JavaScript Callbacks: Introducing Concurrency through Callbacks and Closures

Kenny Hunt
Department of Computer Science
The University of Wisconsin - La Crosse
La Crosse, WI, 54601
khunt@uwlax.edu

Abstract

Computer Science educators have long recognized that teaching students how to write correct programs is difficult, especially when those programs involve parallel constructs. Educators have proposed numerous pedagogical innovations for teaching concurrency; typically involving specialized languages, language-level constructs or framework libraries. This paper show how fundamental aspects of concurrency can be introduced using plain JavaScript. JavaScript is the world's most common programming language, has readily available tool support, is executable within any modern web browser, provides no language-level concurrency constructs, and has conventional design patterns that solve concurrent design problems.

JavaScript supports functions as a data type and hence treats functions as first-class objects. Functions can therefore be created dynamically and passed as arguments and returned as results from other functions. JavaScript allows for the nesting of functions; specifically allowing a function to be created anywhere an expression is allowed. A closure is a function that is defined within an enclosing block and refers to a variable declared within that enclosing block but also outside of its internal scope. Closures are therefore functions that retain their enclosing environment even if the lifetime of that external environment has expired.

Callback functions and AJAX service calls provide the basic architectural building blocks for writing concurrent and parallel code in JavaScript. A callback function is a fragment of code that must be executed only at the conclusion of other code of indeterminate duration has completed. This paper argues that callback functions can be used to introduce parallelism at an early level in a CS curriculum.

1 Introduction

Computer Science educators have long recognized that teaching students how to write correct programs is difficult, especially when those programs involve parallelism and concurrency. Educators have proposed numerous pedagogical innovations for teaching concurrency; typically involving specialized languages [3], language-level constructs [8] or large scaffolding libraries [4] [6] [1].

This paper argues that fundamental aspects of parallelism and concurrency can be introduced using plain JavaScript. JavaScript is the world's most common programming language, has readily available tool support, is executable within any modern web browser, provides no language-level concurrency constructs, and most surprisingly is single-threaded. This combination of qualities grant Javascript significant appeal within an educational setting. The language is widely used in industry, all students already have have access to numerous Javascript interpreters, and the simplicity of the language allows students to focus on the computational essence of an algorithm rather than wading through the oft-criticized syntactic jungle of languages like Java [5] [7].

The terms *concurrent* and *parallel* are largely synonymous in a non-technical context and even computing professionals often use these terms interchangeably [2]. Nonetheless, technical professionals tend to distinguish between concurrency and parallelism. *Concurrency* is defined as the property of a system in which a set of tasks can remain active and make progress at the same time. *Parallelism* is the state of a program or algorithm in which concurrency is exploited to support simultaneous execution on multiple processing elements. In other words, concurrency is in the algorithm; parallelism in the implementation [2]. In this paper, we adopt this distinction between *concurrency* and *parallelism* and argue that JavaScript can effectively illustrate fundamental aspects not only of concurrency but also of parallelism. Closures provide the essential language-level mechanism through which these concepts can be exhibited.

2 JavaScript Closures

JavsScript supports functions as first-class elements. A closure is an *inner function* that has access to all variables that are in the *outer function*, even if the lifetime of the inner function extends beyond that of the outer function.

2.1 Access to variables

Figure 1 defines a function $makeNameTag$ that encloses an inner function named $tagText$. This inner function is a closure and hence all code inside of that function has access to all variables declared in the enclosing function. Line 4, for example, refers to the variable $prefix$ that is declared outside of the $tagText$ function but within the outer function $makeNameTag$. Variable access extends even to variables that are declared as formal parameters of the outer function as illustrated by variables $first$ and $last$ on line 4. Line 9 shows an invokation of the $makeNameTag$ function where the value of variable $nameTag$ is ’Hello. My name is John Adams’.

```
function makeNameTag(first, last) {
  var prefix = 'Hello';
  function tagText( ) {
    return prefix + '. My name is ' + first + ' ' + last;
  }
  return tagText();
}

var nameTag = makeNameTag( 'John', 'Adams' );
```

Figure 1: The function $tagText$ is a closure.

2.2 Lifetime access

Figure 2 re-writes the $makeNameTag$ function such that only the first name is given as a formal parameter and the type of value returned is *function* rather than *string*. Line 9 applies the $makeNameTag$ function to the first name ’John’ and assigns the result to a variable named $johnLastTag$. Since $makeNameTag$ returns a function, it follows that $johnLastTag$ is a function, not a string, and that this function accepts a single input denoting the last name of the name tag. The $johnLastTag$ is applied to the last name ’Adams’ as shown on line 10. The resulting value of variable $johnAdams$ is the string ’Hello. My name is John Adams’. Line 11 once again invokes the $johnLastTag$ function; it to the last name ’Calvin’ resulting in the string ’Hello. My name is John Calvin’.

