Asynchronous JS

SWE 432, Fall 2018

Web Application Development

Review: Classes - Extends

`extends` allows an object created by a class to be linked to a "`super`" class. Can (but don't have to) add parent constructor.

```
class Faculty {
    constructor(first, last, teaches, office)
    {
        this.firstName = first;
        this.lastName = last;
        this.teaches = teaches;
        this.office = office;
    }
    fullname() {
        return this.firstName + " " + this.lastName;
    }
 }

class CoolFaculty extends Faculty {
    fullname() {
        return "The really cool " + super.fullname();
    }
}
```

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)

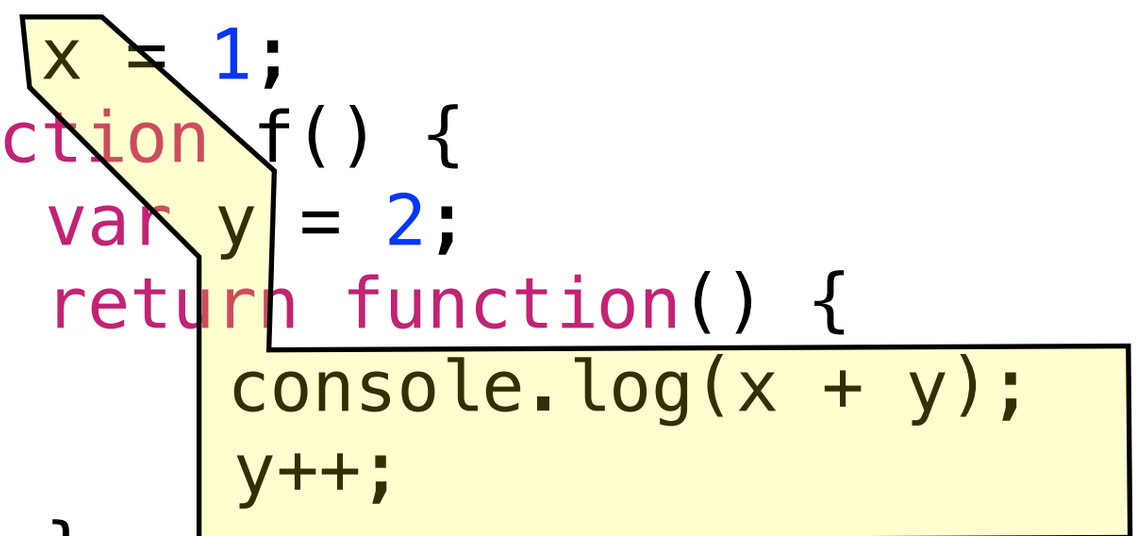
```
var x = 1;
function f() {
    var y = 2;
    return function() {
        console.log(x + y);
        y++;
    };
}
var g = f();
g();              // 1+2 is 3
g();              // 1+3 is 4
```

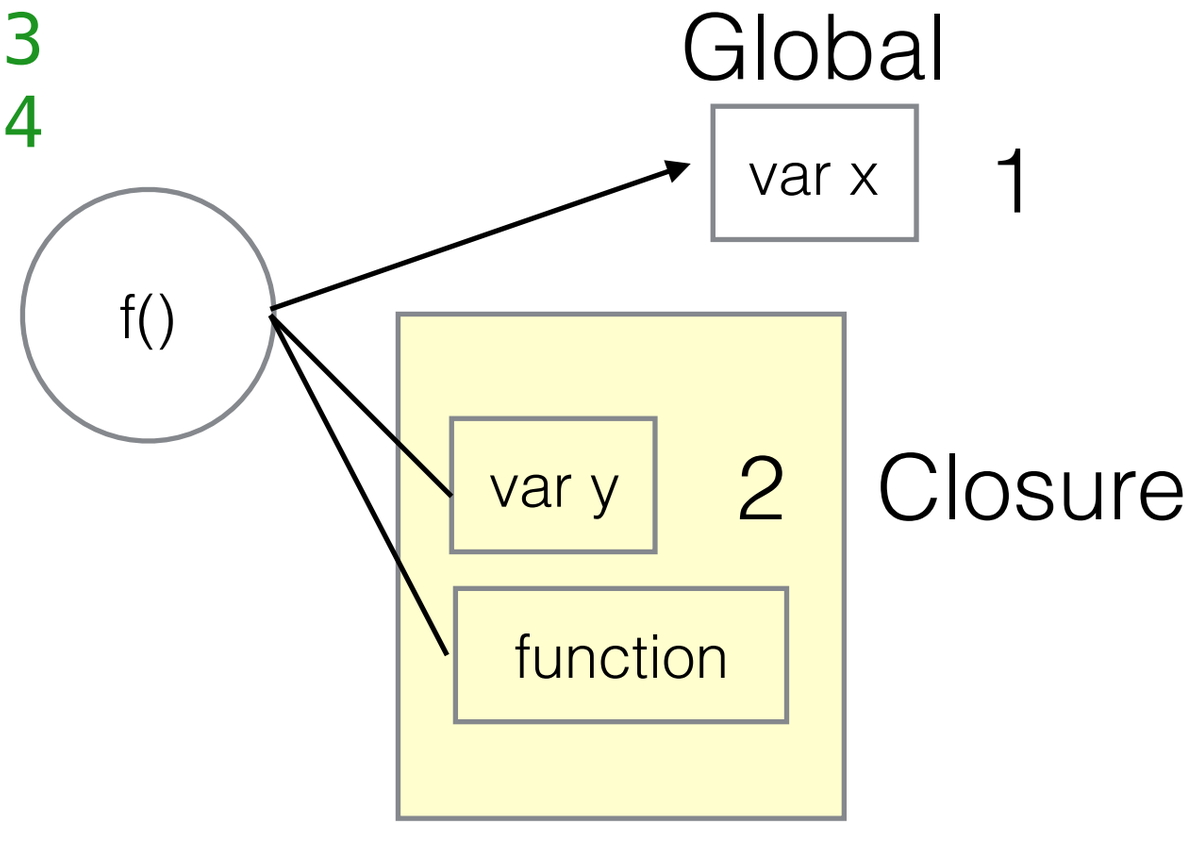
This function attaches itself to x and y so that it can continue to access them.

