Codesmith

CS Prep

Day 1

Day 1 Overview

Introductions

Welcome to Hard Learning

CS Prep Overview

The JavaScript Runtime

JavaScript Execution

Variables

Functions

Pair Programming

Arrow Functions

The CS Prep Team

Will Sentance
CEO & Head of Engineering
at Codesmith

Nat Klein
CS Prep Instructor &
Software Engineer

Phillip Troutman
Chief Academic Officer

Week 1 Schedule

Number	Day	Time	Topics
1	Monday	6:30-9:30 pm ET	Functions, Execution Contexts, Variable Declarations
2	Tuesday	6:30-9:30 pm ET	Objects, Arrays, Prototypal Inheritance, OOP
3	Wednesday	6:30-9:30 pm ET	ES5 vs. ES6+: Arguments Object, Rest, Spread, Arrow Functions, Week 1 Review
---	Thursday/Friday	---	Attend JSHP in person and/or watch JSHP: HOF & Callbacks + CSX
4	Saturday	12-6 ET	Advanced Higher-Order Functions w/ Diagramming, “Big Data” Project

Week 2+ Schedule

Number	Day	Time	Topics
5	Monday	No class - Memorial Day	Recursion...Recursion
6	Tuesday	6:30-9:30 pm ET	Algorithms, Complexity Analysis, Big O Notation
7	Wednesday	6:30-9:30 pm ET	Cumulative Review and Introduction to Closure
---	Thursday/	6:30-9:30 pm ET	Attend JSHP in person or watch JSHP: Closure + CSX
8	Saturday	12-6 pm ET	Advanced Closure w/ Diagramming, Final Assessment, Group Projects
9	Monday	6:30-8 pm ET	Group Project Presentations and Conclusion

The CS Prep Infrastructure

Zoom.us

- Lectures and pair programming

Csbin.io

- Pairing and problem sets

Repl.it

- Problem of the day and assessment

VSCode Liveshare / Repl

- Additional pairing outside of class

Slack

- Community
- Course Communication
- Code Review
- Resource Sharing
- Session Recordings
- GIFs

The Goals of CS Prep

Become a problem-solving engineer (i.e., not just a "technician") by

- Building projects with JavaScript
- Going under the hood of JavaScript (including ES6)

Practice excellent technical communication

Build a community of fellow learners around you

Learn skills used in the development world today

Learn how to learn...everything.

What we expect of you

SUPPORT YOUR COMMUNITY - THIS IS THE MOST IMPORTANT VALUE AT CODESMITH

- Grow yourself – and those around you
- Be professional and fully present
- Trust the process
- Contribute and ask questions - they will be valuable for others

What can you expect of us?

Empathetic Engineering

We will…

- Come prepared
- Listen to your questions, ideas, and feedback
- Support you as fellow learners
- Join you on the learning journey

Our Pillars:

Technical Communication
&
Empathetic Engineering

Why Technical Communication and Engineering Empathy?

Excellence in engineering empathy and technical communication directly correlate to excellence in…

- Codesmith Residency
- Software Engineering
- Engineering Leadership
- Learning….anything

Technical Communication

"Master the vocabulary, and you will have power over the material."

Embrace formal vocabulary; embrace learning.

Formal vocabulary represents a shared knowledge and experience, passed down to all practitioners of a craft or hobby.

Benefits:

- Embracing formal vocabulary subconsciously opens our minds to receive new knowledge.
- Get help from anyone with that same shared experience.
- Give help to anyone else seeking that same shared experience

Engineering Empathy

Inward (to yourself):

- Use Grit and your Growth Mindset
- Measure your progress by how far you've come since your started, not how close you are to the destination.
- First accept the struggle; then embrace the struggle
- Give yourself a break! This is supposed to be hard.

Outward (to your community):

- Recognize that while everyone's challenge is unique, everyone is being challenged. You may not be able to see it.
- Grow your fellow engineer.
- Grow your fellow learner.
- The best way to learn is by teaching, so teach while you learn!

Our Toolbox of Learning Excellence

Caution: these tools require conscious commitment to excellence in technical communication and engineering empathy in order for them to propel you towards your learning goals.

