

# SWE 432 -Web Application Development

Fall 2021

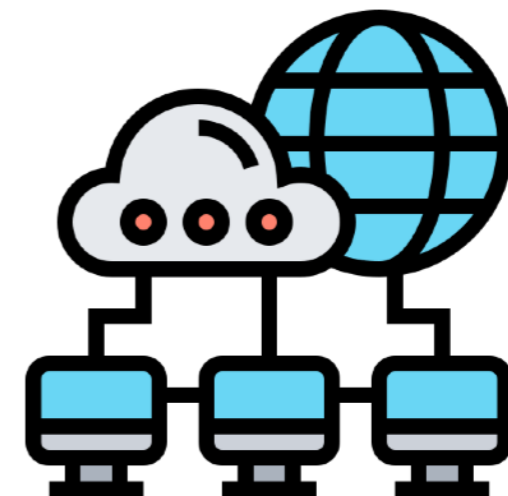


George Mason  
University

---

Dr. Kevin Moran

## Week 3: Asynchronous Programming





# Administrivia

- HW Assignment 1 - Due Today Before Class
- HW Assignment 2 - Out on Thursday, will discuss next class
- Quiz #2: Discussion



# Quiz #2 Review

Given the code snippet below, write code that will log myProp to the console.

```
var object = {  
  foo: 'bar',  
  age: 42,  
  baz: {myProp: 12} }
```



# Quiz #2 Review

Given the code snippet below, write code that will log myProp to the console.

```
var object = {  
  foo: 'bar',  
  age: 42,  
  baz: {myProp: 12} }
```

```
console.log("MyProp: " + object.baz.myProp)
```

Output: "MyProp: 12"



# Quiz #2 Review

Given the code snippet below, using a template literal to access the value of the first (zeroth) element, print the message “Population of ”, and log the name and population of each element.

```
let cities =  
[  
  {name: 'Fairfax', population: 24574},  
  {name: 'Arlington', population: 396394},  
  {name: 'Centreville', population: 71135}];
```



# Quiz #2 Review

Given the code snippet below, using a template literal to access the value of the first (zeroth) element, print the message “Population of ”, and log the name and population of each element.

```
let cities =  
[  
  {name: 'Fairfax', population: 24574},  
  {name: 'Arlington', population: 396394},  
  {name: 'Centreville', population: 71135}];
```

```
console.log(`Population of ${cities[0].name}: ${cities[0].population}`);
```

output: “Population of Fairfax: 24574”



# Quiz #2 Review

What is the output of the code snippet listed below?

```
function makeAdder(x) {  
  return function(y) {  
    return x + y;  
  };  
}  
  
var add5 = makeAdder(5);  
var add10 = makeAdder(10);  
  
console.log(add5(2));  
console.log(add10(2));
```



# Quiz #2 Review

What is the output of the code snippet listed below?

```
function makeAdder(x) {  
  return function(y) {  
    return x + y;  
  };  
}  
  
var add5 = makeAdder(5);  
var add10 = makeAdder(10);  
  
console.log(add5(2));  
console.log(add10(2));
```

Output: “7  
12”





# Review: Closures

- Closures are expressions that work with variables in a specific context
- Closures contain a function, and its needed state
  - Closure is a stack frame that is allocated when a function starts executing and not freed after the function returns
- That state just refers to that state by name (sees updates)



# Review: Closures

- Closures are expressions that work with variables in a specific context
- Closures contain a function, and its needed state
  - Closure is a stack frame that is allocated when a function starts executing and not freed after the function returns
- That state just refers to that state by name (sees updates)

```
var x = 1;
function f() {
  var y = 2;
  return function() {
    console.log(x + y);
    y++;
  };
}
var g = f();
g();           // 1+2 is 3
g();           // 1+3 is 4
```



# Review: Closures

- Closures are expressions that work with variables in a specific context
- Closures contain a function, and its needed state
  - Closure is a stack frame that is allocated when a function starts executing and not freed after the function returns
- That state just refers to that state by name (sees updates)

```
var x = 1;
function f() {
  var y = 2;
  return function() {
    console.log(x + y);
    y++;
  };
}
var g = f();
g();           // 1+2 is 3
g();           // 1+3 is 4
```

This function attaches itself to x and y so that it can continue to access them.

# Review: Closures

- Closures are expressions that work with variables in a specific context
- Closures contain a function, and its needed state
  - Closure is a stack frame that is allocated when a function starts executing and not freed after the function returns
- That state just refers to that state by name (sees updates)

```
var x = 1;
function f() {
  var y = 2;
  return function() {
    console.log(x + y);
    y++;
  };
}
var g = f();
g(); // 1+2 is 3
g(); // 1+3 is 4
```

This function attaches itself to x and y so that it can continue to access them.

# Review: Closures

- Closures are expressions that work with variables in a specific context
- Closures contain a function, and its needed state
  - Closure is a stack frame that is allocated when a function starts executing and not freed after the function returns
- That state just refers to that state by name (sees updates)

```
var x = 1;
function f() {
  var y = 2;
  return function() {
    console.log(x + y);
    y++;
  };
}
var g = f();
g();           // 1+2 is 3
g();           // 1+3 is 4
```

This function attaches itself to x and y so that it can continue to access them.

It “**closes up**” those references



# Closures

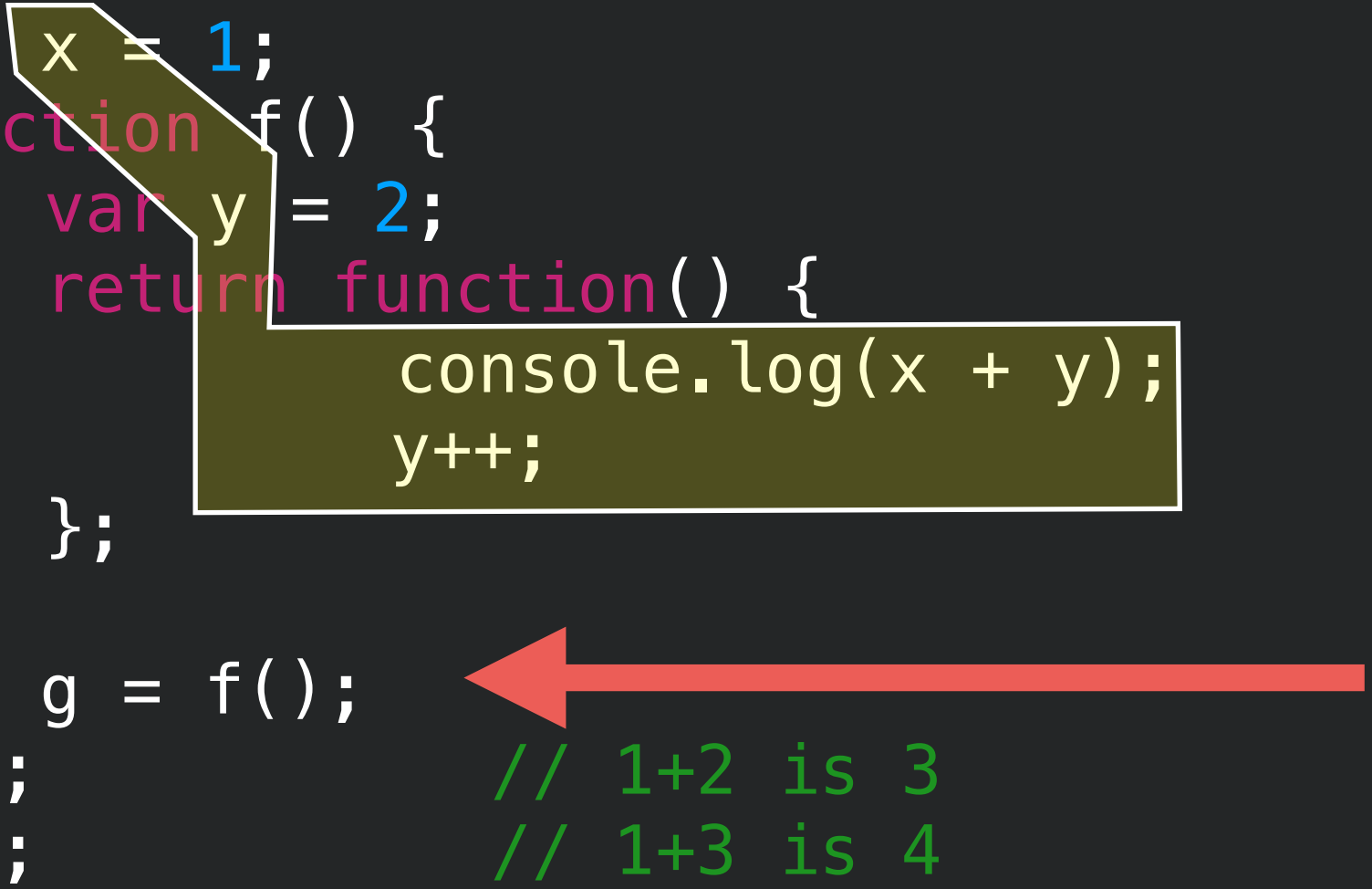


# Closures

```
var x = 1;
function f() {
  var y = 2;
  return function() {
    console.log(x + y);
    y++;
  };
}
var g = f();
g();           // 1+2 is 3
g();           // 1+3 is 4
```

# Closures

```
var x = 1;
function f() {
  var y = 2;
  return function() {
    console.log(x + y);
    y++;
  };
}
var g = f();
g(); // 1+2 is 3
g(); // 1+3 is 4
```



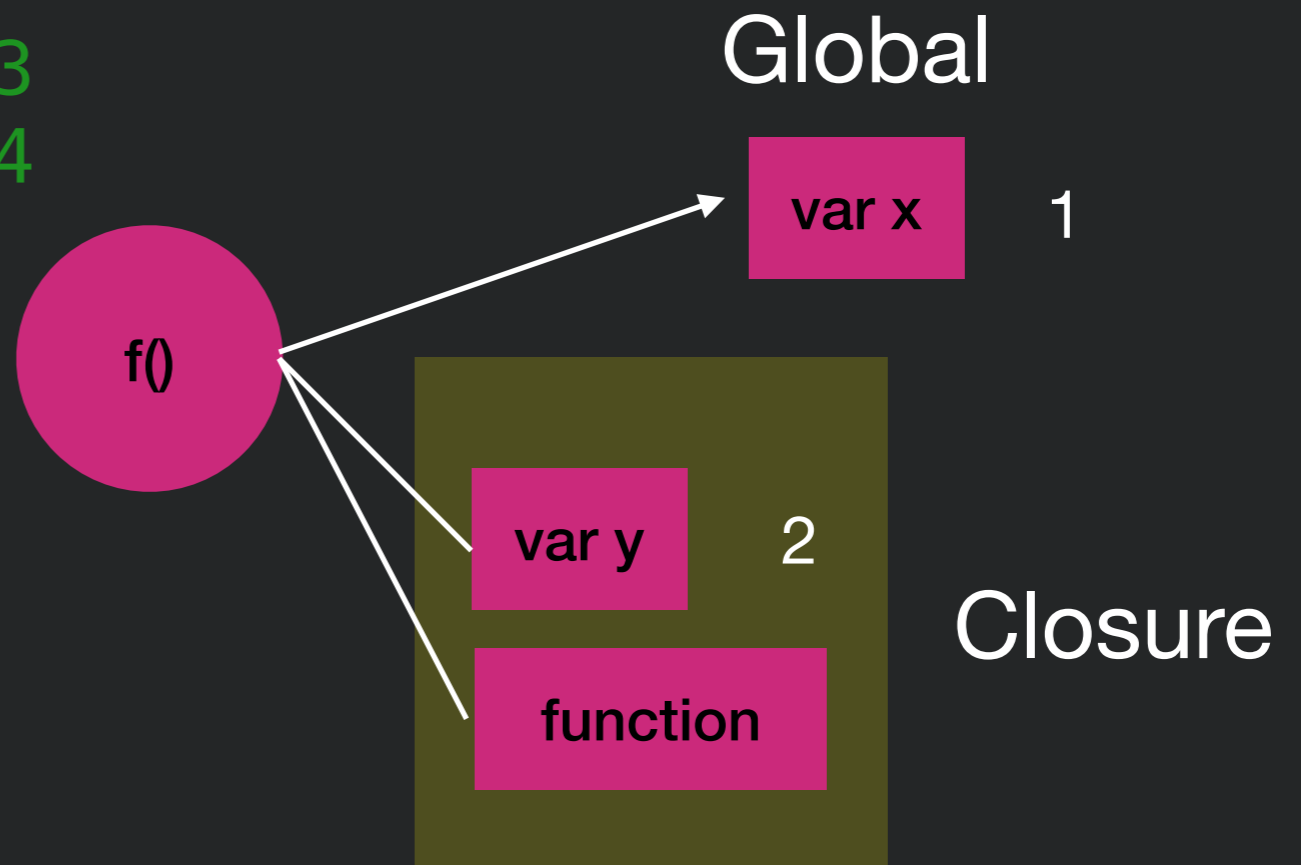


# Closures

```
var x = 1;
function f() {
  var y = 2;
  return function() {
    console.log(x + y);
    y++;
  };
};
```

```
var g = f();
g();
g();
```

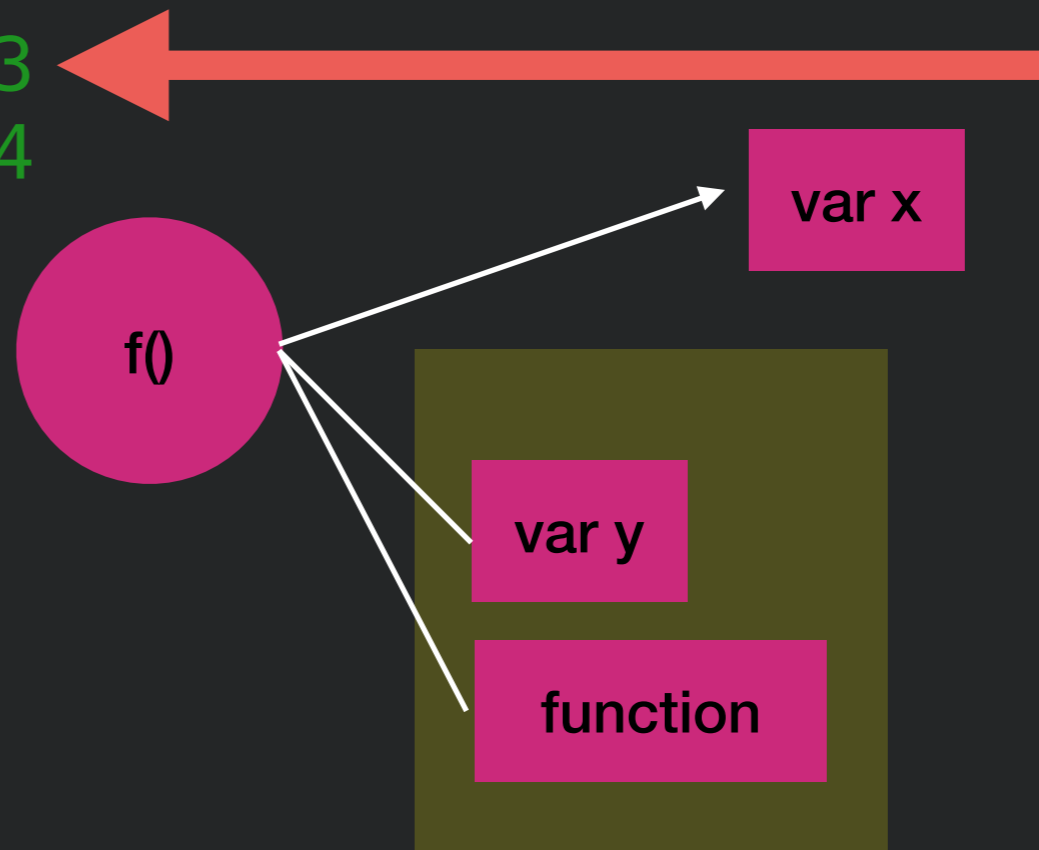
```
// 1+2 is 3
// 1+3 is 4
```



# Closures

```
var x = 1;  
function f() {  
  var y = 2;  
  return function() {  
    console.log(x + y);  
    y++;  
  };  
}
```

```
var g = f();  
g(); // 1+2 is 3  
g(); // 1+3 is 4
```

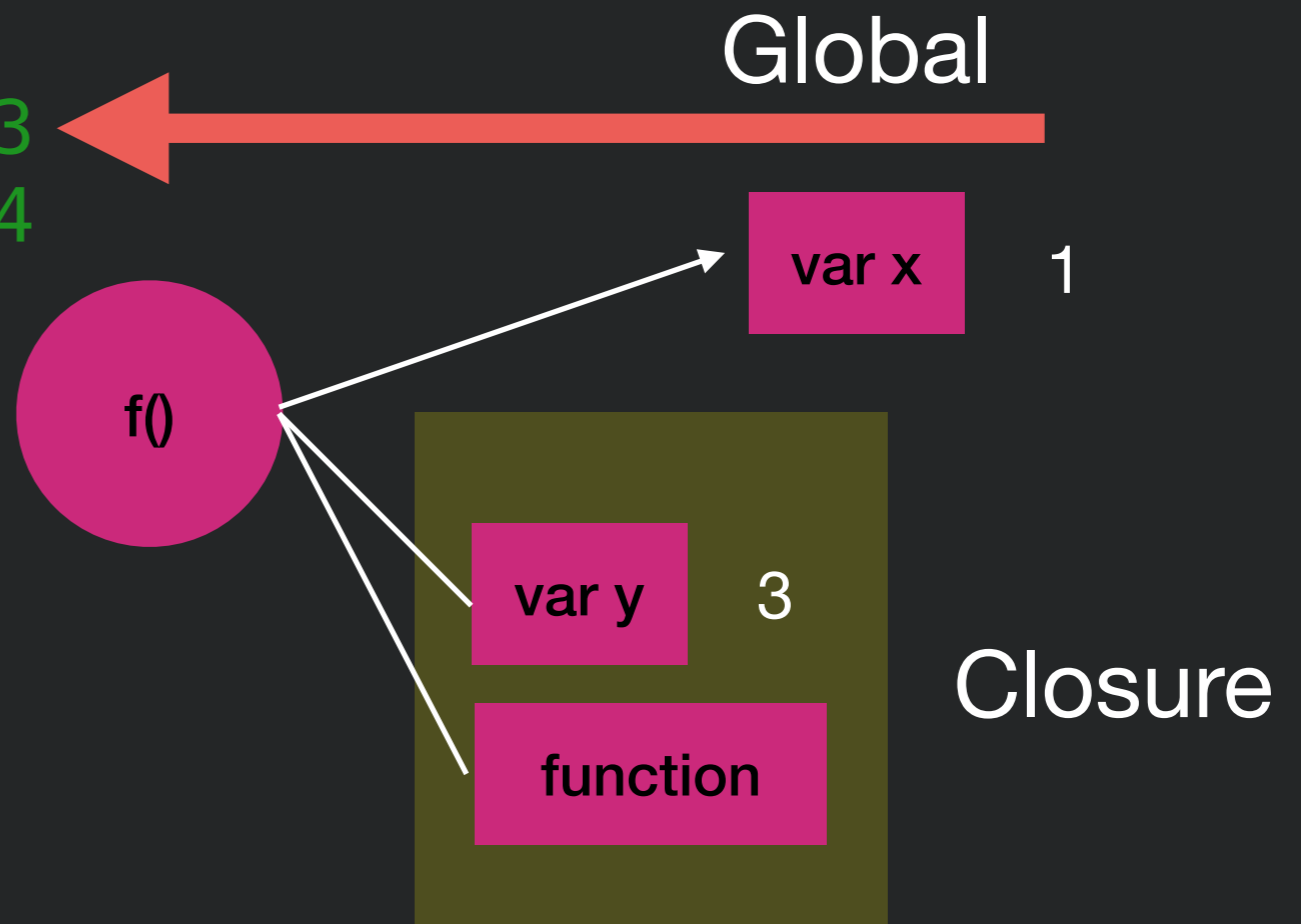




# Closures

```
var x = 1;
function f() {
  var y = 2;
  return function() {
    console.log(x + y);
    y++;
  };
};
```

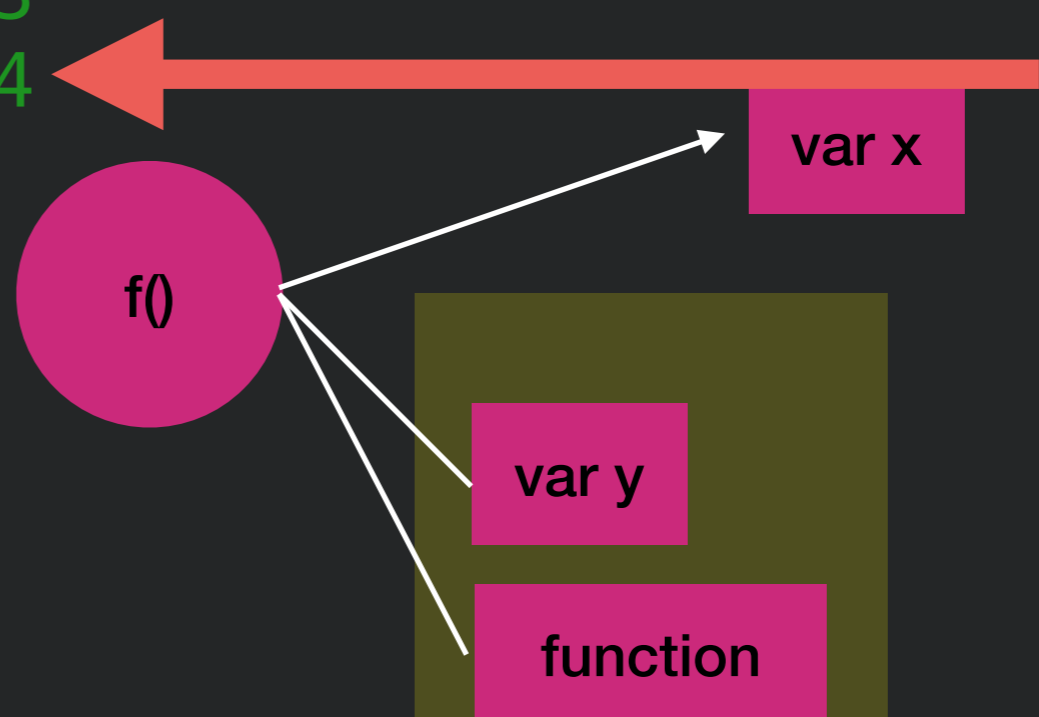
```
var g = f();
g(); // 1+2 is 3
g(); // 1+3 is 4
```



# Closures

```
var x = 1;
function f() {
  var y = 2;
  return function() {
    console.log(x + y);
    y++;
  };
};
```

```
var g = f();
g(); // 1+2 is 3
g(); // 1+3 is 4
```

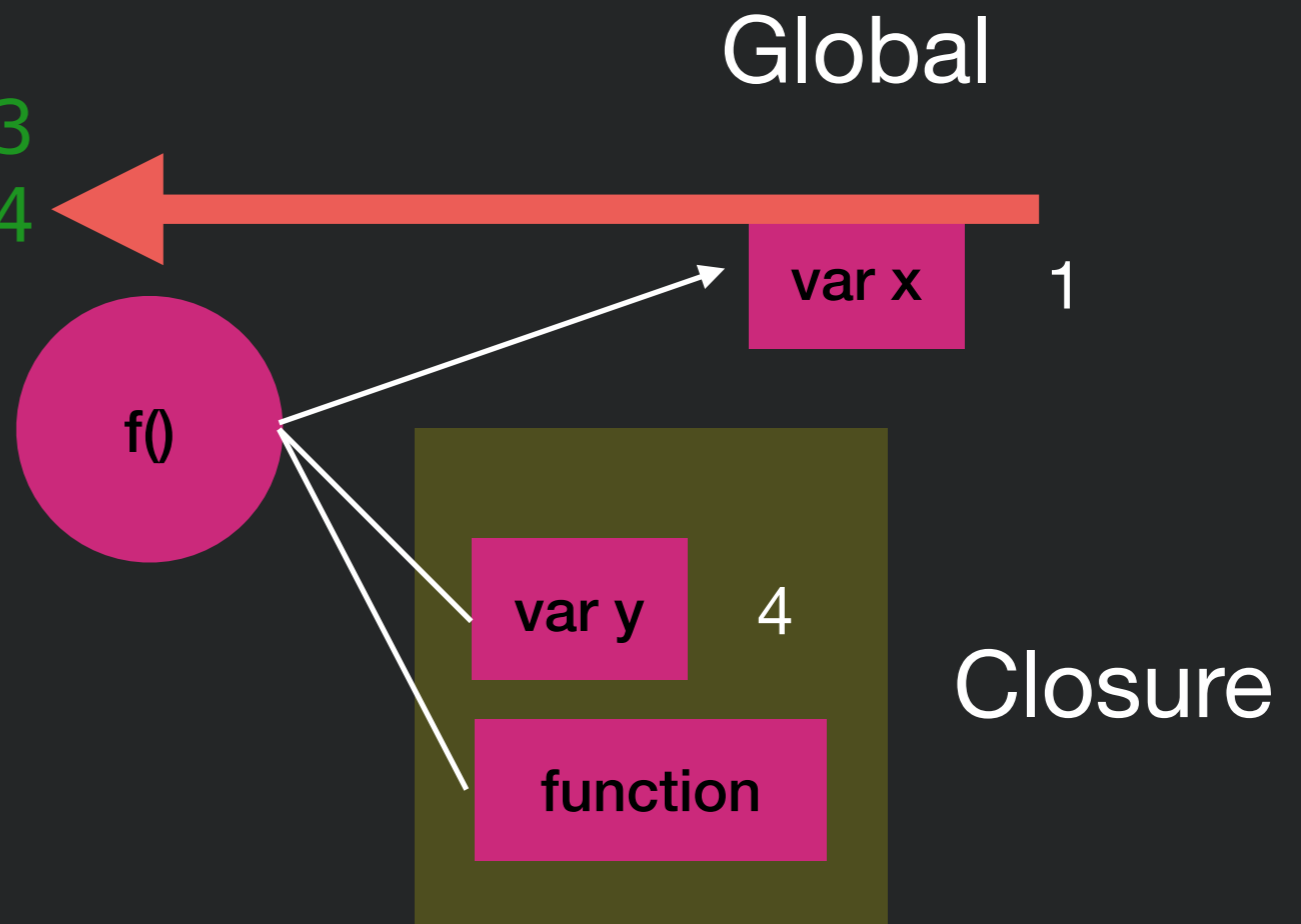




# Closures

```
var x = 1;
function f() {
  var y = 2;
  return function() {
    console.log(x + y);
    y++;
  };
};
```

```
var g = f();
g(); // 1+2 is 3
g(); // 1+3 is 4
```



