CS 4530
Software Engineering

Lecture 6 - Asynchronous Programming

Jonathan Bell, John Boyland, Mitch Wand
Khoury College of Computer Sciences

Zoom Mechanics

- Recording: This meeting is being recorded
- If you feel comfortable having your camera on, please do so! If not: a photo?
- I can see the zoom chat while lecturing, slack while you're in breakout rooms
- If you have a question or comment, please either:
    - "Raise hand" - I will call on you
    - Write "Q: <my question>" in chat - I will answer
      your question, and might mention your name and ask you
      a follow-up to make sure your question is addressed
    - Write "SQ: <my question>" in chat - I will answer
      your question, and not mention your name or expect you to
      respond verbally

Today’s Agenda

Administrative:

- Team formation due Friday
- HW2 posted, due next Friday
- HW1 solution to be posted tomorrow

Today’s session:

- Lecture: Asynchronous Programming
- Activity: Asynchronous Programming with REST client

Why Asynchronous?

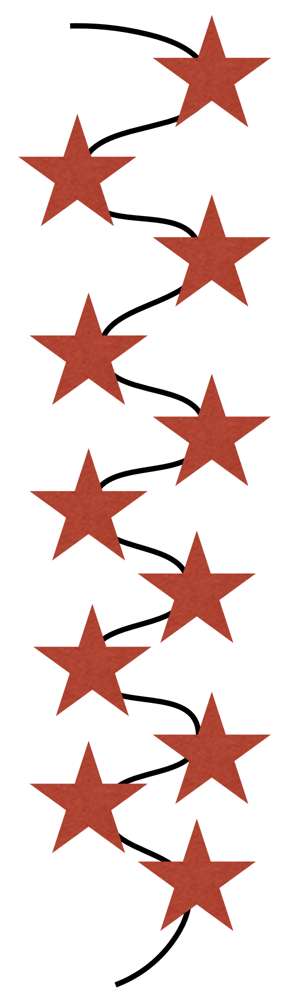
- Maintain an interactive application while still doing stuff
    - Processing data
    - Communicating with remote hosts
    - Timers that countdown while our app is running
- Anytime that an app is doing more than one thing at a time, it is asynchronous

What is a thread?

(Not NodeJS-specific)

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

App Starts



App Ends

What is a thread?

(Not NodeJS-specific)

