JavaScript Core Assessment Framework

Aram Shatakhtsyan and Michael Newman

CODESIGNAL Research

Abstract—At a time when educational resources for mastering JavaScript are increasingly abundant [1] [2] [3], when JavaScript is the most popular programming language [4] and when there are 98,000 [5] open jobs in the the US (and growing) that require familiarity with JavaScript, the need for a well-researched framework defining how to assess JavaScript skill is greater than ever. This paper proposes such a framework with the aim of enabling direct measurement of modern JavaScript development skill and helping the technical recruitment industry go beyond resumes in assessing this specific area of software development.

1 Introduction

The status quo in today's technical recruiting process is to use resume as a proxy for skill, which leads to biased, inefficient recruiting and evaluation practices. Some forward-looking companies are using automated assessment tools to create their own tests, which they send to candidates as the first step of the interview process. However, there are two major problems with that approach. Firstly, the internal teams creating these assessments are not experts in test design, which may lead to creation of tests that are not EEOC-compliant, or focus too much on some skills that might not be as important for the role while neglecting others that could be critical. Secondly, after internal teams spend a substantial amount of time creating these assessments, many of these assessments are eventually posted online on sites like Glassdoor and Stack Overflow, making the validity of the test results questionable.

The JavaScript SCA (Specialized Coding Assessment) framework described in this paper can be used to create standardized tests to measure JavaScript programming skills and knowledge. The tests are based on research that ensures they are highly consistent and EEOC-compliant. In addition, basing the test on a framework allows the SCA to be scaled with a large pool of questions that adhere to the same guidelines. On each test administration, questions can be randomly selected from the pool, reducing the risk that a test-taker gains an unfair advantage by memorizing the questions in advance.

The Coding Score obtained from the test is a singular measure meant to give recruiters, hiring managers and educators – as well as the test-takers themselves – a quick view of the test-taker's skills. It measures each test taker's code-writing and debugging skills as well as knowledge of JavaScript. The test should be administered in a proctored environment, and the solutions should be automatically checked for plagiarism.

In the next section we will discuss which topics are covered and will draw guidelines for creating tasks based on the framework. Afterwards, we will dive deeper into the reasoning of creating the framework and later discuss the possible outcomes and scoring of the test. More details on how Coding Score is calculated can be found in the paper for General Coding Assessment Framework [6].

2 The Framework

The maximum allowed completion time for the test is **80** minutes. Each test has **4** implementation tasks and **16** multiple choice quiz questions; in total, there are **20** tasks. Each test contains an implementation task (requires writing at least 25-40 lines of code, but the description clearly explains what needs to be done), a bugfix task (checking debugging abilities), two recovery tasks (meant to assess a deeper understanding of the JavaScript language by enforcing limitations on code changes), and 16 quiz tasks grouped into 4 categories by topics, with each topic containing 4 tasks.

2.1 Topics

The test will cover the following 8 topics of JavaScript.

1) **Basics**
   String manipulation, array manipulation, loops, function calls, passed arguments, objects, dictionaries.
2) **Callbacks**
   Functions as an argument, callbacks.
3) **Scopes & Hoisting**
   Different scopes, variable definition order, hoisting, `var` and `let`.
4) **Closures**
   Functions that return functions, closures.
5) **Exceptions**
   Throwing an exception, catching exceptions and re-throwing them, `try`/`catch`/`finally`.
6) **Promises**
   Asynchronous operations, throwing/catching promise exceptions, chaining promises, using `async`/`await`.
7) **Binding**
   Understanding the `this` binding, using `apply`/`call`/`bind`, using arrow functions.
8) **Prototypes**
   Classes, constructors, the `new` keyword, the prototype chain.

2.2 Implementation tasks

The 1st task

The expected time for solving this task is **20 minutes**. This is a code-writing task meant to assess the test-taker's implementation skill.

Expected knowledge

- Basic string manipulation.
  - Splitting a string into substrings.
  - Searching inside the string.
  - Comparing strings.
  - Concatenating strings.
- Basic array manipulation.
  - Iterating over an array.
  - Mapping an array using a function.
  - Reversing an array.
- Basic object manipulation.
  - Getting and setting object values, using dot notation or bracket notation.
  - Iterating over keys and/or values of an object.
  - Using an object as a dictionary.

