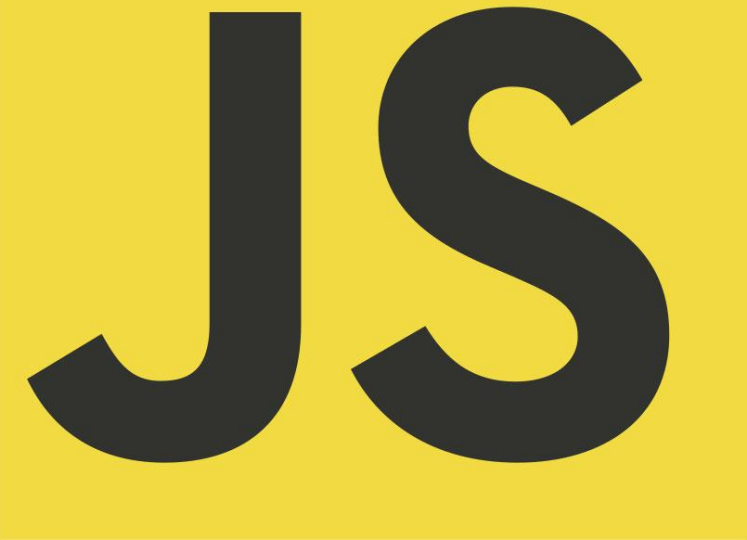
So you want to know more about Javascript?



Compiler Basics

- Although an interpreted language JavaScript is compiled. Just In Time.
- Basic Steps
  - Tokenizing/Lexing
  - Parsing
  - Code Generation

Tokenizing & Lexing

This step is breaking up the code into blocks meaningful to the compiler.

```
1  var bob = 2;
2
```

var	variable definition
bob	variable identifier
=	assignment operator
29	integer literal

LHS / RHS lookups

LHS lookup is on the left side of an =

```
1  bob = 2;
```

RHS lookup isn’t on the left side of an =

```
1  console.log(bob);
```

Hoisting

```
i = "test";

console.log(i)

var i;
```

```
var i;

i = "test";

console.log(i)
```

- var statements are hoisted to the top of the scope by the compiler.
- var a = “test” will be split and the var a hoisted
- duplicate var definitions are ignored
- functions are hoisted above variables

Scope

What is scope?

Basically it is the variables, objects and functions accessible.

```
//global scope
function doo(a) {
    //doo scope
    var b = a * 2;

    function boo (c) {
        //boo scope
        console.log(a, b, c);
        //end boo scope
    }

    boo(5);
    //end doo scope
}
//end glabal scope
```

Global Scope

```
var a = “Bob”
// code here can access a
function foo() {
    // code here can access a
}
```

Variable a is in the global scope, all scripts and functions can access and modify it.

What’s going on here?

```
1   function foo() {
2       a = "bob";
3   }
4
5   function bar() {
6       console.log(a);
7   }
8
9   foo();
10  bar();
11
```

What do you expect to see?

Why is this happening?

Run this and see if you are correct.

Lifetime of Variables

- The lifetime of a variable starts when is it first declared, either explicitly or implicitly.
- Local variables are freed when the function completes
- Global variables are freed when the page is closed or execution terminates

Exercise

- Given the following code, identify the LHS and RHS lookups
- Identify the line that will have a reference error
- What output should we expect?

```
1  function foo(a) {
2      var b = a;
3      return a + b;
4  }
5
6  var c = foo(2);
7  console.log(b);
```

Answers

```
1  function foo(a) {
2      var b = a;
3      return a + b;
4  }
5
6  var c = foo(2);
7  console.log(b);
```

We'd expect a ReferenceError on the last line, b will not be found, it is out of scope.

- LHS (3)
  - **c** = foo(2)
  - **a** = 2 (implicit, param)
  - **b** = a
- RHS (6)
  - **foo**(2)
  - **a** +
  - **+ b**
  - **= a**
  - **log(b)**
  - **console**

So, more Scope

- So far we have seen global scope and function scope, we'll look at other ways to a block scope.
- Leveraging Scope to hide variables and functions

There is no Block Scope

Many languages support block Scope, JavaScript is not one of them. Consider these two examples, what would you expect?

```
for (var i = 0; i< 3; i++) {
    console.log(i);
}
console.log(i);
```

```
var isOnFire = true;

if (isOnFire) {
    var currentAction = "exit building";
    console.log(currentAction);
}

console.log(currentAction);
```

Run the snippets and see if you are correct.

Leveraging Scope

```
function manipulateText(text) {
    if (text) {
        text = text.toUpperCase() + '!!!!!!!';
    }
    return text;
}

function shout(phrase) {
    console.log(manipulateText(phrase));
}


shout('hello');
```

Given this code, we want to assume manipulateText should be a private method.