Note that each time the $johnLastTag$ is invoked, it is accessing variables declared in an outer function that is no longer active. Specifically, the $first$ and $prefix$ variables of the function $makeNameTag$ are available to the $johnLastTag$ function even though the $makeNameTag$ function has completed and is no longer active.

```
 1  function makeNameTag(first) {
 2    var prefix = 'Hello';
 3    function tagText( last ) {
 4      return prefix + '. My name is ' + first + ' ' + last;
 5    }
 6    return tagText;
 7  }
 8
 9  var johnLastTag = makeNameTag( 'John' );
10  var johnAdamsTag = johnLastTag( 'Adams' );
11  var johnCalvinTag = johnLastTag( 'Calvin' );
```

Figure 2: The function $tagText$ has permanent access to outer variables.

2.3 References not Values

Closures store references to the outer function's variables; they do not store actual values. This behavior is illustrated in Figure 3 where we re-write the outer function such that it returns two inner functions wrapped in an object. More specifically, the $makeNameTag$ function returns an object having properties: $tagText$ and $setPrefix$, both of which are inner functions.

The $makeNameTag$ function is once applied to the first name 'John' as shown in line 13. We then apply the $tagText$ function of the $johnLastTag$ object to the last name 'Adams' as shown on line 14. The result is the value 'Hello. My name is John Adams'. We then apply the $tagText$ function to the last name 'Calvin' as shown on line 16. Before this application, however, line 15 changes the value of the $prefix$ variable by invoking the $setPrefix$ function of the $johnLastTag$ object. Once the prefix has been changed, the resulting value of the $johnCalvinTag$ is then 'Bonjour. My name is John Calvin'.

```
 1  function makeNameTag(first) {
 2    var prefix = 'Hello';
 3    function tagText( last ) {
 4      return prefix + '. My name is ' + first + ' ' + last;
 5    }
 6
 7    return {
 8      tagText : tagText,
 9      setPrefix : function( p ) { prefix = p; }
10    }
11  }
12
13  var johnLastTag = makeNameTag( 'John' );
14  var johnAdamsTag = johnLastTag.tagText( 'Adams' );
15  johnLastTag.setPrefix( 'Bonjour' );
16  var johnCalvinTag = johnLastTag.tagText( 'Calvin' );
```

Figure 3: Closures store references, not values.

3 JavaScript Callbacks

Since functions are first-class citizens of JavaScript, functions can be passed as arguments to other functions. A callback function F_{cb} is a function that is passed to another function F_{async} such that when F_{async} completes its computation, the callback function F_{cb} is invoked on the result. We use the notation F_{async} to denote a function F_{async} that is understood to be a non-blocking function. In other words, when F_{async} is invoked, it initiates a (possibly) long computational process but returns immediately. The process that it initiates will terminate at some indeterminate future time. Since functions always return a value, the asynchronous function may return the value $undefined$, since the result of the computation is not yet available, while producing the result at some later time via invocation of the callback function.

3.1 The $setTimeout$ Function

The $setTimeout$ function is an asynchronous function that accepts two inputs: a callback function F_{cb} and a $delay$ denoting a time in milleseconds [9]. When invoked, $setTimeout$ will execute F_{cb} after $delay$ milliseconds. Figure 4 gives an example this $setTimeout$ behavior. Line 10 invokes $setTimeout$ by passing in an anonymous callback function that applies $printResult$ to the result object. The time delay $runtime$ is set to 500 milliseconds on line 1.

Note that the $setTimeout$ function returns immediately but will execute $printResult$ 500 milliseconds later. The value of x, the result of executing the $setTimeout$ function, is clearly not related to the value returned by $printResult$ since that function won't be called until long after $setTimeout$ has produced its value for x. While $setTimeout$ does return a meaningful value, that value is not relevant to our discussion and we will therefore disingenuously claim that the value of x is $undefined$.

```
 1  var delay = 500;
 2
 3  function printResult(result) {
 4    var runtime = Date.now() - result.start;
 5    console.log('invocation', result.n, 'finished in',
 6                runtime, 'ms after a delay of', result.delay);
 7    return runtime;
 8  }
 9
10  var result = { n:0, start:Date.now(), delay:delay };
11  var x = setTimeout( () => printResult(result), delay );
```

Figure 4: Example of the $setTimeout$ function

When the code of Figure 4 is invoked, the $printResult$ function will output a message that includes the actual time taken between the start of the code and its completion in addition to the time delay imposed by the $setTimeout$ function. These two timings will not typically be identical as shown in the sample output of Figure 5. Also, the $printResult$ function returns the actual runtime (given in milliseconds) but this returned value is unused.

```
invocation 0 finished in 504 ms after a delay of 500
```

Figure 5: Sample execution of Figure 4

4 JavaScript Concurrency

We can now write customized concurrent functions by wrapping calls to $setTimeout$ and using callbacks. While these functions are truly concurrent, they are contrived since the concurrent behavior relies on the injection of an unnecessary and arbitrary time delay. Nonetheless, the pattern that we introduce here is easily extened to accomodate truly parallel techniques that form the heart of modern JavaScript applications.