✓ Can Include

- Tasks that require a combination of 3 to 5 basic concepts.
  - Splitting a string into substrings, modifying each substring and comparing with other strings.
  - Iterating over the given array, counting occurrences using a dictionary object and creating a new array based on the dictionary object.

✗ Should Exclude

- Anything that requires optimizing an algorithm.
- Closures.
- Async operations.

Example 2.2.1.

Given an array of words and a number K, return an array with length K where the ith element of the array is the number of unique prefixes with length i among the given words ($i = 1,2,\ldots,K$) (including only words that are at least i characters long).

```
function getUniquePrefixCount(words, len) {
  const prefixMap = {};
  words.forEach(word => {
    if (word.length >= len) {
      const prefix = word.substr(0, len);
      prefixMap[prefix] = true;
    }
  });
  return Object.keys(prefixMap).length;
}

function getFrequentWordCount(words, k) {
  return [...Array(k)].map(
    (_, i) => getUniquePrefixCount(words, i + 1)
  );
}
```

The 2nd task

The expected time for solving this task is **5 minutes**. This is a bugfix task meant to assess the test-taker's ability to find minor bugs in others' code.

Expected knowledge

- Includes everything from the code-writing task.
- Understanding how callbacks work.
- Understanding how `this` bindings work.

✓ Can Include

- Array prototype methods like `forEach`/`map`/`reduce`.
- Throwing errors and handling exceptions.
- Using the `new` keyword.

✗ Should Exclude

- Anything that requires optimizing an algorithm.

Example 2.2.2.

You are given a function `createLowerCaseSentence` that returns a sentence object which can be used to create a sentence from given strings that contain words. There is a bug on one of the lines - find and fix it.

```
function createSentence() {
  return {
    words: [],

    add(str) {
      str.split(' ').forEach(word => {
        const trimmedWord = word.trim()
        if (trimmedWord !== '') {
          this.words.push(trimmedWord);
        }
      });
    },

    get() {
      return this.words.join(' ') + '.';
    }
  };
};

function createLowerCaseSentence() {
  const sentence = createSentence();
  return {
    add(str) {
      sentence.add(str.toLowerCase())
    },

    get: sentence.get
  };
}
```

Line 27 should be changed to

```
    get: () => sentence.get()
```

The 3rd task

The expected time for solving this task is **10 minutes**. This is a recovery task meant to assess a deeper understanding of the JavaScript language, closures, bindings, dictionary objects.

Expected knowledge

- Closures

✓ Can Include

- Everything from the code-writing task.
- Implementing/using callbacks.
- Implementing a closure.
- Using functional Array prototype methods like forEach/map/reduce.
- Using function bindings and working with `this`.
- Using closures.
- Using dictionary objects.
- Using exceptions.
- Storing values in a dictionary object, using it as a counter and iterating over it.
- Modifying `this` binding of a function by using any binding mechanism.
- Throwing errors and handling exceptions.

✗ Should Exclude

- Anything that requires optimizing an algorithm.
- Async operations.
- Classes.

Example 2.2.3.

Implement the `createCountdown` function so that it returns a new countdown object with a `tick()` method that returns the current value and decreases it by 1 after each call and an `isOver` method that returns true or false indicating whether the countdown reached to 0.

```
function createCountdown(initialValue) {
  if (!Number.isInteger(initialValue) || initialValue < 1) {
    throw new Error('Invalid initial value');
  }
  let value = initialValue;
  let isOver = false;
  return {
    tick() {
      const returnValue = value;
      if (value > 0) {
        value--;
      } else {
        isOver = true;
      }
      return returnValue;
    },
    isOver() {
      return isOver;
    }
  };
}
```

The 4th task

The expected time for solving this task is **10 minutes**. This is a recovery task meant to assess a deeper understanding of exception handling, async operations and promises.

Expected knowledge

- Async operations and promises.

✓ Can Include

- Everything from the code-writing task.
- Implementing/using callbacks.
- Using functional Array prototype methods like `forEach/map/reduce`.
- Using exceptions.
- Using promises.
- Throwing errors and handling exceptions.