- Interactive Diagramming
- Pair Programming
- Technical Communication Videos
- Freeform Projects
- Code Review (Yes! You're already ready to do code review)

Best Practices in Coding

Writing clean, readable code is an important part of good communication! Here are some best practices to follow during CS Prep and beyond:

- Use proper indentation and spacing
- Semantically name variables
- Leave comments in your code
- Make strategic use of console logs to test your code as you write
- Refactor often - keep your code DRY!

JavaScript

Let’s go under the hood...

JavaScript

The language is...

- High-level
- Interpreted
- Dynamically typed
- Multi-Paradigm
  - Object-oriented
  - Functional
- Memory-managed
- Garbage collected

Its execution is...

- Single-threaded
- Synchronous
- Blocking

Where do we execute JavaScript Code?

Since JavaScript is an interpreted language, it needs a runtime environment. There are two distinct runtimes for JavaScript Code:

- The Browser (client-side)
- NodeJS (server-side)

In order to begin executing code, we need two items:

- The Global Execution Context
- A Call Stack to manage control of the interpreter

Execution Contexts & The Call Stack

Execution Contexts:

- Execution contexts always consist of a thread of execution and variable environment, or store of data.
- The Global Context is always first, and its variable environment is referred to as Global Memory
- Whenever a function is invoked, it creates a new function-level execution context
    - Its variable environment is referred to as Local Memory
- Multiple execution contexts can exist simultaneously

The Call Stack:

- Tells the interpreter which execution context is in control of the thread of execution
- It is a Stack (data structure following the LIFO model, opposite of a Queue)
- Each execution context is represented by a Stack Frame
- When a function is invoked, a new frame is pushed to the call stack.
- When a function returns, its frame is popped off the call stack, and the top most frame takes control of the interpreter.

Let’s Diagram!

Setup

- Call Stack
- Global Context
  - Thread of Execution
  - Variable Environment / Global Memory

Function Invoked

- Push a new frame to the call stack
- Create function-level execution context
  - Thread of Execution
  - Variable Environment / Local Memory

```
const myAge = 5;
const yourAge = 5;

function addAges(age1, age2) {
  return age1 + age2;
}

const sumOfAges = addAges(myAge, yourAge);
```

Functions and Return Values

Every function in JavaScript returns something.

- The return statement tells the interpreter that the invocation of the current function is complete.
- When the interpreter hits the return statement, the thread of execution leaves that context, its frame is popped off the call stack, and the thread of execution will enter the context belonging to the top most remaining frame on the stack.
- Question: what happens if there is no return statement?

```
function return5() {
  return 5;
}
// returns the number 5
```

```
function justReturn() {
  return;
}
// returns undefined
```

Functions and Side Effects

- A side effect is any application state change that is or remains observable outside a function invocation.
- Usually, developers try to minimize side effects.
- Side effects can make it more difficult to track down bugs or adapt to changing requirements.

Functions and Side Effects

- Examples:
  - Mutating arguments' values that exist outside of the function
  - Console.log
    - Return value: undefined
    - Side effect: outputs messages to the web console (a window into our code running that we as developers can use - but has no consequences in javascript)

How does the JavaScript interpreter work?

The interpreter runs through our code in two “passes”:

The Allocation or Creation Phase

- Variable and Function declarations are stored in memory, “hoisted” to the top of their scopes
- Variable declarations are stored with the label only
- Function declarations are stored with both the label and its definition.

The Assignment or Execution Phase

- This is the part we diagram.
- Values are assigned.
- Expressions are evaluated.
- Function invocations are executed.
- Synchronous. Single threaded. Blocking. Boom.

These two passes happen with every execution context created.

Hoisting

Allocation Phase:

- Variable thing is declared in global memory. Its current value is undefined.
- Function declaration sayHello is stored in global memory.
- Variable otherThing is declared in global memory. Its current value is undefined.

Assignment Phase:

- The string ‘hello’ is assigned to thing in global memory.
- The string ‘bye bye’ is assigned to otherThing in global memory.

```
var thing = 'hello';
function sayHello() {
  console.log('hello');
}
var otherThing = 'bye bye';
```

A quick note on functions:

Not Hoisted:

- Variable sayHello is declared in global memory. Its value is inaccessible and uninitialized.
- An anonymous function definition is assigned to the label sayHello in global memory.
- This behaves no differently from assigning a value to any other variable's label.

```
// function expression
const sayHello = function() {
  console.log('hello')
}
```

Hoisted:

- The entire function declaration sayHello is stored in global memory.
- Just chillin'.

```
// function declaration
function sayHello() {
  console.log('hello');
}
```

Hoisting...again

Allocation Phase:

- Variable thing is declared in global memory. Its current value is inaccessible and uninitialized.
- Function declaration sayHello is stored in global memory.

