Asynchronous Programming

SWE 432, Fall 2016

Design and Implementation of Software for the Web

GEORGE MASON UNIVERSITY

Today

- What is asynchronous programming?
- What are threads?
- How does JS keep the page interactive?
- Writing asynchronous code

For further reading:

Book: Programming HTML5 Applications, Chapter 5, “Web Workers” (Safari books online)

Book: Javascript with Promises, Chapters 1-2 (Safari books online)

https://developer.mozilla.org/en-US/docs/Web/API/Web_Workers_API/Using_web_workers

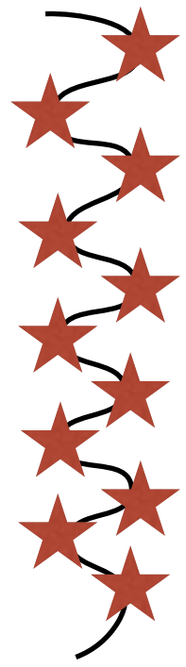
Why Asynchronous?

- 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)

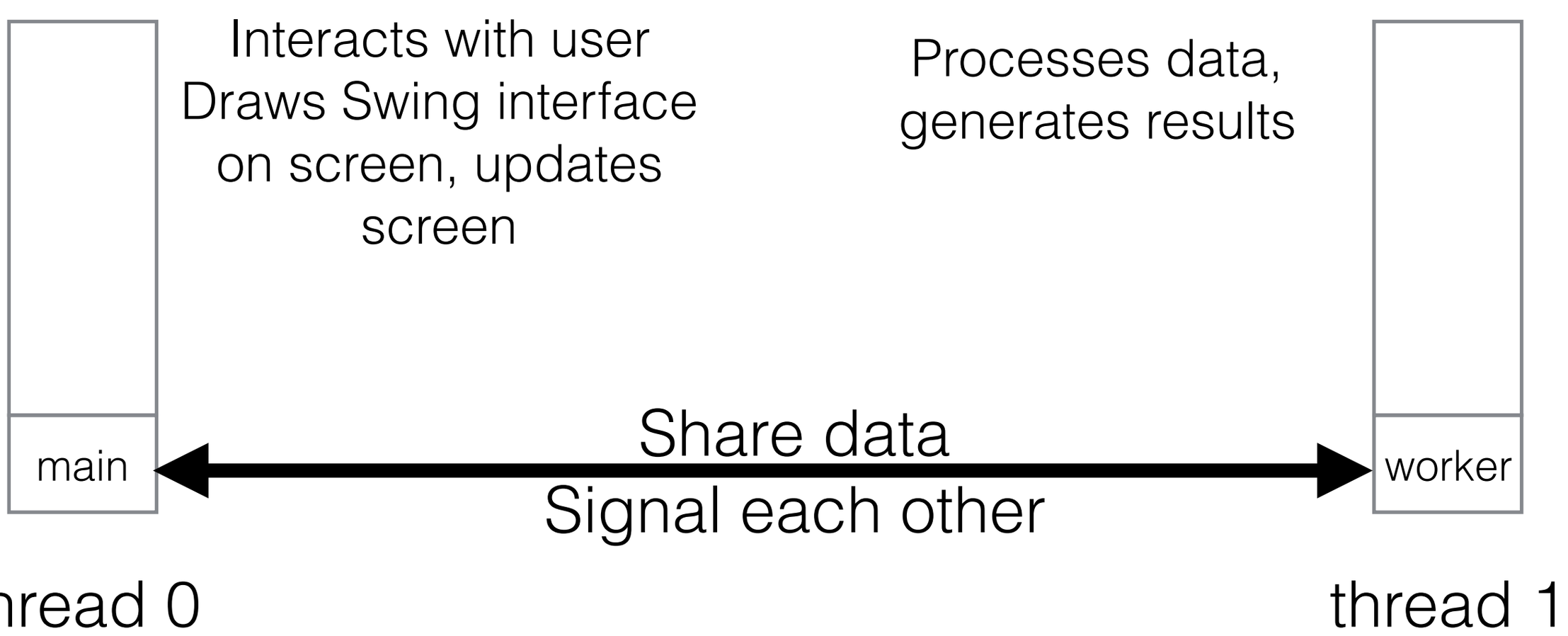
App Starts

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;
}
```

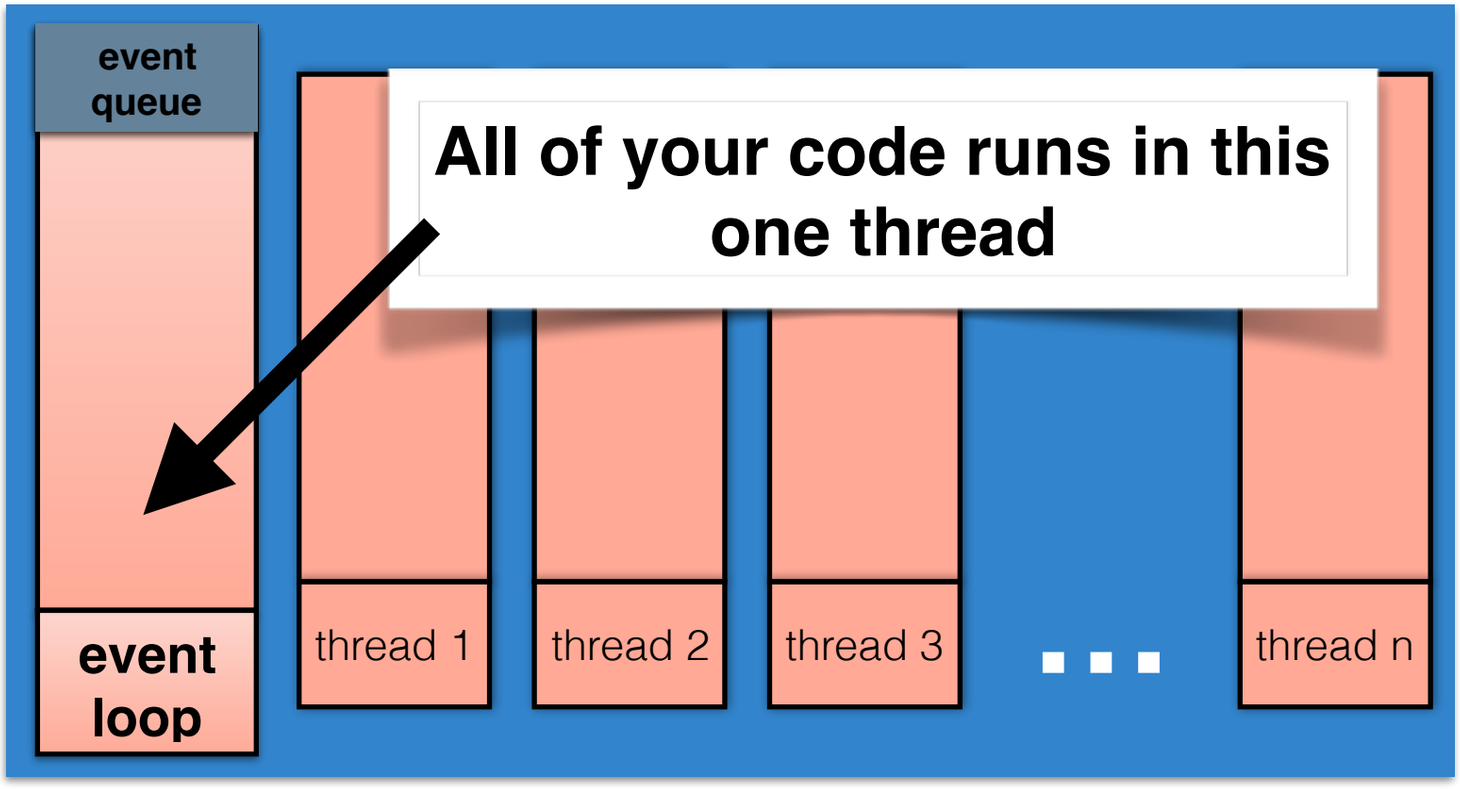