✗ Should Exclude

- Anything that requires optimizing an algorithm.
- Closures.
- Classes.

Example 2.2.4.

Implement a function that accepts an arbitrary number of promises as an input and returns a promise that is rejected with the number *o* if all of the input promises get rejected, or is resolved with the sum of the values of the resolved input promises otherwise.

```
function promiseCombiner(...promises) {
  if (promises.length === 0) {
    throw new Error('At least one argument is required');
  }

  let hasOneResolved = false;

  const mapPromise = p => p
    .then((e) => {
      hasOneResolved = true;
      return e;
    })
    .catch(() => 0);

  return Promise.all(promises.map(mapPromise))
    .then(vals => hasOneResolved ? vals.reduce((sum, e)
    ↪  => e + sum) : Promise.reject(0));
}
```

2.3 Quiz tasks

For the quiz tasks, 8 topics are divided into 4 groups, each containing 4 quiz tasks. So, in total there are 16 different quiz tasks. Table 1 shows the description of the groups and the task distribution between the topics.

Quiz Group 1: (tasks 5-8)

The expected total time for solving all the tasks of this group is **8 minutes**.

Expected knowledge

- The difference between passing by value and passing by reference.
- Function declarations and using passed arguments.
- Function calls.
- Callbacks.
- Using simple objects/maps.

Example 2.3.1.

Which of the listed options would prevent the original `u` object from being modified? (Select all that apply.)

```
function getNormalizedUser(user) {
  user.username = user.username.toLowerCase();
  return user;
}

let u = {
  id: 12,
  username: 'SomeUser',
};
let u2 = getNormalizedUser(u);
```

1) In `getNormalizedUser`, copy `user` using `JSON.parse(JSON.stringify(user))` then modify and return the copied version.
2) Change the implementation of `getNormalizedUser` to this:
   `return ...user, username: user.username.toLowerCase() ;`
3) Change the declaration from `let u` to `const u`.

Answer: Either (1) or (2) would work.

Example 2.3.2.

With the following code, which of the given options will run without an error? (Select all that apply.)

```
function sayHi(fullName, callback) {
  console.log(`Hi, ${fullName}!`);
  if (typeof callback === 'function') {
    callback();
  }
}

function sayBye(fullName, callback) {
  console.log(`Bye, ${fullName}!`);
  callback();
}

function printMessage(firstName, lastName, callback) {
  const fullName = `${firstName} ${lastName}`;
  if (typeof callback === 'function') {
    callback(fullName);
  }
}
```

```
A. printMessage("Andrew", "Johnson", sayHi);
B. printMessage("Mark", "Newman", sayBye);
C. printMessage("Jon", "Snow", sayYouKnowNothing);
D. printMessage("John", "Doe", x => console.log(x));
```

Answer: A and D.

Quiz Group 2: (tasks 9-12)

The expected total time for solving all the tasks of this group is **8 minutes**.

Groups		Quiz tasks count	Total tasks in group
JavaScript Fundamentals	Basics	2	4
	Callbacks	2	
Scopes, Hoisting and Closures	Scopes & Hoisting	2	4
	Closures	2	
Exceptions and Asynchronous operations	Exceptions	1	4
	Promises	3	
OOP, this and binding	Binding	2	4
	Prototypes	2	

Table 1: Quiz groups

Expected knowledge

- Multiple variable definitions in different scopes.
- Variable definition order and variable hoisting.
- Function definition order and hoisting.
- Functions that return functions.
- Closures.

Example 2.3.3.

Which of the listed options would make `counter.tick()` print distinct subsequent numbers when called consecutively? (Select all that apply.)

```
const counter = {
  next: 1,
  tick() {
    setTimeout(() => console.log(this.next), 1000);
    this.next++;
  }
};
```

1) Assign `this.next` to a local variable in the `tick()` method and pass the variable to `console.log` instead.
2) Wrap the `setTimeout` line inside another function and call it by passing `this.next`.
3) Move `this.next++` into the `setTimeout` callback, after `console.log`.

Answer: Either (1), (2) or (3) would work.

Example 2.3.4.