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

App Starts

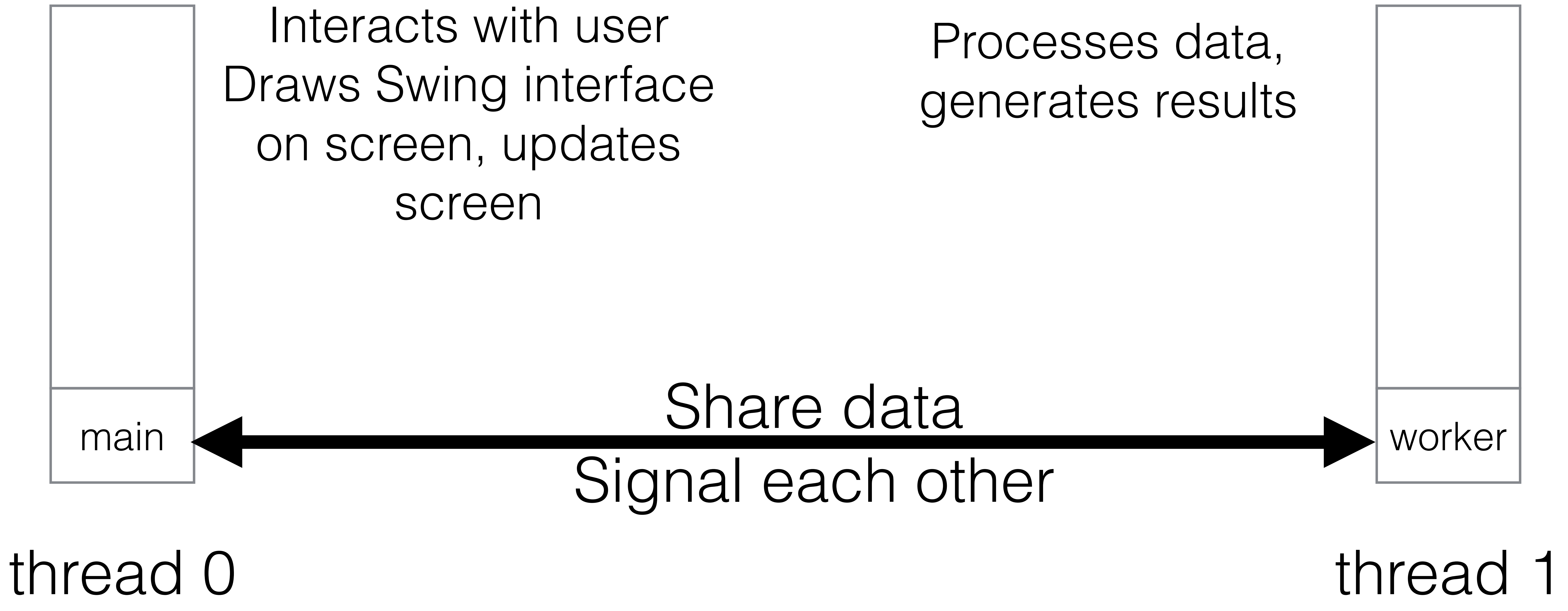
App Ends

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

Asynchronous Computation with Threads

Typical Java Example

- 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


Interacts with user
Draws Swing interface on screen, updates screen
Processes data, generates results
main
worker
Share data
Signal each other
thread 0
thread 1

Asynchronous Programming in JS/TS

How do we make a network request? Isn't that a slow thing?

```
console.log('Making a request to rest-example');
axios.get('https://rest-example.covey.town/') // axios is a popular library for making HTTP requests
  .then((response) =>{
  console.log('Heard back from server');
  console.log(response.data);
});
console.log('Response sent!');
```

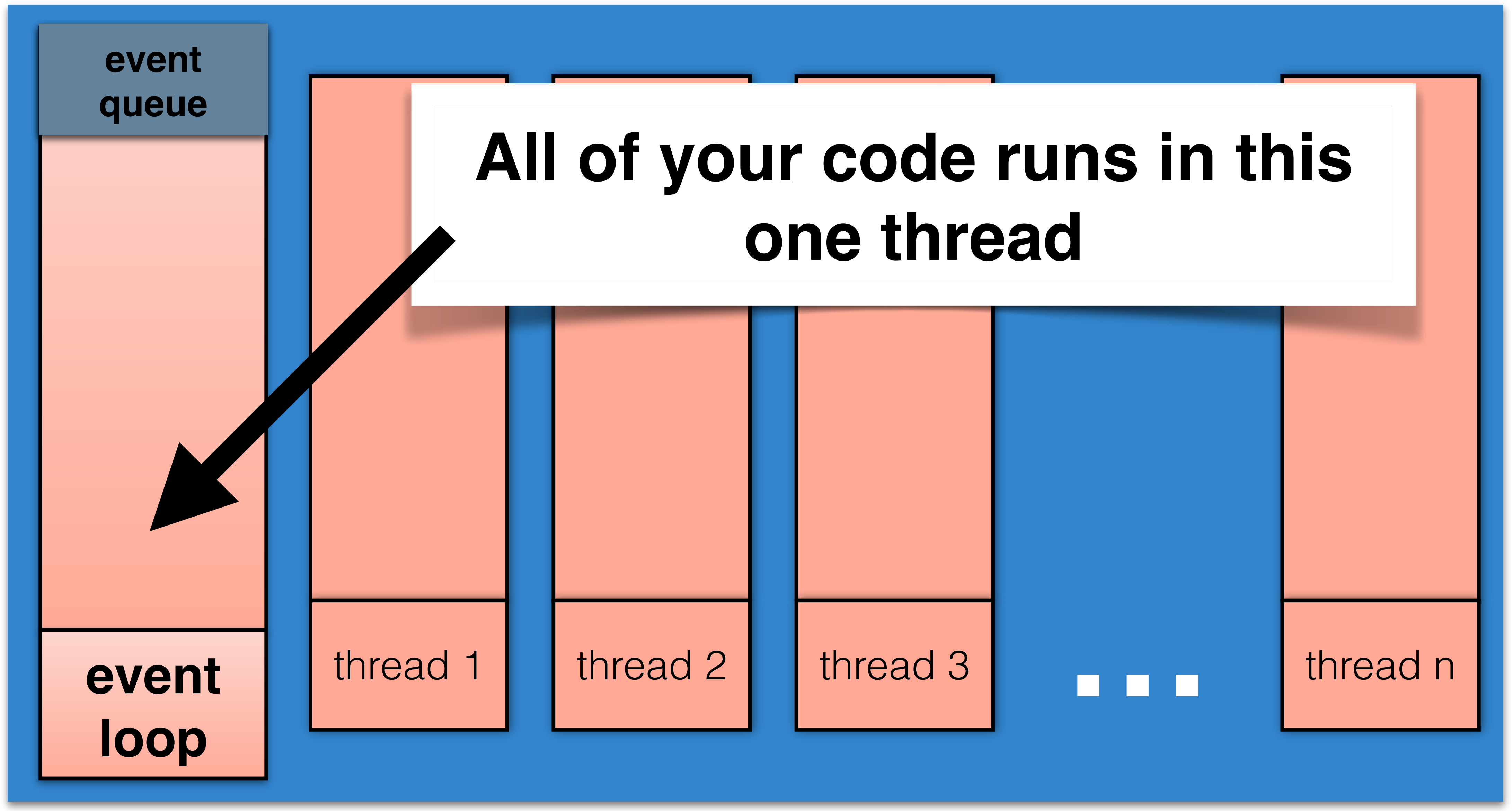
Output:

```
Making a request to rest-example
Response sent!
Heard back from server
This is GET number 4 on the current server
```