This is a data race: the println in thread1 might see either 2 OR 4

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

- 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:

window: hashChange

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:

#newButton: onClick

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:

AJAX data received

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**

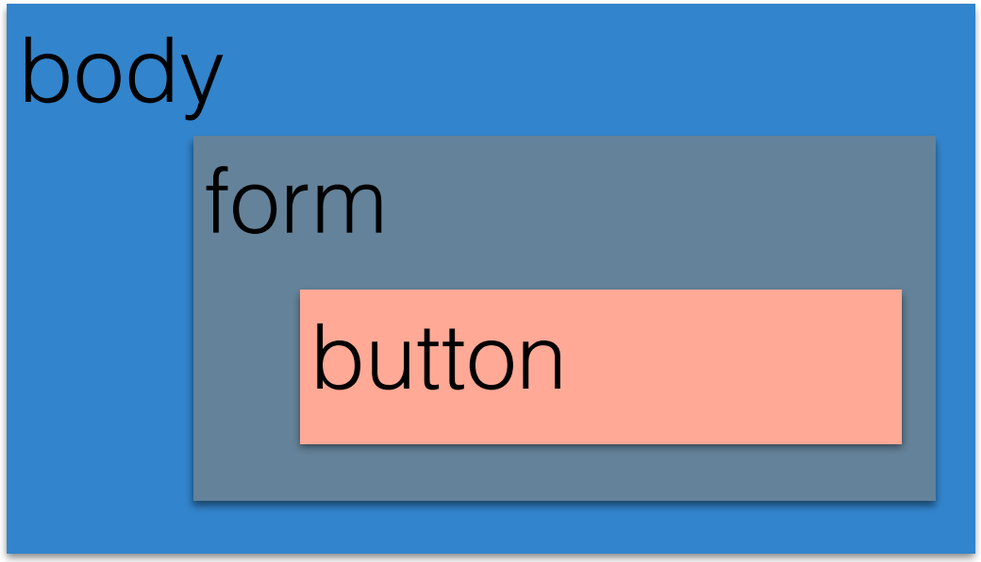
```
$(window).on('hashchange', function () {
   show(location.hash);
});
```

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

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

Event Dispatching

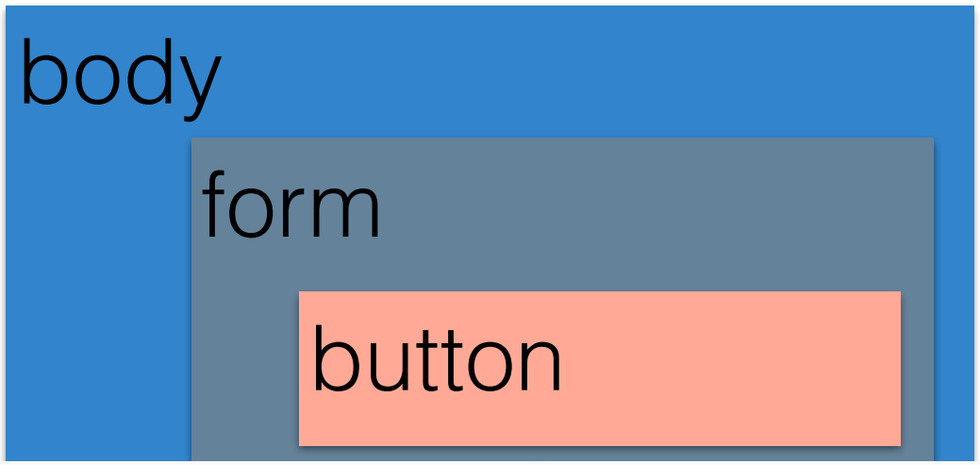
- Each event target can have (0…n) listeners registered for any given event type, called in arbitrary order
- What happens with nested elements?




