UCLA, drone technology at Airmap while others work on large systems at the top technology companies in the country
- Postgraduate education in advanced software architecture and Machine Learning

Where are Codesmith graduates from recent cohorts working?

LinkedIn	FullStack Developer III
Google	Web Software Developer
NIKE	Fullstack Developer x3
amazon	Senior Software Engineer
PayPal	Fullstack Software Engineer
Microsoft	Senior FullStack Developer
Anami BLOCKCHAIN	Technical Lead Blockchain
NASA JPL	Software Engineer

How is it possible?

Pedagogy

- Residents become engineers not technicians through a first principles understanding of JavaScript, computer science (algorithms & data structures), OOP & functional programming
- Placing **analytical problem-solving** and **technical communication** at the heart of the program - the most valued capacities of a software engineer
- The most valuable contemporary technologies - React, Redux, Node, build tools, machine learning

Supporting each other - Engineering empathy is the most important value at Codesmith

- The program is pair-programmed throughout - vital for improving technical communication
- Residents stay each day until midnight building together

Production-level Environment

- Codesmith residents build some of the most prominent tools in the React/Node and broader developer ecosystem
- Other graduates have built tools serving hundreds of thousands of users - including Veritas Prep, YouDescribe and the University of Wisconsin
- Projects have collected over 15,000 Github stars in total and been featured by Brendon Eich (the founder of JavaScript) InfoWorld and TechWorld

The 5 capacities we look for in candidates

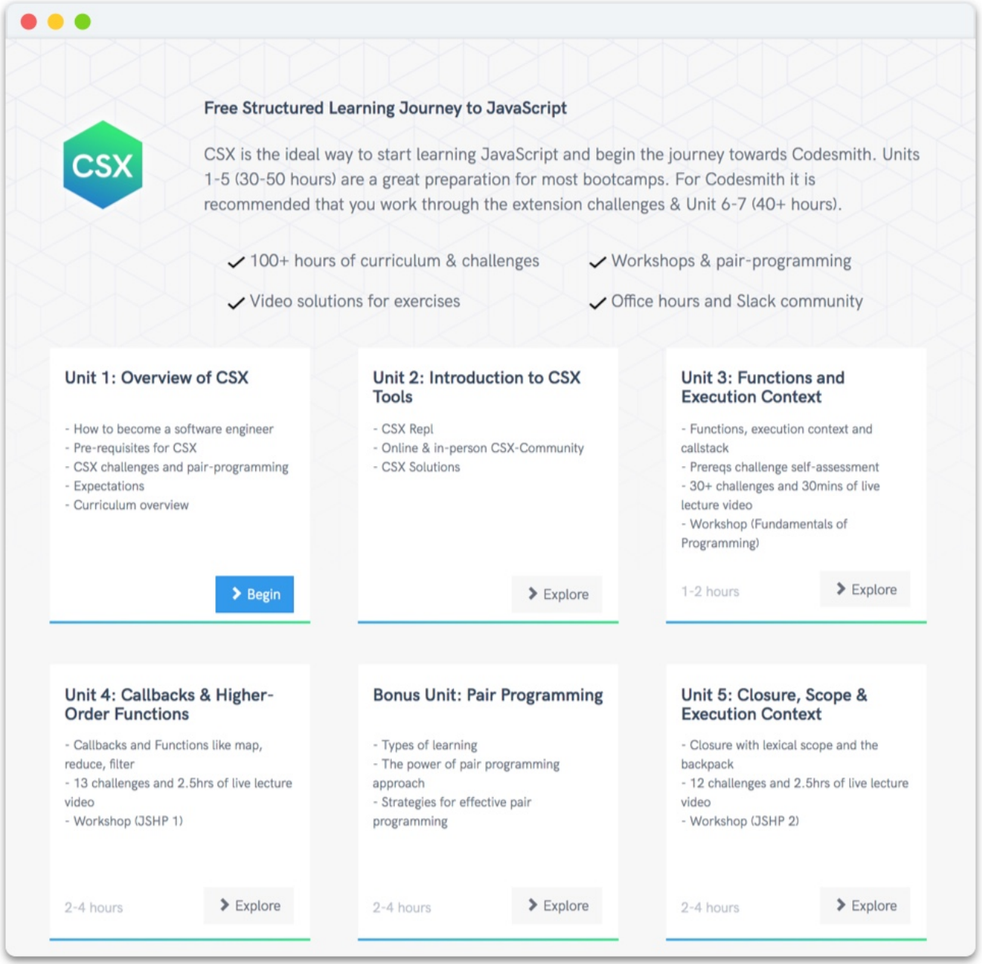
1. Analytical problem solving with code
2. Technical communication (can I implement your approach just from your explanation)
3. Engineering best practices and approach (Debugging, code structure, patience and reference to documentation)
4. Non-technical communication (empathetic and thoughtful communication)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

How to prepare for your Codesmith interview?

