SAMSUNG

NOV 2018

# Practical JavaScript

Irene Libby

- Scope
- Closures
- Module Pattern
- Unit Test
- Asynchronous JS: Callbacks and Promises

- Javascript have two scopes: Global and Local
- **Global Scope** - variables and functions belongs to window object of the browser, can be accessed on any level of your code
- **Function Level Scope** - variables declared within a function are only accessible within that function or by functions inside that function

Examples: https://codesandbox.io/s/xovy28mkyo (scope.js)

# Scope - Tips

Practical Javascript

- Global variables: avoid using generic names to avoid clashing of scope with local variables
- Declare variables with explicit values is preferred even if it’s an “empty” or null value
- Nested functions: try to not use same variable or function names on any levels of your nested functions
- Instead of too many levels of nested functions, use CLOSURES!

- An inner function that has access to the outer (enclosing) function's variables scope access
- Allows maintaining state of contained variables
- Closure has three scope accesses: its own, outer function's variable and parameters and all global variables
- Think of Closures like writing a class that has private and public variables

Example: https://codesandbox.io/s/xovy28mkyo (closures.js)

- **Use Closures to:**
  - keep track of state of particular variables
  - use less global variables
  - Give proper access to variable and functions for your application

# Module Pattern

Practical Javascript

- Design pattern that emulates concept of “class” similar to other languages
- Allows us to create a public facing API methods we want to expose while encapsulating private variables and methods in a closure scope.
- Utilizes an immediately-invoked function expression

- A very common pattern used by libraries like jQuery, Underscore etc.
- Reduces clutter with the global namespace
- Enables unit testability of code

Example: https://codesandbox.io/s/xovy28mkyo (module.js)

- Check small snippets of code to ensure result of implemented logic is what the application expects
- Self documents behaviour of application
- May give indication of whether a dependency could be affected by code change while test fails
- Module pattern lends itself for code to be more “testable”

Example: https://codesandbox.io/s/xovy28mkyo

(module.test.js)

- JavaScript program is single threaded and all code is executed in a sequence
- While the execution of JavaScript is blocking, I/O operations are not (asynchronous non-blocking I/O model”)
- Asynchronous operation has a result that points to a function that will be executed once that result is ready and that function is what is called a “Callback” function.

# Asynchronous Javascript: Callbacks

Practical Javascript

- Callbacks are necessary but has a few pitfalls:
  - Passing of return values and errors involves nested callbacks
  - Heavily interdependent async calls can easily create ”Callback hell”
  - when callbacks are further nested, it’s hard to read/understand
  - Nested callbacks make unit testing impossible because we have no reference of the nested functions
  - We use Promises to “flatten” nested Callbacks

- An object which is used to hold the state of the result of an asynchronous action that will eventually be completed
- Based on the Promise A+ spec, these states are:
  - fulfilled (value returned)
  - rejected (error returned)
  - unfulfilled (in progress, but shall eventually become fulfilled or rejected)

- We use Promises because:
  - code will move from continuation-passing style to one where your functions return a value, called a promise, that returns eventual results of that operation.
  - can catching errors like synchronized functions
  - Are much more "unit testable" and clean to read

Example: https://codesandbox.io/s/xovy28mkyo (async.js)

# Asynchronous Javascript: Callback VS Promises

Practical Javascript

```
asyncCall(function(err, data1){
    if(err) return callback(err);
    anotherAsyncCall(function(err2, data2){
        if(err2) return calllback(err2);
        oneMoreAsyncCall(function(err3, data3){
            if(err3) return callback(err3);
            // are we done yet?
        });
    });
});
```

```
asyncCall().then(function(data1){
    return anotherAsyncCall();
}).then(function(data2){
    return oneMoreAsyncCall();
}).then(function(data3){
   // the third and final async response
}).catch(function (err) {
  // errors resulting from any calls above
});
```