# Class Overview





# Class Overview

- Part 1 - Asynchronous Programming I: Communicating between web app components?
- 10 minute Break
- Part 2 - Asynchronous Programming II: More communication strategies
- Part 3 - In-Class Activity: Exploring Asynchronous Programming

# Asynchronous Programming I







# Lecture 1

- What is asynchronous programming?
- What are threads?
- Writing asynchronous code

For further reading:

- **Using Promises:** [https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Using\\_promises](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Using_promises)
- **Node.js event loop:** <https://nodejs.org/en/docs/guides/event-loop-timers-and-nexttick/>



# Why Asynchronous?

- Maintain an interactive application while still doing stuff
  - Processing data
  - Communicating with remote hosts
  - Timers that countdown while our app is running
- Anytime that an app is doing more than one thing at a time, it is asynchronous



# What is a thread?

**Program execution: a series of sequential method calls (★s)**

# What is a thread?

**Program execution: a series of sequential method calls (★s)**

App Starts



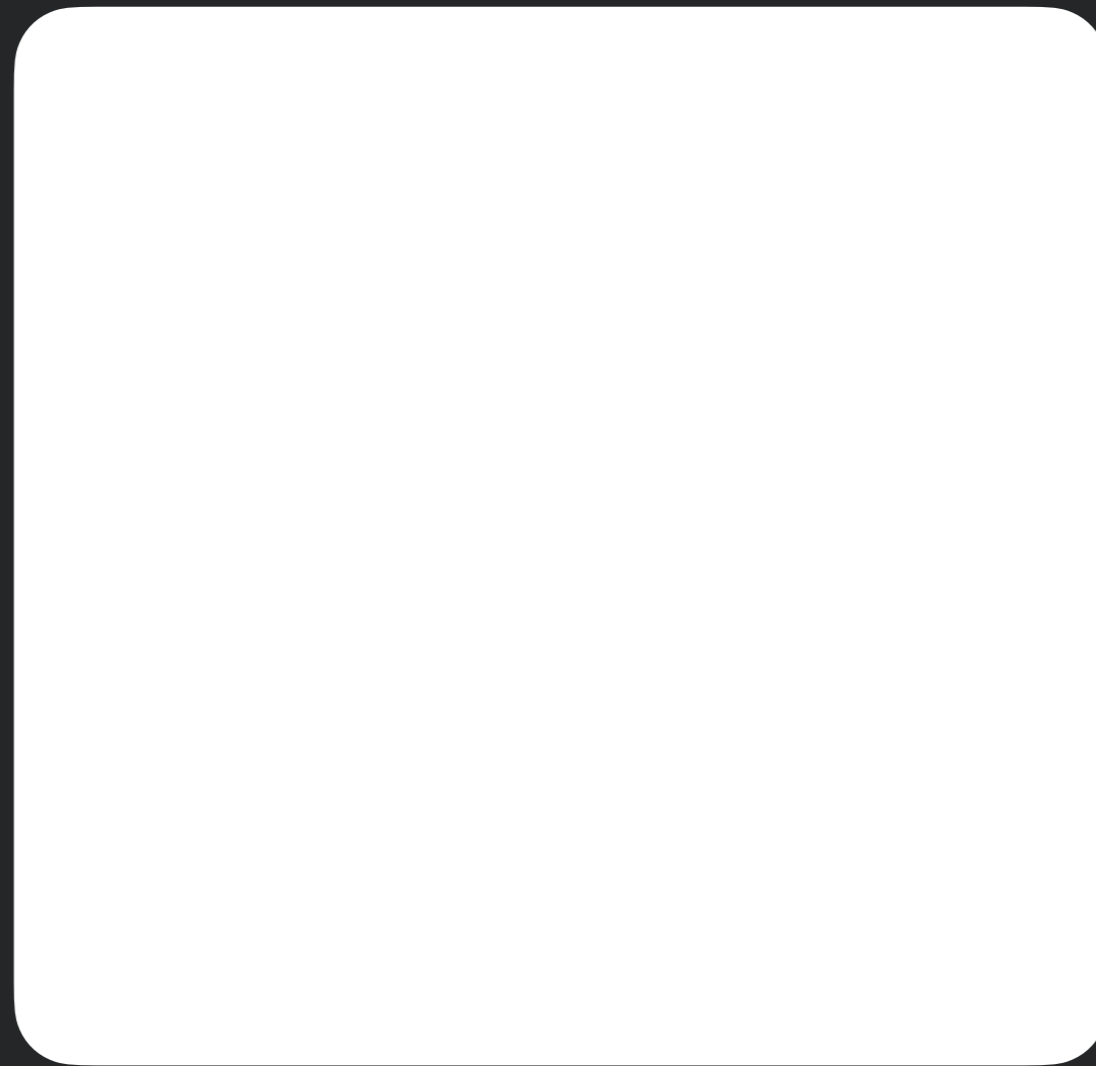
App Ends



# What is a Thread?

**Program execution: a series of sequential method calls (★s)**

App Starts



App Ends

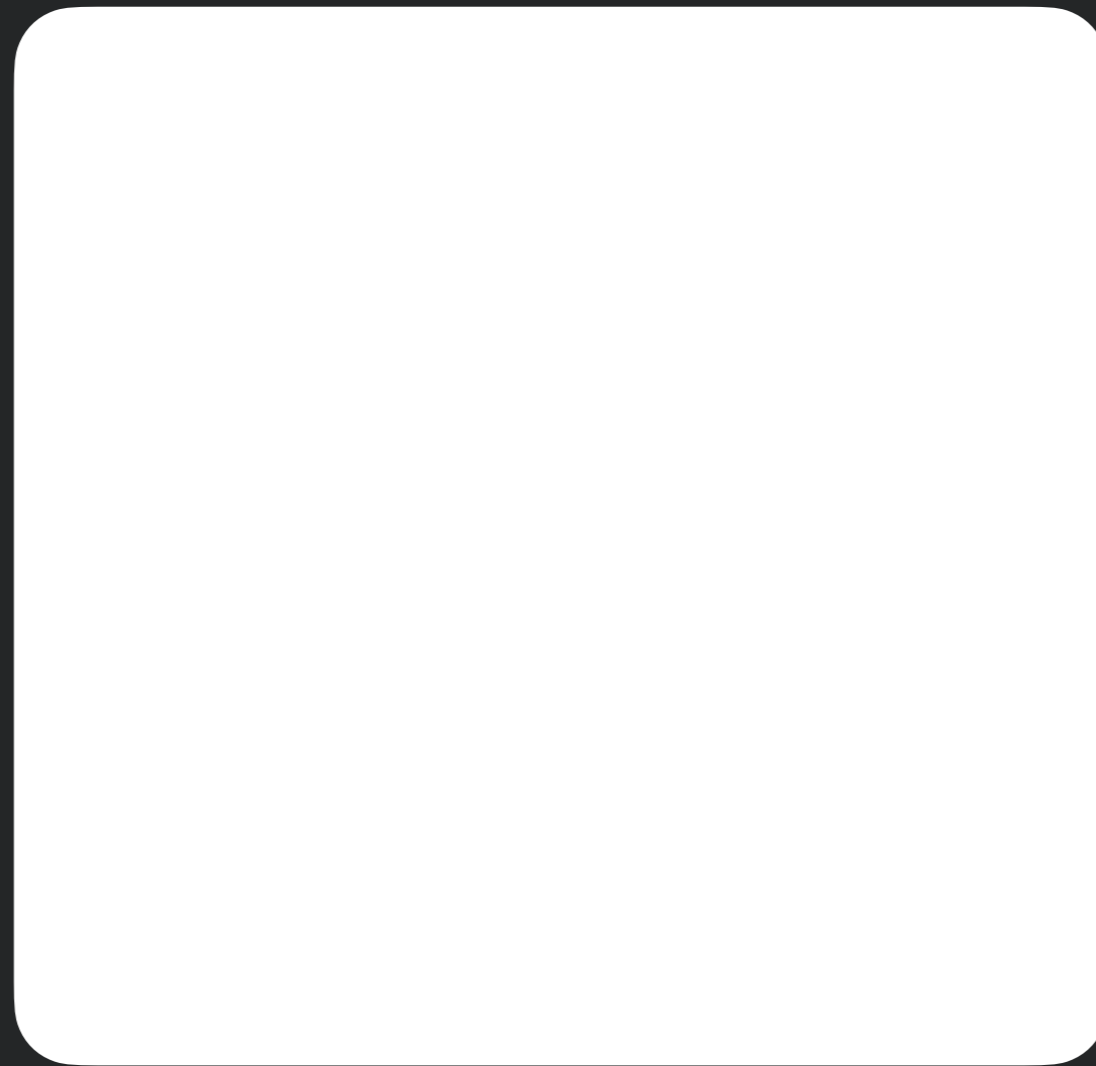
**Multiple threads can run at once -> allows for asynchronous code**



# What is a Thread?

**Program execution: a series of sequential method calls (★s)**

App Starts



App Ends

**Multiple threads can run at once -> allows for asynchronous code**

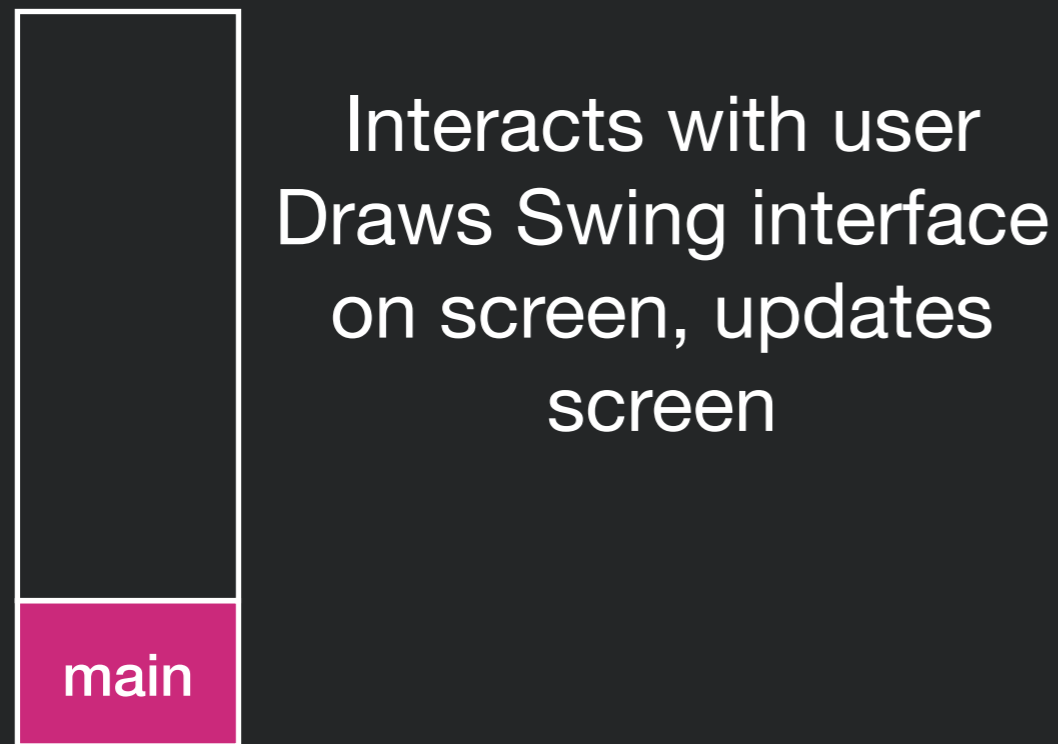


# Multi-Threading in Java

- Multi-Threading allows us to do more than one thing at a time
- Physically, through multiple cores and/or OS scheduler
- Example: Process data while interacting with user

# Multi-Threading in Java

- Multi-Threading allows us to do more than one thing at a time
- Physically, through multiple cores and/or OS scheduler
- Example: Process data while interacting with user

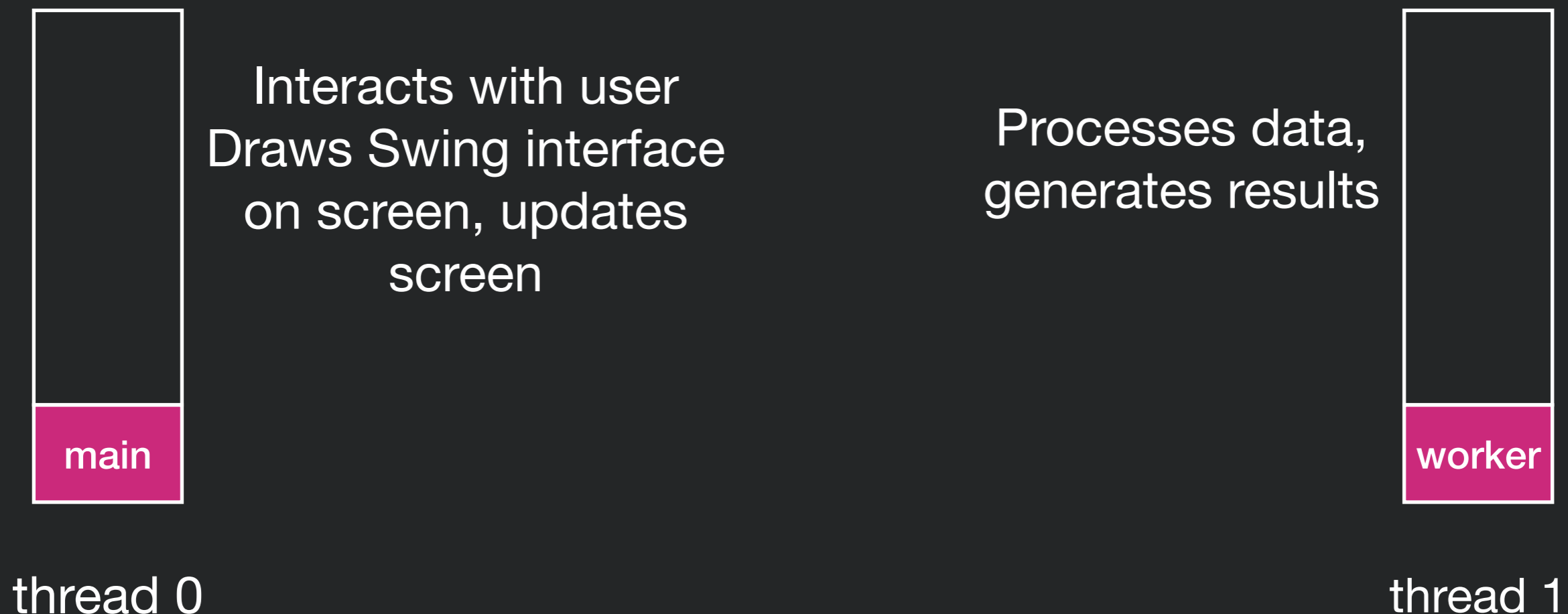


thread 0



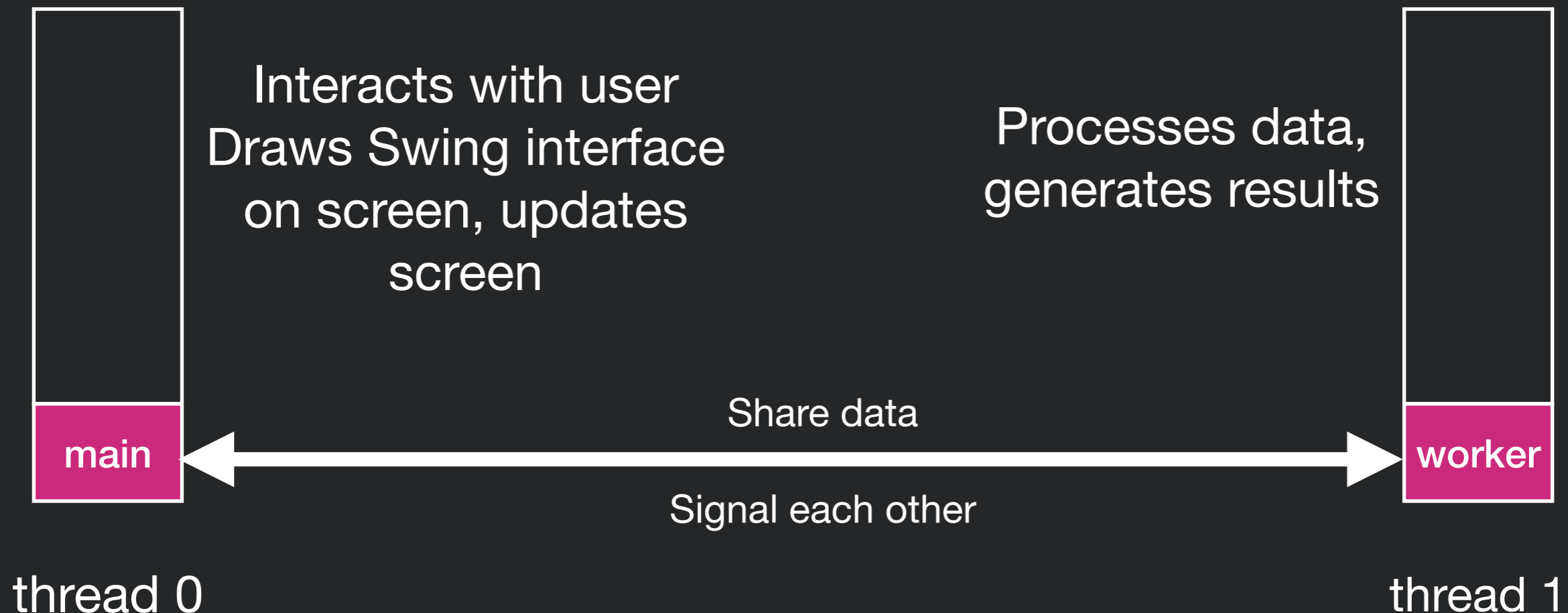
# Multi-Threading in Java

- Multi-Threading allows us to do more than one thing at a time
- Physically, through multiple cores and/or OS scheduler
- Example: Process data while interacting with user



# Multi-Threading in Java

- Multi-Threading allows us to do more than one thing at a time
- Physically, through multiple cores and/or OS scheduler
- Example: Process data while interacting with user





# Woes of Multi-Threading

```
public static int v;  
public static void thread1()  
{  
    v = 4;  
    System.out.println(v);  
}
```

```
public static void thread2()  
{  
    v = 2;  
}
```

This is a data race: the `println` in `thread1` might see either 2 OR 4



# Woes of Multi-Threading

```
public static int v;  
public static void thread1()  
{  
    v = 4;  
    System.out.println(v);  
}
```

```
public static void thread2()  
{  
    v = 2;  
}
```

This is a data race: the `println` in `thread1` might see either 2 OR 4

Thread 1

Thread 2



# Woes of Multi-Threading

```
public static int v;  
public static void thread1()  
{  
    v = 4;  
    System.out.println(v);  
}
```

```
public static void thread2()  
{  
    v = 2;  
}
```

This is a data race: the `println` in `thread1` might see either 2 OR 4

Thread 1	Thread 2
Write V = 4	



# Woes of Multi-Threading

```
public static int v;  
public static void thread1()  
{  
    v = 4;  
    System.out.println(v);  
}
```

```
public static void thread2()  
{  
    v = 2;  
}
```

This is a data race: the `println` in `thread1` might see either 2 OR 4

Thread 1	Thread 2
Write V = 4	
	Write V = 2



# Woes of Multi-Threading

```
public static int v;  
public static void thread1()  
{  
    v = 4;  
    System.out.println(v);  
}
```

```
public static void thread2()  
{  
    v = 2;  
}
```

This is a data race: the `println` in `thread1` might see either 2 OR 4

Thread 1	Thread 2
Write V = 4	
	Write V = 2
Read V (2)	



# Woes of Multi-Threading

```
public static int v;  
public static void thread1()  
{  
    v = 4;  
    System.out.println(v);  
}
```

```
public static void thread2()  
{  
    v = 2;  
}
```

This is a data race: the `println` in `thread1` might see either 2 OR 4

Thread 1	Thread 2
Write V = 4	
	Write V = 2
Read V (2)	

Thread 1	Thread 2
----------	----------





# Woes of Multi-Threading

```
public static int v;  
public static void thread1()  
{  
    v = 4;  
    System.out.println(v);  
}
```

```
public static void thread2()  
{  
    v = 2;  
}
```

This is a data race: the `println` in `thread1` might see either 2 OR 4

Thread 1	Thread 2
Write V = 4	
	Write V = 2
Read V (2)	

Thread 1	Thread 2
	Write V = 2



# Woes of Multi-Threading

```
public static int v;  
public static void thread1()  
{  
    v = 4;  
    System.out.println(v);  
}
```

```
public static void thread2()  
{  
    v = 2;  
}
```

This is a data race: the `println` in `thread1` might see either 2 OR 4

Thread 1	Thread 2
Write V = 4	
	Write V = 2
Read V (2)	

Thread 1	Thread 2
	Write V = 2
Write V = 4	



# Woes of Multi-Threading

```
public static int v;  
public static void thread1()  
{  
    v = 4;  
    System.out.println(v);  
}
```

```
public static void thread2()  
{  
    v = 2;  
}
```

This is a data race: the `println` in `thread1` might see either 2 OR 4

Thread 1	Thread 2
Write V = 4	
	Write V = 2
Read V (2)	

Thread 1	Thread 2
	Write V = 2
Write V = 4	
Read V (4)	



# Multi-Threading in JS

```
var request = require('request');  
request('http://www.google.com', function (error, response,  
body) {  
    console.log("Heard back from Google!");  
});  
console.log("Made request");
```

Request is an asynchronous call



# Multi-Threading in JS

```
var request = require('request');  
request('http://www.google.com', function (error, response,  
body) {  
    console.log("Heard back from Google!");  
});  
console.log("Made request");
```

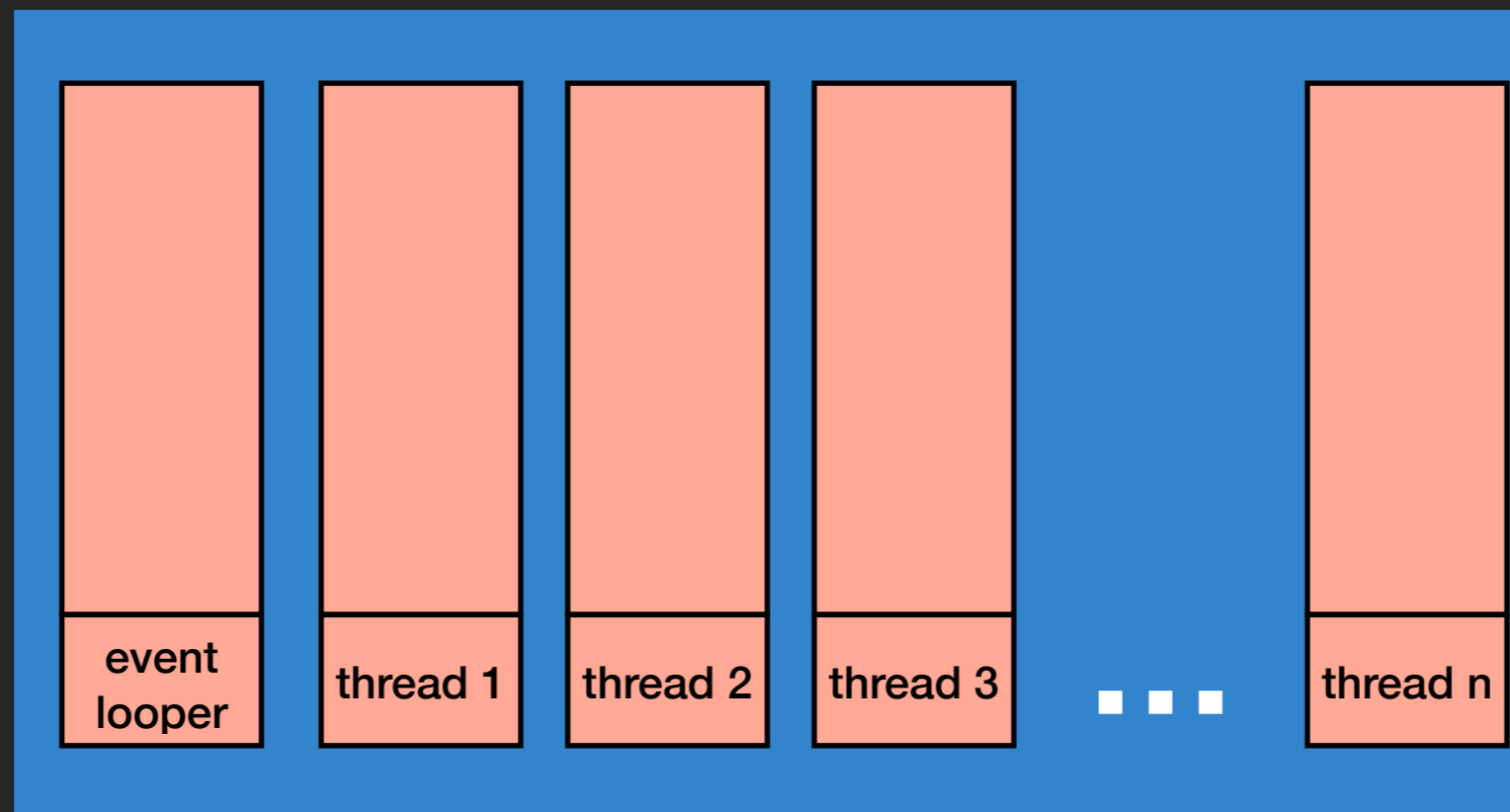
## Output:

Made request  
Heard back from Google!

