David Shariff

MANAGE THE UI PLATFORM TEAM @ AMAZON.COM

Follow @davidshariff 2,089 followers

I'm hiring! Send your resumé to:
primenow-hiring@amazon.com

Previously at Yahoo, RBS, Richi and Trend Micro.

Blog Homepage **View my Github** **See my LinkedIn**

Views are my own, and don't represent those of my employer.

What is the Execution Context & Stack in JavaScript?

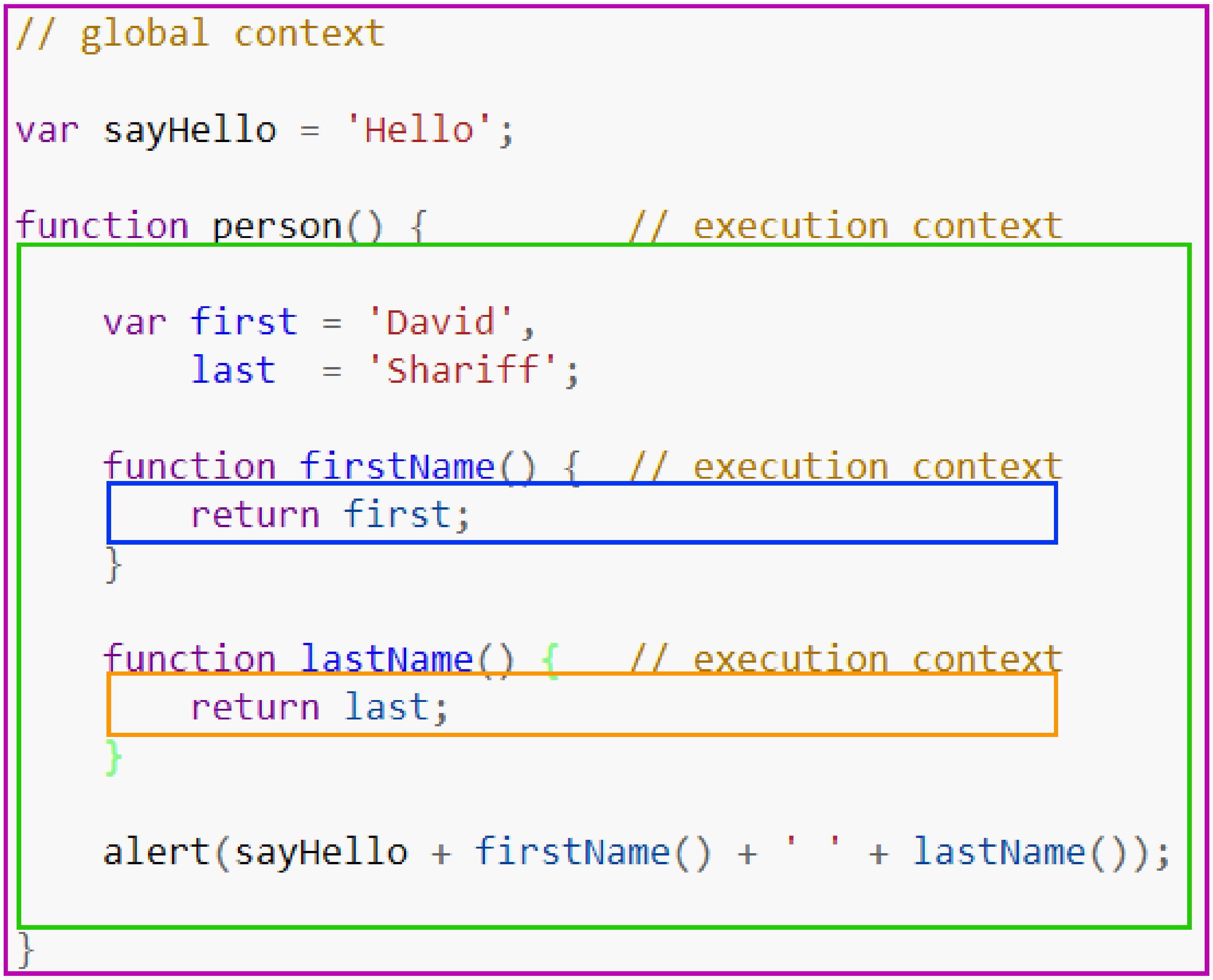
In this post I will take an in-depth look at one of the most fundamental parts of JavaScript, the `Execution Context`. By the end of this post, you should have a clearer understanding about what the interpreter is trying to do, why some functions / variables can be used before they are declared and how their value is really determined.

What is the Execution Context?

When code is run in JavaScript, the environment in which it is executed is very important, and is evaluated as 1 of the following:

- **Global code** – The default envionment where your code is executed for the first time.
- **Function code** – Whenever the flow of execution enters a function body.
- **Eval code** – Text to be executed inside the internal eval function.