Hiding with Scope

```
function manipulateText(text) {
    if (text) {
        text = text.toUpperCase() + '!!!!!!!';
    }
    return text;
}

function shout(phrase) {
    console.log(manipulateText(phrase));
}


shout('hello');
```

```
function shout(phrase) {

    //this is now hidden from the
    //global scope within the scope of shout
    function manipulateText(text) {
        if (text) {
            text = text.toUpperCase() + '!!!!!!!';
        }
        return text;
    }

    console.log(manipulateText(phrase));
}


shout('hello');
```

On the left we can call manipulateText directly from global scope, on the right manipulateText is within the scope of the shout function and cannot be called from the global scope, making it behave as a private function. Declare variables inside shout and they'll be variables only accessible within shout.

Modules

```
/**
 * Snippet 06 - "Modules"
 */

function myModule() {

    var a = 20;

    function log() {
        console.log("a is now: " + a);
    }

    function increment() {
        b = 20;
        a++;
        log();
    }

    function decrement() {
        a--;
        log();
    }

    clear = function() {
        a = 0;
        log();
    }

    return {
        decrement: decrement,
        increment: increment
    }
}

var module = myModule();

module.increment();
module.increment();
module.decrement();

clear();
```

A quick example on how we can put to use what we have learned so far.

How is is possible that we can call the clear method?

Modules

```
/**
 * Snippet 06 - "Modules"
 */

function myModule() {

    var a = 20;

    function log() {
        console.log("a is now: " + a);
    }

    function increment() {
        b = 20;
        a++;
        log();
    }

    function decrement() {
        a--;
        log();
    }

    clear = function() {
        a = 0;
        log();
    }

    return {
        decrement: decrement,
        increment: increment
    }
}

var module = myModule();

module.increment();
module.increment();
module.decrement();

clear();
```

A quick example on how we can put to use what we have learned so far.

How is is possible that we can call the clear method?

The clear variable is been leaked to the global scope, and the variable is assigned as a function. With the function defined within the scope of myModule it will have access to the scope of myModule, even when called from the global scope as we can see from the output.

```
a is now: 21
a is now: 22
a is now: 21
a is now: 0
[Finished in 0.114s]
```

We need some closure

You have just witnessed an example of closure, the clear method was executed outside of the scope it was declared within, but it was able to remember the scope it has access to.

Sounds simple, right?

So clear has access to the scope of myModule, which includes variable a. We return a reference to clear, which happens to be a function. This is then executed out of scope.

We would expect the scope of myModule to go away after it has executed, but it doesn't clear still holds a **closure** over it.

```
1  function myModule() {
2
3      var a = 20;
4
5      function clear() {
6          a = 0;
7          console.log(a);
8      }
9
10     return clear
11 }
12
13 var module = myModule();
14
15 module();
16
```

Real-life Closure

```
/**
 * Add the current request to the Print Request queue
 */
printRequestClick: function (role, queue) {
    this._clearMessages();

    var appConfig = this.get(APP_CONFIG),
        queueURL = appConfig.addToQueueUrl.replace(REQUEST_ID_TOKEN, this.get(REQUEST_ID)).replace(ROLE_TOKEN, role).replace(QUEUE_TYPE_TOKEN, queue),
        request = new Y.Merlins.JsonRequest({url: queueURL, method: Y.Merlins.AsyncRequest.METHOD.GET}),
        requestSuccessHandle,
        requestFailHandle,
        requestEndHandle,
        eventHandles = this.get(EVENT_HANDLES),
        inst = this;

    request.addParam(SELECTED_INSTITUTION_ID, this.get(REQUEST_INSTITUTION_ID));

    // Bind to the success event
    requestSuccessHandle = request.on(Y.Merlins.JsonRequest.Event.Success, function (e) {
        if (e.data.result === OK) {
            inst._printRequestSuccessCallback(queue);
        } else if (e.data.serviceErrors) {
            // Service errors
            inst.updateServiceErrors(e.data);
        } else {
            // Anything else is considered a failure
            inst._saveFailureCallback();
        }
    });

    // Bind to the failure event
    requestFailHandle = request.on(Y.Merlins.JsonRequest.Event.Failure, function () {
        inst._printRequestFailureCallback(queue);
    });

    requestEndHandle = request.on(Y.Merlins.JsonRequest.Event.End, function () {
        request.destroy();
    });

    // Send the request
    request.go();
    eventHandles.push(requestSuccessHandle);
```

⇐

This function holds a closure over the scope of printRequestClick by using the inst variable, so the entire scope chain must hang around until this callback function has finished with it.

So, what's the problem?

```
var i = 1;

while (i < 10) {

    setTimeout( function timer() {
        console.log(i);
    }, i * 500);

    i++;
}
```

So, now we know about scope & closures, what's the output here?

Huh?

```
10
10
10
10
10
10
10
10
10
[Finished in 4.533s]
```

Didn’t expect that?

Let’s look again.

The timer function holds a closure over the global scope with i in it. When the timeout executes the timer function i is already at 10

Can we fix it?

```
var i = 1;

while (i < 10) {
    (function() {
        var b = i;
        setTimeout( function timer() {
            console.log(b);
        }, i * 500);
    })();
    i++;
}
```

Each iteration gets a new scope with a variable b.

Summary

- Scope and how it is walked
- Leveraging Scope
- Hoisting
- Closures

Now you are ready to make some promises….