```
Listener1: body onClick
Listener2: form onClick
Listener3: button onClick
```

What happens when we click in `button`?

Event Bubbling



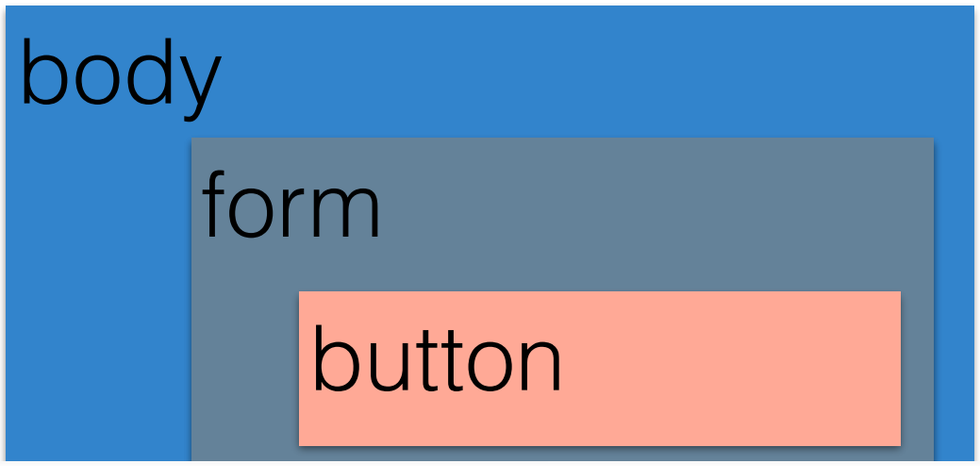
What happens when we click in **`button`**?

Called →