SAMSUNG

NOV 2018

# Event Loop

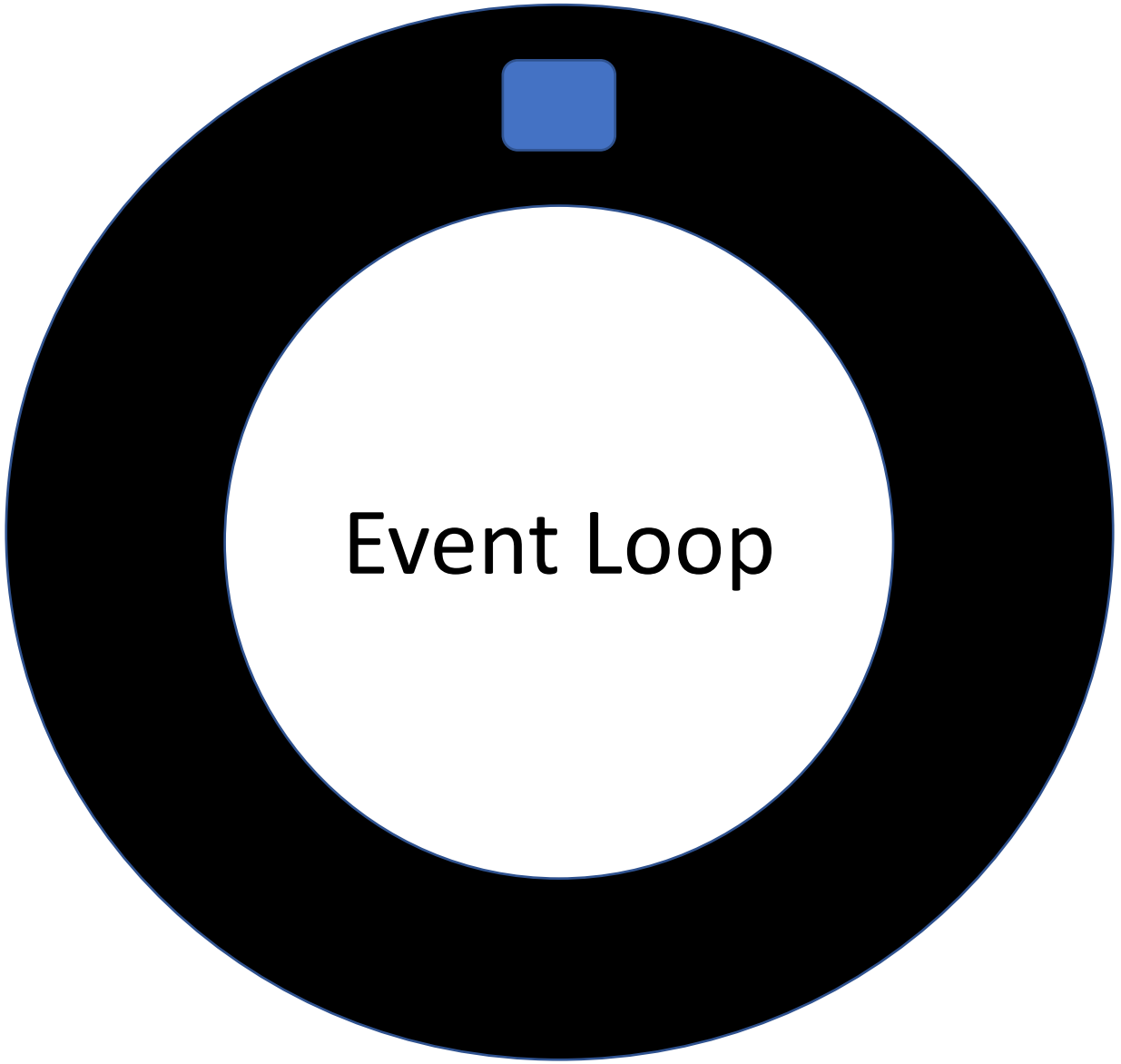
Raymond Leung

# What is JavaScript

JavaScript is Asynchronous

JavaScript is Single Threaded

How is this possible?