Request is an asynchronous call

# Multi-Threading in JS

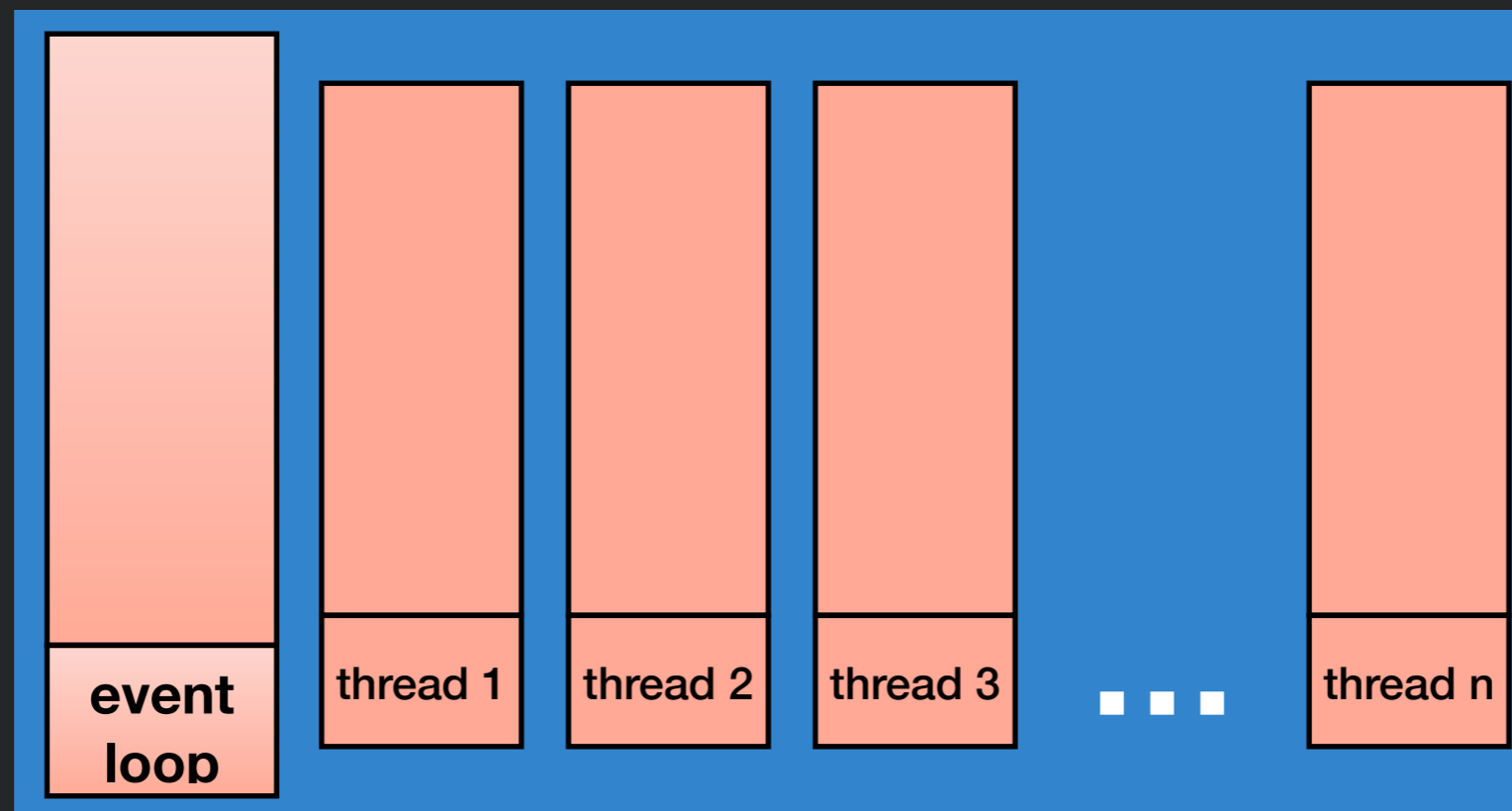
- Everything you write will run in a single thread\* (event loop)
- Since you are not sharing data between threads, races don't happen as easily
- Inside of JS engine: many threads
- Event loop processes events, and calls your callbacks



JS Engine

# Multi-Threading in JS

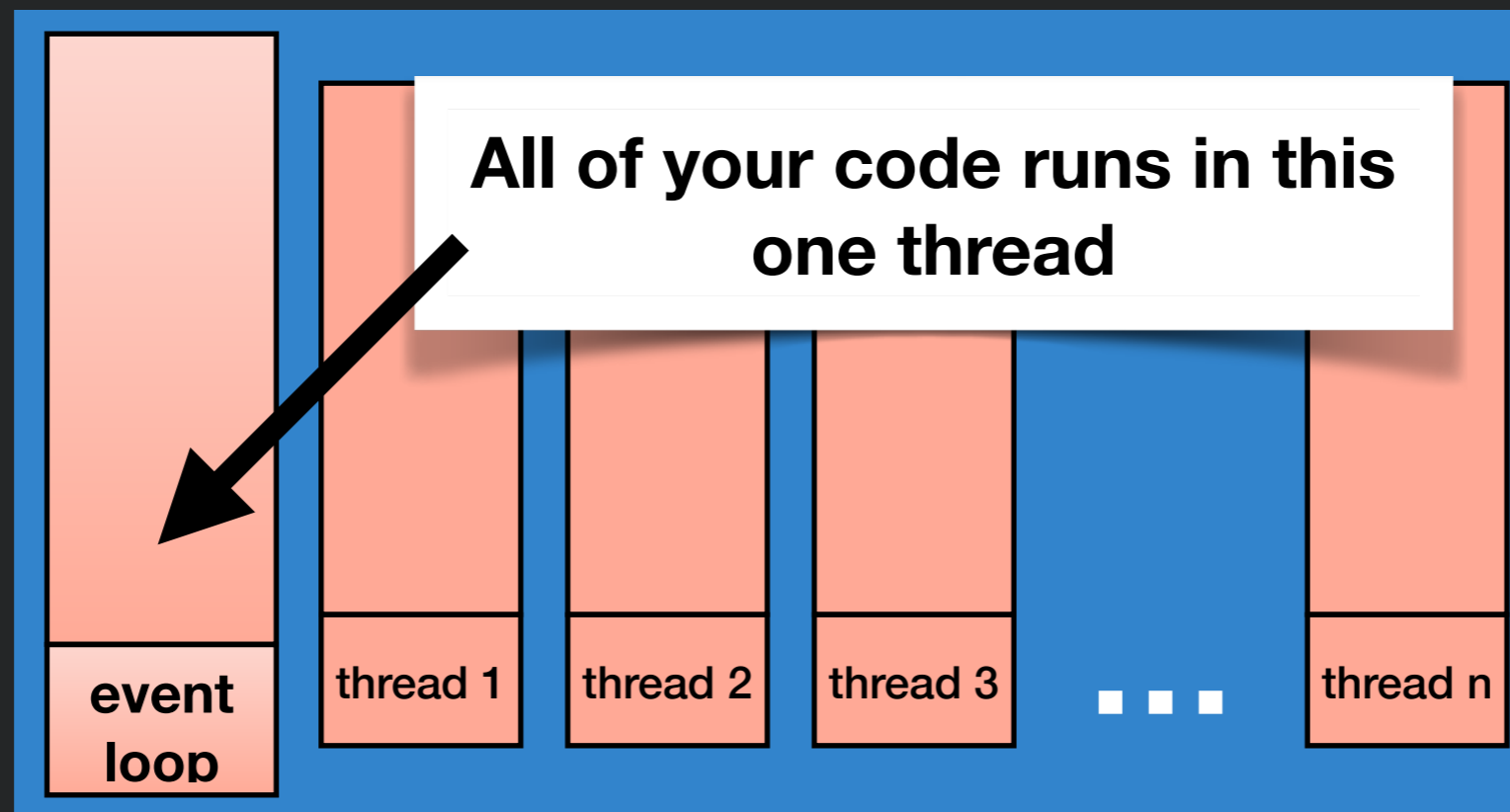
- Everything you write will run in a single thread\* (event loop)
- Since you are not sharing data between threads, races don't happen as easily
- Inside of JS engine: many threads
- Event loop processes events, and calls your callbacks



JS Engine

# Multi-Threading in JS

- Everything you write will run in a single thread\* (event loop)
- Since you are not sharing data between threads, races don't happen as easily
- Inside of JS engine: many threads
- Event loop processes events, and calls your callbacks

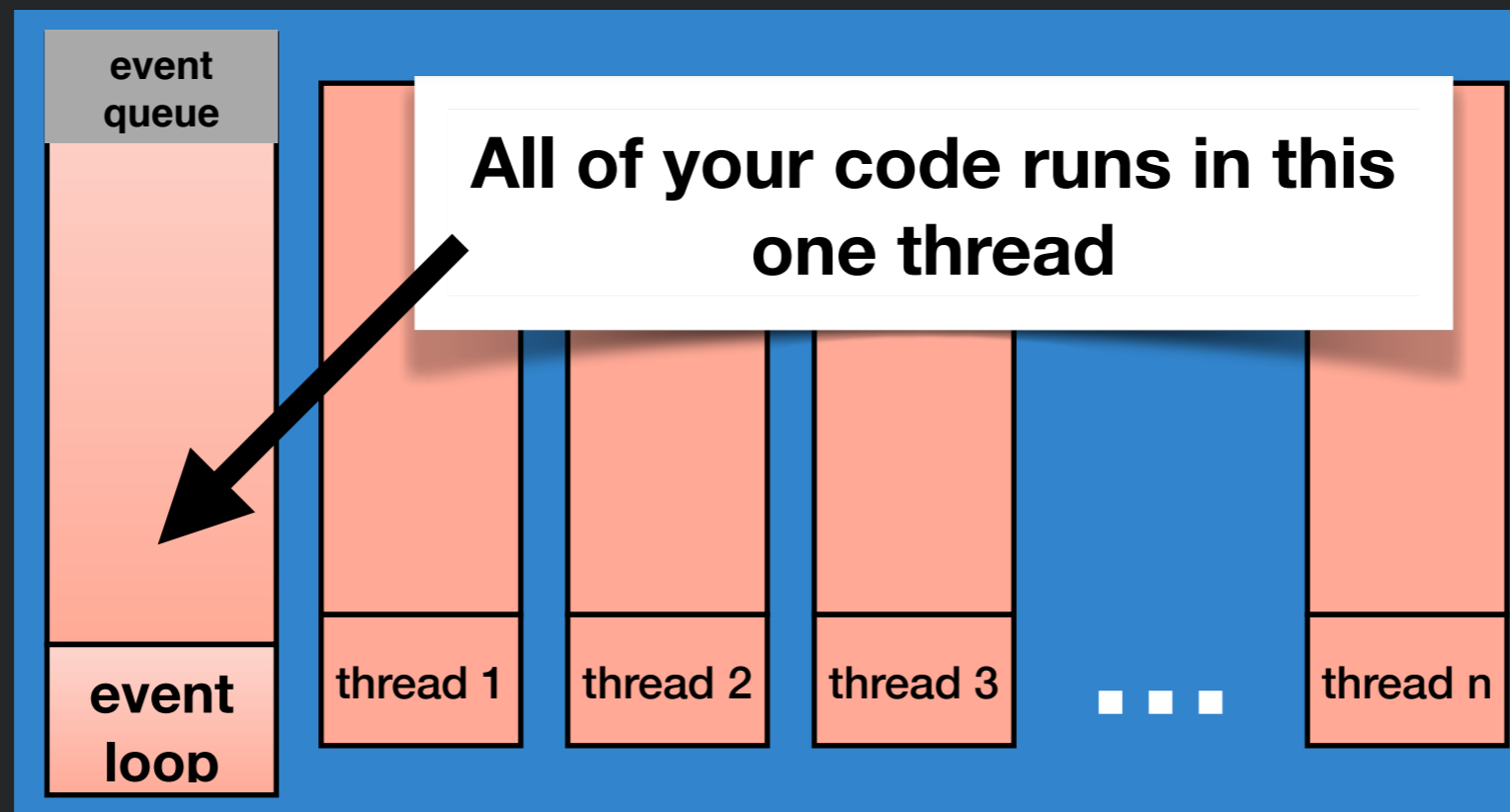


JS Engine



# Multi-Threading in JS

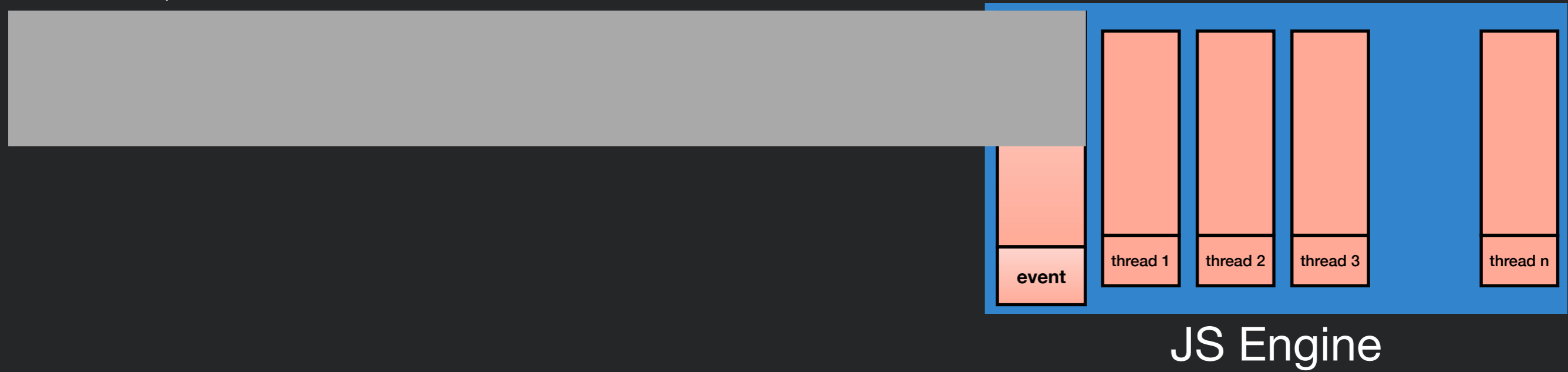
- Everything you write will run in a single thread\* (event loop)
- Since you are not sharing data between threads, races don't happen as easily
- Inside of JS engine: many threads
- Event loop processes events, and calls your callbacks





# The Event Loop

*Event Queue*





# The Event Loop

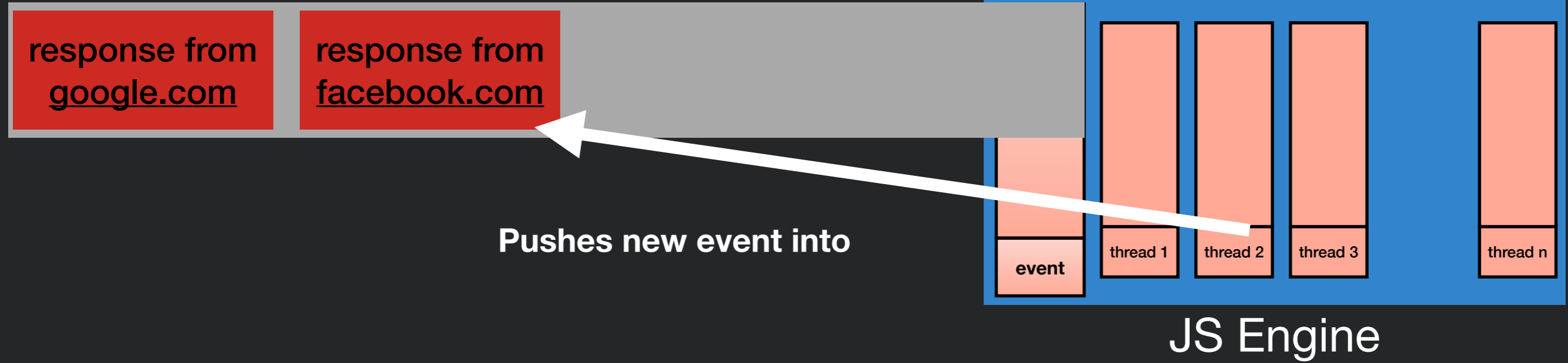
## Event Queue





# The Event Loop

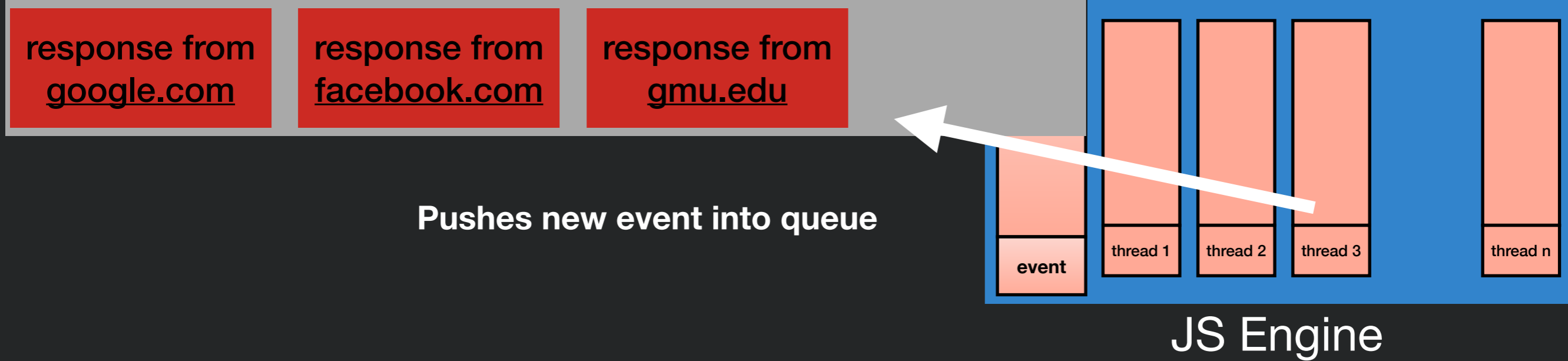
## Event Queue





# The Event Loop

## Event Queue





# The Event Loop

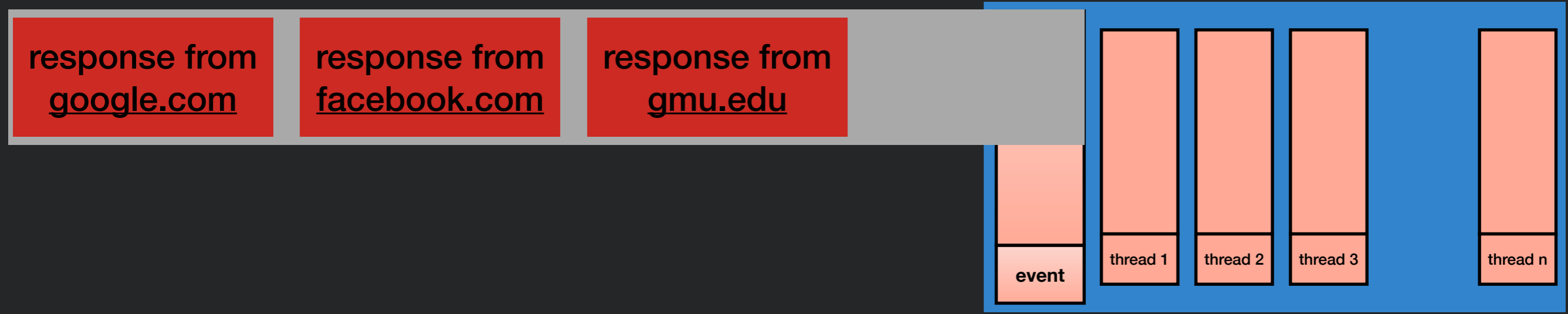
## Event Queue





# The Event Loop

## Event Queue



**Event Being Processed:**

JS Engine

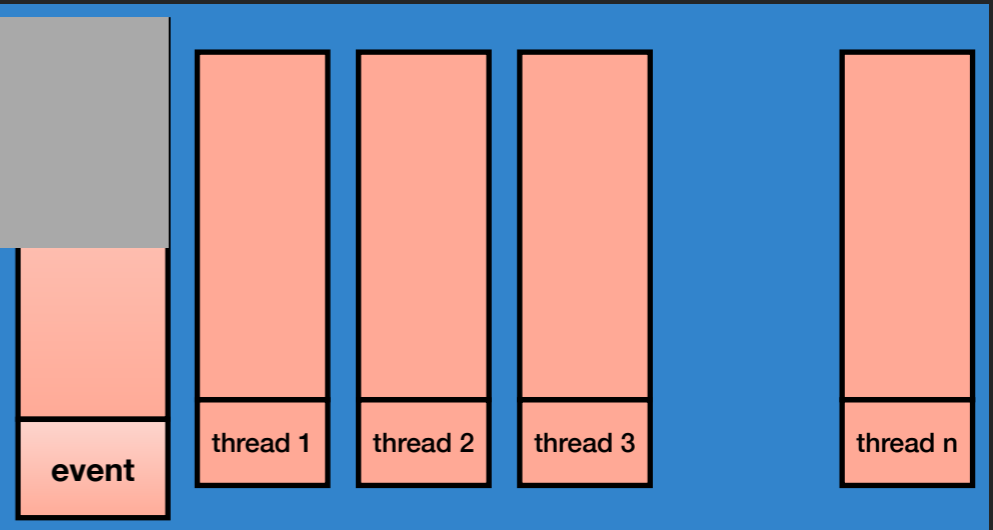


# The Event Loop

## Event Queue

response from  
[facebook.com](https://www.facebook.com)

response from  
[gmu.edu](https://www.gmu.edu)



JS Engine

response from  
[google.com](https://www.google.com)



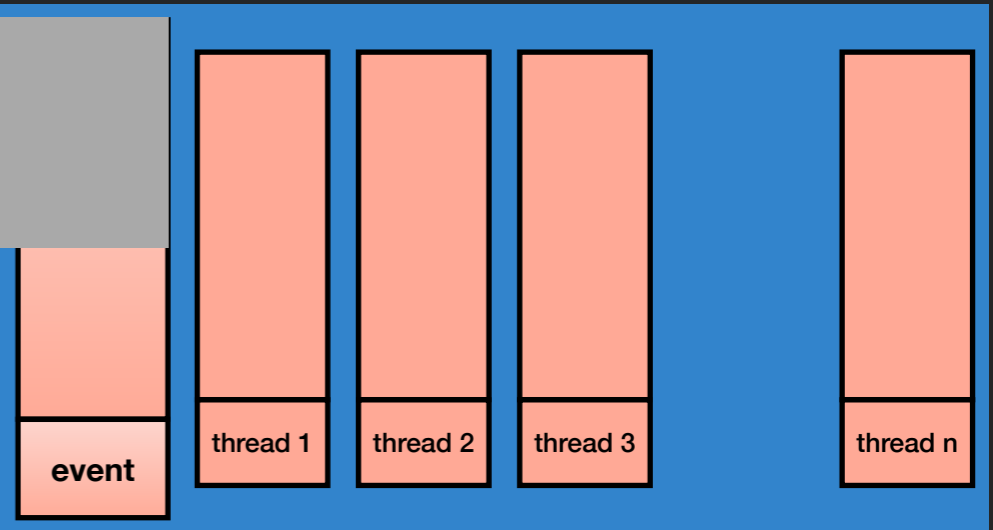


# The Event Loop

## Event Queue

response from  
[facebook.com](https://www.facebook.com)

response from  
[gmu.edu](https://www.gmu.edu)



## Event Being Processed:

response from  
[google.com](https://www.google.com)

JS Engine

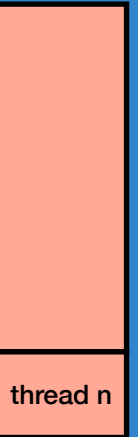
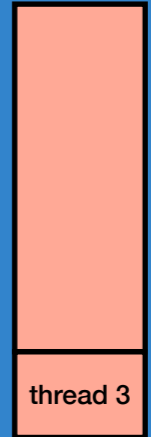
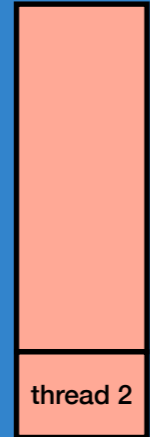
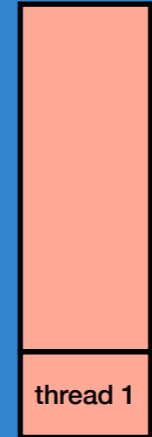
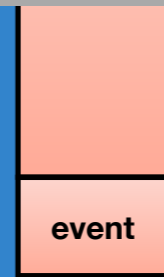


# The Event Loop

## Event Queue

response from  
[facebook.com](https://www.facebook.com)

response from  
[gmu.edu](https://www.gmu.edu)



JS Engine

## Event Being Processed:

response from  
[google.com](https://www.google.com)

Are there any listeners registered for this event?