It "**closes up**" those references

Review: Closures

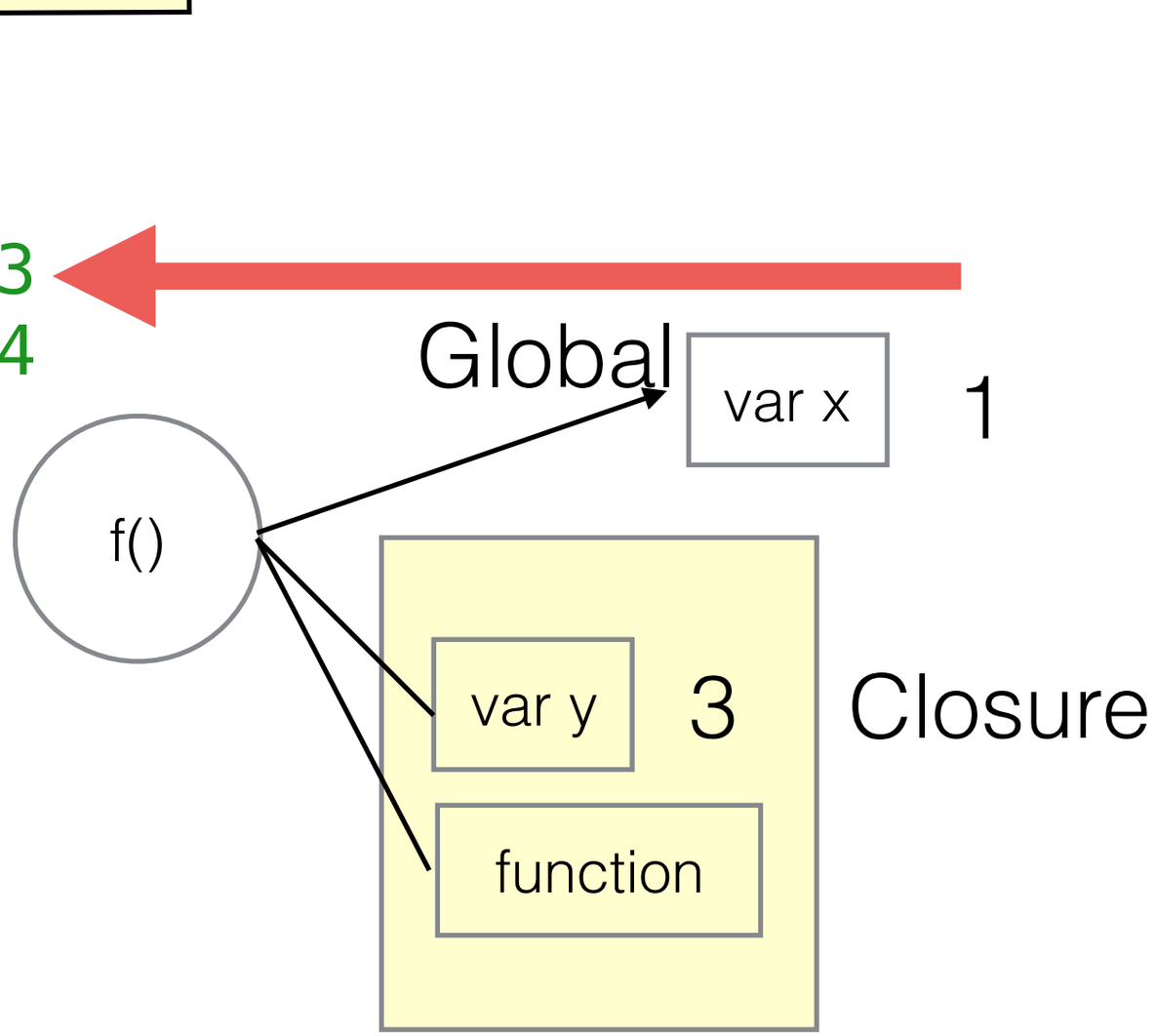
```
var x = 1;
function f() {
    var y = 2;
    return function() {
        console.log(x + y);
        y++;
    };
}
var g = f();
g();          // 1+2 is 3
g();          // 1+3 is 4
```