- Co-ordinates timing of all tasks and actions, from running code to repainting the DOM
- Follows the principle of “Run to Completion”
- Can only move tasks to Call Stack ONLY when the Call Stack is empty

# Main Thread

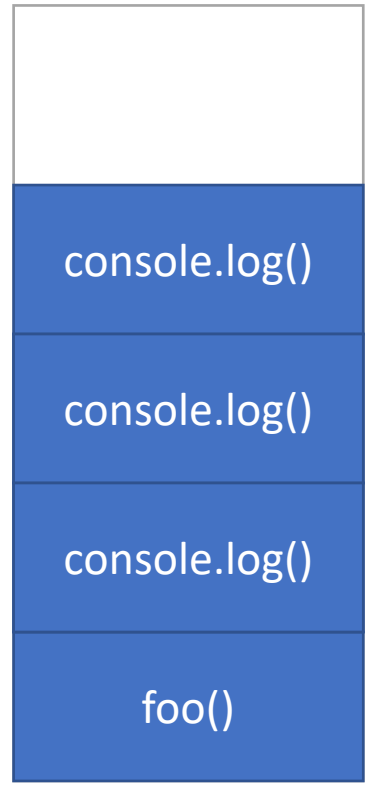
## Call Stack – Run to completion

```
function foo() {
    bar();
    console.log('hello from foo');
}

function bar() {
    baz();
    console.log('hello from bar');
}

function baz() {
    console.log('hello from baz');
}

foo();
```




Call Stack

~~Rule #1~~ The only rule:
Don’t block the main thread!

# Async Calls

## Callbacks - Message Queue

```
<button id='button'>while(true)</button>


const button = document.getElementById('button');

button.addEventListener('click', () => {
    while(true);
});
```

# Blocking Main Thread

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Aenean commodo ligula eget dolor. Aenean massa. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Donec quam felis, ultricies nec, pellentesque eu, pretium quis, sem. Nulla consequat massa quis enim. Donec pede justo, fringilla vel, aliquet nec, vulputate eget, arcu. In enim justo, rhoncus ut, imperdiet a, venenatis vitae, justo. Nullam dictum felis eu pede mollis pretium. Integer tincidunt. Cras dapibus. Vivamus elementum semper nisi. Aenean vulputate eleifend tellus. Aenean leo ligula, porttitor eu, consequat vitae, eleifend ac, enim. Aliquam lorem ante, dapibus in, viverra quis, feugiat a, tellus. Phasellus viverra nulla ut metus varius laoreet. Quisque rutrum. Aenean imperdiet. Etiam ultricies nisi vel augue. Curabitur ullamcorper ultricies nisi. Nam eget dui. Etiam rhoncus. Maecenas tempus, tellus eget condimentum rhoncus, sem quam semper libero, sit amet adipiscing sem neque sed ipsum. Nam quam nunc, blandit vel, luctus pulvinar, hendrerit id, lorem. Maecenas nec odio et ante tincidunt tempus. Donec vitae sapien ut libero venenatis faucibus. Nullam quis ante. Etiam sit amet orci eget eros faucibus tincidunt. Duis leo. Sed fringilla mauris sit amet nibh. Donec sodales sagittis magna. Sed consequat, leo eget bibendum sodales, augue velit cursus nunc,

```
while(true)
```

# What about Async?

setTimeout()

setInterval()

requestAnimationFrame()