```
Listener1: body onClick
Listener2: form onClick
Listener3: button onClick
```

This is the default behavior

Event Capturing



What happens when we click in **`button`**?

Called →

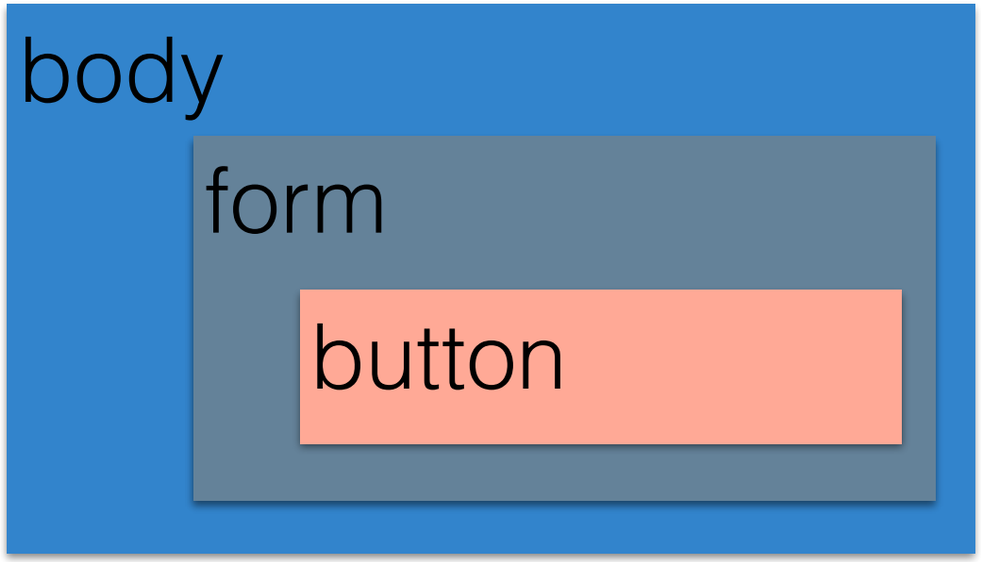
```
Listener1: body onClick
Listener2: form onClick
Listener3: button onClick
```

Enable event capturing when you register your listener:

```
element.addListener('click', myListener, true);
```

Event Dispatching

- An individual listener can stop bubbling/capturing by calling
- `event.stopPropagation();`
  - Assuming that `event` is the name of your handler's parameter
- Or in jQuery, simply `return false;`




```
Listener1: body onClick
Listener2: form onClick
Listener3: button onClick
```

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

Example: Preload some data for a div

Event Queue

Potential problem: div will go away before data comes back

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

When good requests go bad

- It can be tricky to keep track of the status of our asynchronous requests: what happens if they cause an error?
- Most async functions let you register a second callback to be used in case of errors
- Example (Firebase, retrieves a todo):