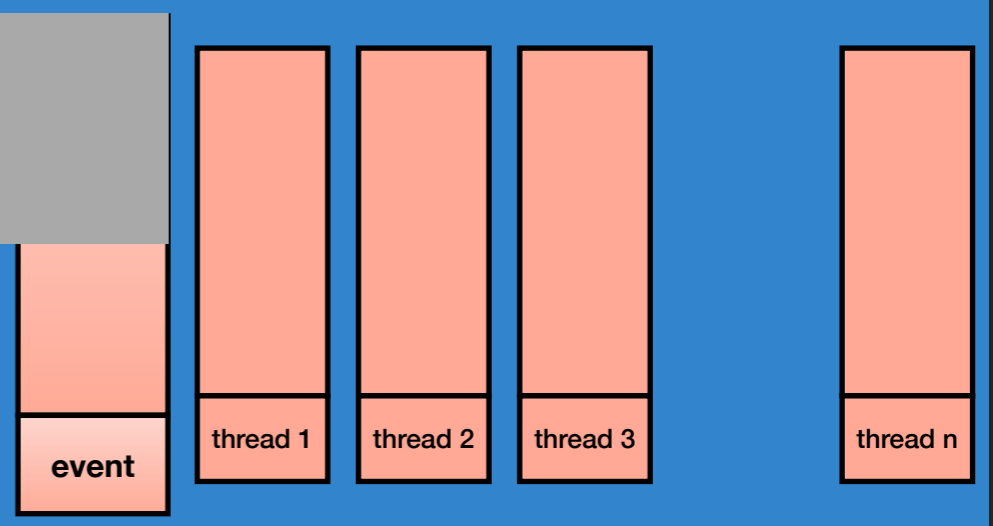


# The Event Loop

## Event Queue

response from  
[facebook.com](https://www.facebook.com)

response from  
[gmu.edu](https://www.gmu.edu)



## Event Being Processed:

response from  
[google.com](https://www.google.com)

JS Engine

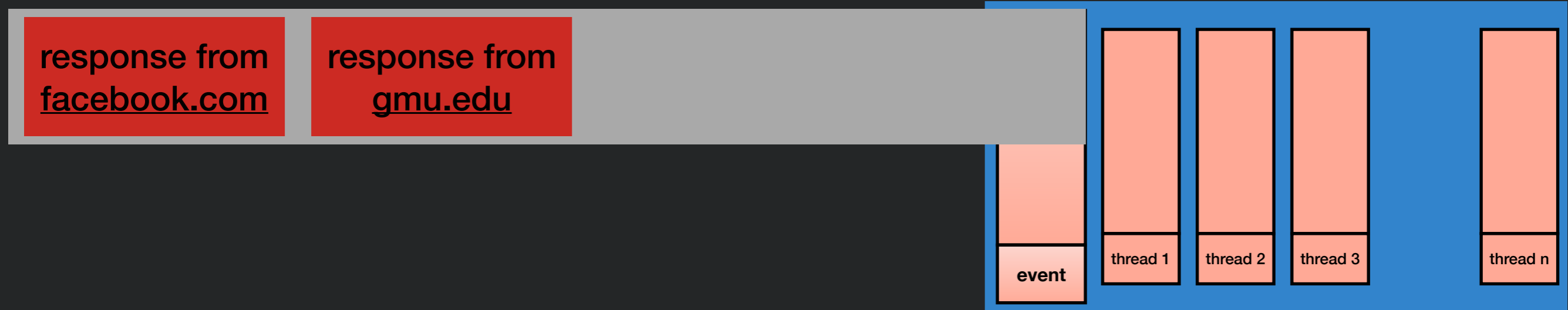
Are there any listeners registered for this event?

If so, call listener with event



# The Event Loop

## Event Queue



## Event Being Processed:

response from  
[google.com](https://www.google.com)

Are there any listeners registered for this event?

If so, call listener with event

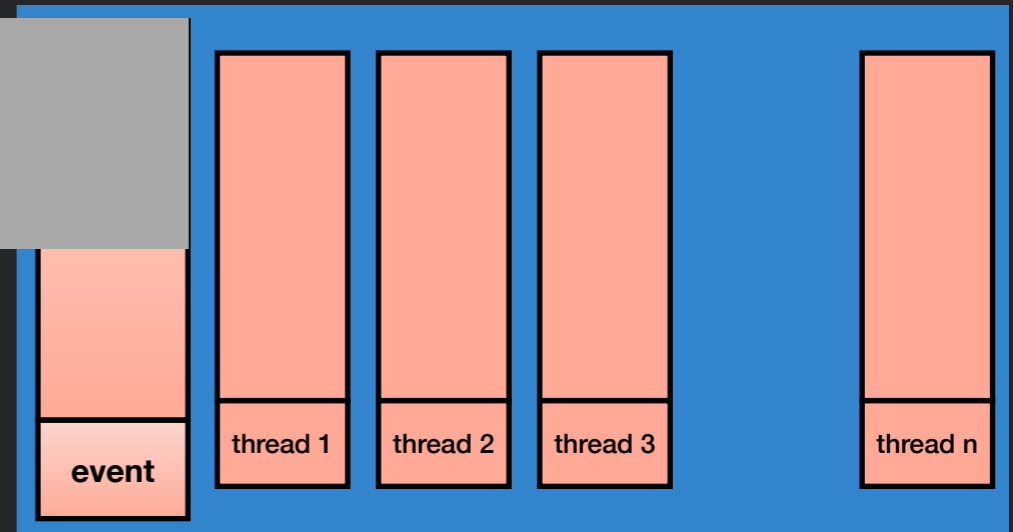
After the listener is finished, repeat



# The Event Loop

## Event Queue

response from [gmu.edu](http://gmu.edu)



JS Engine

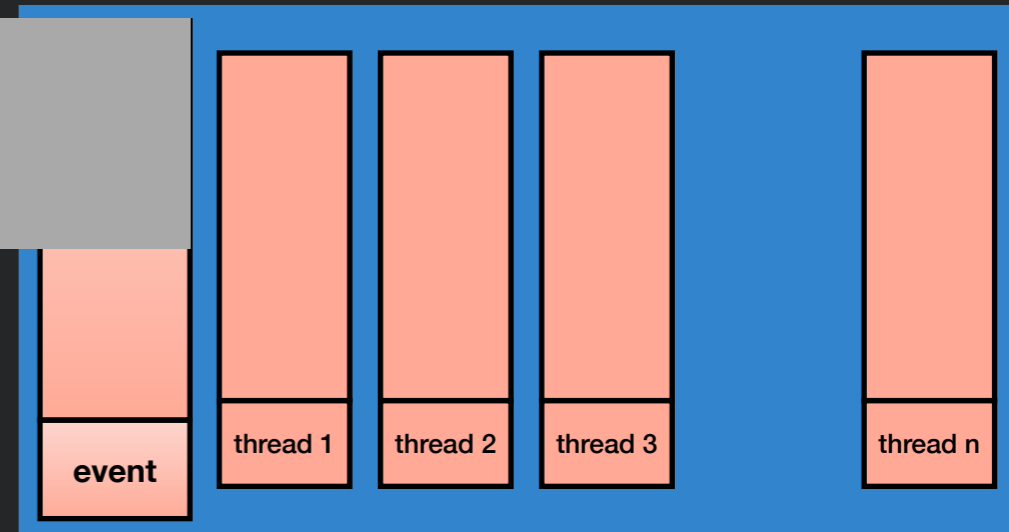
response from [facebook.com](http://facebook.com)



# The Event Loop

## Event Queue

response from  
[gmu.edu](http://gmu.edu)



## Event Being Processed:

response from  
[facebook.com](http://facebook.com)

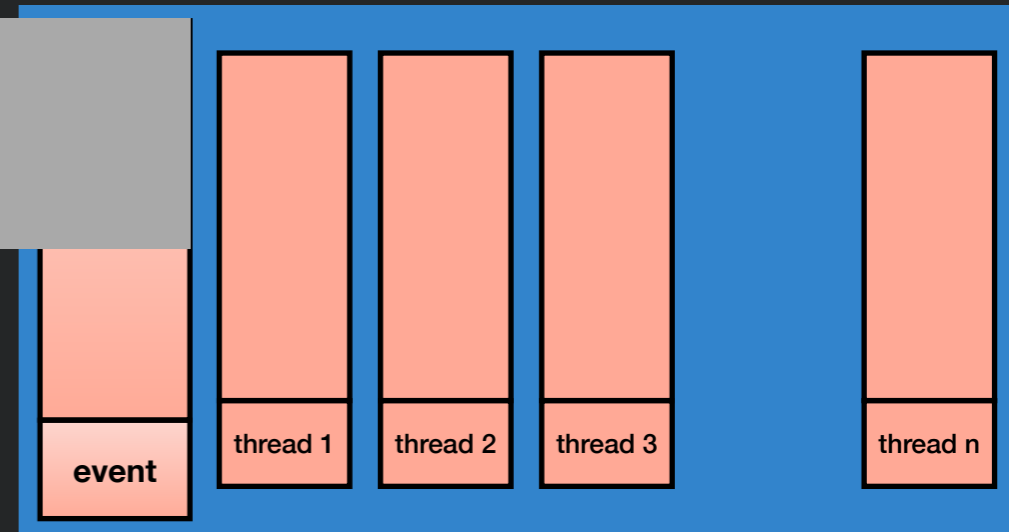
JS Engine



# The Event Loop

## Event Queue

response from  
[gmu.edu](http://gmu.edu)



## Event Being Processed:

response from  
[facebook.com](http://facebook.com)

JS Engine

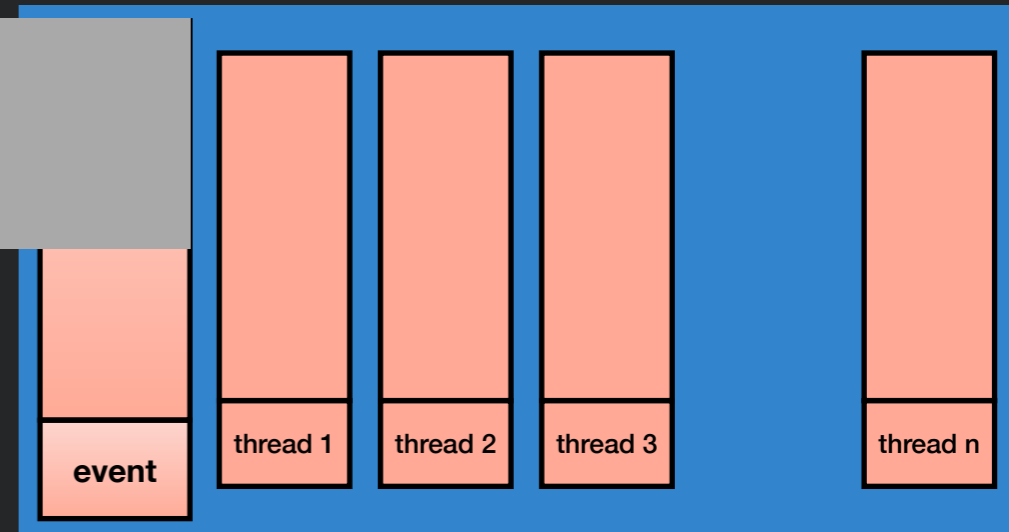
Are there any listeners registered for this event?



# The Event Loop

## Event Queue

response from  
[gmu.edu](http://gmu.edu)



## Event Being Processed:

response from  
[facebook.com](http://facebook.com)

JS Engine

Are there any listeners registered for this event?

If so, call listener with event

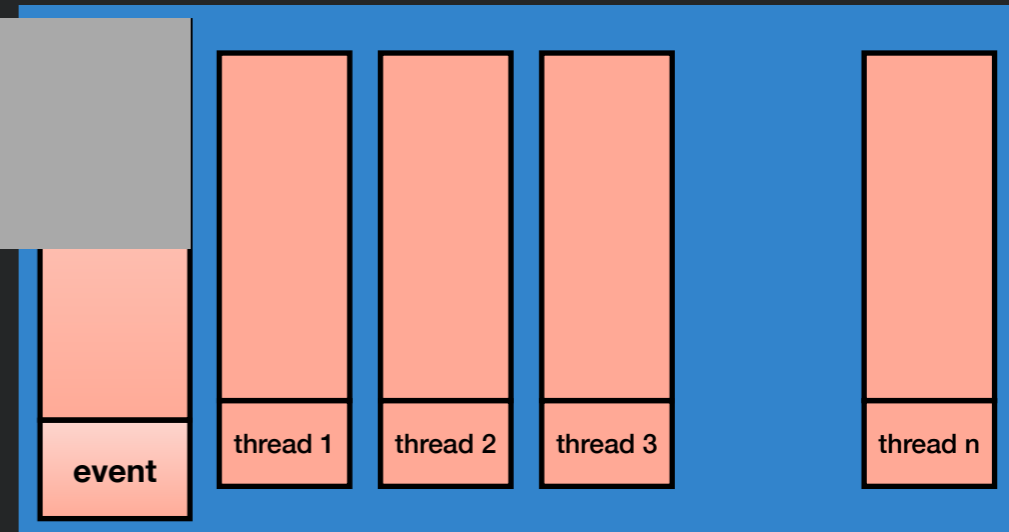




# The Event Loop

## Event Queue

response from  
[gmu.edu](http://gmu.edu)



## Event Being Processed:

response from  
[facebook.com](http://facebook.com)

JS Engine

Are there any listeners registered for this event?

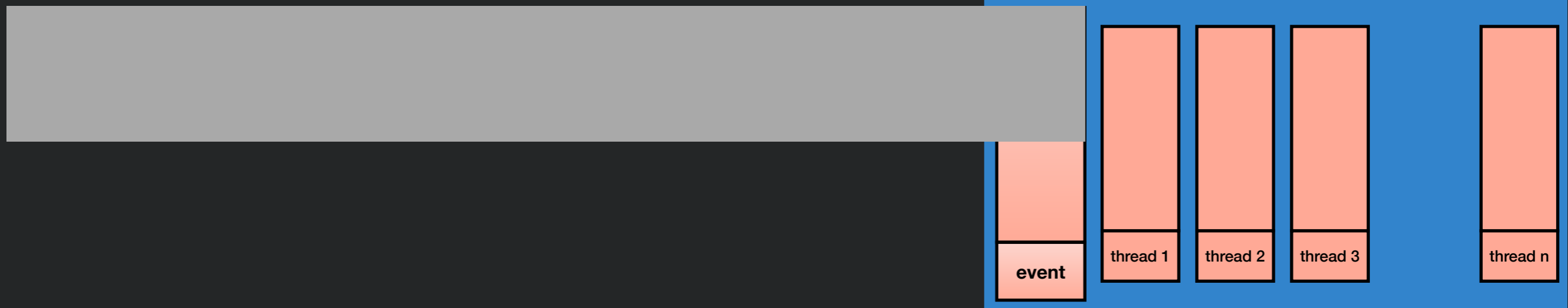
If so, call listener with event

After the listener is finished, repeat



# The Event Loop

*Event Queue*



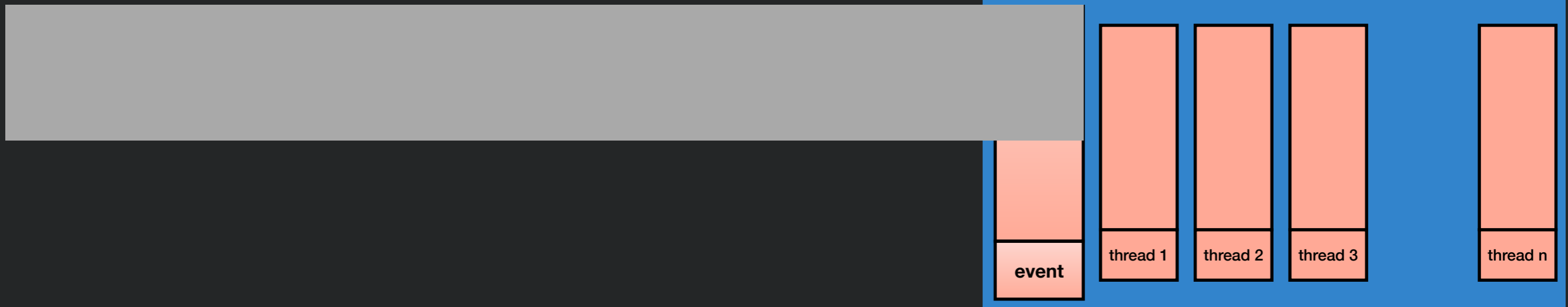
JS Engine

response from  
[gmu.edu](http://gmu.edu)



# The Event Loop

## Event Queue



## Event Being Processed:

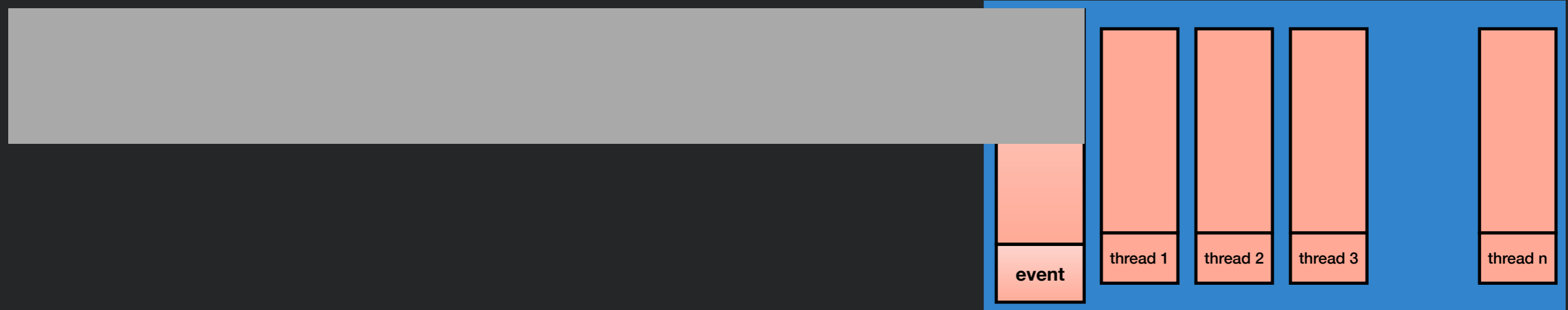
response from  
[gmu.edu](http://gmu.edu)

JS Engine



# The Event Loop

*Event Queue*



**Event Being Processed:**

response from  
[gmu.edu](http://gmu.edu)

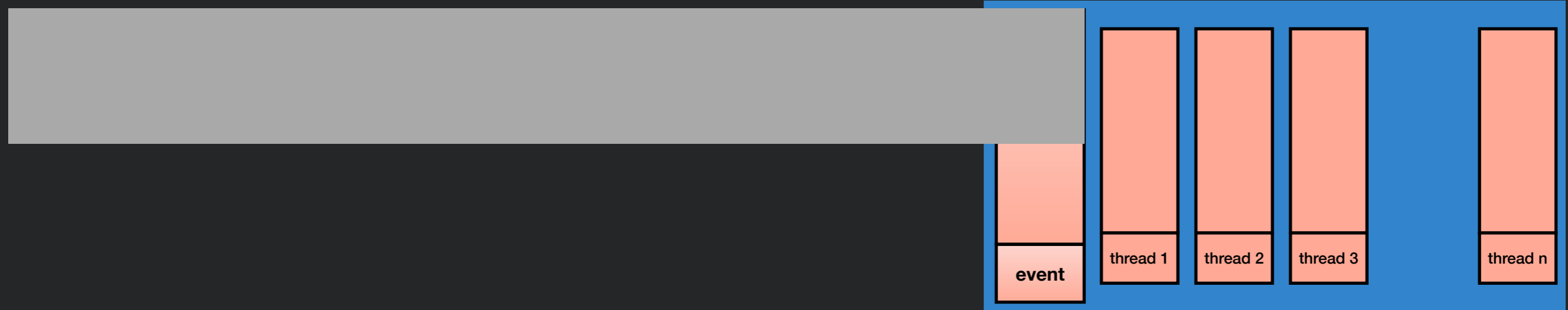
JS Engine

Are there any listeners registered for this event?



# The Event Loop

*Event Queue*



**Event Being Processed:**

response from  
[gmu.edu](http://gmu.edu)

JS Engine

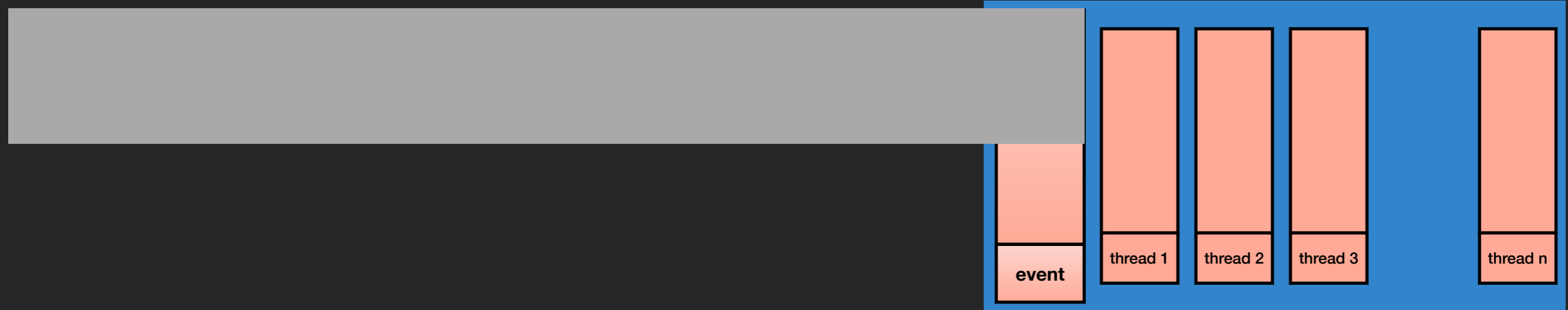
Are there any listeners registered for this event?

If so, call listener with event



# The Event Loop

*Event Queue*



**Event Being Processed:**

response from  
[gmu.edu](http://gmu.edu)

JS Engine

Are there any listeners registered for this event?