What will this code print?

```
function math(operation, x) {
  const OPERATIONS = {
    '*': (a, b) => a * b,
    '/': (a, b) => a / b,
    '+': (a, b) => a + b,
    '-': (a, b) => a - b,
  }

  return function(y) {
    return OPERATIONS[operation](x, y);
  }
}

const mul = math("*", 5);
const add = math("+", mul(2));

console.log(typeof add);
console.log(add(math("-", 25)(20)));
```

Answer:

```
function
15
```

Quiz Group 3: (tasks 13-16)

The expected total time for solving all the tasks of this group is **8 minutes**.

Expected knowledge

- Throwing an exception and what happens to the execution context.
- Catching exceptions and re-throwing them.
- Using the finally block after catch.
- Combining closures with asynchronous operations .
- Using a promise, throwing/catching promise exceptions, chaining promises.
- The event loop.

Example 2.3.5.

Given that `fetchUser` and `fetchPage` are both functions that return a Promise, `processData` is a regular synchronous function and `onLoaded` is a function that needs to be called after both user and page data are loaded, which of the listed options are true? (Select all that apply.)

```
// **Version 1**
fetchUser().then(user => {
  processData(user);
  return fetchPage();
}).then(page => {
  processData(page);
  onLoaded();
```

```
});
// **Version 2**
fetchUser().then(user => processData(user));
fetchPage().then(page => processData(page));
onLoaded();
// **Version 3**
Promise.all([fetchUser(), fetchPage()]).then(values => {
  processData(values[0]);
  processData(values[1]);
  onLoaded();
});
```

1) Version 1 will not always call `processData` for the page after the page data is loaded.
2) Version 2 will call `onLoaded` before the data is loaded.
3) Version 3 will not always call `processData` for the user data and the page data in the same order.

Answer: Only (2) is true.

Example 2.3.6.

What will this code print?

```
function prefix(str) {
  try {
    return str.substring(0, 3);
  } catch (err) {
    throw err;
    return "catch";
  }
}

try {
  console.log(prefix(5));
} catch (err) {
  console.log("error");
}
```

Answer:

`error`

Quiz Group 4: (tasks 17-20)

The expected total time for solving all the tasks of this group is **8 minutes**.

Expected knowledge

- Using `this` in a function.
- Using any mechanism to change `this` context: `bind`, arrow functions, or invoking with dot notation.
- Using `apply` and `call` to execute functions.
- Chaining method calls.
- How the `new` keyword works on built-in types like `Date` or `Array`.
- Creating a function that can be instantiated and used as a class, how constructor functions work.
- How prototype chains work, using `__proto__` and `prototype`.
- Using the `class` keyword.

Example 2.3.7.

Which of the listed options would make `obj.addMultiple(...)` work and correctly add all the numbers from the passed array to `obj.sum`? (Select all that apply.)

```
const obj = {
  sum: 0,
  addMultiple(numbers) {
    numbers.forEach(function (x) {
      this.sum += x;
    });
  }
};
```

1) Replace the callback function with an arrow function.
2) Bind `addMultiple` to `obj`.
3) Use a `for` loop instead of using `forEach` with a callback.

Answer: Either (1) or (3) would work.

Example 2.3.8.

What will this code print?

```
const Pair = function(first, second) {
  this.first = first;
  this.second = second;
};

Pair.prototype.setFirst = function(newFirst) {
  this.first = newFirst;
  return this;
};

Pair.prototype.setSecond = function(newSecond) {
  this.second = newSecond;
  return this;
};

const arr = [];
arr[0] = new Pair("first", "second");
arr[1] = arr[0].setFirst("second");
arr[2] = arr[1].setSecond("first");
if (arr[0] === arr[1] || arr[0] === arr[2] || arr[1] === arr[2]) {
  arr[0].setSecond("second").setFirst("first");
} else {
  arr[1].setFirst("third").setSecond("third");
}

console.log(arr[0].first);
console.log(arr[2].second);
```

Answer:

`first`

`second`

Topic	Importance
Basics	The basic building blocks for doing anything in JavaScript.
Callbacks	Callbacks are the foundation of asynchronous programming in JavaScript, both in the browser and in Node.
Scopes & Hoisting	Understanding scopes and hoisting is necessary for understanding how data is defined and manipulated in your code.
Closures	Closures are a powerful tool for writing expressive and elegant JavaScript code.
Exceptions	Clearly understanding exceptions and how to handle them is important in any language.
Promises	Asynchronous programming is necessary for any production JavaScript application, and promises along with async/await are the new standard for dealing with asynchronicity.
Binding	The `this` keyword is one of the most confusing mechanisms in JavaScript, but understanding it is necessary for using objects and classes effectively.
Prototypes	Prototypal inheritance and delegation are the cornerstone of re-using functionality and applying OOP principles in JavaScript.