You can read a lot of resources online that refer to `scope`, and for the purpose of this article to make things easier to understand, let's think of the term `execution context` as the envionment / scope the current code is being evaluated in. Now, enough talking, let's see an example that includes both `global` and `function / local` context evaluated code.



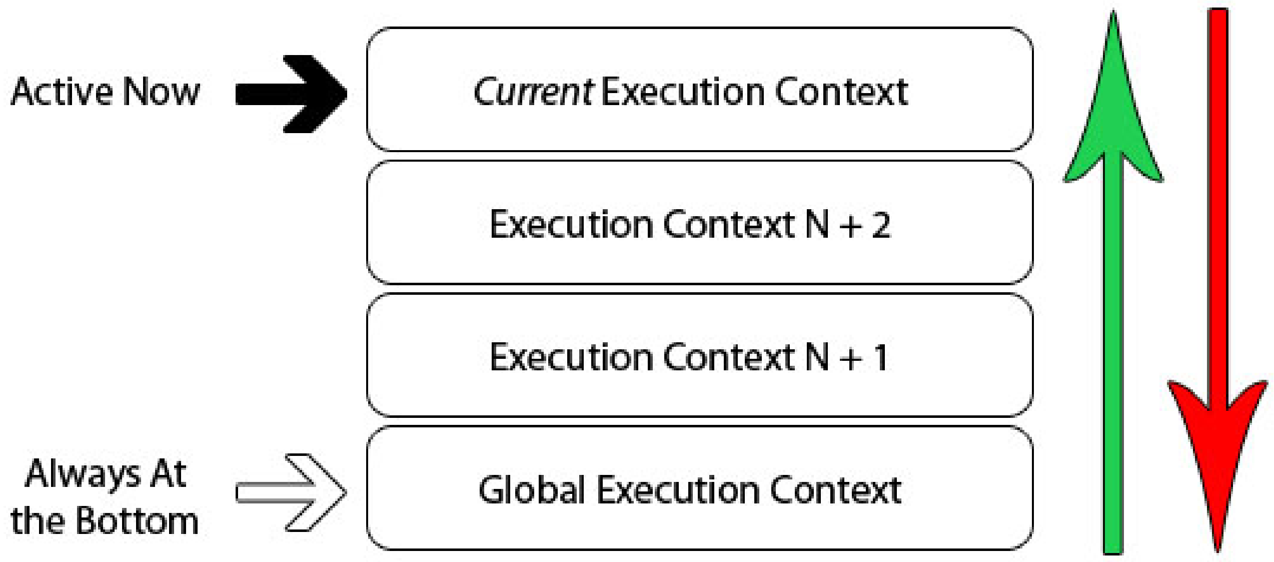
```
// global context

var sayHello = 'Hello';

function person() {          // execution context

    var first = 'David',
        last  = 'Shariff';

    function firstName() {   // execution context
        return first;
    }

    function lastName() {    // execution context
        return last;
    }

    alert(sayHello + firstName() + ' ' + lastName());

}
```

Nothing special is going on here, we have 1 `global context` represented by the purple border and 3 different `function contexts` represented by the green, blue and orange borders. There can only ever be 1 `global context`, which can be accessed from any other context in your program.

You can have any number of `function contexts`, and each function call creates a new context, which creates a private scope where anything declared inside of the function can not be directly accessed from outside the current function scope. In the example above, a function can access a variable declared outside of its current context, but an outside context can not access the variables / functions declared inside. Why does this happen? How exactly is this code evaluated?

Execution Context Stack

The JavaScript interpreter in a browser is implemented as a single thread. What this actually means is that only 1 thing can ever happen at one time in the browser, with other actions or events being queued in what is called the `Execution Stack`. The diagram below is an abstract view of a single threaded stack:



As we already know, when a browser first loads your script, it enters the `global execution context` by default. If, in your global code you call a function, the sequence flow of your program enters the function being called, creating a new `execution context` and pushing that context to the top of the `execution stack`.

If you call another function inside this current function, the same thing happens. The execution flow of code enters the inner function, which creates a new `execution context` that is pushed to the top of the existing stack. The browser will always execute the current `execution context` that sits on top of the stack, and once the function completes executing the current `execution context`, it will be popped off the top of the stack, returning control to the context below in the current stack. The example below shows a recursive function and the program's `execution stack`:

```
(function foo(i) {
    if (i === 3) {
        return;
    }
    else {
        foo(++i);
    }
}(0));
```

Global
Executing Now

The code simply calls itself 3 times, incrementing the value of i by 1. Each time the function `foo` is called, a new execution context is created. Once a context has finished executing, it pops off the stack and control returns to the context below it until the `global context` is reached again.

There are 5 key points to remember about the `execution stack`:

- Single threaded.
- Synchronous execution.
- 1 Global context.
- Infinite function contexts.
- Each function call creates a new `execution context`, even a call to itself.