If so, call listener with event

After the listener is finished, repeat



# The Event Loop

- Remember that JS is **event-driven**

```
var request = require('request');
request('http://www.google.com', function (error, response, body) {
  console.log("Heard back from Google!");
});
console.log("Made request");
```

- Event loop is responsible for dispatching events when they occur
- Main thread for event loop:

```
while(queue.waitForMessage()){
  queue.processNextMessage();
}
```



# How do you write a “good” event handler?

- Run-to-completion
  - The JS engine will not handle the next event until your event handler finishes
- **Good news:** no other code will run until you finish (no worries about other threads overwriting your data)
- **Bad/OK news:** Event handlers must not block
  - Blocking -> Stall/wait for input (e.g. alert(), non-async network requests)
  - If you *\*must\** do something that takes a long time (e.g. computation), split it up into multiple events





# More Properties of Good Handlers

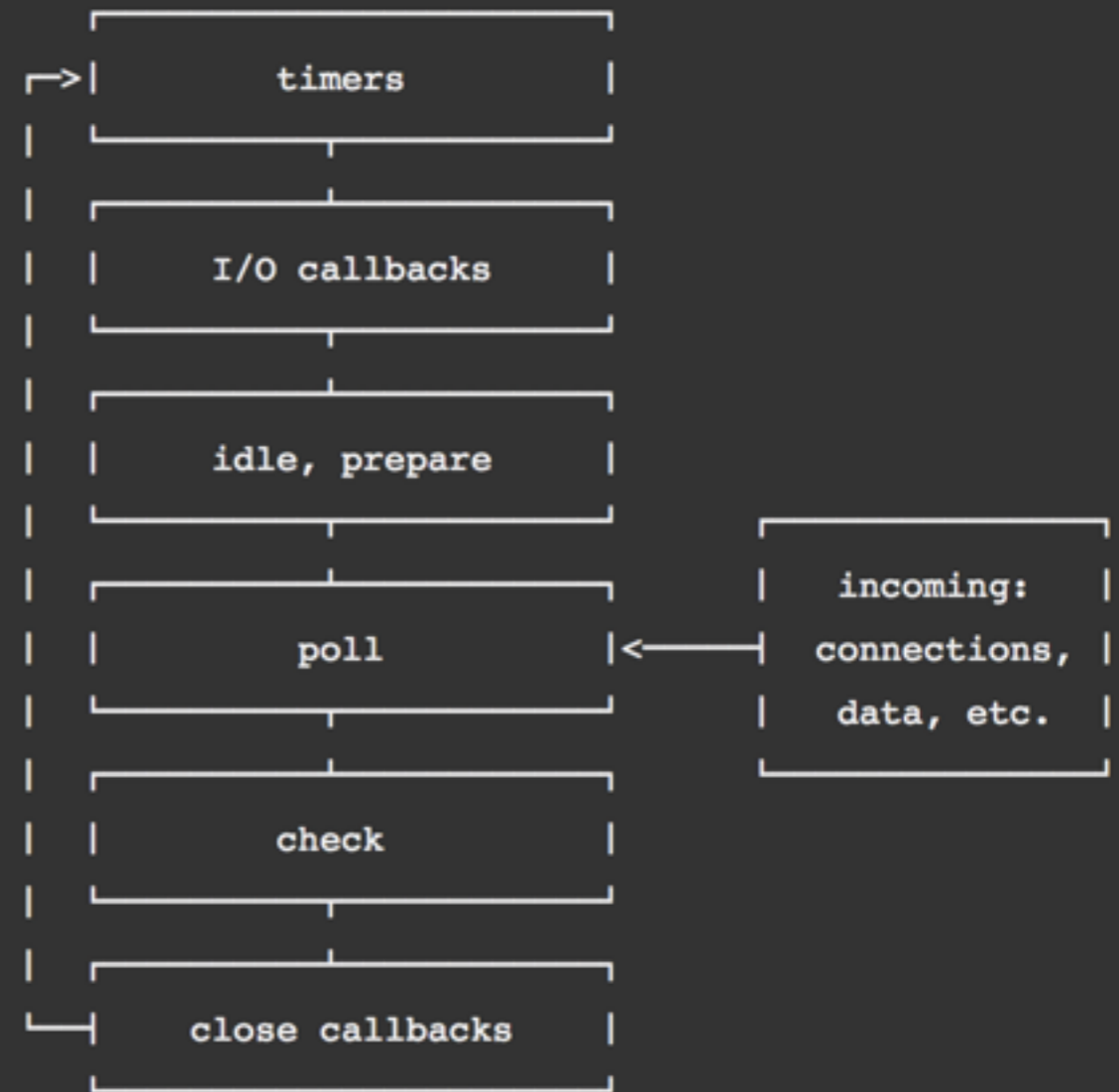
- Remember that event events are processed in the order they are received
- Events might arrive in unexpected order
- Handlers should check the current state of the app to see if they are still relevant



# Prioritizing Events in node.js

- Some events are more important than others
- Keep separate queues for each event "phase"
- Process all events in each phase before moving to next

First



<https://nodejs.org/en/docs/guides/event-loop-timers-and-nexttick/>



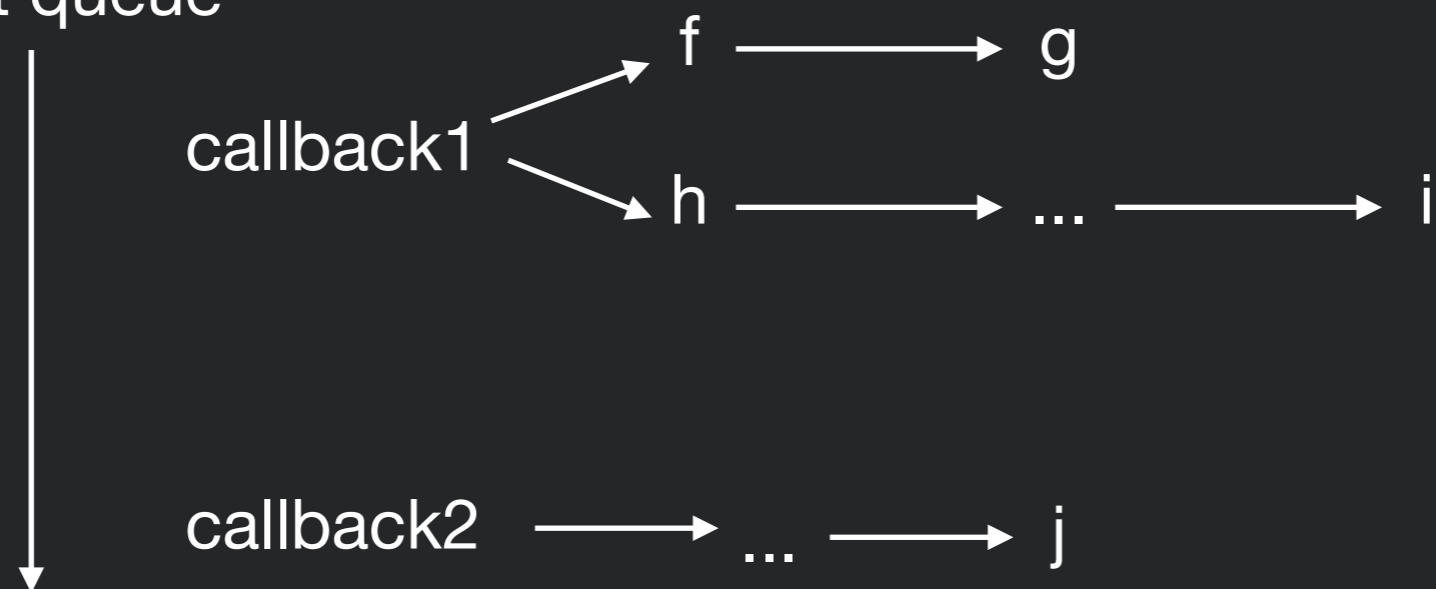
# Benefits vs. Explicit Threading (Java)

- Writing your own threads is *difficult* to reason about and get right:
  - When threads share data, need to ensure they correctly *synchronize* on it to avoid race conditions
- Main downside to events:
  - Can not have slow event handlers
  - Can still have races, although easier to reason about

# Run-to-Completion Semantics

- Run-to-completion
  - The function handling an event and the functions that it (transitively) synchronously calls will keep executing until the function finishes.
  - The JS engine will not handle the next event until the event handler finishes.

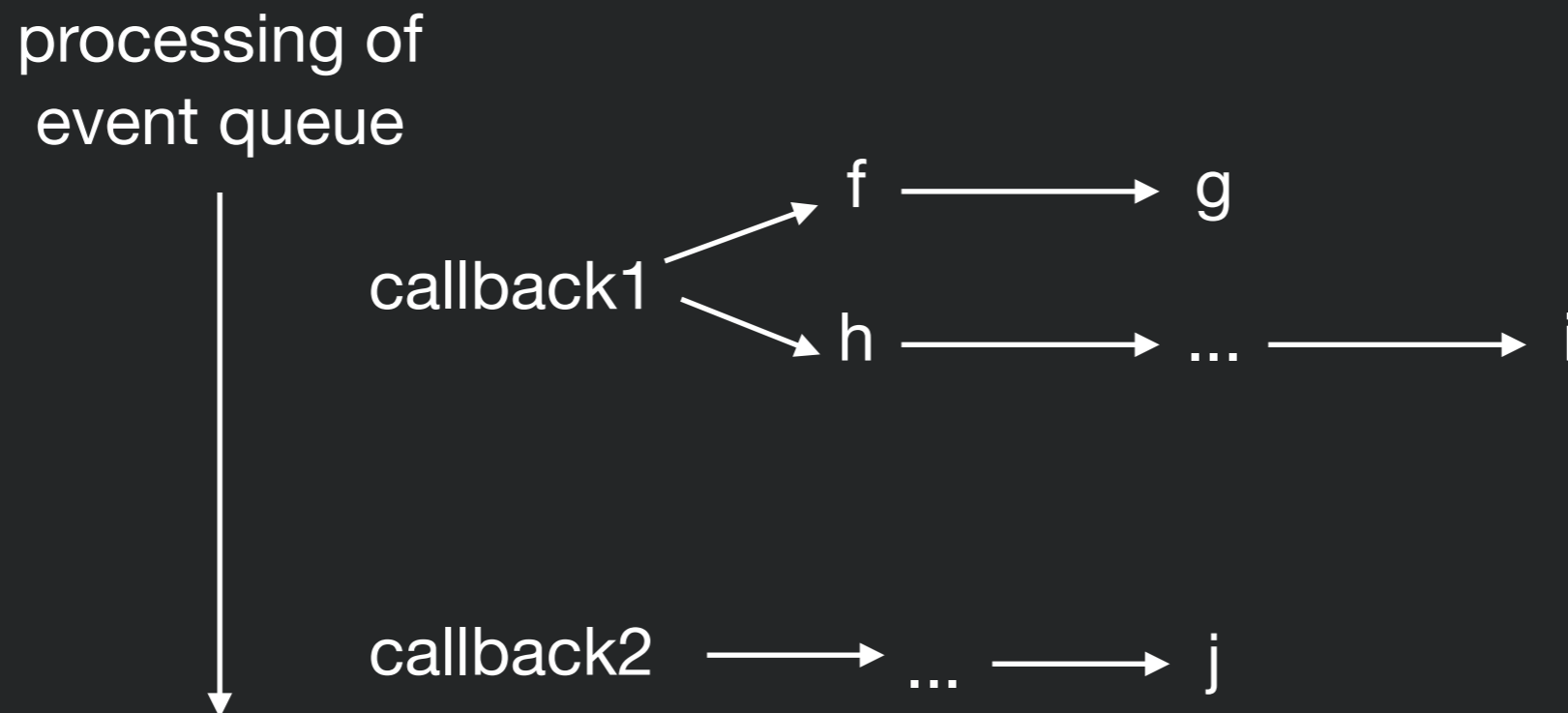
processing of  
event queue





# Implications of Run-to-Completion

- Good news: no other code will run until you finish (no worries about other threads overwriting your data)

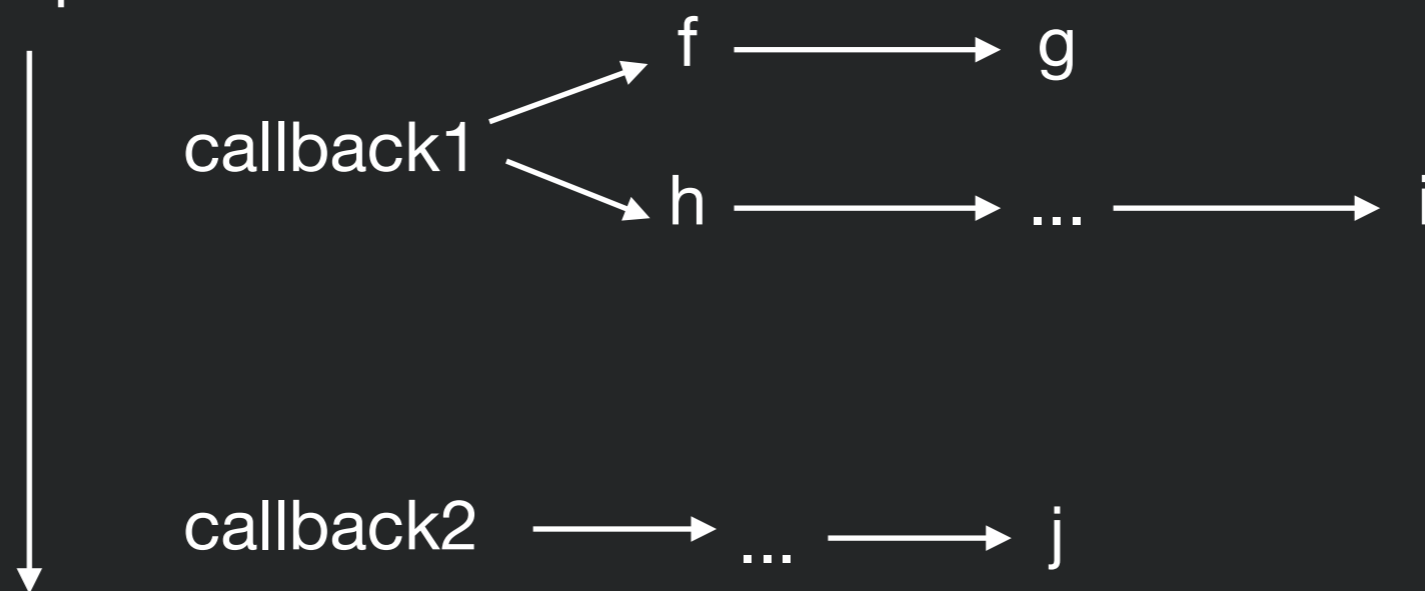


*j will not execute until after i*

# Implications of Run-to-Completion

- Bad/OK news: Nothing else will happen until event handler returns
- Event handlers should never block (e.g., wait for input) --> all callbacks waiting for network response or user input are **always** asynchronous
- Event handlers shouldn't take a long time either

processing of  
event queue



*j will not execute until i finishes*



# Decomposing a long-running computation

- If you ***must*** do something that takes a long time (e.g. computation), split it into multiple events
  - `doSomeWork()`;
  - ... [let event loop process other events]..
  - `continueDoingMoreWork()`;
  - ...



# Dangers of Decomposition

- Application state may *change* before event occurs
  - Other event handlers may be interleaved and occur before event occurs and mutate the same application state
  - --> Need to check that update still makes sense
- Application state may be in *inconsistent* state until event occurs
- leaving data in inconsistent state...
- Loading some data from API, but not all of it...





# Sequencing events

- We'd like a better way to sequence events.
- Goals:
  - Clearly distinguish *synchronous* from *asynchronous* function calls.
  - Enable computation to occur only *after* some event has happened, without adding an additional nesting level each time (no pyramid of doom).
  - Make it possible to handle *errors*, including for multiple related async requests.
  - Make it possible to *wait* for multiple async calls to finish before proceeding.



# Sequencing events with Promises

- Promises are a wrapper around async callbacks
- Promises represents how to get a value
- Then you tell the promise what to do when it gets it
- Promises organize many steps that need to happen in order, with each step happening asynchronously
- At any point a promise is either:
  - Unresolved
  - Succeeds
  - Fails



# Using a Promise

- Declare what you want to do when your promise is completed (**then**), or if there's an error (**catch**)

```
fetch('https://github.com/')  
  .then(function(res) {  
    return res.text();  
  });
```

```
fetch('http://domain.invalid/')  
  .catch(function(err) {  
    console.log(err);  
  });
```

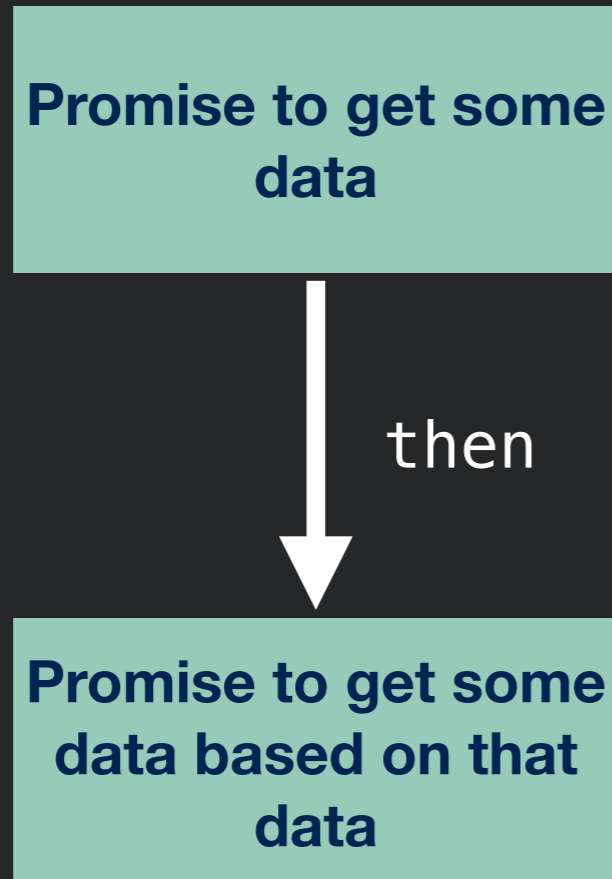


# Promise One Thing Then Another

**Promise to get some  
data**

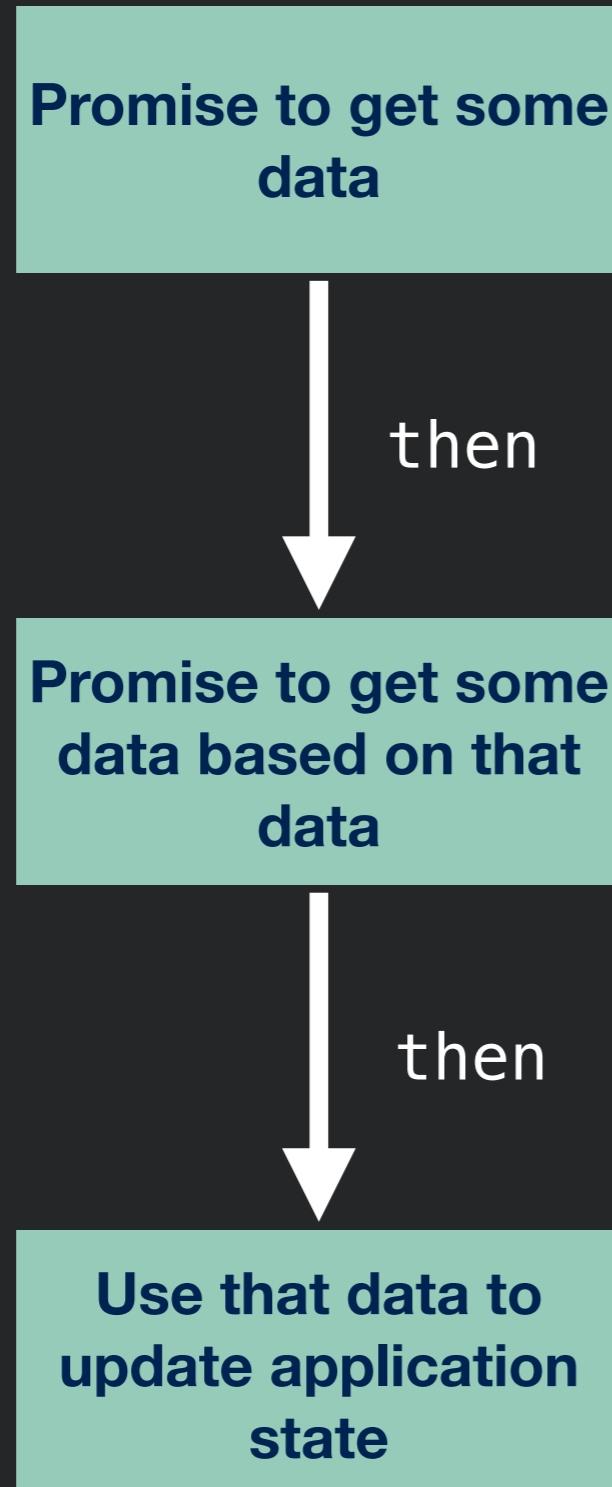


# Promise One Thing Then Another



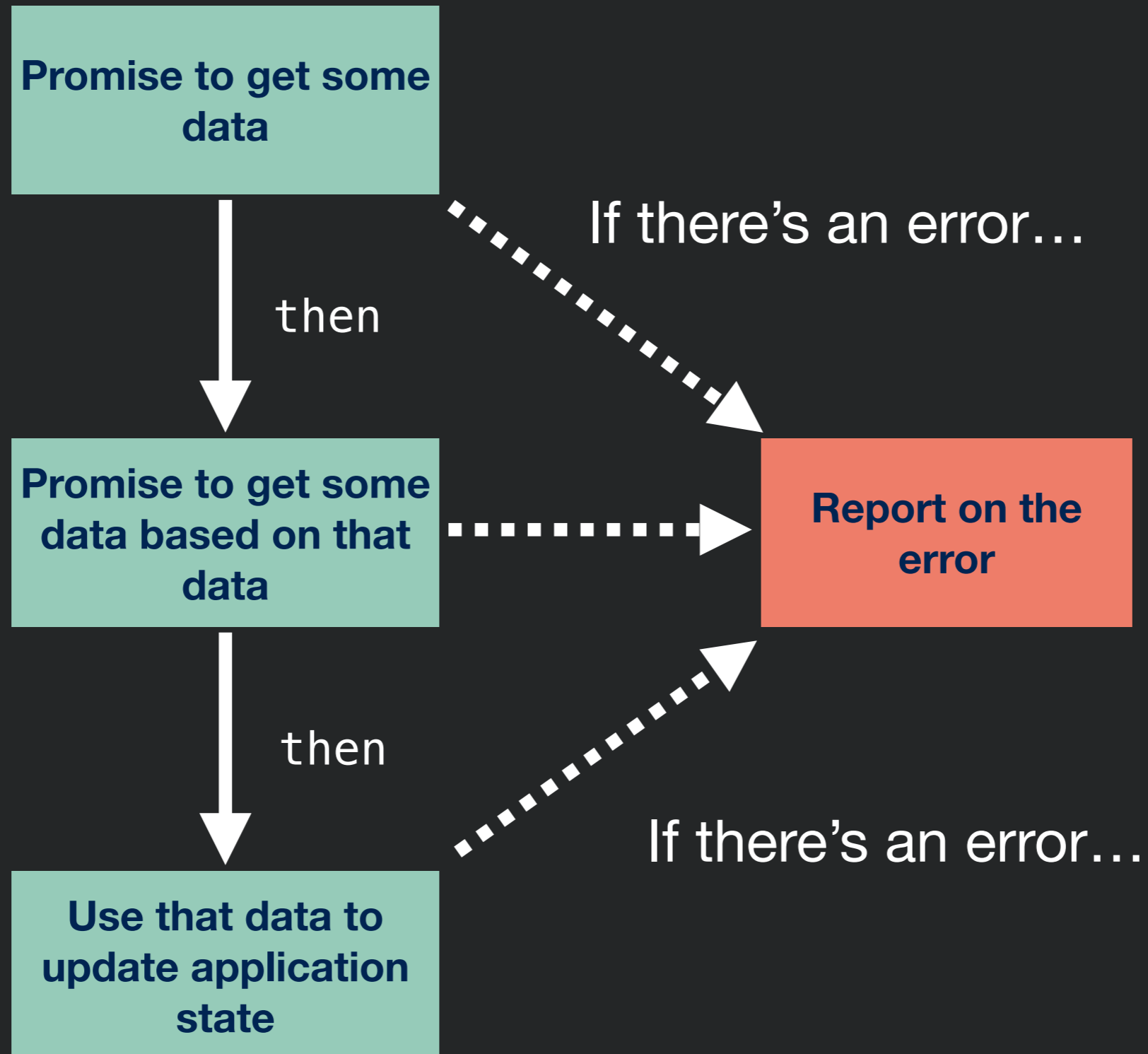


# Promise One Thing Then Another





# Promise One Thing Then Another





# Chaining Promises







# Chaining Promises

```
myPromise.then(function(resultOfPromise){  
    //Do something, maybe asynchronously  
    return theResultOfThisStep;  
})
```



