Asynchronous Programming

SWE 432, Fall 2017
Design and Implementation of Software for the Web
Today

• What is asynchronous programming?
• What are threads?
• How does JS keep programs responsive?
• Writing asynchronous code with Promises and timers
What does this code do?

```javascript
var global;
var fetch = require('node-fetch');

fetch('https://api.wmata.com/Incidents.svc/json/ElevatorIncidents',
    { headers: { api_key: "e1eee2b5677f408da40af8480a5fd5a8" } })
    .then(function(res) {
        return res.json();
    }).then(function(json) {
    global = json;
});
console.log(global.ElevatorIncidents[0]);
```
Why write asynchronous programs?

- Asynchronous programs occur when there are events which occur outside the control flow of your program
  - Data arrived back from an HTTP request made earlier
  - A timer went off
  - The OS sent a message
  - (Client-side) The user clicked a button
What's wrong with this program?

```javascript
var global;
var fetch = require('node-fetch');

global = fetch('https://api.wmata.com/Incidents.svc/json/ElevatorIncidents',
   { headers: { api_key: "e1eee2b5677f408da40af8480a5fd5a8" } }).json();

console.log(global.ElevatorIncidents[0]);
```

Note: this code is hypothetical and does not reflect the actual behavior of fetch.
The perils of blocking

• Asynchronous events might take a long time to occur
  • Waiting for data from a server
  • A button that the user never clicks on

• Want to execute code that occurs after such an event occurs

• But, in the meantime, want application to be responsive so that other computation can occur and over events can be handled
Solution 1: Threads

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

App Starts

App Ends
Solution 1: Threads

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

- Allowing more than one thread is **multi-threading**
- Multi-Threading enables responsiveness by allowing computation to occur in parallel
- May occur physically through multiple cores and/or logically through OS scheduler
- Example: Process data while interacting with user

<table>
<thead>
<tr>
<th>thread 0</th>
<th>thread 1</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>main</strong></td>
<td><strong>worker</strong></td>
</tr>
<tr>
<td>Interacts with user</td>
<td>Processes data, generates results</td>
</tr>
<tr>
<td>Draws Swing interface on screen, updates screen</td>
<td>Share data</td>
</tr>
<tr>
<td>Signal each other</td>
<td></td>
</tr>
</tbody>
</table>
Woes of Multi-Threading

```java
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
Solution 2: Single thread w/ event loop

• All of your code will run in a **single thread**
• Since you are not sharing data between threads, races don’t happen as easily
• **Event-driven:** Event loop maintains queue of events, and invokes handler for each event
• (JavaScript engine itself may still be multithreaded)
The Event Loop

Event Queue

GET resource1
POST resource5
timer fired

Event Being Processed:
The Event Loop

Event Queue

POST resource5

timer fired

JS Engine

event loop

thread 1

thread 2

thread 3

thread n

Event Being Processed:

GET resource1

Are there any listeners registered for this event?

If so, call listener with event

After the listener is finished, repeat
Event Queue

timer fired

JS Engine

event loop
thread 1
thread 2
thread 3
thread n

Event Being Processed:

POST
resource5

Are there any listeners registered for this event?
If so, call listener with event
After the listener is finished, repeat
Event Being Processed:

Are there any listeners registered for this event? If so, call listener with event. After the listener is finished, repeat.
The Event Loop

```javascript
fetch('https://api.wmata.com/Incidents.svc/json/ElevatorIncidents',
    { headers: { api_key: "e1eee2b5677f408da40af8480a5fd5a8" } })
    .then(function(res) {
        return res.json();
    }).then(function(json) {
        global = json;
    });
```

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

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

Advantages of single-threaded event loops

• Managing dependencies between data in different threads is difficult to understand and get right and tricky to debug
  • When threads share data, need to ensure they correctly synchronize on it to avoid race conditions
• But there are downsides
  • Can not have slow event handlers
  • Can still have data races, although easier to reason about (data races can only occur at event boundaries)
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.

```
<table>
<thead>
<tr>
<th>processing of event queue</th>
</tr>
</thead>
<tbody>
<tr>
<td>callback1</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>f</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>g</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>callback2</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>i</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>j</td>
</tr>
</tbody>
</table>
```
Implications of run-to-completion

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

\[ j \text{ will not execute until after } i \]
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

\[
\text{processing of event queue} \\
\text{callback1} \\
\text{callback2} \\
\]

\[
\text{f} \rightarrow \text{g} \\
\text{h} \rightarrow \ldots \rightarrow \text{i} \\
\]

\[
\text{...} \rightarrow \text{j} \\
\]

\text{*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...
When good requests go bad

• What happens if an error occurs in an asynchronous function?
• Most async functions let you register a second callback to be used in case of errors
• You *must* check for errors and fail gracefully
  • Not ok to assume that errors will never happen.
fs.readdir(source, function (err, files) {
  if (err) {
    console.log('Error finding files: ' + err)
  } else {
    files.forEach(function (filename, fileIndex) {
      console.log(filename)
      gm(source + filename).size(function (err, values) {
        if (err) {
          console.log('Error identifying file size: ' + err)
        } else {
          console.log(filename + ' : ' + values)
          aspect = (values.width / values.height)
          widths.forEach(function (width, widthIndex) {
            height = Math.round(width / aspect)
            console.log('resizing ' + filename + ' to ' + height + 'x' + height)
            this.resize(width, height).write(dest + 'w' + width + '_ ' + filename, function(err) {
              if (err) console.log('Error writing file: ' + err)
            }).bind(this)
          })
        }
      })
    })
  }
})

From http://callbackhell.com/
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
Using a Promise

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

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

```javascript
fetch('http://domain.invalid/')
  .catch(function(err) {
    console.log(err);
  });
```
Promise one thing then another

1. Promise to get some data
2. Promise to get some data based on that data
3. Use that data to update application state

If there’s an error…

- Report on the error
Chaining Promises

```javascript
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){
});
```
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:

```javascript
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);
 });
```
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.

```javascript
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

```javascript
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.
Timers

```javascript
function myFunc(arg) {
    console.log('arg was => ' + arg);
}
setTimeout(myFunc, 1500, 'funky');
```

Run myFunc no sooner than 1500 ms

```javascript
setInterval(() => {
    console.log('interval executing');
}, 500);
```

Run code every 500 ms

```javascript
setImmediate((arg) => {
    console.log('executing immediate: ' + arg);
}, 'so immediate');
```

Run code after any I/O operations in current cycle and before timers from next cycle
const timeoutObj = setTimeout(() => {
    console.log('execute after 1500 ms');
}, 1500);

const intervalObj = setInterval(() => {
    console.log('execute every 500 ms');
}, 500);

const immediateObj = setImmediate(() => {
    console.log('execute after IO operations');
});

clearTimeout(timeoutObj);
clearImmediate(immediateObj);
clearInterval(intervalObj);
Demo: Promises and Timers
Readings for next time

• Using Promises

• Node.js event loop