Execution Context in Detail

So we now know that everytime a function is called, a new `execution context` is created. However, inside the JavaScript interpreter, every call to an `execution context` has 2 stages:

1. **Creation Stage** [when the function is called, but before it executes any code inside]:
   - Create the Scope Chain.
   - Create variables, functions and arguments.
   - Determine the value of `"this"`.
2. **Activation / Code Execution Stage**:
   - Assign values, references to functions and interpret / execute code.

It is possible to represent each `execution context` conceptually as an object with 3 properties:

```
executionContextObj = {
    'scopeChain': { /* variableObject + all parent execution context's variableObject
    'variableObject': { /* function arguments / parameters, inner variable and functic
    'this': {}
}
```

Activation / Variable Object [AO/VO]

This `executionContextObj` is created when the function is invoked, but *before* the actual function has been executed. This is known as stage 1, the `Creation Stage`. Here, the interpreter creates the `executionContextObj` by scanning the function for parameters or arguments passed in, local function declarations and local variable declarations. The result of this scan becomes the `variableObject` in the `executionContextObj`.

Here is a pseudo-overview of how the interpreter evaluates the code:

1. Find some code to invoke a function.
2. Before executing the `function` code, create the `execution context`.
3. Enter the creation stage:
   - Initialize the `Scope Chain`.
   - Create the `variable object`:
     - Create the `arguments object`, check the context for parameters, initialize the name and value and create a reference copy.
     - Scan the context for function declarations:

- For each function found, create a property in the `variable object` that is the exact function name, which has a reference pointer to the function in memory.
- If the function name exists already, the reference pointer value will be overwritten.

- Scan the context for variable declarations:
    - For each variable declaration found, create a property in the `variable object` that is the variable name, and initialize the value as undefined.
    - If the variable name already exists in the `variable object`, do nothing and continue scanning.

- Determine the value of `"this"` inside the context.

4. Activation / Code Execution Stage:
    - Run / interpret the function code in the context and assign variable values as the code is executed line by line.

Let's look at an example:

```
function foo(i) {
    var a = 'hello';
    var b = function privateB() {

    };
    function c() {

    }
}

foo(22);
```

On calling `foo(22)`, the `creation stage` looks as follows:

```
fooExecutionContext = {
    scopeChain: { ... },
    variableObject: {
        arguments: {
            0: 22,
            length: 1
        },
        i: 22,
        c: pointer to function c()
        a: undefined,
        b: undefined
    },
    this: { ... }
}
```

As you can see, the `creation stage` handles defining the names of the properties, not assigning a value to them, with the exception of formal arguments / parameters. Once the `creation stage` has finished, the flow of execution enters the function and the activation / code `execution stage` looks like this after the function has finished execution:

```
fooExecutionContext = {
    scopeChain: { ... },
    variableObject: {
        arguments: {
            0: 22,
            length: 1
        },
        i: 22,
        c: pointer to function c()
        a: 'hello',
        b: pointer to function privateB()
    },
    this: { ... }
}
```

A Word On Hoisting

You can find many resources online defining the term `hoisting` in JavaScript, explaining that variable and function declarations are *hoisted* to the top of their function scope. However, none explain in detail why this happens, and armed with your new knowledge about how the interpreter creates the `activation object`, it is easy to see why. Take the following code example:

```
(function() {

    console.log(typeof foo); // function pointer
    console.log(typeof bar); // undefined

    var foo = 'hello',
        bar = function() {
            return 'world';
        };

    function foo() {
        return 'hello';
    }

}());
```

The questions we can now answer are:

- **Why can we access foo before we have declared it?**
  - If we follow the `creation stage`, we know the variables have already been created before the `activation / code execution stage`. So as the function flow started executing, `foo` had already been defined in the `activation object`.
- **Foo is declared twice, why is foo shown to be `function` and not `undefined` or `string`?**
  - Even though `foo` is declared twice, we know from the `creation stage` that functions are created on the `activation object` before variables, and if the property name already exists on the `activation object`, we simply bypass the decleration.
  - Therefore, a reference to `function foo()` is first created on the `activation object`, and when the interpreter gets to `var foo`, we already see the property name `foo` exists so the code does nothing and proceeds.
- **Why is bar `undefined`?**
  - `bar` is actually a variable that has a function assignment, and we know the variables are created in the `creation stage` but they are initialized with the value of `undefined`.

Summary

Hopefully by now you have a good grasp about how the JavaScript interpreter is evaluating your code. Understanding the execution context and stack allows you to know the reasons behind why your code is evaluating to different values that you had not initially expected.

Do you think knowing the inner workings of the interpreter is too much overhead or a necessity to your JavaScript knowledge ? Does knowing the execution context phase help you write better JavaScript ?

Note: Some people have been asking about closures, callbacks, timeout etc which I will cover in the next post, focusing more on the Scope Chain in relation to the `execution context`.

Further Reading

- ECMA-262 5th Edition
- ECMA-262-3 in detail. Chapter 2. Variable object
- Identifier Resolution, Execution Contexts and scope chains