# Chaining Promises

```
myPromise.then(function(resultOfPromise){  
    //Do something, maybe asynchronously  
    return theResultOfThisStep;  
})  
.then(function(resultOfStep1){  
    //Do something, maybe asynchronously  
    return theResultOfStep2;  
})
```



# Chaining Promises

```
myPromise.then(function(resultOfPromise){  
    //Do something, maybe asynchronously  
    return theResultOfThisStep;  
})  
    .then(function(resultOfStep1){  
        //Do something, maybe asynchronously  
        return theResultOfStep2;  
    })  
    .then(function(resultOfStep2){  
        //Do something, maybe asynchronously  
        return theResultOfStep3;  
    })
```



# Chaining Promises

```
myPromise.then(function(resultOfPromise){  
    //Do something, maybe asynchronously  
    return theResultOfThisStep;  
})  
  .then(function(resultOfStep1){  
    //Do something, maybe asynchronously  
    return theResultOfStep2;  
})  
  .then(function(resultOfStep2){  
    //Do something, maybe asynchronously  
    return theResultOfStep3;  
})  
  .then(function(resultOfStep3){  
    //Do something, maybe asynchronously  
    return theResultOfStep4;  
})
```



# Chaining Promises

```
myPromise.then(function(resultOfPromise){  
    //Do something, maybe asynchronously  
    return theResultOfThisStep;  
})  
.then(function(resultOfStep1){  
    //Do something, maybe asynchronously  
    return theResultOfStep2;  
})  
.then(function(resultOfStep2){  
    //Do something, maybe asynchronously  
    return theResultOfStep3;  
})  
.then(function(resultOfStep3){  
    //Do something, maybe asynchronously  
    return theResultOfStep4;  
})  
.catch(function(error){  
  
});
```

# Writing a Promise

- Most often, Promises will be generated by an API function (e.g., `fetch`) and returned to you.
- But you can also create your own Promise.

```
var p = new Promise(function(resolve, reject) {  
  if (/* condition */) {  
    resolve(/* value */); // fulfilled successfully  
  }  
  else {  
    reject(/* reason */); // error, rejected  
  }  
});
```

# Example: Writing a Promise

- loadImage returns a promise to load a given image

```
function loadImage(url){  
    return new Promise(function(resolve, reject) {  
        var img = new Image();  
        img.src = url;  
        img.onload = function(){  
            resolve(img);  
        }  
        img.onerror = function(e){  
            reject(e);  
        }  
    });  
}
```

Once the image is loaded, we'll resolve the promise

If the image has an error, the promise is rejected



# Writing a Promise

- Basic syntax:
  - do something (possibly asynchronous)
  - when you get the result, call `resolve()` and pass the final result
  - In case of error, call `reject()`

```
var p = new Promise( function(resolve, reject){  
    // do something, who knows how long it will take?  
    if(everythingIsOK)  
    {  
        resolve(stateIWantToSave);  
    }  
    else  
        reject(Error("Some error happened"));  
} );
```





# Promises in Action



# Promises in Action

- Firebase example: get some value from the database, then push some new value to the database, then print out “OK”



# Promises in Action

- Firebase example: get some value from the database, then push some new value to the database, then print out “OK”



# Promises in Action

- Firebase example: get some value from the database, then push some new value to the database, then print out “OK”





# Promises in Action

- Firebase example: get some value from the database, then push some new value to the database, then print out “OK”

```
todosRef.child(keyToGet).once('value')
  .then(function(foundTodo){
    return foundTodo.val().text;
  })
  .then(function(theText){
    todosRef.push({'text' : "Seriously: " + theText});
  })
  .then(function(){
    console.log("OK!");
  })
  .catch(function(error){
    //something went wrong
  });
```



# Promises in Action

- Firebase example: get some value from the database, then push some new value to the database, then print out “OK”

```
todosRef.child(keyToGet).once('value')  
  .then(function(foundTodo){  
    return foundTodo.val().text; Do this  
  })  
  .then(function(theText){  
    todosRef.push({'text' : "Seriously: " + theText});  
  })  
  .then(function(){  
    console.log("OK!");  
  })  
  .catch(function(error){  
    //something went wrong  
  });
```



# Promises in Action

- Firebase example: get some value from the database, then push some new value to the database, then print out “OK”

```
todosRef.child(keyToGet).once('value')
  .then(function(foundTodo){
    return foundTodo.val().text; Do this
  })
  .then(function(theText){ Then, do this
    todosRef.push({'text' : "Seriously: " + theText});
  })
  .then(function(){
    console.log("OK!");
  })
  .catch(function(error){
    //something went wrong
  });
```



# Promises in Action

- Firebase example: get some value from the database, then push some new value to the database, then print out “OK”

```
todosRef.child(keyToGet).once('value')
  .then(function(foundTodo){
    return foundTodo.val().text; Do this
  })
  .then(function(theText){ Then, do this
    todosRef.push({'text' : "Seriously: " + theText});
  })
  .then(function(){ Then do this
    console.log("OK!");
  })
  .catch(function(error){
    //something went wrong
  });
```





# Promises in Action

- Firebase example: get some value from the database, then push some new value to the database, then print out “OK”

```
todosRef.child(keyToGet).once('value')
  .then(function(foundTodo){
    return foundTodo.val().text; Do this
  })
  .then(function(theText){ Then, do this
    todosRef.push({'text' : "Seriously: " + theText});
  })
  .then(function(){ Then do this
    console.log("OK!");
  })
  .catch(function(error){
    //something went wrong
  });
```

**And if you ever had an error, do this**



# Testing Promises

```
function getUsername(userID) {  
  return request-promise('/users/' + userID).then(user => user.name);  
}
```



# Testing Promises

```
function getUser_name(userID) {  
  return request-promise('/users/' + userID).then(user => user.name);  
}
```

```
it('works with promises', () => {  
  expect(user.getUser_name(4).toEqual('Mark'));  
});
```



# Testing Promises

```
function getUser_name(userID) {  
  return request-promise('/users/' + userID).then(user => user.name);  
}
```

```
it('works with promises', () => {  
  expect(user.getUser_name(4).toEqual('Mark'));  
});
```

```
it('works with promises', () => {  
  expect.assertions(1);  
  return user.getUser_name(4).then(data => expect(data).toEqual('Mark'));  
});
```



# Testing Promises

```
function getUserName(userID) {  
  return request-promise('/users/' + userID).then(user => user.name);  
}
```

```
it('works with promises', () => {  
  expect(user.getUserName(4).toEqual('Mark'));  
});
```

```
it('works with promises', () => {  
  expect.assertions(1);  
  return user.getUserName(4).then(data => expect(data).toEqual('Mark'));  
});
```

```
it('works with resolves', () => {  
  expect.assertions(1);  
  return expect(user.getUserName(5)).resolves.toEqual('Paul');  
});
```



# Testing Promises

```
function getUsername(userID) {  
  return request-promise('/users/' + userID).then(user => user.name);  
}
```

```
it('works with promises', () => {  
  expect(user.getUsername(4)).toEqual('Mark');  
});
```



```
it('works with promises', () => {  
  expect.assertions(1);  
  return user.getUsername(4).then(data => expect(data).toEqual('Mark'));  
});
```

```
it('works with resolves', () => {  
  expect.assertions(1);  
  return expect(user.getUsername(5)).resolves.toEqual('Paul');  
});
```

# SWE 432 - Web Application Development

---



George Mason  
University

---

Instructor:  
Dr. Kevin Moran

Teaching Assistant:  
David Gonzalez Samudio

Class will start in:

10:01

# SWE 432 - Web Application Development

---



George Mason  
University

---

Instructor:  
Dr. Kevin Moran

Teaching Assistant:  
David Gonzalez Samudio

Class will start in:

10:01



# Asynchronous Programming II





# Review: Asynchronous

- Synchronous:
  - Make a function call
  - When function call returns, the work is done
- Asynchronous:
  - Make a function call
  - Function returns immediately, before completing work!



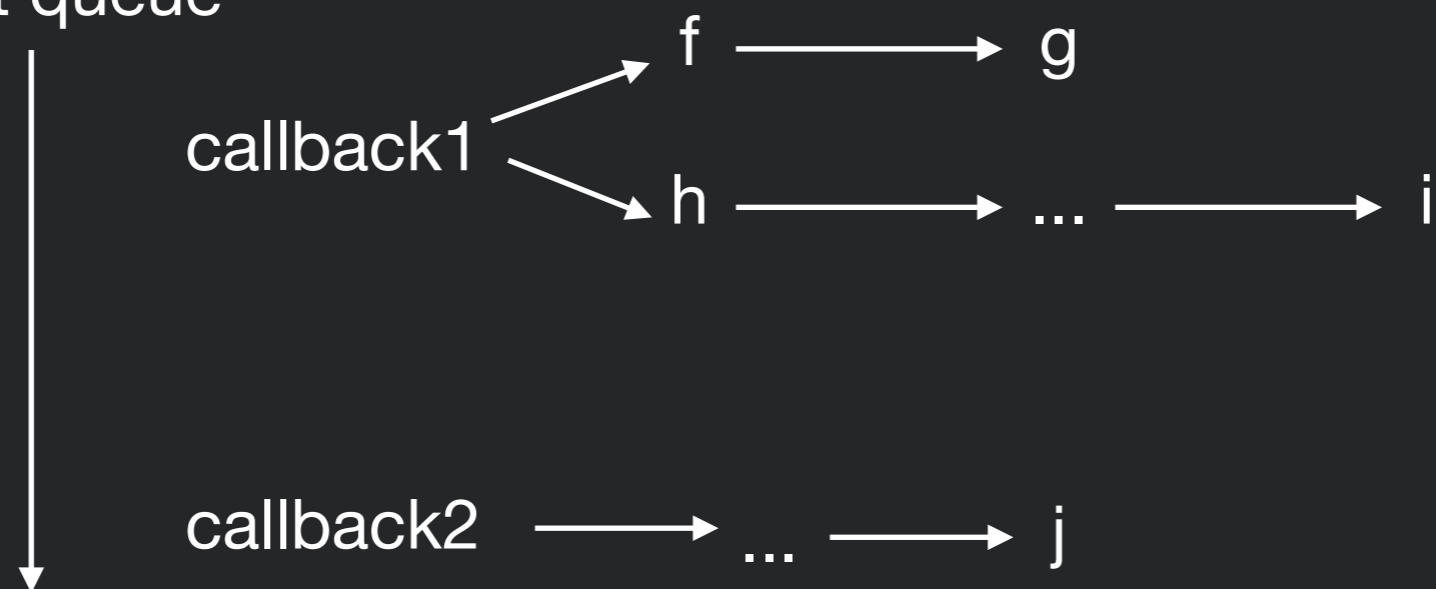
# Review: Asynchronous

- How we do multiple things at a time in JS
- NodeJS magically handles these asynchronous things in the background
- Really important when doing file/network input/output

# Review: Run-to-completion semantics

- Run-to-completion
  - The function handling an event and the functions that it (transitively) synchronously calls will keep executing until the function finishes.
  - The JS engine will not handle the next event until the event handler finishes.

processing of  
event queue

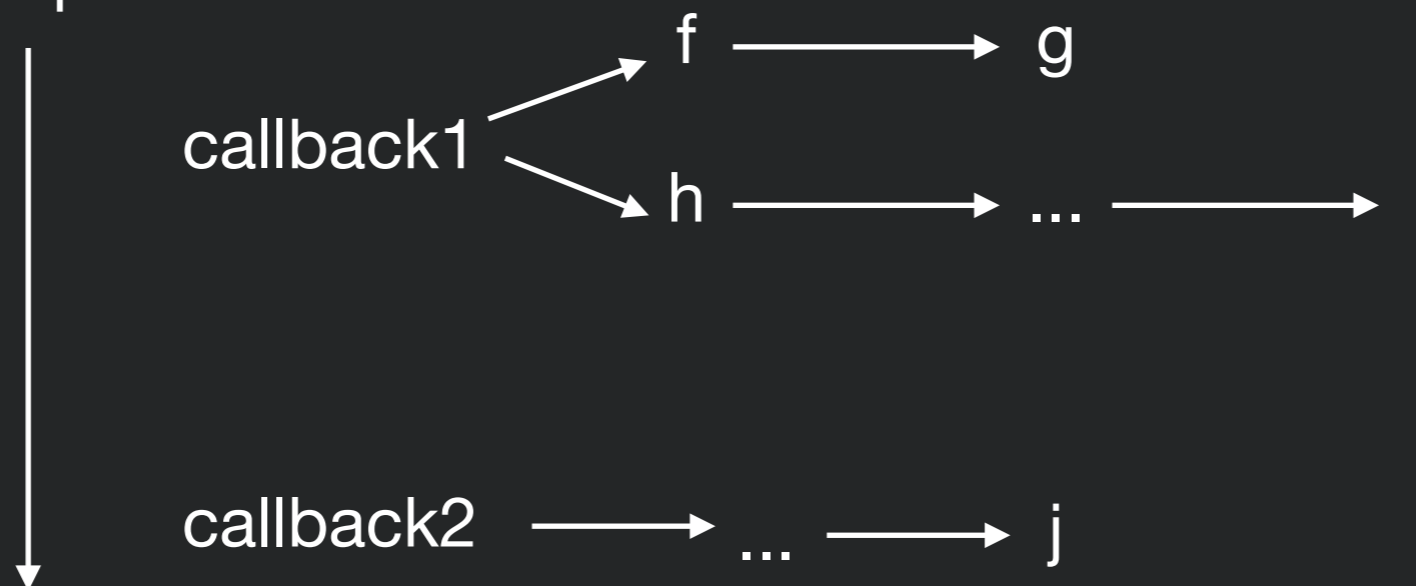




# Review: Implications of run-to-completion

- Good news: no other code will run until you finish (no worries about other threads overwriting your data)

processing of  
event queue

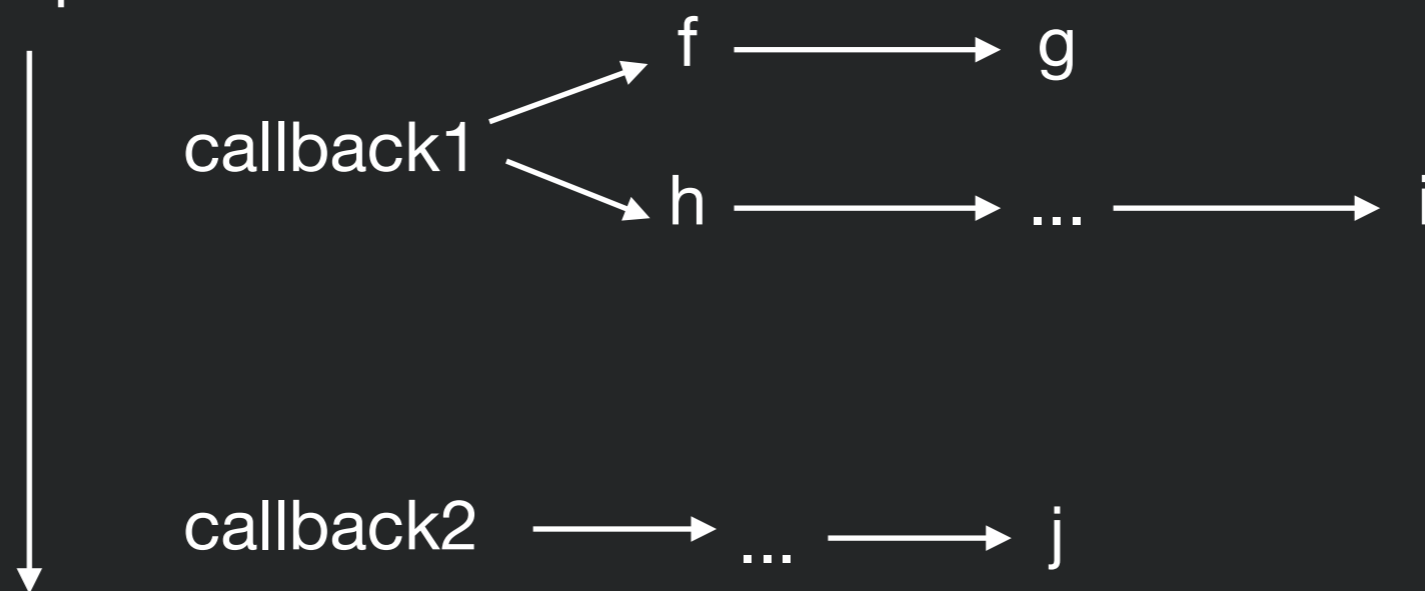


*j will not execute until after i*

# Review: Implications of run-to-completion

- Bad/OK news: Nothing else will happen until event handler returns
  - Event handlers should never block (e.g., wait for input) --> all callbacks waiting for network response or user input are **always** asynchronous
  - Event handlers shouldn't take a long time either

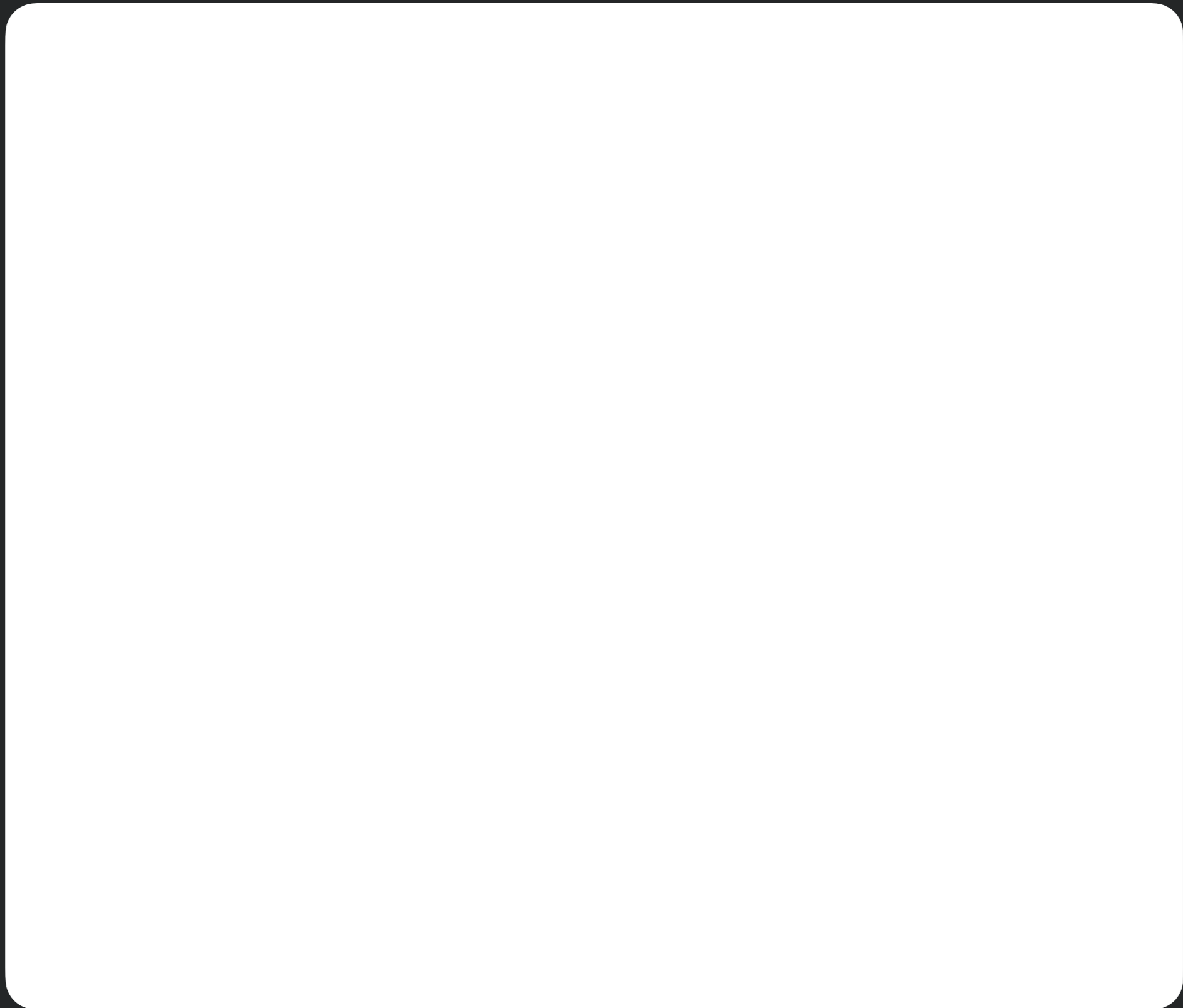
processing of  
event queue



*j will not execute until i finishes*



# Review: Chaining Promises





# Review: Chaining Promises

```
myPromise.then(function(resultOfPromise){  
    //Do something, maybe asynchronously  
    return theResultOfThisStep;  
})
```





# Review: Chaining Promises

```
myPromise.then(function(resultOfPromise){  
    //Do something, maybe asynchronously  
    return theResultOfThisStep;  
})  
    .then(function(resultOfStep1){  
        //Do something, maybe asynchronously  
        return theResultOfStep2;  
    })
```



# Review: Chaining Promises

```
myPromise.then(function(resultOfPromise){  
    //Do something, maybe asynchronously  
    return theResultOfThisStep;  
})  
  .then(function(resultOfStep1){  
    //Do something, maybe asynchronously  
    return theResultOfStep2;  
})  
  .then(function(resultOfStep2){  
    //Do something, maybe asynchronously  
    return theResultOfStep3;  
})
```



# Review: Chaining Promises

```
myPromise.then(function(resultOfPromise){  
    //Do something, maybe asynchronously  
    return theResultOfThisStep;  
})  
.  
then(function(resultOfStep1){  
    //Do something, maybe asynchronously  
    return theResultOfStep2;  
})  
.  
then(function(resultOfStep2){  
    //Do something, maybe asynchronously  
    return theResultOfStep3;  
})  
.  
then(function(resultOfStep3){  
    //Do something, maybe asynchronously  
    return theResultOfStep4;  
})
```



# Review: Chaining Promises

```
myPromise.then(function(resultOfPromise){
    //Do something, maybe asynchronously
    return theResultOfThisStep;
})
  .then(function(resultOfStep1){
    //Do something, maybe asynchronously
    return theResultOfStep2;
})
  .then(function(resultOfStep2){
    //Do something, maybe asynchronously
    return theResultOfStep3;
})
  .then(function(resultOfStep3){
    //Do something, maybe asynchronously
    return theResultOfStep4;
})
  .catch(function(error){

  });
```



# Current Lecture

- Async/await
- Programming activity

# Promising many things

- Can also specify that \*many\* things should be done, and then something else
- Example: load a whole bunch of images at once:

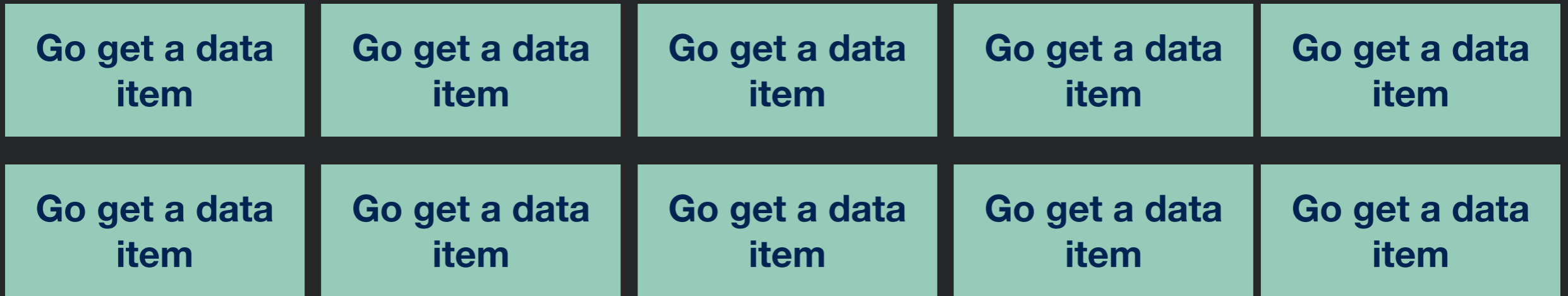
Promise

```
.all([loadImage("GMURGB.jpg"), loadImage("CS.jpg")])  
.then(function (imgArray) {  
    imgArray.forEach(img => {document.body.appendChild(img)})  
})  
.catch(function (e) {  
    console.log("Oops");  
    console.log(e);  
});
```