CSX and Javascript the Hard Parts



Top resources for technical communication, problem solving and best practices

- Pair-programming and going under-the-hood of JavaScript at JS the Hard Parts - Thursdays 6:30pm
- Coding Challenge sites like: Codewars, Coderbyte, LeetCode, HackerRank, Project Euler
- PythonTutor - Allows you to run through your code line by line similar to the JS the Hard Parts white boarding approach
- JavaScript30 - Build a project each day for 30 days. Or build chrome extensions to focus on your problem-solving

Top resources for JavaScript knowledge

- JS the Hard Parts, JS the Weird Parts
- ReactiveX LearnRX challenges (Higher order functions)
- Angus Croll on JavaScript (Closures, Execution context etc)
- FreeCodeCamp

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

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

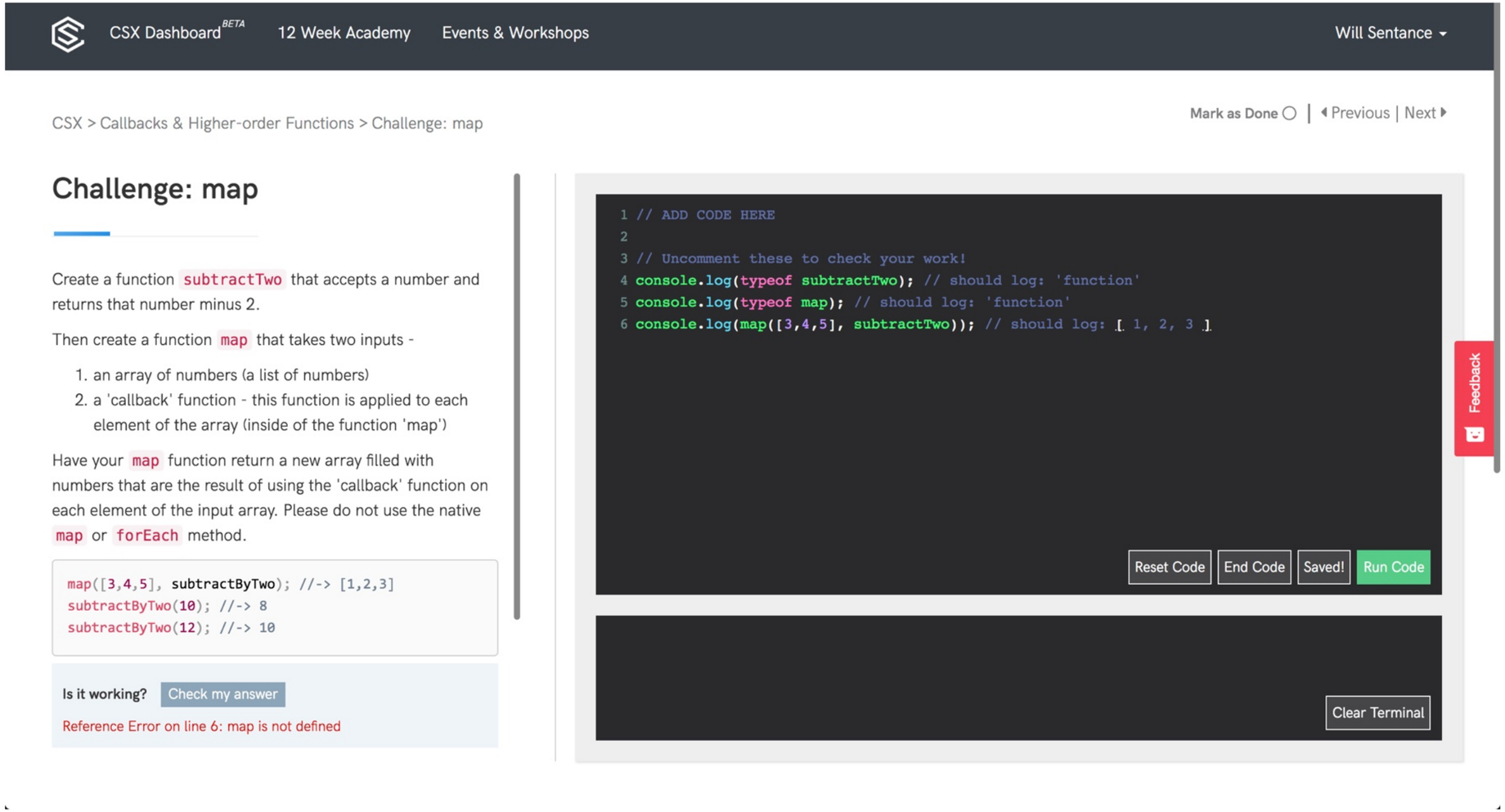
Pair Programming

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

Pair-programming Challenges on CSX

csx.codesmith.io



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’ applications

The Hard Parts Challenge Code

— We created the Hard Parts Challenge code to guarantee an interview for the Hard Parts community members

— We invite ~35% of regular online applications to interview. Completion of the Hard Parts challenge code *guarantees* interview for Codesmith

— It builds upon the content you worked on today

— Drinks now 🍹🍸🍻

How to continue your JavaScript journey

JS functions, objects, arrays, loops

- Workshop: JavaScript Easier Parts
- CSX Unit 1: Functions and Execution Context

Web Development: HTML, CSS & jQuery

- (CSX Project) Building a Chrome extension

JS Callbacks & Higher order functions

- Workshop: JS the Hard Parts: Callbacks
- CSX Unit 2: Callbacks & Higher order functions

Traditional Coding School

+

JS Closures

- Workshop: JS the Hard Parts: Closures
- CSX Unit 3: Closures

Algorithms & recursion

- Workshop: Recursion
- CSX Unit 4: Recursion

Technical communication

- All JSHP workshops
- (Online) Office hours and pair-programming

Codesmith

- In-person & online workshops
- Online activities
- CSX (codesmith.io/learn)
- Self-guided projects

OOP in JS: Workshop & CSX

Asynchronous JS: Workshop & CSX