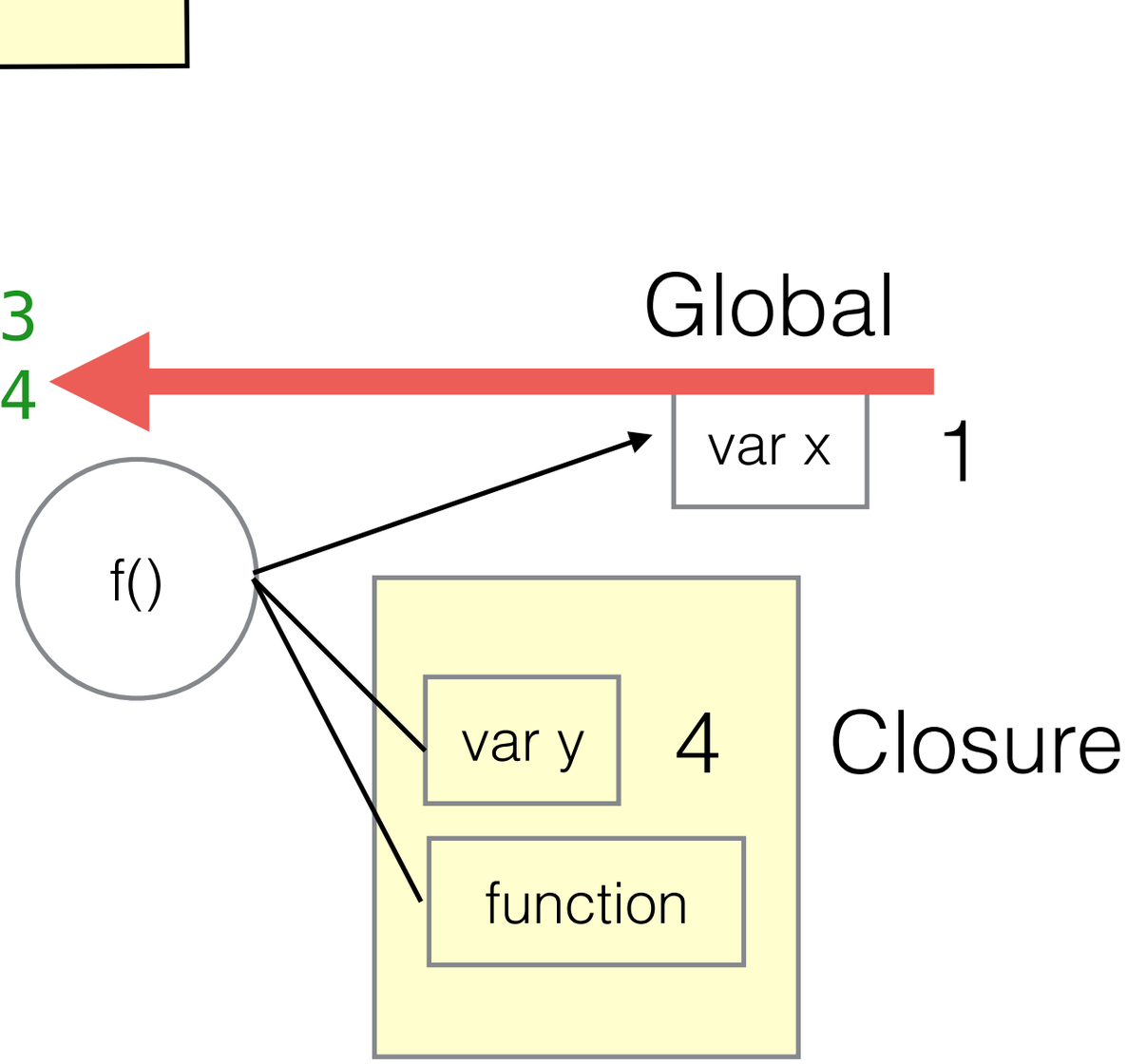
Review: Closures

```
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
```