Table 2: Topic importance

Level 0	Level 1	Level 2	Level 3	Level 4	Level 5	Level 6
600-649	**650-699**	**700-749**	**750-774**	**775-799**	**800-824**	**825-850**

Table 3: Score levels.

Score	Code-Writing	Quiz Groups	Description
700	**1st**	**1st**	The test-taker understands the fundamentals of JavaScript and can solve data manipulation tasks but does not have a deep understanding of important concepts.
735	**1st, 2nd**	**1st, 2nd**	The test-taker can write code using fundamental knowledge, can find bugs and is familiar with some of the most common topics like scopes and closures.
780	**1st, 2nd, 3rd**	**1st, 2nd, 3rd**	The test-taker understands and can use the most important JavaScript concepts like scopes, closures, exceptions and promises. They can find bugs and maintain code written by other developers.
830	**1st, 2nd, 3rd, 4th**	**1st, 2nd, 3rd, 4th**	The test-taker is an experienced professional JavaScript developer who understands the most advanced topics of the language.

Table 4: Score guide. The second column is the set of solved code-writing tasks; the third column is the set of fully solved quiz groups. The first column is the expected score. The actual score may vary depending on the performance.

3 Deep Dive

The topics of JavaScript SCA were selected by summarizing the following 3 main factors:

1) Topics covered in University courses (including online courses provided via edX [7] and Udacity [3]).
2) What are the most important skills that are required to build a JavaScript-based web application? We used MDN [1] and the book series You Don't Know Js [2].
3) What are the most common topics covered during technical interviews at successful US-based companies? We've used CareerCup [8] interview questions.

After analyzing all the data, we can came to the conclusion that the topics of Table 2 cover the most important aspects of JavaScript.

4 Results

The result of the standardized test is a Coding Score based on the test-taker's solved tasks, speed, coding style and failed attempts. Coding Score is a number from 600 to 850, and there are 7 score levels, as shown in Table 3.

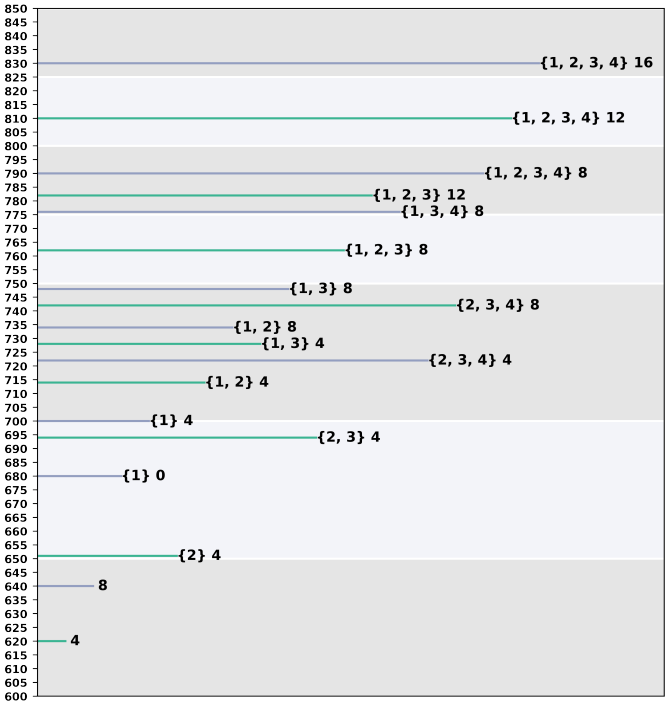

Figure 1: Most common outcomes.