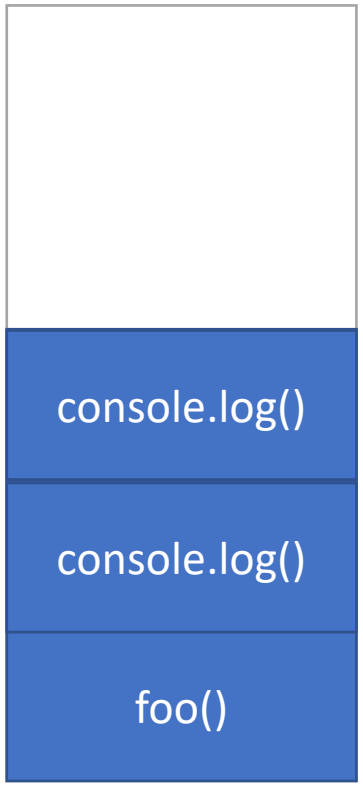
# Async Calls

## Callbacks - Message Queue

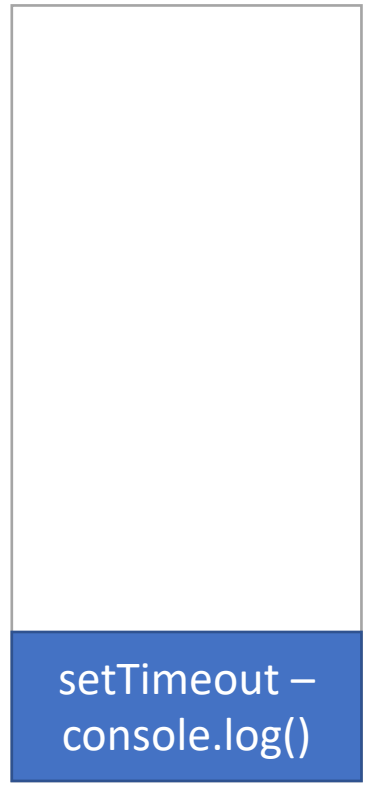
```
function foo() {
    bar();
    console.log('hello from foo');
}

function bar() {
    baz();
    console.log('hello from bar');
}

function baz() {
    setTimeout(() => {
        console.log('hello from baz');
    }, 0);
}
```




Call Stack




Message Queue

# Async Calls

## Callbacks - Message Queue

```
<button id='button'>while(true)</button>

function loop() {
    console.log('calling loop');
    setTimeout(loop, 0);
}

const button = document.getElementById('button');

button.addEventListener('click', () => {
    loop();
});
```

# Practical JavaScript

## Module Pattern

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Aenean commodo ligula eget dolor. Aenean massa. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Donec quam felis, ultricies nec, pellentesque eu, pretium quis, sem. Nulla consequat massa quis enim. Donec pede justo, fringilla vel, aliquet nec, vulputate eget, arcu. In enim justo, rhoncus ut, imperdiet a, venenatis vitae, justo. Nullam dictum felis eu pede mollis pretium. Integer tincidunt. Cras dapibus. Vivamus elementum semper nisi. Aenean vulputate eleifend tellus. Aenean leo ligula, porttitor eu, consequat vitae, eleifend ac, enim. Aliquam lorem ante, dapibus in, viverra quis, feugiat a, tellus. Phasellus viverra nulla ut metus varius laoreet. Quisque rutrum. Aenean imperdiet. Etiam ultricies nisi vel augue. Curabitur ullamcorper ultricies nisi. Nam eget dui. Etiam rhoncus. Maecenas tempus, tellus eget condimentum rhoncus, sem quam semper libero, sit amet adipiscing sem neque sed ipsum. Nam quam nunc, blandit vel, luctus pulvinar, hendrerit id, lorem. Maecenas nec odio et ante tincidunt tempus. Donec vitae sapien ut libero venenatis faucibus. Nullam quis ante. Etiam sit amet orci eget eros faucibus tincidunt. Duis leo. Sed fringilla mauris sit amet nibh. Donec sodales sagittis magna. Sed consequat, leo eget bibendum sodales, augue velit cursus nunc,

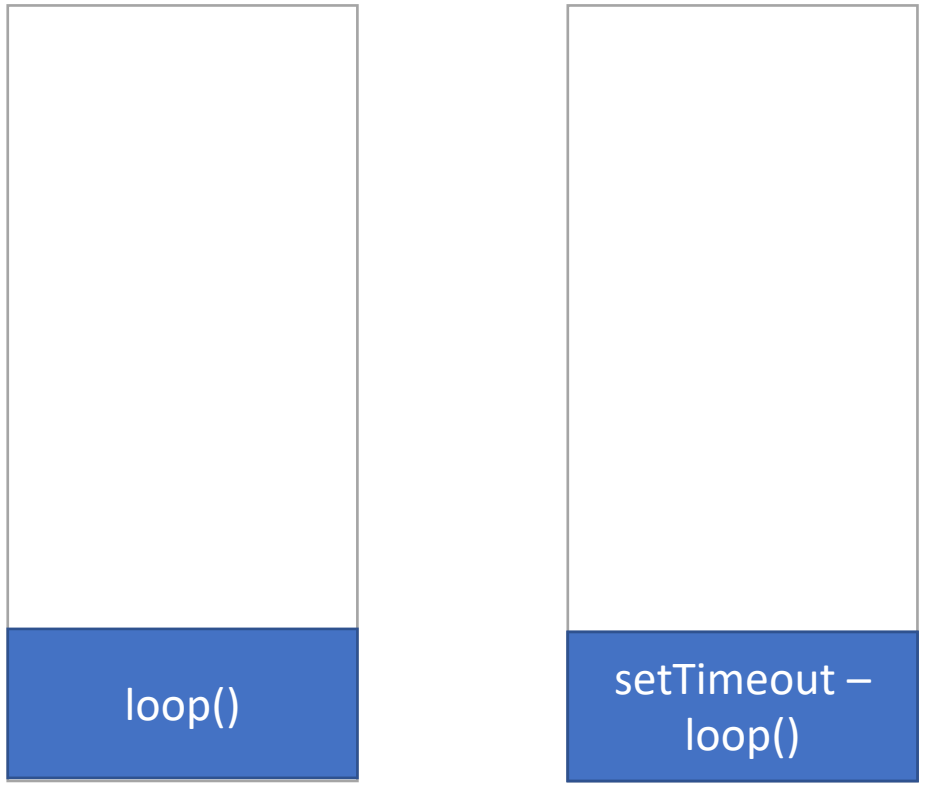
setTimeout(loop, 0)