Review: 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
```



Today

- What is asynchronous programming?
- What are threads?
- Writing asynchronous code
- HW1 Discussion

For further reading:

- 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/

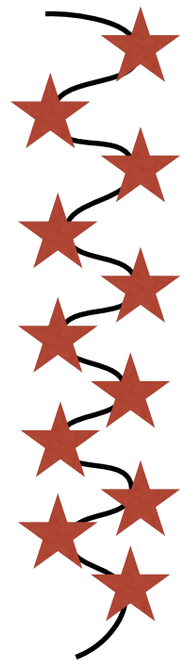
Why Asynchronous?

- 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)

App Starts



App Ends

What is a thread?

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

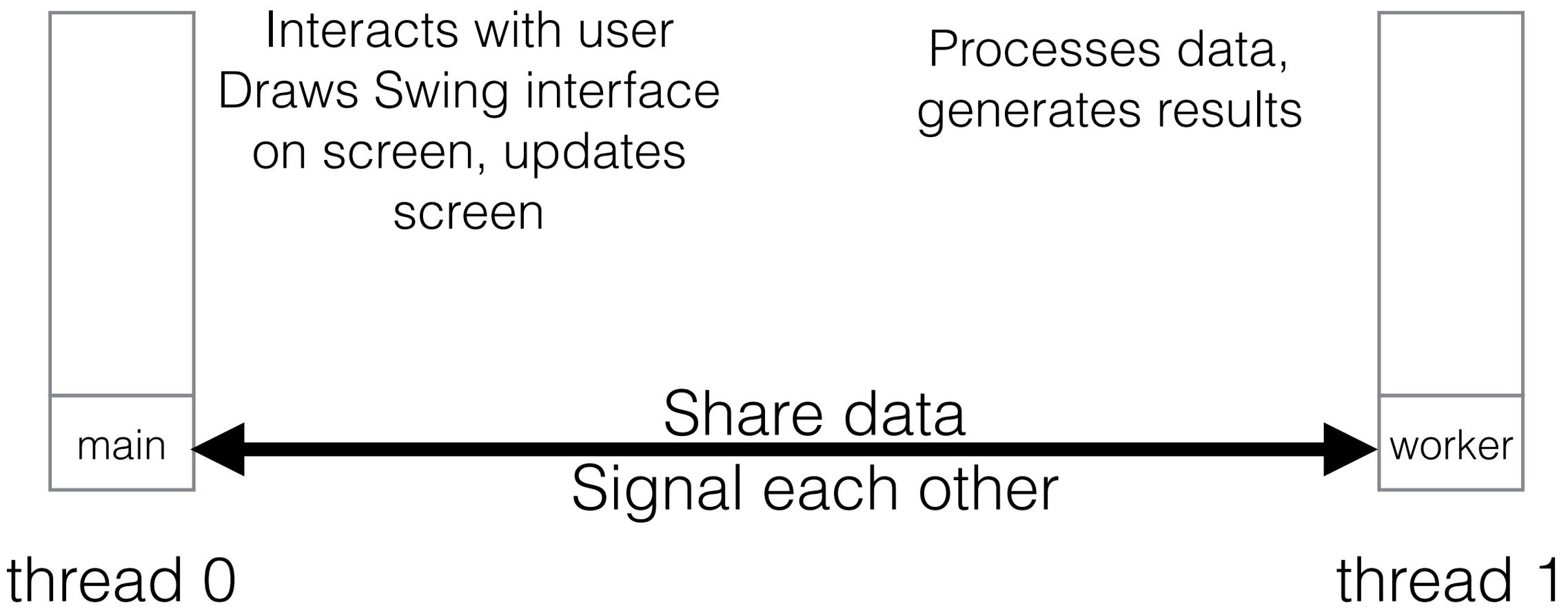
App Starts

App Ends

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

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

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");
```

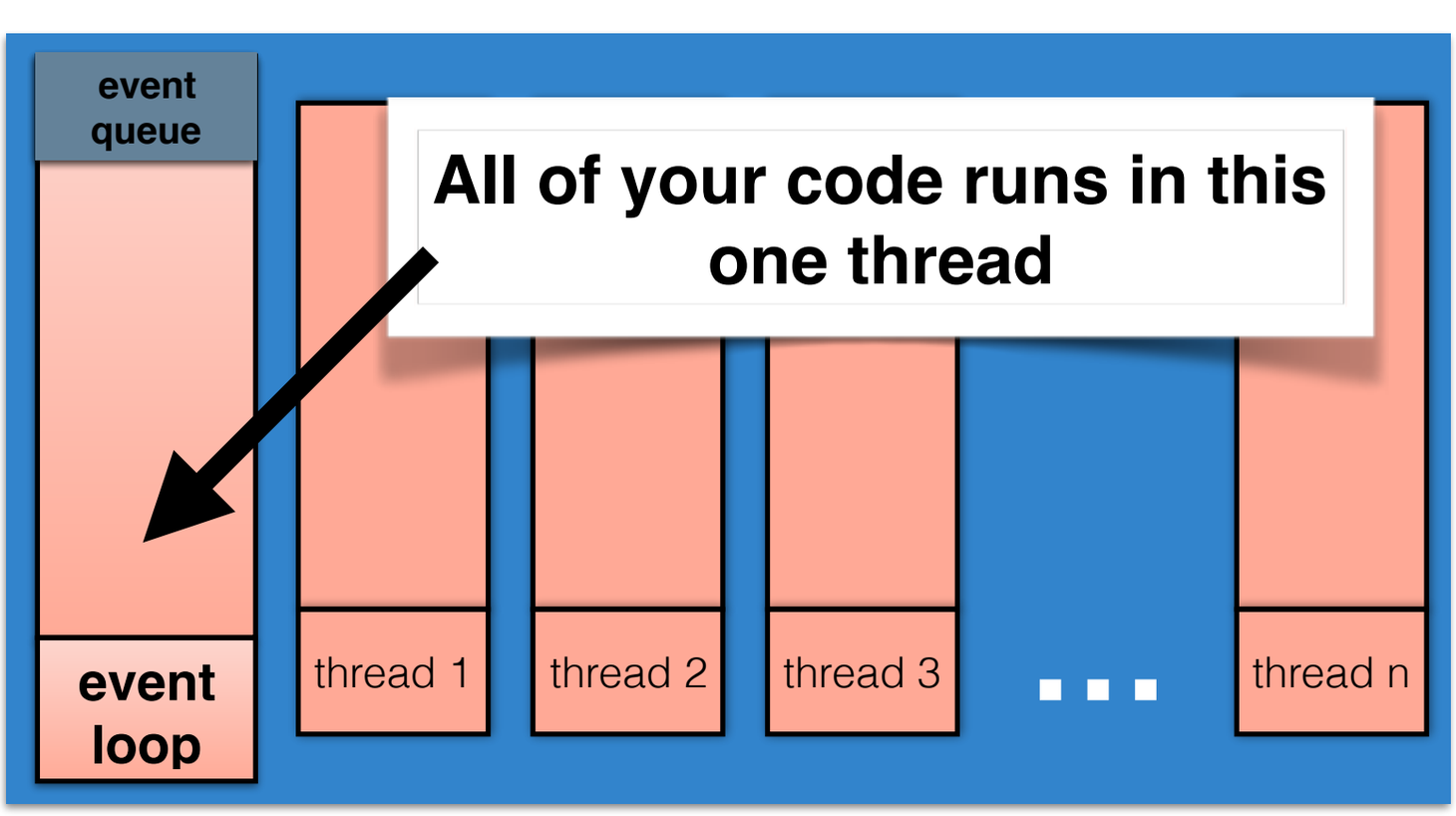
Output:

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

The Event Loop

Event Being Processed:

The Event Loop

Event Being Processed:

response from 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 Being Processed:

response from facebook.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 Being Processed:

response from gmu.edu

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?

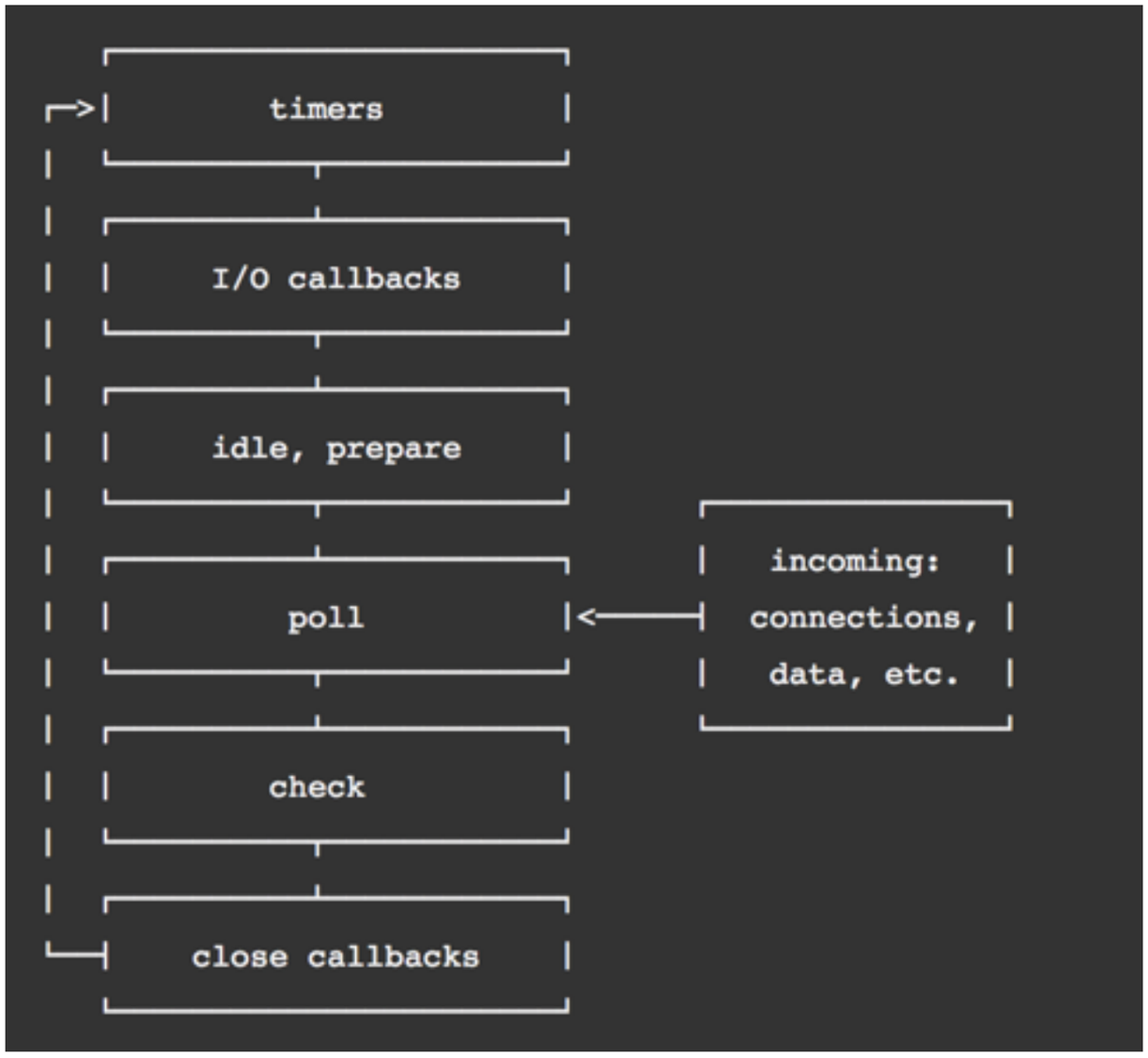
- Run-to-completion
  - The JS engine will not handle the next event until your event handler finishes
- Good news: no other code will run until you finish (no worries about other threads overwriting your data)
- Bad/OK news: Event handlers must not block
  - Blocking -> Stall/wait for input (e.g. alert(), 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