4.1 A Concurrent Function

Figure 6 defines $asyncFunction$; an asynchronous function that accepts a single integer n. When invoked, as it is on line 16, the function generates an arbitrary delay and prints a message to the console after that delay transpires. The value n is merely a label that keeps track of different invocations of the $asyncFunction$ itself. Of course, $asyncFunction$ returns the value $undefined$ immediately so that the value of $result$ is indeed always $undefined$.

```
 1  function asyncFunction( n ) {
 2    var delay = Math.floor( 100 + Math.random() * 900 );
 3
 4    function printResult( result ) {
 5      var runtime = Date.now() - result.start;
 6      console.log( 'invocation', n, 'finished in', runtime,
 7                   'ms after a delay of', delay );
 8      return runtime;
 9    }
10
11    var result = { n:n, start:Date.now(), delay:delay }
12    setTimeout( () => printResult( result ), delay );
13  }
14
15  for( var rank = 0; rank < 10; rank++ ) {
16    var result = asyncFunction( rank );
17  }
```

Figure 6: A contrived asynchronous function

The asynchronous behavior of this code is exhibited when the loop is executed. Although the $asyncFunction$ is sequentially invoked 10 times using ranks of 0 through 9, the output generated by these 10 sequential calls does not follow the order in which they are invoked. Additionally, the difference between time delay and actual time to complete varies across invocations. Figure 7 shows the output of a single execution of the code in Figure 6.

While Figure 6 does illustrate concurrency, the code is poorly designed since the $asyncFunction$ 1) fails to communicate the computed result back to the caller and 2) assumes that the computed result should be printed to the console. Functions are expected to communicate results back to the caller and allow the caller to decide what further actions to take on those

```
invocation 2 finished in 170 ms after a delay of 166
invocation 0 finished in 185 ms after a delay of 184
invocation 6 finished in 201 ms after a delay of 199
invocation 9 finished in 333 ms after a delay of 330
invocation 1 finished in 481 ms after a delay of 476
invocation 5 finished in 501 ms after a delay of 498
invocation 3 finished in 519 ms after a delay of 518
invocation 4 finished in 794 ms after a delay of 792
invocation 7 finished in 847 ms after a delay of 844
invocation 8 finished in 953 ms after a delay of 951
```

Figure 7: Sample execution of Figure 6

computed values. Callback functions provide a mechanism for correcting both of these design flaws.

4.2 Concurrency with Callbacks

Figure 8 re-writes $asyncFunction$ as a function that accepts two arguments: an integer value n and a callback function cb. The callback function accepts a single input, the computed result of the asyncFunction, and is executed after an arbitrary delay of between 100 and 999 milliseconds. Client code is then free to define its own callback function that 1) is given access to the computed result via the passed argument and 2) process the result in any what that the client requires. In this example, the callback function is defined in the scope of the client (lines 8-13) such that the client has access to the computed result via the callbacks formal parameter $result$.

```
 1  function asyncFunction( n, cb ) {
 2      var delay = Math.floor( 100 + Math.random() * 900 );
 3      var result = { n:n, start:Date.now(), delay:delay };
 4      setTimeout( () => cb( result ), delay );
 5  }
 6
 7  for( var rank = 0; rank < 10; rank++ ) {
 8    function printResult( result ) {
 9      var runtime = Date.now() - result.start;
10      console.log( 'invocation', result.n, 'finished in', runtime,
11                   'ms after a delay of', result.delay);
12      return runtime;
13    }
14    var result = asyncFunction( rank, printResult );
15  }
```

Figure 8: Concurrent function with a callback

5 JavaScript Parallelism

We can now transform the *asyncFunction* into a truly parallel function by using AJAX calls to offload computation to other servers. For this paper, we will eschew non-standard libraries and adhere to the rather pedestrian but standard *XMLHttpRequest* library [10]. This library can easily be used in both a Node and browser environment.

A well known web api located at *https://api.whitehouse.gov* allows third-party clients to create, view, and sign politically motivated petitions that may prompt action from the executive branch of the United States government. One endpoint, *api/v1/petititions/:id* produces a single *petition* document that describes a desired political outcome along with a count of the number of signatures the petition has garnered. The petition is identified by a unique *id* that can obtained elsewhere from the api. Figure 9 shows how can write parallel code that obtains numerous such petitions and prints them when they become available from the whitehouse server.