# Async Calls

## Callbacks - Message Queue

```
function loop() {
    setTimeout(loop, 0);
}

loop();
```



# Async Calls

## Promises

```
setTimeout(() => {
    console.log('setTimeout done');
}, 0);

Promise.resolve().then(() => {
    console.log('promise done');
});
```

# Promises

## Micro Task Queue

- Promises resolve to a Micro Task rather than Task
- Micro Tasks run in a separate queue
- Micro Tasks are checked just before the current Event Loop tick is complete.
- Like Tasks, they can only be processed once the Call Stack is empty
- Unlike the Task Queue, all new spawned Micro Tasks will be processed before yielding the Call Stack

# Micro Tasks

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Aenean commodo ligula eget dolor. Aenean massa. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Donec quam felis, ultricies nec, pellentesque eu, pretium quis, sem. Nulla consequat massa quis enim. Donec pede justo, fringilla vel, aliquet nec, vulputate eget, arcu. In enim justo, rhoncus ut, imperdiet a, venenatis vitae, justo. Nullam dictum felis eu pede mollis pretium. Integer tincidunt. Cras dapibus. Vivamus elementum semper nisi. Aenean vulputate eleifend tellus. Aenean leo ligula, porttitor eu, consequat vitae, eleifend ac, enim. Aliquam lorem ante, dapibus in, viverra quis, feugiat a, tellus. Phasellus viverra nulla ut metus varius laoreet. Quisque rutrum. Aenean imperdiet. Etiam ultricies nisi vel augue. Curabitur ullamcorper ultricies nisi. Nam eget dui. Etiam rhoncus. Maecenas tempus, tellus eget condimentum rhoncus, sem quam semper libero, sit amet adipiscing sem neque sed ipsum. Nam quam nunc, blandit vel, luctus pulvinar, hendrerit id, lorem. Maecenas nec odio et ante tincidunt tempus. Donec vitae sapien ut libero venenatis faucibus. Nullam quis ante. Etiam sit amet orci eget eros faucibus tincidunt. Duis leo. Sed fringilla mauris sit amet nibh. Donec sodales sagittis magna. Sed consequat, leo eget bibendum sodales, augue velit cursus nunc,

Promises

# Event Loop

## Queue timing and processing

**Call Stack** – Runs one thing at a time.  Runs to completion so will not yield.

**Micro Task** – Will pop onto the Call Stack only when the Call Stack is empty. Micro Task Queue is evaluated at the end of the current loop tick.  Will not only run to completion, but all new Micro Tasks that were spawned from this Micro Task will be evaluated before yielding.

**Message Queue** – Will pop onto the Call Stack only when the Call Stack is empty.  This is evaluated at the beginning of the next Event Loop tick.  Only one Task in the queue will be evaluated.  Any new Tasks that were spawned from this will be put into the Queue to be processed whenever the Message Queue can be evaluated again.

# JavaScript is weird

```
button.addEventListener('click', () => {
    Promise.resolve().then(() => { console.log('Micro Task 1') });
    console.log('Listener 1');
});

button.addEventListener('click', () => {
    Promise.resolve().then(() => { console.log('Micro Task 2') });
    console.log('Listener 2');
});


button.click()
```

# Resources

**Jake Archibald**
In The Loop – JSConf Asia 2018
https://www.youtube.com/watch?v=cCOL7MC4Pl0

**Shelley Vohr**
Asynchrony: Under the Hood
https://www.youtube.com/watch?v=SrNQS8J67zc

# Questions?