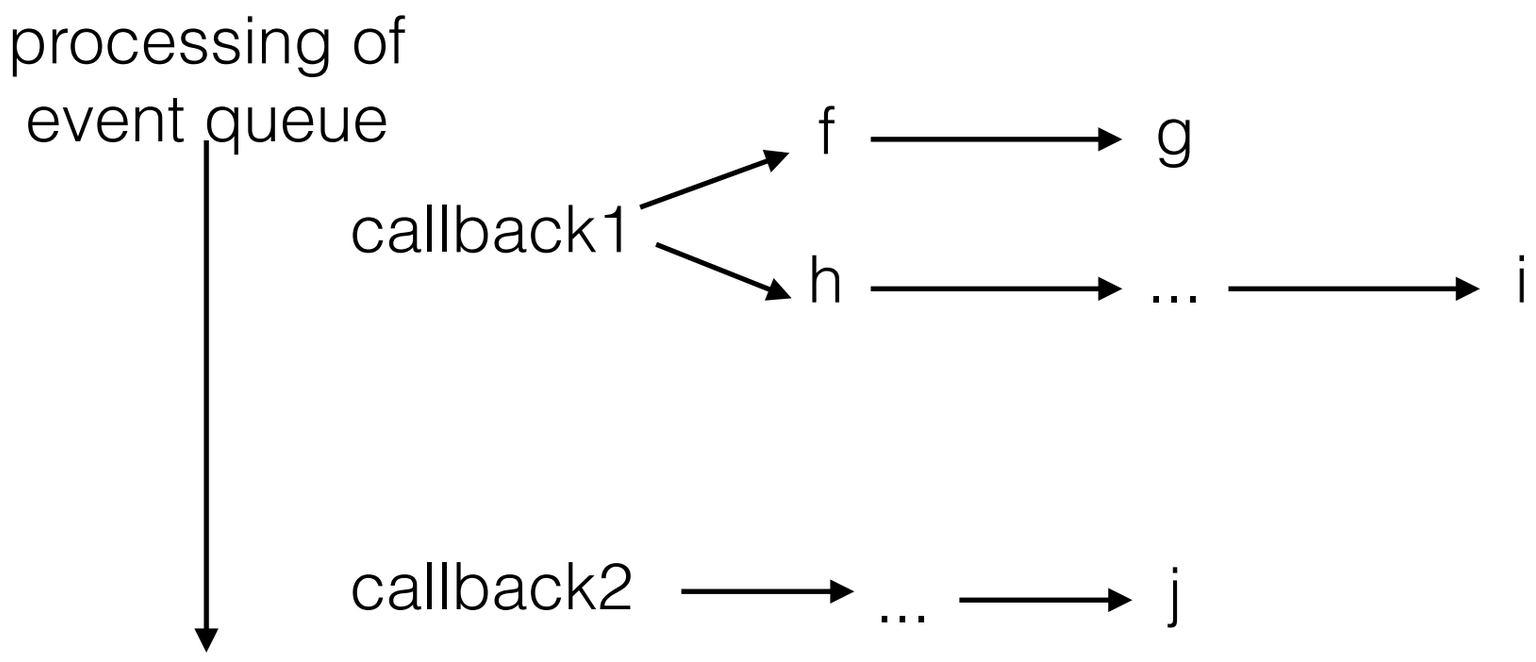
https://nodejs.org/en/docs/guides/event-loop-timers-and-nexttick/

Benefits vs. Explicit Threading (Java)

- Writing your own threads is 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