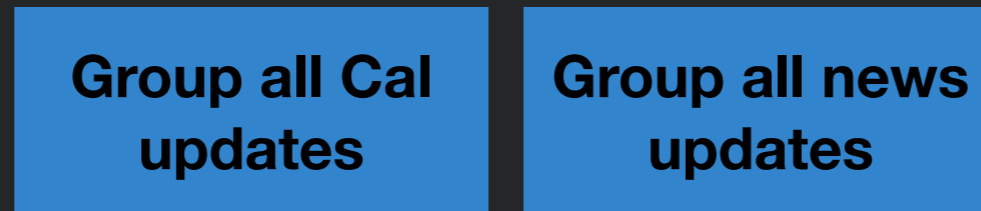
# Async Programming Example

1 second each

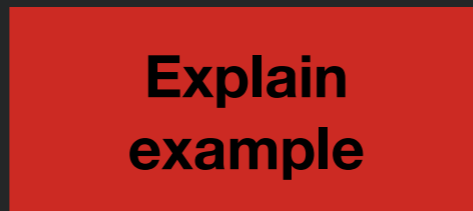
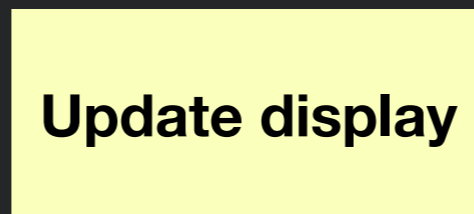


2 seconds each

thenCombine



when done





# Synchronous Version





# Synchronous Version

**Go get a data  
item**



# Synchronous Version

**Go get a data  
item**

**Go get a data  
item**



# Synchronous Version

**Go get a data  
item**

**Go get a data  
item**

**Go get a data  
item**



# Synchronous Version

**Go get a data  
item**

**Go get a data  
item**

**Go get a data  
item**

**Go get a data  
item**



# Synchronous Version

**Go get a data  
item**

**Go get a data  
item**

**Go get a data  
item**

**Go get a data  
item**

**Go get a data  
item**



# Synchronous Version

**Go get a data  
item**

**Go get a data  
item**

**Go get a data  
item**

**Go get a data  
item**

**Go get a data  
item**

**Go get a data  
item**



# Synchronous Version

**Go get a data  
item**

**Go get a data  
item**

**Go get a data  
item**

**Go get a data  
item**

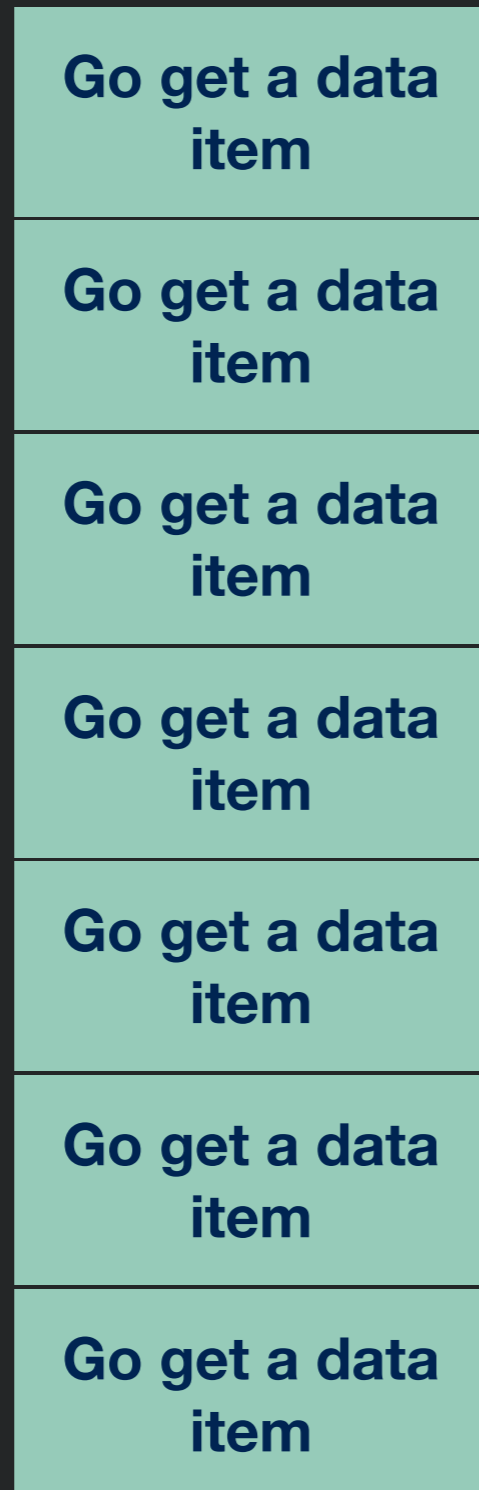
**Go get a data  
item**

**Go get a data  
item**

**Go get a data  
item**



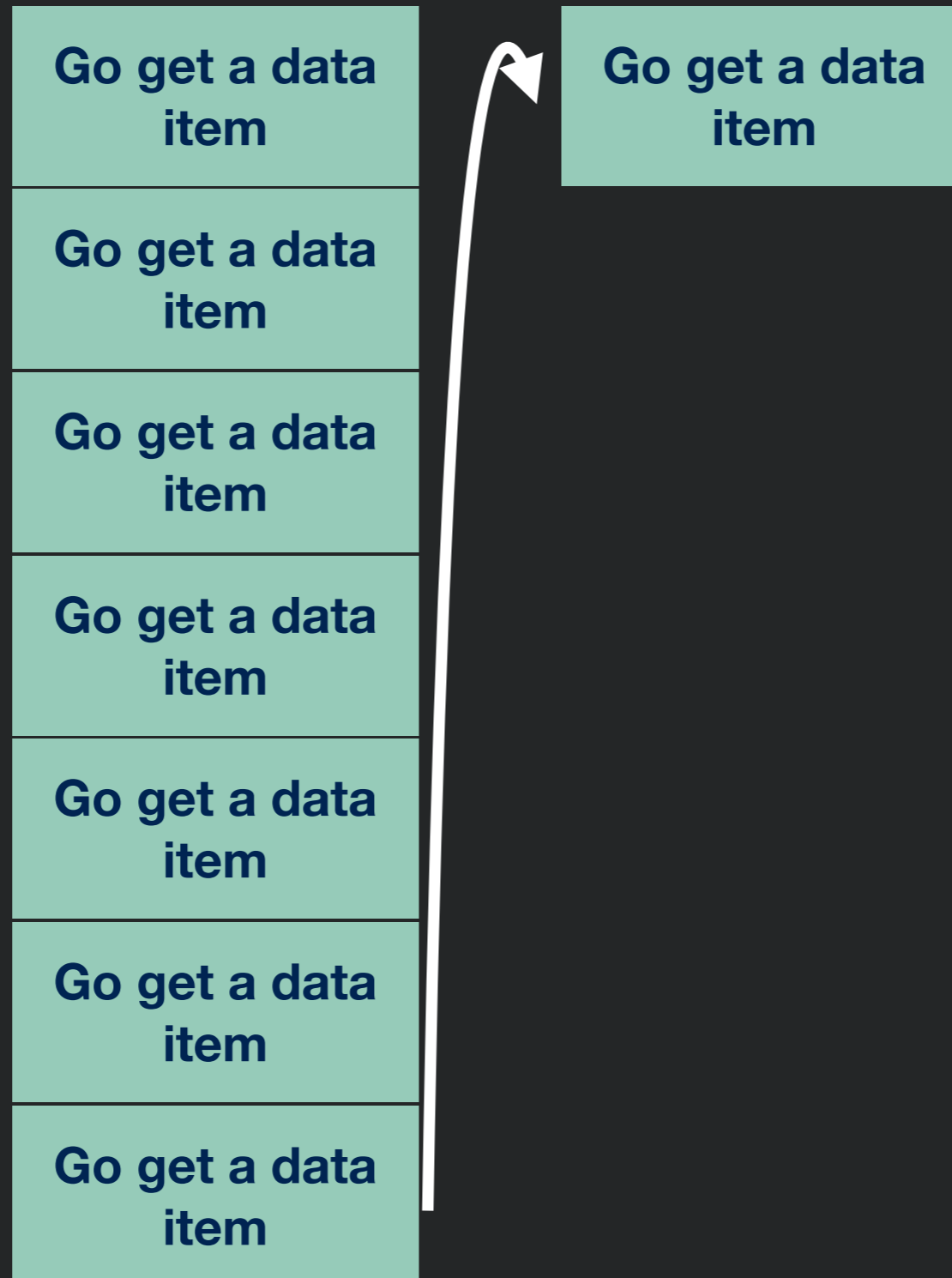
# Synchronous Version





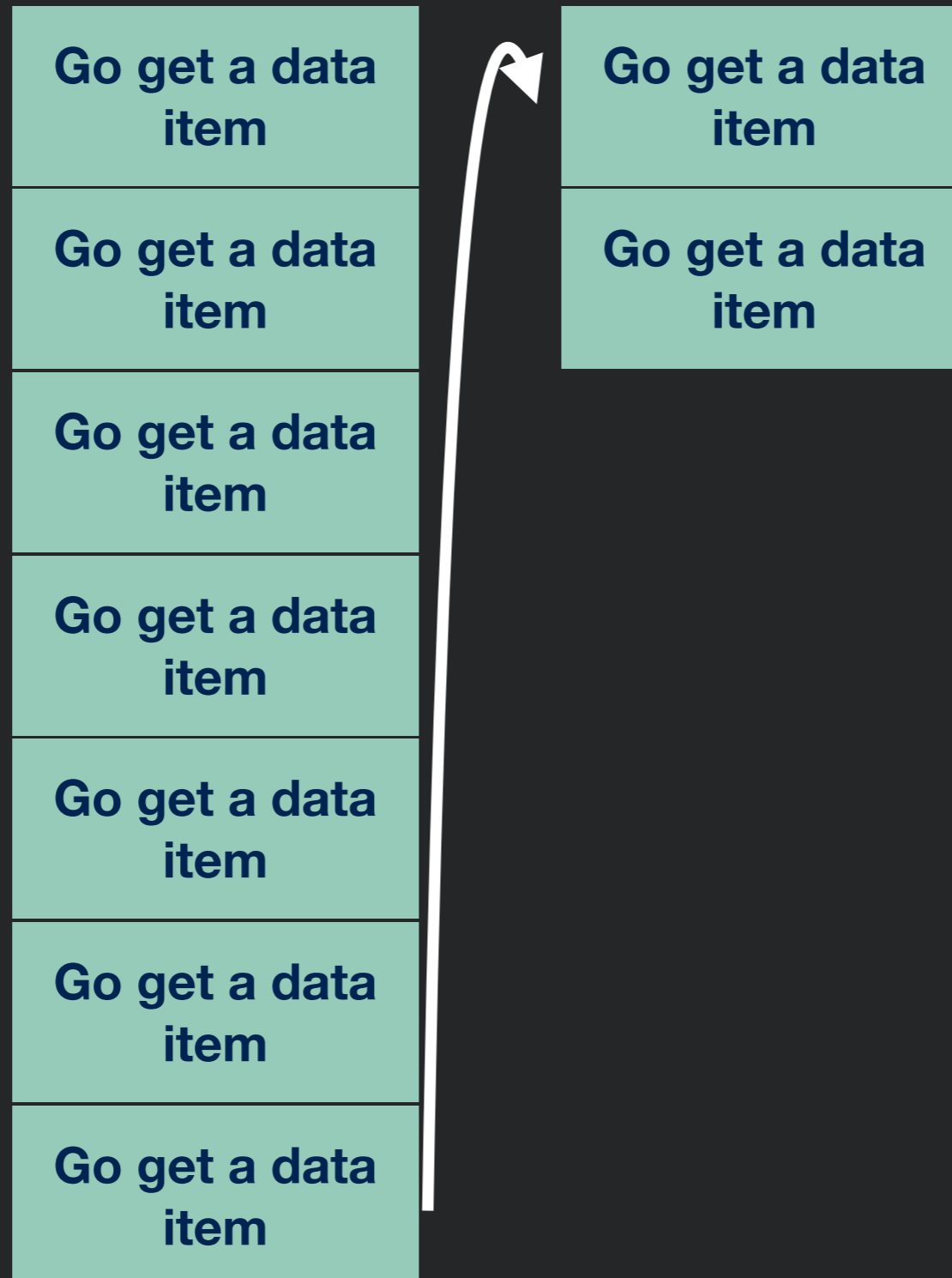


# Synchronous Version



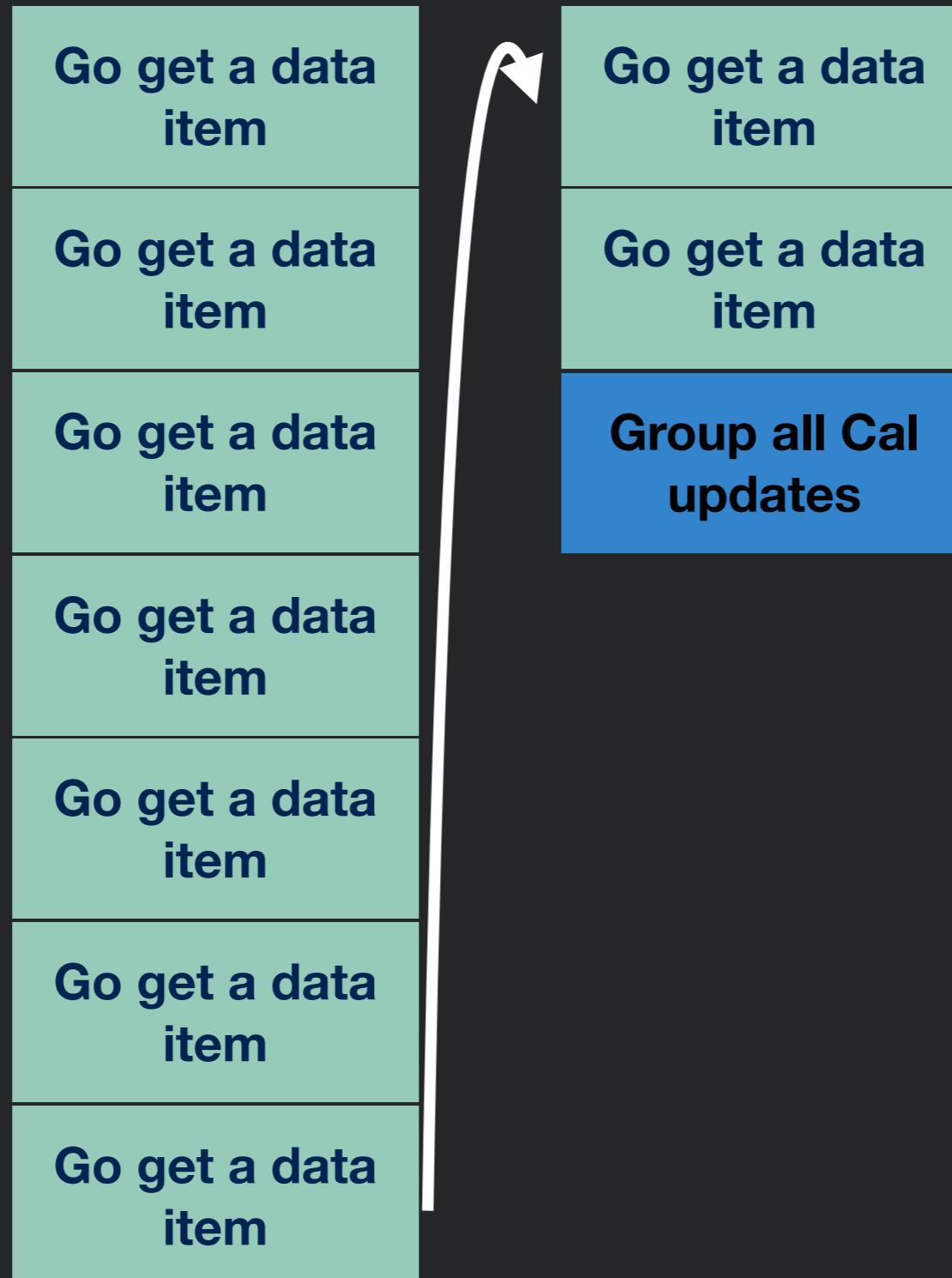


# Synchronous Version





# Synchronous Version



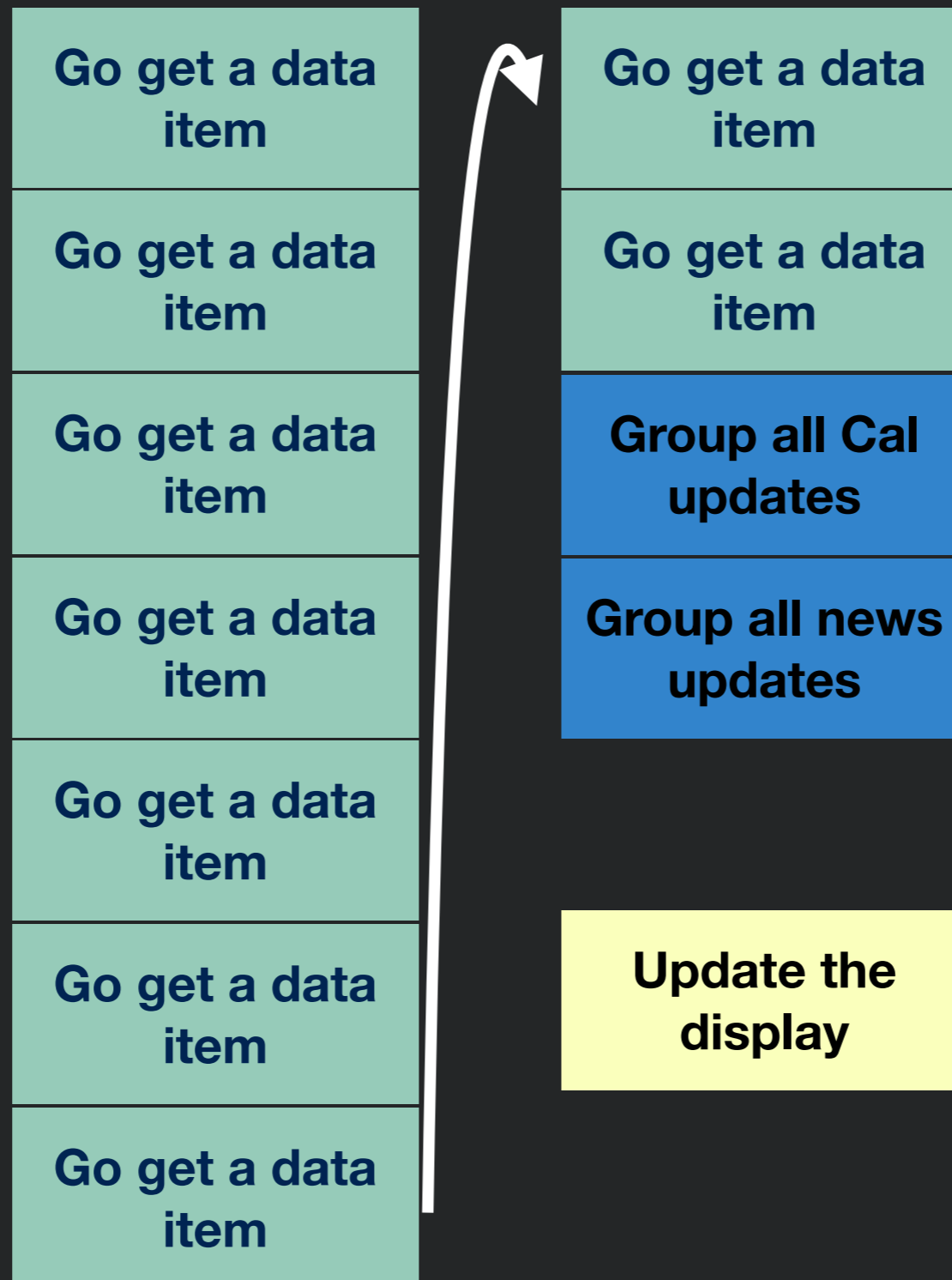


# Synchronous Version



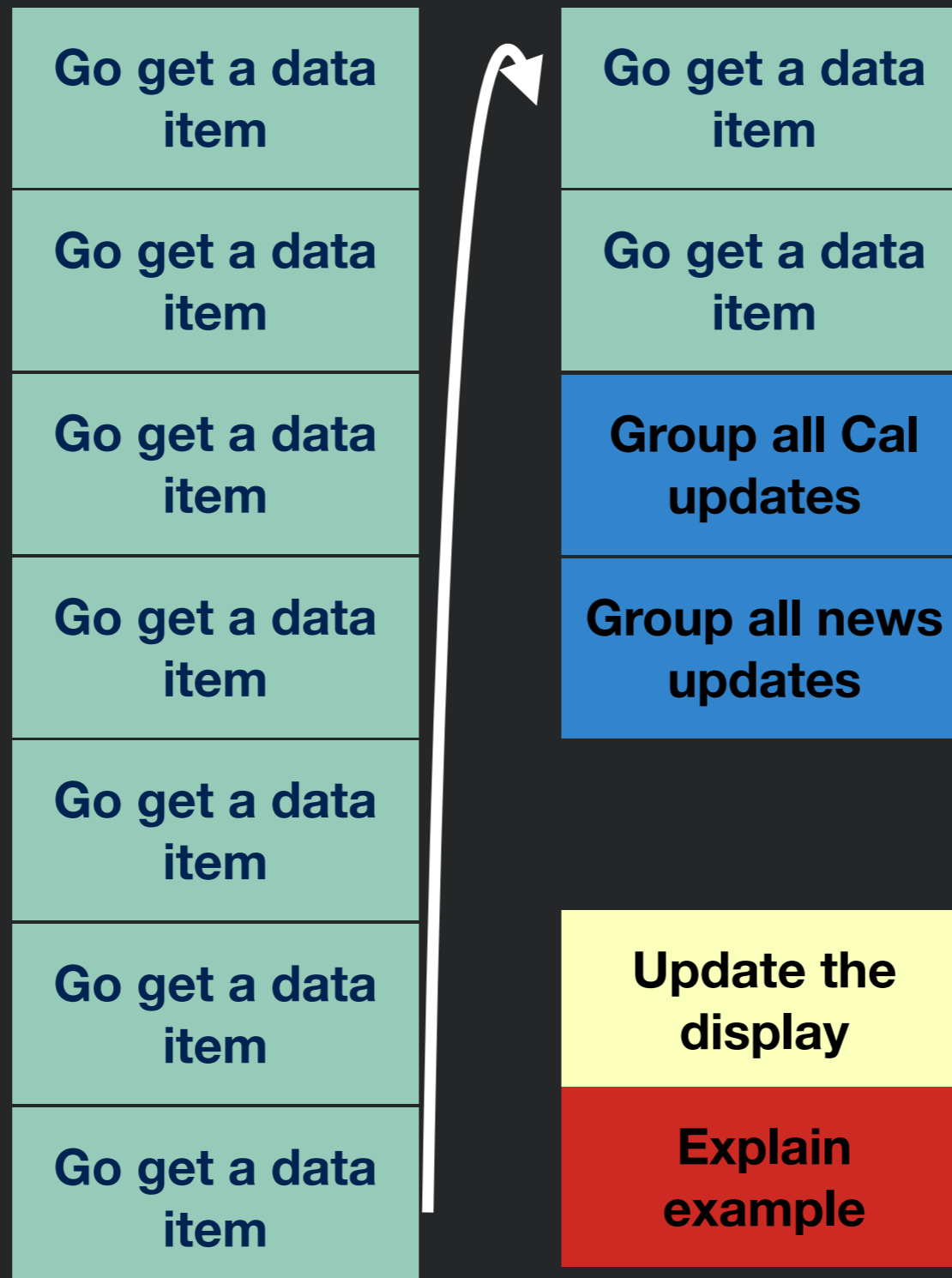


# Synchronous Version





# Synchronous Version





# Asynchronous Version



# Asynchronous Version

Go get a data  
item

Go get a data  
item

Go get a data  
item

Go get a data  
item

Go get a data  
item

Go get a data  
item

Go get a data  
item

Go get a data  
item

Go get a data  
item

Go get a data  
item

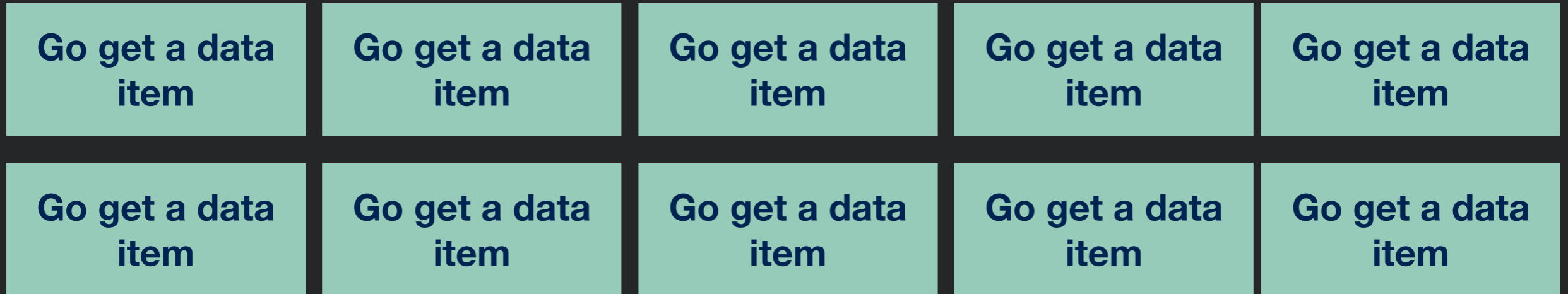
...