Assignment Phase:

- sayHello is invoked.
  - ‘Hello’ is logged to the console.
  - undefined is logged to the console.
- The string ‘hello’ is assigned to thing in global memory.
- sayHello is invoked.
  - ‘hello’ is logged to the console.
  - ‘hello’ is logged to the console.

```
sayHello();
var thing = 'hello';
function sayHello() {
  console.log('hello');
  console.log(thing);
}
sayHello();
```

Hoisting Part III

Allocation Phase:

- Variable thing is declared in global memory. Its current value is inaccessible and uninitialized.
- Function declaration sayHello is stored in global memory.

Assignment Phase:

- sayHello is invoked.
  - 'Hello' is logged to the console.
  - undefined is logged to the console.
- ReferenceError: thing is not defined is logged to the console.
- The program has now crashed and exited.

```
sayHello();
const thing = 'hello';
function sayHello() {
  console.log('hello');
  console.log(thing);
}
```

What happened here?

- Variables declared with var return undefined when referenced before their values are assigned.
- Variables declared with let and const throw a ReferenceError when referenced before their values are assigned.

Variables and their declarations

- Three variable initialization keywords in JavaScript: var, and in ES6, let and const.
- What happens to variables declared without an initialization keyword?
- Two ways to “change” stored data: mutation and reassignment.
- Primitive data types cannot be mutated.
    - Primitive Types: String, Number, Boolean, null, undefined, symbol
    - Ex: adding a character to a string
- Composite or reference data types can be mutated.
    - Composite Types: Objects, Functions
    - Ex: adding an element to an array

Variables and their declarations

- Variables declared with var…
  - Lexical (function-level) scope
  - Hoisted - access before assignment returns undefined
  - Can be reassigned

- Variables declared with let…
  - Block scope
  - Hoisted - access before assignment throws ReferenceError
  - Can be Reassigned

- Variables declared with const…
  - Block scope
  - Hoisted - access before assignment throws ReferenceError
  - Cannot be reassigned

Let and Const Examples

```
// block scoping
if (true) {
  const x = 5;
}
console.log(x);
// ReferenceError: X is not defined
```

```
// reassignment
let marioBro = 'Mario';
marioBro = 'Luigi';
console.log(marioBro) // 'Luigi'
```

```
// mutation
const nums = [1, 22, 3];
nums.push(18);
console.log(nums); // [1, 22, 3, 18];
```

```
// reassignment part 2
const marioBro = 'Mario';
marioBro = 'Luigi';
// TypeError: Assignment to constant variable.
```

Pair Programming Time!

Pair Programming Partners

The Navigator:

- Gives declarative instructions in plain English.
- Is responsible for using excellent technical communication and engineering empathy to explain and re-explain their vision as needed.
- Does NOT dictate code or implementation details.
- Does not correct typos :).

The Driver:

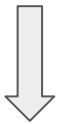
- Implements (converts to code) the navigator's instructions.
- Is responsible for speaking up and asking questions when stuck or unclear about the directions, and doing so with excellent technical communication and engineering empathy.
- Does not implement beyond the navigator's instructions, or "go rogue".

Arrow Functions (ES6+)

Arrow Functions and the Implicit Return

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

```
// function expression
const multiplyBy2 = function(num) {
  return num * 2;
}
```



```
// arrow function
const multiplyBy2 = (num) => {
  return num * 2;
}
```

```
// => and implicit return
const multiplyBy2 = num => num * 2;
```

What's on deck?

- Continue pairing outside of class as much as you want!
- Problem of the Day (POD) will go out tomorrow at 4pm Pacific/ 7pm Eastern
- Day 2 will begin in Zoom at 5pm Pacific/ 8pm Eastern
- Topics:
    - Objects and Arrays
    - Prototypal Inheritance
    - Object-Oriented Programming (OOP)
- Day 1: complete!