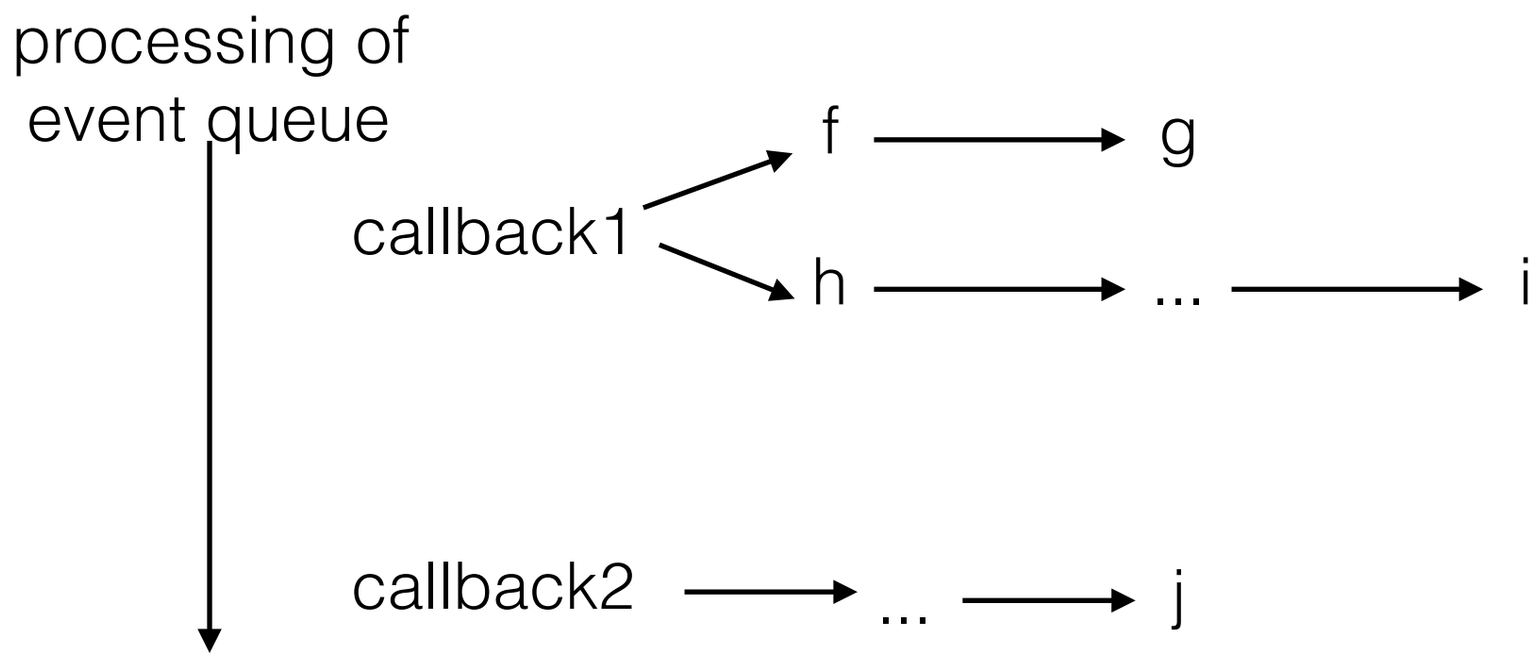
Run-to-completion semantics

- 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.



Implications of run-to-completion

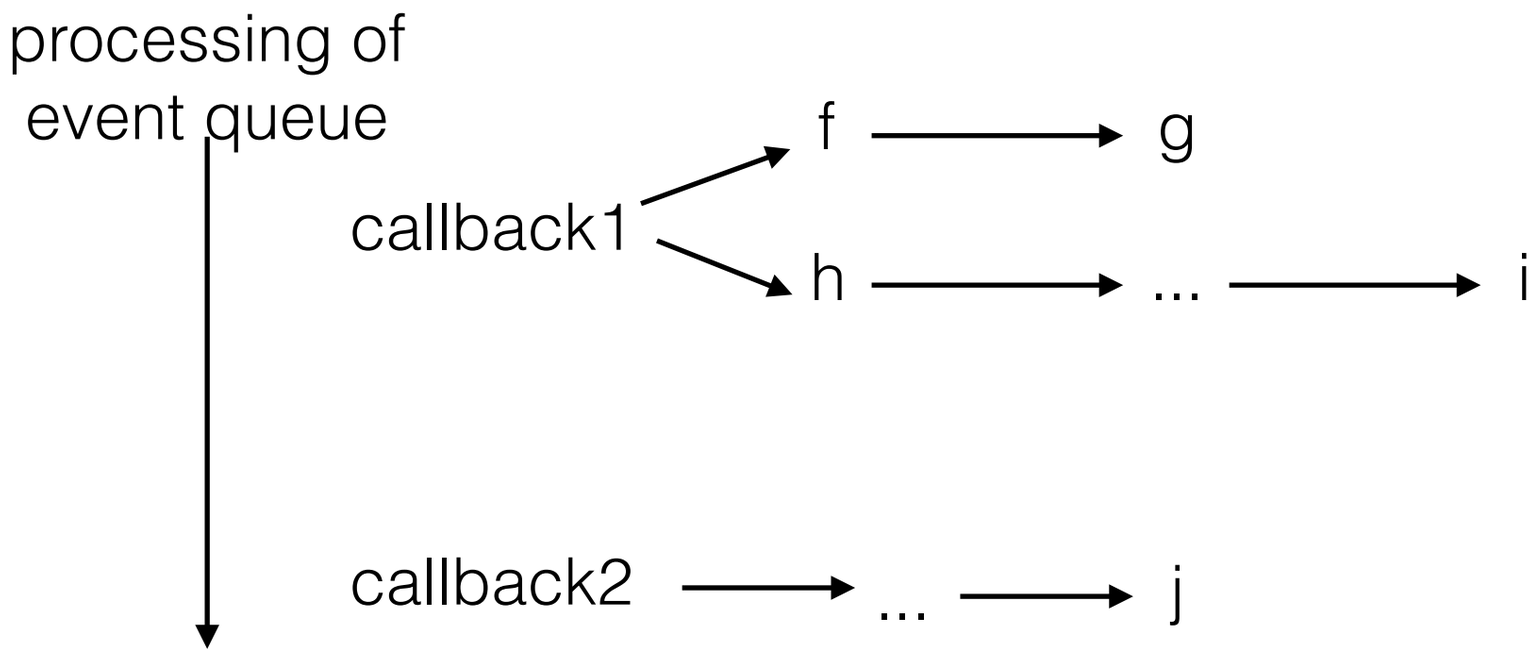
- Good news: no other code will run until you finish (no worries about other threads overwriting your data)



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



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
  - Application
- leaving data in inconsistent state...
- Loading some data from API, but not all of it...