Explain  
example





# Asynchronous Version



...

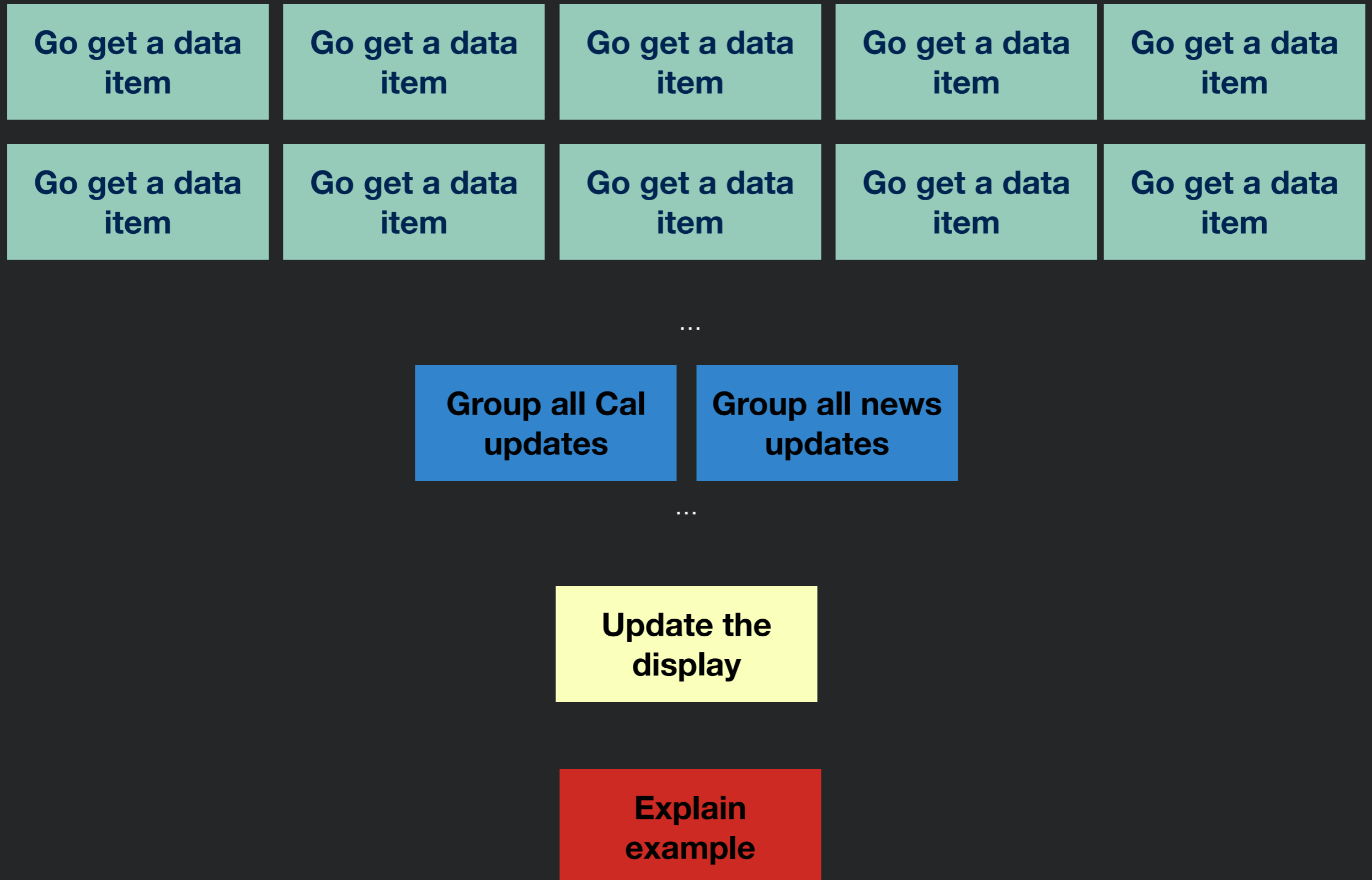


...

**Explain example**



# Asynchronous Version





# Async Programming Example (Sync)

```
let lib = require("./lib.js");

let thingsToFetch = ['t1', 't2', 't3', 's1', 's2',
  's3', 'm1', 'm2', 'm3', 't4'];
let stuff = [];
for(let thingToGet of thingsToFetch)
{
  stuff.push(lib.getSync(thingToGet));
  console.log("Got a thing");
}
//Got all my stuff
let ts = lib.groupSync(stuff, "t");
console.log("Grouped");
let ms = lib.groupSync(stuff, "m");
console.log("Grouped");
let ss = lib.groupSync(stuff, "s");
console.log("Grouped");

console.log("Done");
```

```
node v12.16.1
```

```
□
```



# Async Programming Example (Sync)

```
let lib = require("./lib.js");

let thingsToFetch = ['t1', 't2', 't3', 's1', 's2',
  's3', 'm1', 'm2', 'm3', 't4'];
let stuff = [];
for(let thingToGet of thingsToFetch)
{
  stuff.push(lib.getSync(thingToGet));
  console.log("Got a thing");
}
//Got all my stuff
let ts = lib.groupSync(stuff, "t");
console.log("Grouped");
let ms = lib.groupSync(stuff, "m");
console.log("Grouped");
let ss = lib.groupSync(stuff, "s");
console.log("Grouped");

console.log("Done");
```

```
node v12.16.1
```

```
█
```



# Async Programming Example (Callbacks, no parallelism)

```
let lib = require("./lib.js");

let thingsToFetch = ['t1', 't2', 't3', 's1', 's2', 's3', 'm1', 'm2', 'm3', 't4'];
let stuff = [];
let ts, ms, ss;
let outstandingStuffToGet = thingsToFetch.length;

lib.getASync(thingsToFetch[0], (v) => {
  stuff.push(v);
  console.log("Got a thing")
  lib.getASync(thingsToFetch[1], (v) => {
    stuff.push(v);
    console.log("Got a thing")
    lib.getASync(thingsToFetch[2], (v) => {
      stuff.push(v);
      console.log("Got a thing")
      lib.getASync(thingsToFetch[3], (v) => {
        stuff.push(v);
        console.log("Got a thing")
        lib.getASync(thingsToFetch[4], (v) => {
          stuff.push(v);
          console.log("Got a thing")
          lib.getASync(thingsToFetch[5], (v) => {
            stuff.push(v);
            console.log("Got a thing")
            lib.getASync(thingsToFetch[6], (v) => {
              stuff.push(v);
              console.log("Got a thing")
              lib.getASync(thingsToFetch[7], (v) => {
                stuff.push(v);
                console.log("Got a thing")
                lib.getASync(thingsToFetch[8], (v) => {
                  stuff.push(v);
                  console.log("Got a thing")
                  lib.getASync(thingsToFetch[9], (v) => {
                    stuff.push(v);
                    console.log("Got a thing")
                    lib.groupASync(stuff, "t", (t) => {
                      ts = t;
                      console.log("Grouped");
                      lib.groupASync(stuff, "m", (m) => {
                        ss = s;
                        console.log("Grouped");
```

```
node v12.16.1
```

```
█
```



# Async Programming Example (Callbacks, no parallelism)

```
let lib = require("./lib.js");

let thingsToFetch = ['t1', 't2', 't3', 's1', 's2', 's3', 'm1', 'm2', 'm3', 't4'];
let stuff = [];
let ts, ms, ss;
let outstandingStuffToGet = thingsToFetch.length;

lib.getASync(thingsToFetch[0], (v) => {
  stuff.push(v);
  console.log("Got a thing")
  lib.getASync(thingsToFetch[1], (v) => {
    stuff.push(v);
    console.log("Got a thing")
    lib.getASync(thingsToFetch[2], (v) => {
      stuff.push(v);
      console.log("Got a thing")
      lib.getASync(thingsToFetch[3], (v) => {
        stuff.push(v);
        console.log("Got a thing")
        lib.getASync(thingsToFetch[4], (v) => {
          stuff.push(v);
          console.log("Got a thing")
          lib.getASync(thingsToFetch[5], (v) => {
            stuff.push(v);
            console.log("Got a thing")
            lib.getASync(thingsToFetch[6], (v) => {
              stuff.push(v);
              console.log("Got a thing")
              lib.getASync(thingsToFetch[7], (v) => {
                stuff.push(v);
                console.log("Got a thing")
                lib.getASync(thingsToFetch[8], (v) => {
                  stuff.push(v);
                  console.log("Got a thing")
                  lib.getASync(thingsToFetch[9], (v) => {
                    stuff.push(v);
                    console.log("Got a thing")
                    lib.groupAsync(stuff, "t", (t) => {
                      ts = t;
                      console.log("Grouped");
                      lib.groupAsync(stuff, "m", (m) => {
                        ss = s;
                        console.log("Grouped");
```

```
node v12.16.1
```

```
█
```



# Async Programming Example (Callbacks)

```
let lib = require("./lib.js");

let thingsToFetch = ['t1', 't2', 't3', 's1', 's2', 's3', 'm1', 'm2', 'm3', 't4'];
let stuff = [];
let ts, ms, ss;
let outstandingStuffToGet = thingsToFetch.length;
for (let thingToGet of thingsToFetch) {
  lib.getAsync(thingToGet, (v) => {
    stuff.push(v);
    console.log("Got a thing")
    outstandingStuffToGet--;
    if (outstandingStuffToGet == 0) {
      let groupsOfStuffToGetStill = 3;
      lib.groupAsync(stuff, "t", (t) => {
        ts = t;
        console.log("Grouped");
        groupsOfStuffToGetStill--;
        if (groupsOfStuffToGetStill == 0)
          console.log("Done");
      });
      lib.groupAsync(stuff, "m", (m) => {
        ms = m;
        console.log("Grouped");
        groupsOfStuffToGetStill--;
        if (groupsOfStuffToGetStill == 0)
          console.log("Done");
      });
      lib.groupAsync(stuff, "s", (s) => {
        ss = s;
        console.log("Grouped");
        groupsOfStuffToGetStill--;
        if (groupsOfStuffToGetStill == 0)
          console.log("Done");
      });
    }
  });
}
```

node v12.16.1

```
[ ]
```



# Async Programming Example (Callbacks)

```
let lib = require("./lib.js");

let thingsToFetch = ['t1', 't2', 't3', 's1', 's2', 's3', 'm1', 'm2', 'm3', 't4'];
let stuff = [];
let ts, ms, ss;
let outstandingStuffToGet = thingsToFetch.length;
for (let thingToGet of thingsToFetch) {
  lib.getAsync(thingToGet, (v) => {
    stuff.push(v);
    console.log("Got a thing")
    outstandingStuffToGet--;
    if (outstandingStuffToGet == 0) {
      let groupsOfStuffToGetStill = 3;
      lib.groupAsync(stuff, "t", (t) => {
        ts = t;
        console.log("Grouped");
        groupsOfStuffToGetStill--;
        if (groupsOfStuffToGetStill == 0)
          console.log("Done");
      });
      lib.groupAsync(stuff, "m", (m) => {
        ms = m;
        console.log("Grouped");
        groupsOfStuffToGetStill--;
        if (groupsOfStuffToGetStill == 0)
          console.log("Done");
      });
      lib.groupAsync(stuff, "s", (s) => {
        ss = s;
        console.log("Grouped");
        groupsOfStuffToGetStill--;
        if (groupsOfStuffToGetStill == 0)
          console.log("Done");
      });
    }
  });
}
```

```
node v12.16.1
```

```
█
```





# Async Programming Example (Promises, no parallelism)

```
let lib = require("./lib.js");

let thingsToFetch = ['t1', 't2', 't3', 's1', 's2', 's3', 'm1', 'm2', 'm3', 't4'];
let stuff = [];
let ts, ms, ss;
let outstandingStuffToGet = thingsToFetch.length;
lib.getPromise(thingsToFetch[0]).then(
  (v)=>{
    stuff.push(v);
    console.log("Got a thing");
    return lib.getPromise(thingsToFetch[1]);
  }
).then(
  (v)=>{
    stuff.push(v);
    console.log("Got a thing");
    return lib.getPromise(thingsToFetch[1]);
  }
).then(
  (v)=>{
    stuff.push(v);
    console.log("Got a thing");
    return lib.getPromise(thingsToFetch[1]);
  }
).then(
  (v)=>{
    stuff.push(v);
    console.log("Got a thing");
    return lib.getPromise(thingsToFetch[2]);
  }
).then(
  (v)=>{
    stuff.push(v);
    console.log("Got a thing");
    return lib.getPromise(thingsToFetch[3]);
  }
).then(
  (v)=>{
    stuff.push(v);
    console.log("Got a thing");
    return lib.getPromise(thingsToFetch[4]);
  }
);
```

```
node v12.16.1
```

```
█
```



# Async Programming Example (Promises, no parallelism)

```
let lib = require("./lib.js");

let thingsToFetch = ['t1', 't2', 't3', 's1', 's2', 's3', 'm1', 'm2', 'm3', 't4'];
let stuff = [];
let ts, ms, ss;
let outstandingStuffToGet = thingsToFetch.length;
lib.getPromise(thingsToFetch[0]).then(
  (v)=>{
    stuff.push(v);
    console.log("Got a thing");
    return lib.getPromise(thingsToFetch[1]);
  }
).then(
  (v)=>{
    stuff.push(v);
    console.log("Got a thing");
    return lib.getPromise(thingsToFetch[1]);
  }
).then(
  (v)=>{
    stuff.push(v);
    console.log("Got a thing");
    return lib.getPromise(thingsToFetch[1]);
  }
).then(
  (v)=>{
    stuff.push(v);
    console.log("Got a thing");
    return lib.getPromise(thingsToFetch[2]);
  }
).then(
  (v)=>{
    stuff.push(v);
    console.log("Got a thing");
    return lib.getPromise(thingsToFetch[3]);
  }
).then(
  (v)=>{
    stuff.push(v);
    console.log("Got a thing");
    return lib.getPromise(thingsToFetch[4]);
  }
);
```

```
node v12.16.1
```

```
█
```



# Async Programming Example (Promises)

```
let lib = require("./lib.js");

let thingsToFetch = ['t1', 't2', 't3', 's1', 's2', 's3',
                    'm1', 'm2', 'm3', 't4'];
let stuff = [];
let ts, ms, ss;

let promises = [];
for (let thingToGet of thingsToFetch) {
  promises.push(lib.getPromise(thingToGet));
}
Promise.all(promises).then((data) => {
  console.log("Got all things");
  stuff = data;
  return Promise.all([
    lib.groupPromise(stuff, "t"),
    lib.groupPromise(stuff, "m"),
    lib.groupPromise(stuff, "s")
  ])
})
  .then((groups) => {
    console.log("Got all groups");
    ts = groups[0];
    ms = groups[1];
    ss = groups[2];
    console.log("Done");
  });
```

```
node v12.16.1
```

```
█
```



# Async Programming Example (Promises)

```
let lib = require("./lib.js");

let thingsToFetch = ['t1', 't2', 't3', 's1', 's2', 's3',
                    'm1', 'm2', 'm3', 't4'];
let stuff = [];
let ts, ms, ss;

let promises = [];
for (let thingToGet of thingsToFetch) {
  promises.push(lib.getPromise(thingToGet));
}
Promise.all(promises).then((data) => {
  console.log("Got all things");
  stuff = data;
  return Promise.all([
    lib.groupPromise(stuff, "t"),
    lib.groupPromise(stuff, "m"),
    lib.groupPromise(stuff, "s")
  ])
})
  .then((groups) => {
    console.log("Got all groups");
    ts = groups[0];
    ms = groups[1];
    ss = groups[2];
    console.log("Done");
  });
```

```
node v12.16.1
```

```
█
```



# Problems with Promises

```
const makeRequest = () => {
  try {
    return promise1()
      .then(value1 => {
        // do something
      }).catch(err => {
        //This is the only way to catch async errors
        console.log(err);
      })
  } catch(ex) {
    //Will never catch async errors!!
  }
}
```



# Async/Await

- The latest and greatest way to work with async functions
- A programming pattern that tries to make async code look more synchronous
- Just “await” something to happen before proceeding
- <https://javascript.info/async-await>



# Async keyword

- Denotes a function that can block and resume execution later

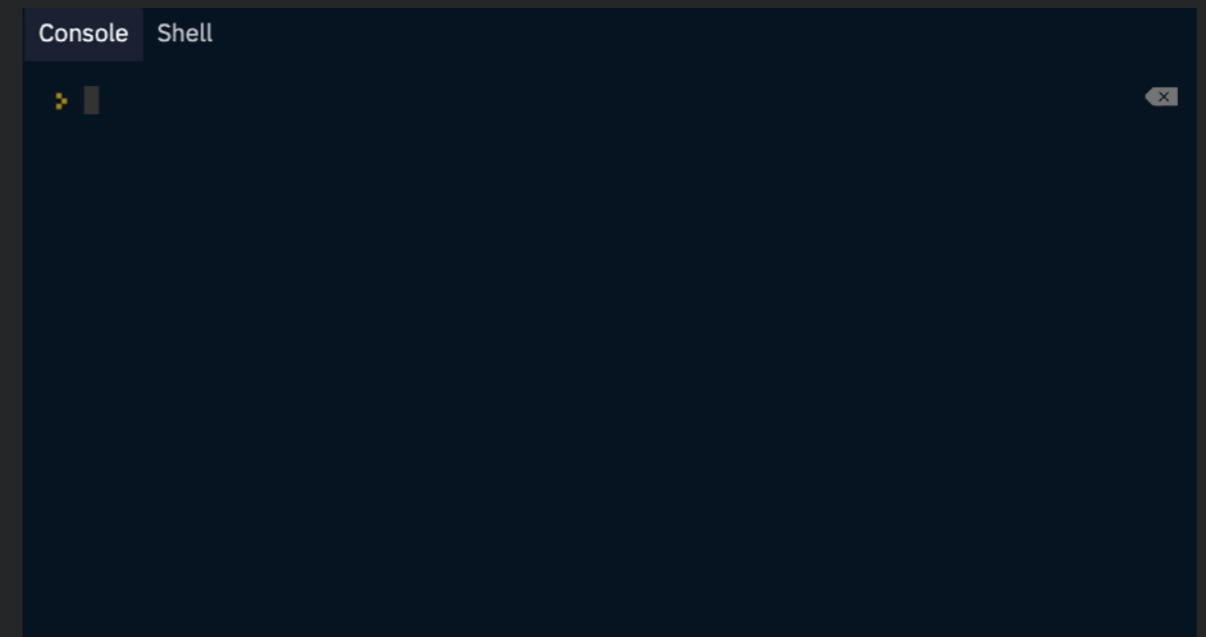
```
async function hello() { return "Hello" };  
hello();
```

- Automatically turns the return type into a Promise



# Async/Await Example

```
function resolveAfter2Seconds() {  
  return new Promise(resolve => {  
    setTimeout(() => {  
      resolve('resolved');  
    }, 2000);  
  });  
}  
  
async function asyncCall() {  
  console.log('calling');  
  var result = await  
  resolveAfter2Seconds();  
  console.log(result);  
  // expected output: 'resolved'  
}
```



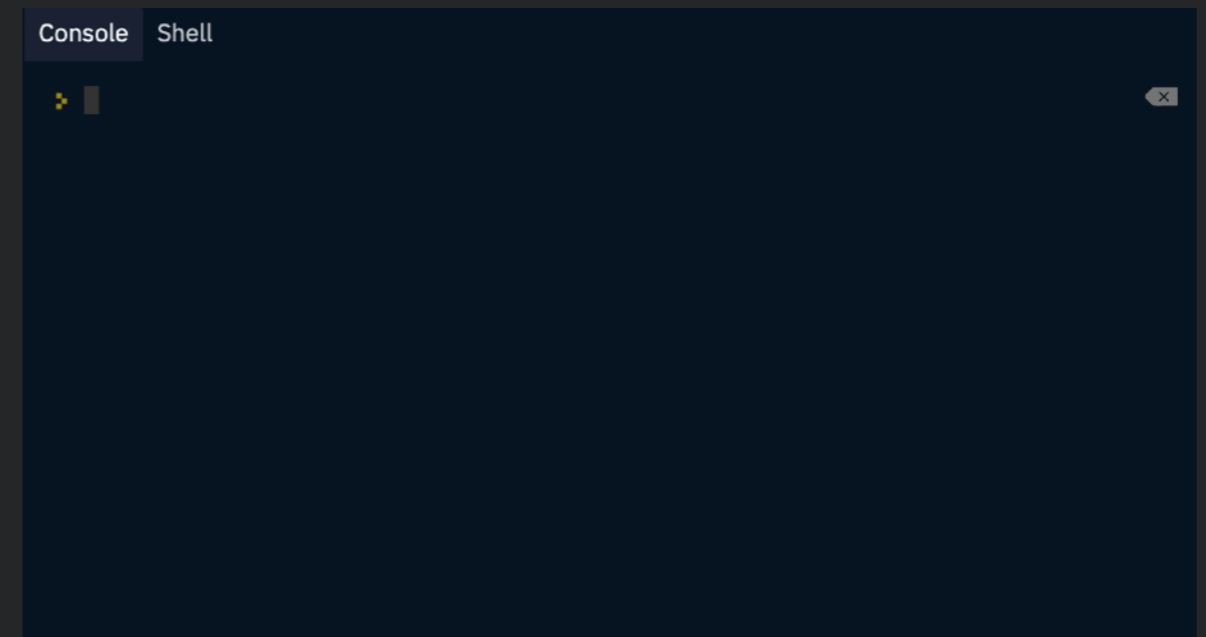
<https://replit.com/@kmoran/async-ex#script.js>





# Async/Await Example

```
function resolveAfter2Seconds() {  
  return new Promise(resolve => {  
    setTimeout(() => {  
      resolve('resolved');  
    }, 2000);  
  });  
}  
  
async function asyncCall() {  
  console.log('calling');  
  var result = await  
  resolveAfter2Seconds();  
  console.log(result);  
  // expected output: 'resolved'  
}
```



<https://replit.com/@kmoran/async-ex#script.js>



# Async/Await -> Synchronous

```
let lib = require("./lib.js");

async function getAndGroupStuff() {
  let thingsToFetch = ['t1', 't2', 't3', 's1', 's2',
    's3', 'm1', 'm2', 'm3', 't4'];
  let stuff = [];
  let ts, ms, ss;

  let promises = [];
  for (let thingToGet of thingsToFetch) {
    stuff.push(await lib.getPromise(thingToGet));
    console.log("Got a thing");
  }
  ts = await lib.groupPromise(stuff, "t");
  console.log("Made a group");
  ms = await lib.groupPromise(stuff, "m");
  console.log("Made a group");
  ss = await lib.groupPromise(stuff, "s");
  console.log("Made a group");
  console.log("Done");
}

getAndGroupStuff();
```

```
node v12.16.1
□
```



# Async/Await -> Synchronous

```
let lib = require("./lib.js");

async function getAndGroupStuff() {
  let thingsToFetch = ['t1', 't2', 't3', 's1', 's2',
's3', 'm1', 'm2', 'm3', 't4'];
  let stuff = [];
  let ts, ms, ss;

  let promises = [];
  for (let thingToGet of thingsToFetch) {
    stuff.push(await lib.getPromise(thingToGet));
    console.log("Got a thing");
  }
  ts = await lib.groupPromise(stuff, "t");
  console.log("Made a group");
  ms = await lib.groupPromise(stuff, "m");
  console.log("Made a group");
  ss = await lib.groupPromise(stuff, "s");
  console.log("Made a group");
  console.log("Done");
}

getAndGroupStuff();
```

```
node v12.16.1
□
```



# Async/Await

- Rules of the road:
  - You can only call **await** from a function that is **async**
  - You can only **await** on functions that return a **Promise**
  - Beware: await makes your code synchronous!

```
async function getAndGroupStuff() {  
    ...  
    ts = await lib.groupPromise(stuff, "t");  
    ...  
}
```



# Async/Await Activity

Rewrite this code so that all of the things are fetched (in parallel) and then all of the groups are collected using async/await

```
let lib = require("./lib.js");

async function getAndGroupStuff() {
  let thingsToFetch = ['t1', 't2', 't3', 's1', 's2', 's3', 'm1', 'm2', 'm3', 't4'];
  let stuff = [];
  let ts, ms, ss;

  let promises = [];
  for (let thingToGet of thingsToFetch) {
    stuff.push(await lib.getPromise(thingToGet));
    console.log("Got a thing");
  }
  ts = await lib.groupPromise(stuff, "t");
  console.log("Made a group");
  ms = await lib.groupPromise(stuff, "m");
  console.log("Made a group");
  ss = await lib.groupPromise(stuff, "s");
  console.log("Made a group");
  console.log("Done");
}

getAndGroupStuff();
```

<https://replit.com/@kmoran/SWE-Week-3-Activity#index.js>

I will also post to Ed right now!



# Acknowledgements

Slides adapted from Dr. Thomas LaToza's  
SWE 632 course