```
todosRef.child(keyToGet).once('value', function(foundTodo){
   //found the TODO, here it is: foundTodo.val().text
}, function(error){
    //something went wrong
});
```

- You *must* check for errors and fail gracefully

Error handling can get messy

- Let's take the example from the last slide and do something with the returned value, like copy it

```
todosRef.child(keyToGet).once('value', function(foundTodo){
  todosRef.push({'text' : "Seriously: " + foundTodo.val().text},
    function(error)
    {
        if(error != null)
        {
            //something went w
        }
        else
        {
            console.log("OK!");
        }
    });
}, function(error){
        //something went wrong

});
```

Problems:

Will have repeated error handlers

Starts to look nasty after a lot of nesting!

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"));
} );
```

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

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);
});
```

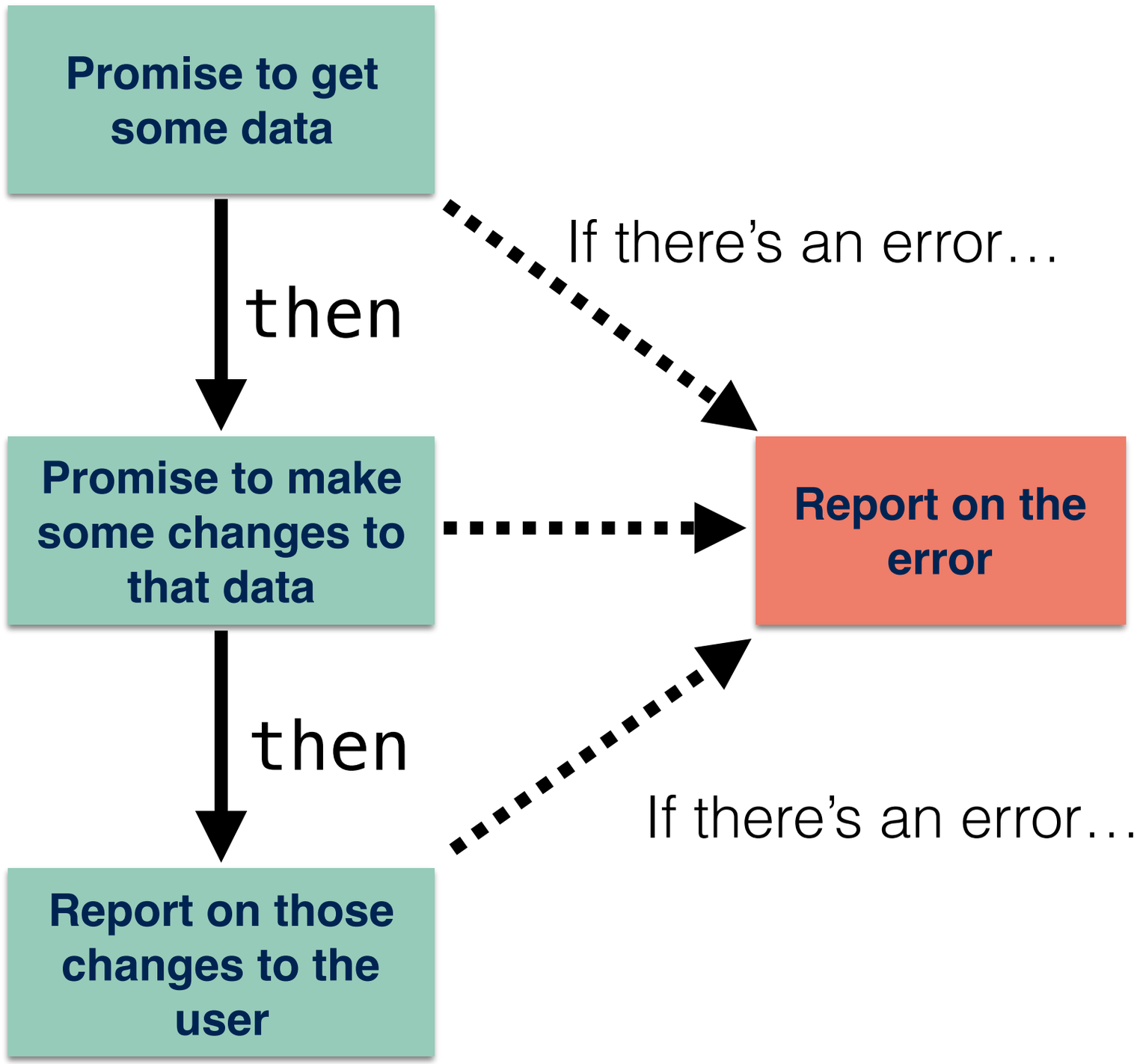
- Advantages:
  - Easier to read
  - Can be used to chain *many* actions together that might happen asynchronously

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

- Example: Firebase interactions can be used as promises, rather than directly using callbacks
- Old:

```
todosRef.child(keyToGet).once('value', function(foundTodo){
   //found the TODO, here it is: foundTodo.val().text
}, function(error){
    //something went wrong
});
```

- With Promises:

A *promise* to return a value

```
todosRef.child(keyToGet).once('value').then(function(foundTodo){

}).catch(
function(error)
{

});
```

- Starts to read more like a sentence

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

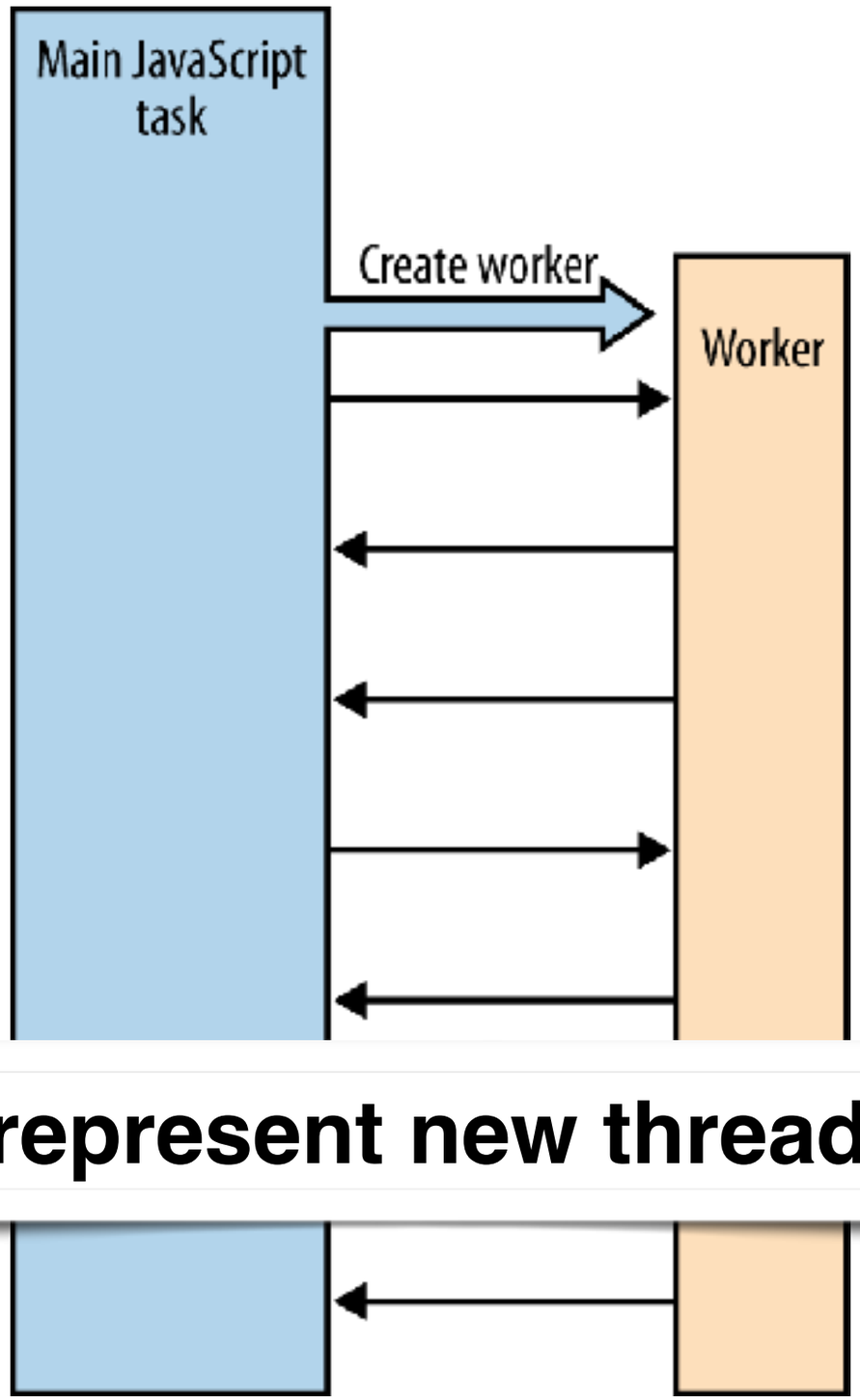
And if you ever had an error, do this

Demo: Promises

- Update our running Todo App
- Add new button: Make important
  - Will add !! to start and end of each todo item
  - We want to show a loading icon until *all* of the todo items are updated
    - But: need to handle error case: what happens if the request doesn't succeed?

https://gmu-swe432.github.io/lecture8demos/public/lecture8Demo1Finished.html
https://github.com/gmu-swe432/lecture8demos/tree/master/public

Web Workers



Web Workers represent new threads of execution

Web Workers

- Web Workers allow you to run arbitrary code in the background, without affecting the performance of your page
- Web Workers:
  - Must be defined in separate files
  - Can not access `document`, `window`, or `parent` objects (so no DOM manipulation)
  - Should mainly be used for performing long, intensive computation (text parsing, image processing, big data)
  - Communicate with the rest of your app with messages

Web Worker API

- Defining a new worker