`axios.get` 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 (or "event handlers")



NodeJS

Asynchronous Programming in JS/TS

Promises

axios.get returns a Promise for an AxiosResponse

```
console.log('Making a request to rest-example');
axios.get('https://rest-example.covey.town/') // axios is a popular library for making HTTP requests
  .then((response) =>{
  console.log('Heard back from server');
  console.log(response.data);
});
console.log('Response sent!');
```

Promise.then will run the event handler provided once the value that is promised becomes available

Output:

```
Making a request to rest-example
Response sent!
Heard back from server
This is GET number 4 on the current server
```

axios.get is an **asynchronous call**

Making lots of requests

3 Requests: What is the output?

```
console.log('Making a requests');
axios.get('https://rest-example.covey.town/')
  .then((response) =>{
  console.log('Heard back from server');
  console.log(response.data);
});
axios.get('https://www.google.com/')
  .then((response) =>{
    console.log('Heard back from Google');
  });
axios.get('https://www.facebook.com/')
  .then((response) =>{
    console.log('Heard back from Facebook');
  });
console.log('Requests sent!');
```

These 2 lines ALWAYS first (same handler)

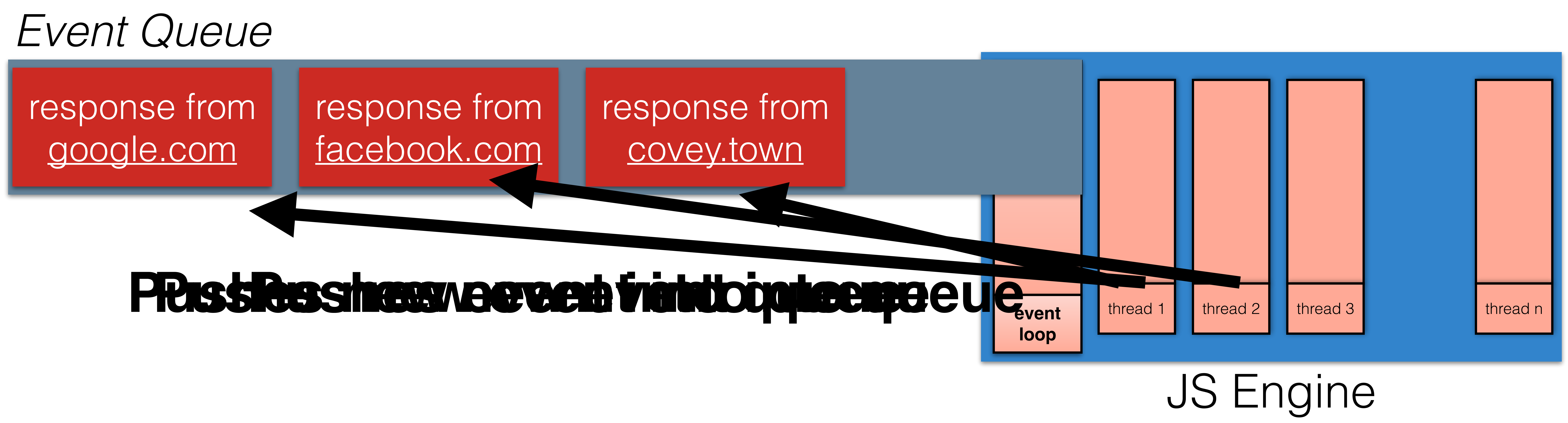
Sample Output:

```
Making a requests
Requests sent!
Heard back from Google
Heard back from server
This is GET number 6 on the current server
Heard back from Facebook
```