Example: Writing Asynchronous Tasks

- From an array of 10 URL's:
  - Request each URL
  - Then for each page, save it to disk
    - Then once all of the pages are downloaded and saved, print out the total size of all of the files that were saved

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:
        - Is unresolved
        - Succeeds
        - Fails

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"));
} );
```

Using a Promise

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

```
var imgPromise = loadImage("GMURGB.jpg");
imgPromise.then(function (img){
    document.body.appendChild(img);
}).catch(function(e){
    console.log("Oops");
    console.log(e);
});
```

- Advantages:
  - Easier to read
  - Can be used to chain *many* actions together that might happen asynchronously

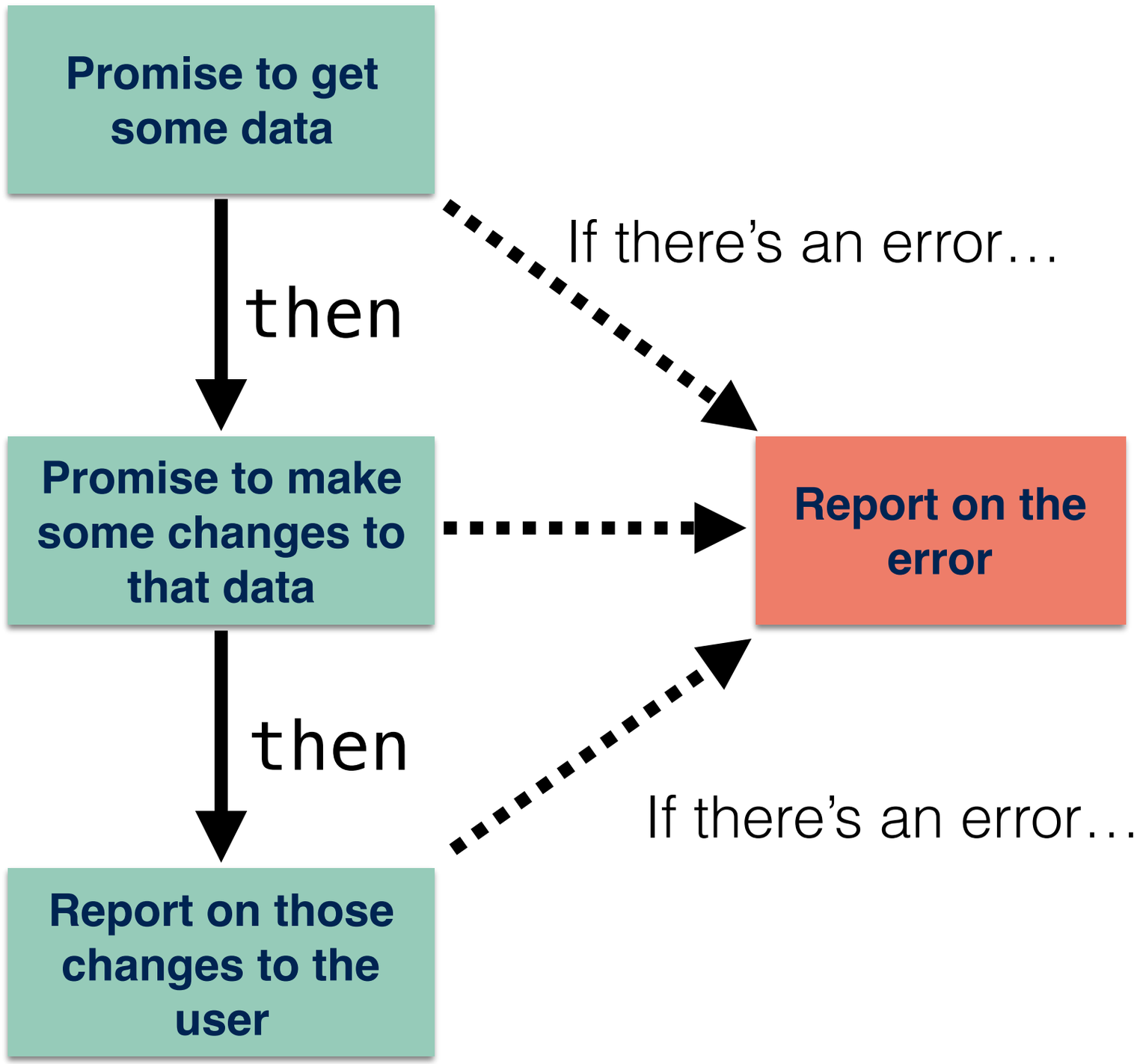
Promising many things

- 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(“JonBell.jpg")])
.then(function (imgArray) {
        imgArray.forEach(img => {document.body.appendChild(img)})
    })
.catch(function (e) {
  console.log("Oops");
  console.log(e);
});
```

Promise one thing then another!



Chaining Promises

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

});
```

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
});
```

Do this

Then, do this

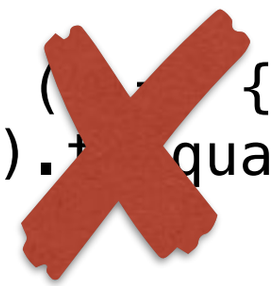
Then do this

And if you ever had an error, do this

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');
});
```

https://jestjs.io/docs/en/tutorial-async

Next Time

- More asynchronous examples
- async/wait keywords
- Threading in JS