Fig. 1 illustrates some of the most common scenarios. For simplicity we will assume that a test-taker either solves all the tasks from a group or does not solve any, and the groups are solved in order (so if one solved the 2nd group we will assume that they solved the 1st group too). Possible outcomes of Fig. 1 are the set of solved code-writing tasks and the number of solved quiz tasks (solved in the same order).

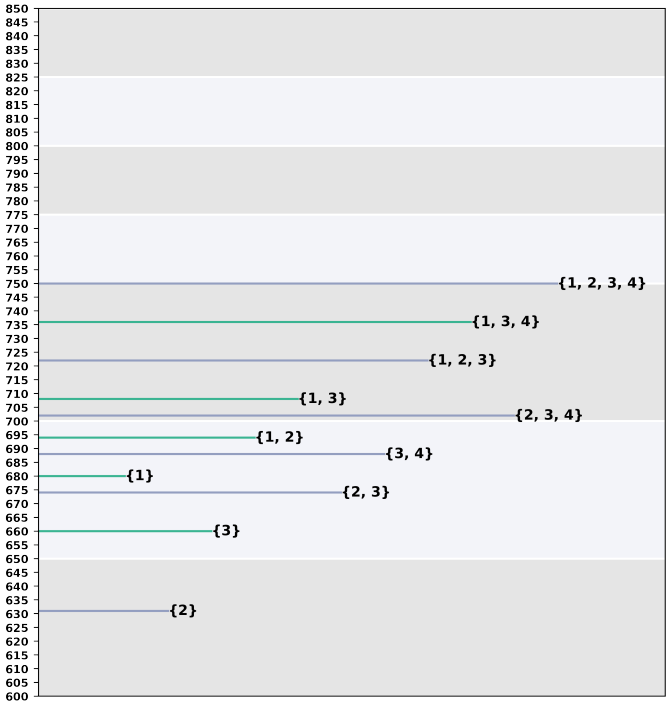

Figure 2: All score outcomes for code-writing tasks.

Solving all the tasks of a quiz group increases the score by 20 points, which means that all quiz tasks can increase the score by at most 80 points. Fig. 2 illustrates all possible outcomes for only code-writing tasks (the 3rd and 4th tasks are considered to be equivalent). The actual score may vary from the illustrated score by at most 20 points (depending on speed, code-quality and number of incorrect attempts).

Table 4 is a guide that describes some possible test results.

References

[1] MDN Web Docs. [Online]. Available: https://developer.mozilla.org/en-US/docs/Learn/JavaScript

[2] You Don't Know JS (6 book series). [Online]. Available: https://www.amazon.com/gp/product/B07FK9VBD7/?ie=UTF8&%2AVersion%2A=1&%2Aentries%2A=0

[3] Udacity, ES6 - JavaScript Improved. [Online]. Available: https://www.udacity.com/course/es6-javascript-improved--ud356

[4] Stack Overflow Developer Survey 2018. [Online]. Available: https://insights.stackoverflow.com/survey/2018/#most-popular-technologies

[5] LinkedIn, JavaSscript Developer jobs in United States. [Online]. Available: https://www.linkedin.com/jobs/sear

ch/?keywords=javascript&location=United%20States&locationId=us%3A0
[6] A. Sahakyan and T. Sloyan. General Coding Assessment Framework. [Online]. Available: https://codesignal.com/general-coding-assessment-framework/
[7] edX, javascript introduction. [Online]. Available: https://www.edx.org/course/javascript-introduction
[8] CareerCup, JavaScript Interview Questions. [Online]. Available: https://www.careercup.com/page?pid=javascript-interview-questions

Aram Shatakhtsyan is the CTO and Co-Founder of CodeSignal and has been working as a Software Engineer over the last 10 years, predominantly in full-stack JavaScript roles using Node, React, MeteorJS and other related technologies. He also has various achievements from programming competitions, including a Silver medal from International Olympiad in Informatics held in Zagreb, Croatia in 2007.

Michael Newman is the Director of Engineering at CodeSignal and has been programming professionally in JavaScript – across multiple frameworks, in-browser and in Node – for more than five years, on production applications ranging from EMR clinical documentation tools to virtual reality call center applications to technical assessment platforms.