These 2 lines ALWAYS together (same handler)

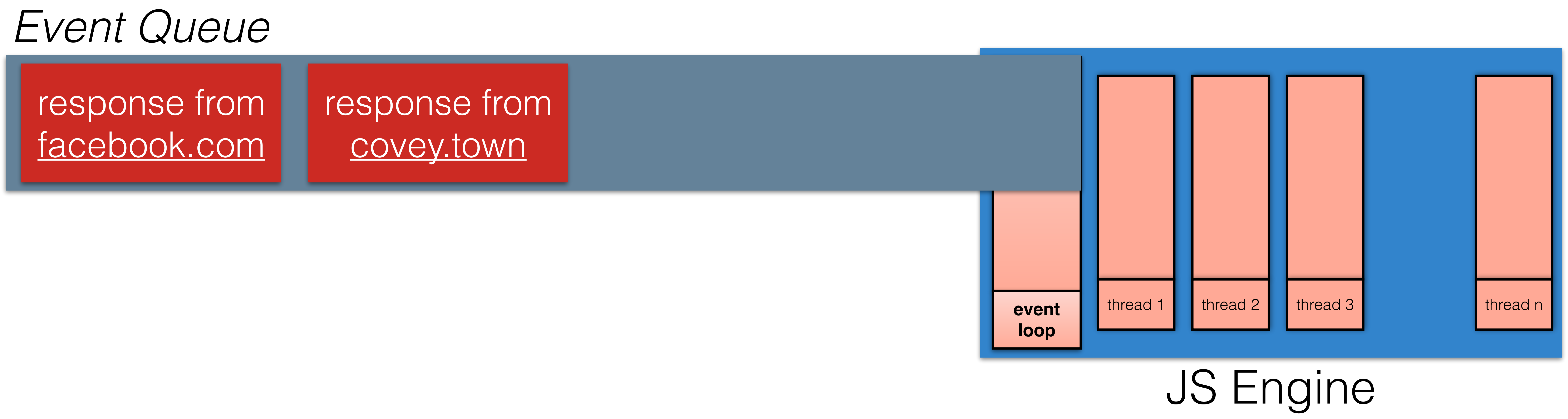
No guarantee on order of hearing back from Google, our server, or Facebook (new handlers)

The Event Loop



Event Being Processed:

The Event Loop



Event Being Processed:

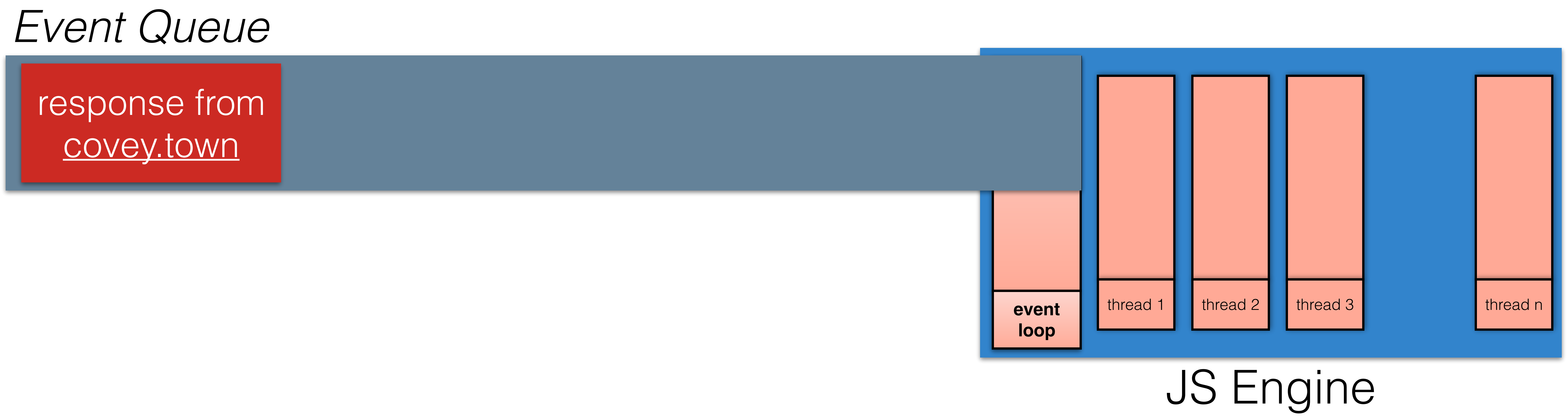
response from google.com

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:

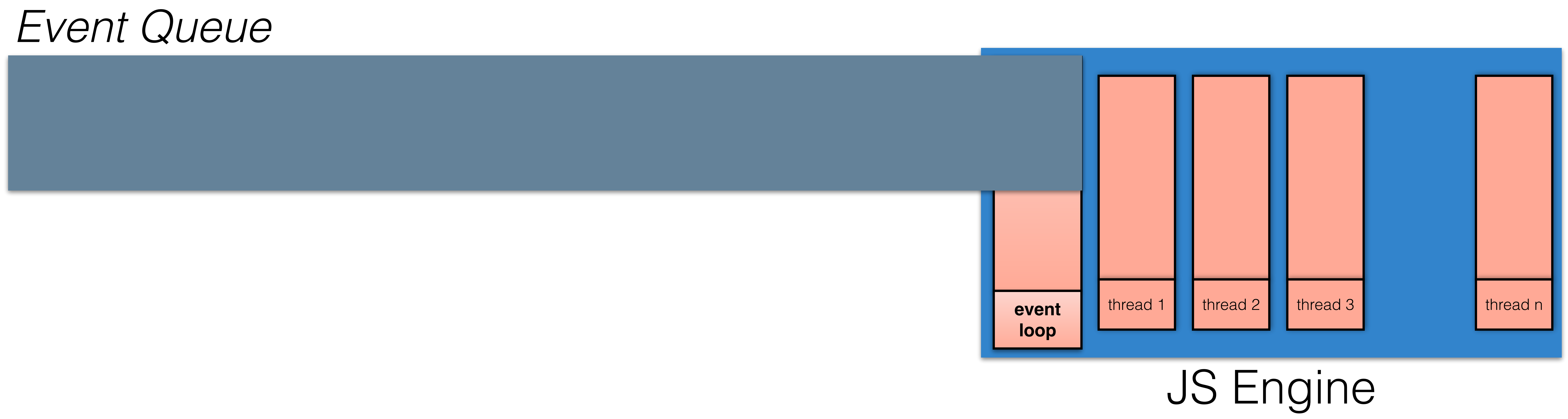
response from facebook.com

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 covey.town

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**

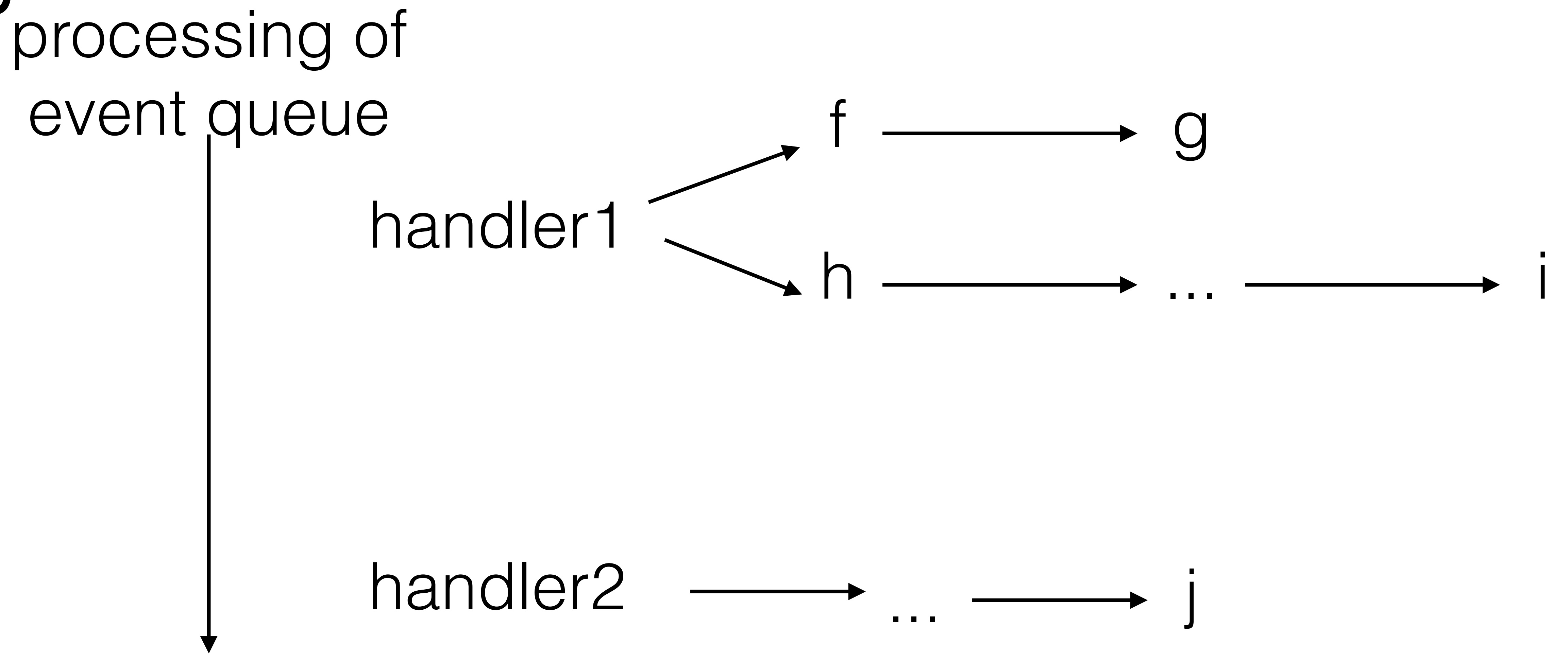
```
axios.get('https://rest-example.covey.town/') // axios is a popular library for making HTTP requests
  .then((response) =>{
  console.log('Heard back from server');
  console.log(response.data);
});
```

- Event loop is responsible for dispatching events when they occur
- Main thread for event loop (buried somewhere in NodeJS) :