```
var worker = new Worker('worker.js');
```

- Registering a listener to hear results from the worker

```
worker.addEventListener("message", function(e){
    console.log("Message from worker: <" + e.data + ">");
});
worker.addEventListener("error", function(e){
    console.log(“Uh oh");
});
```

- Sending data to the worker

```
worker.postMessage("Hello");
```

- In the worker: registering for messages from the main thread, sending responses

```
self.addEventListener('message', function(e) { doSomething(); });
self.postMessage(“Greetings from the Worker");
```

- Including additional scripts:

```
importScripts('script2.js');
```

- Kill a worker:

```
worker.terminate();
```

Passing Messages with Web Workers

- Can pass string or object
- Objects are *passed by value*
  - Good news: reduces concurrency programming errors
  - Bad news: passing a big (10's of MB's) object will be *slow*
- Alternative: *transfer* an object
  - No longer exists in the original thread
  - Syntax:

```
worker.postMessage(myObject, [myObject]);
```

Web Workers: Example

Defining a web worker in worker.js

```
self.addEventListener('message', function(e) {
    self.postMessage("Worker is sending back the text:" + e.data);
}, false);
```

Using a web worker in our web app

```
<script language="javascript">
    "use strict";
    var worker = new Worker('worker.js');
    worker.addEventListener("message", function(e){
        console.log("Message from worker: <" + e.data + ">");
    });
    worker.postMessage("Hello");
    worker.postMessage("How's it going over there, worker?");
    worker.terminate();
</script>
```

When should you use web workers?

- Mainly for computational stuff:
  - Image manipulation
  - Map routing (without going off to server)
  - Numerical methods
- Remember: can *not* interact with DOM in web worker

Web Workers Demo

Calculating Pi iteratively

```
function CalculatePi(loop)
{
    var n=1;
    var c = parseInt(loop);
    console.log(loop);
    for (var i=0,Pi=0;i<=c;i++) {
        Pi=Pi+(4/n)-(4/(n+2));
        n=n+4;
    }
    return Pi;
}
```

https://gmu-swe432.github.io/lecture8demos/public/WebWorkerDemoFinished.html

https://github.com/gmu-swe432/lecture8demos/tree/master/public

Exit-Ticket Activity

Go to socrative.com and select “Student Login”

Class: SWE432001 (Prof LaToza) or SWE432002 (Prof Bell)

ID is your @gmu.edu email

1: How well did you understand today's material

2: What did you learn in today's class?

For question 3: What is a promise used for?