The function *getPetition* accepts a petition id and callback function. An HTTP AJAX call is generated and, at some indetermine later time later when the requested petition is made available by the whitehouse server, the callback function is invoked on the resulting data. This invocation occurs on line 5.

Lines 10-13 show how the *getPetition* function can be applied to several petition ids (these id's were active as of the date of this writing but may be inactive as of the date this document is read). The order in which these three petitions are printed depends solely on the load and performance of the whitehouse server and cannot be predicted.

```
 1  function getPetition(pid, cb) {
 2   var host = 'https://api.whitehouse.gov';
 3   var path = `/v1/petitions/${pid}.json`;
 4   var req = new XMLHttpRequest();
 5   req.addEventListener('load', () => cb( req.responseText ) );
 6   req.open("GET", host + path );
 7   req.send();
 8  }
 9
10  ['2434701', '2436006', '2451411'].forEach( id => {
11    var result = getPetition( id, (result) => {
12      console.log( result );
13    });
14  } );
```

Figure 9: Parallel function with a callback

The code of Figure 9 is truly parallel and is designed in a conventional manner for JavaScript applications. The *getPetition* method is invoked three times and each invocation is, in a sense, simultanenously active. While each of these invocations returns immediately, nonetheless their computed results are not made available until after a relatively large and indetermine delay during which time a great deal of computation is occuring to generate the results. It is almost certain that the *www.whitehouse.gov* server executes each request on different hardware; either different servers sitting behind a load balancer or different cores on a single server. In either case, these three invocations are simulatenously active and also executing simultaneously. Eventually, the server responds to the request for a *petition* document and replies to our code on line 5. Upon receipt of a response from any of the three $getPetition$ methods, the corresonding result is printed to the console window.

6 Conclusion

JavaScripts closures, in addition to the standard $setTimeout$ and $XMLHttpRequest$ functions, allow the introduction of concurrency and parallelism at a relatively early stage in a CS curriculum. JavaScript comes with very mild syntactic overhead and benefits greatly from similarity to widespread languages such as Java and C. Since students learning JavaScript *must* already learn closures since they are fundamental to JavaScript programming, students already have the knowledge base necessary to understand issues of concurrency. If students are also introduced to functions such as $setTimeout$ andlibraries such as the standardized $XMLHttpRequest$ library, students can be introduced to true parallelism. It is the authors experience that even upper-level students who learn JavaScript can benefit by requiring them to understand and author callback functions that are applied in an AJAX setting.

References

[1] BRUCE, K. B., DANYLUK, A., AND MURTAGH, T. Introducing concurrency in cs 1. In *Proceedings of the 41st ACM Technical Symposium on Computer Science Education* (New York, NY, USA, 2010), SIGCSE '10, ACM, pp. 224–228.

[2] CASSEL, L., LEBLANC, R., MCGETTRICK, A., AND WRINN, M. Concurrency and parallelism in the computing ontology. *SIGCSE Bull. 41*, 3 (July 2009), 402–402.

[3] DANN, W., COSGROVE, D., AND SLATER, D. Tutorial: Concurrency with alice 3 and java. In *Proceedings of the 46th ACM Technical Symposium on Computer Science Education* (New York, NY, USA, 2015), SIGCSE '15, ACM, pp. 78–79.

[4] HUNT, J. M., AND WILLISON, T. California speedway: A concurrent programming project for beginners. In *Proceedings of the 49th Annual Southeast Regional Conference* (New York, NY, USA, 2011), ACM-SE '11, ACM, pp. 7–12.

[5] MOTH, A. L. A., VILLADSEN, J., AND BEN-ARI, M. Syntaxtrain: Relieving the pain of learning syntax. In *Proceedings of the 16th Annual Joint Conference on Innovation and Technology in Computer Science Education* (New York, NY, USA, 2011), ITiCSE '11, ACM, pp. 387–387.

[6] REGES, S. Conservatively radical java in cs1. *SIGCSE Bull. 32*, 1 (Mar. 2000), 85–89.

[7] STEFIK, A., AND SIEBERT, S. An empirical investigation into programming language syntax. *Trans. Comput. Educ. 13*, 4 (Nov. 2013), 19:1–19:40.

[8] VON PRAUN, C. Parallel programming: Design of an overview class. In *Proceedings of the 2011 ACM SIGPLAN X10 Workshop* (New York, NY, USA, 2011), X10 '11, ACM, pp. 2:1–2:6.

[9] HTML5, a vocabulary and associated APIs for HTML and XHTML. Recommendation, W3C, Nov. 2014. http://www.w3.org/TR/REC-html5-20141028.

[10] XMLHttpRequest level 1. Note, W3C, Nov. 2014. http://www.w3.org/TR/NOTE-XMLHttpRequest-20161006.