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

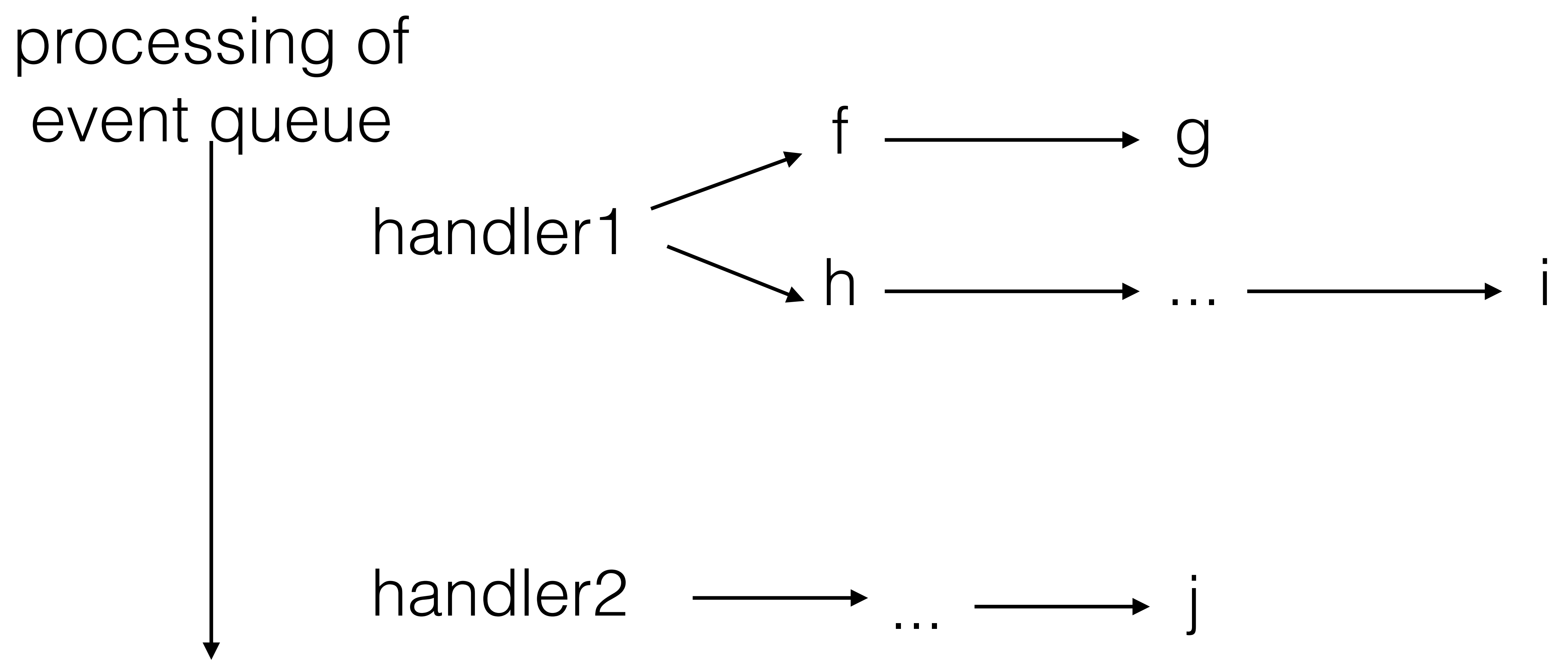
Run-to-completion semantics

- 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

Run-to-completion: first 2 lines ALWAYS first, covey.town handler lines always together

```
console.log('Making a requests');
axios.get('https://rest-example.covey.town/')
  .then((response) =>{
  console.log('Heard back from server');
  console.log(response.data);
});
axios.get('https://www.google.com/')
  .then((response) =>{
    console.log('Heard back from Google');
  });
axios.get('https://www.facebook.com/')
  .then((response) =>{
    console.log('Heard back from Facebook');
  });
console.log('Requests sent!');
```

These 2 lines ALWAYS first (same handler)

Sample Output:

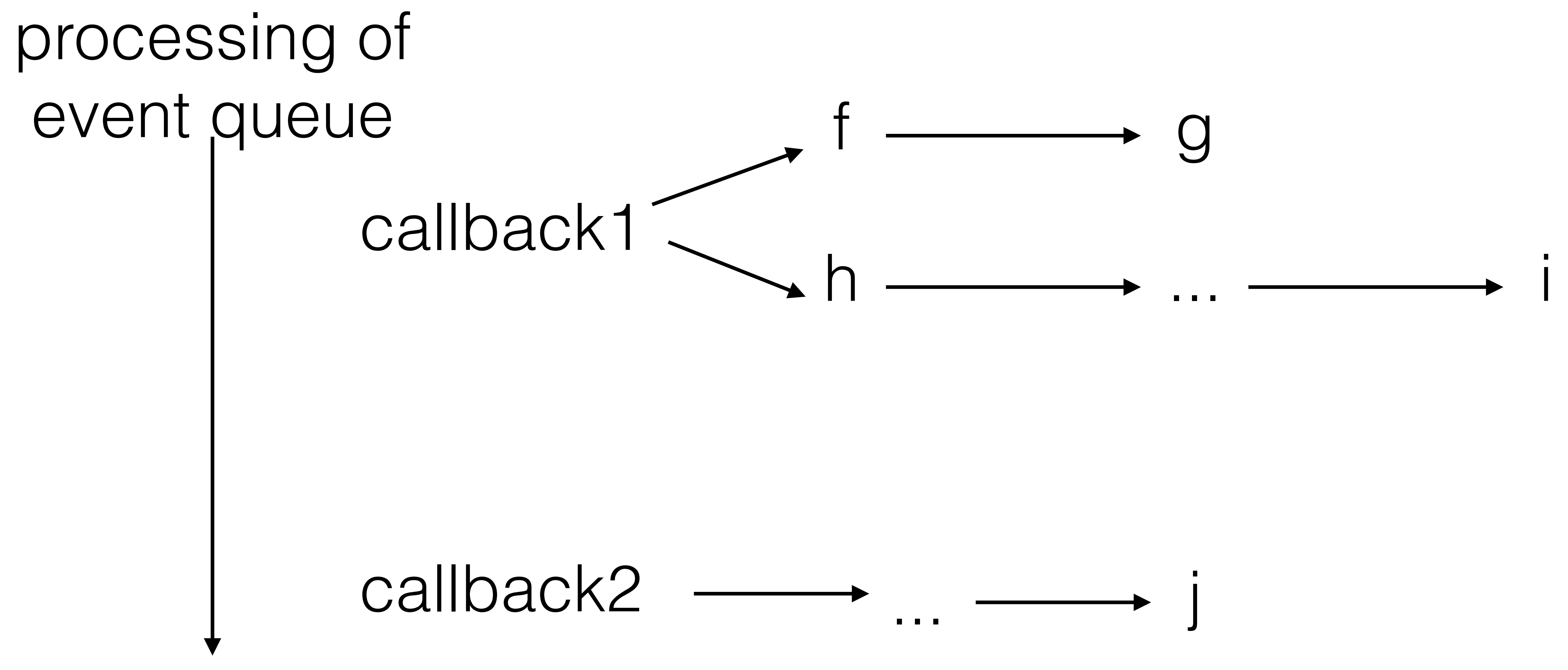
```
Making a requests
Requests sent!
Heard back from Google
Heard back from server
This is GET number 6 on the current server
Heard back from Facebook
```

These 2 lines ALWAYS together (same handler)

No guarantee on order of hearing back from Google, our server, or Facebook (new handlers)

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

What NOT to do in an event handler?

Run-to-completion: Slow handlers are really bad.

```
axios.get('https://rest-example.covey.town/')
  .then((response) =>{
  console.log('Heard back from server');
  console.log(response.data);
});
axios.get('https://www.google.com/')
  .then((response) =>{
    console.log('Heard back from Google');
    fs.writeFileSync("google-response.txt",response.data);
  });
axios.get('https://www.facebook.com/')
  .then((response) =>{
    console.log('Heard back from Facebook');
    fs.writeFileSync("facebook-response.txt",response.data);
  });
```

❌

3 seconds

Write a file *synchronously*
(write it in this event handler)

```
axios.get('https://rest-example.covey.town/')
  .then((response) =>{
  console.log('Heard back from server');
  console.log(response.data);
});
axios.get('https://www.google.com/')
  .then((response) =>{
    console.log('Heard back from Google');
    return fsPromises.writeFile("google-response.txt",response.data);
  });
axios.get('https://www.facebook.com/')
  .then((response) =>{
    console.log('Heard back from Facebook');
    return fsPromises.writeFile("facebook-response.txt",response.data);
  });
```

✓

2.1 seconds

Write a file *asynchronously*
(Ask NodeJS to write it in the background, this returns a new Promise to tell us when it's done)

Good news: You usually have to go out of your way to use synchronous I/O in NodeJS (the methods all have the word "Sync" in them) ✓

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
- Always add an error handler:

```
axios.get('https://www.facebook.com/')
  .then((response) =>{
    console.log('Heard back from Facebook');
  }).catch((error) => {
    console.log("Uh oh, I guess we should have an error handler!");
    console.trace(error);
});
```

Example: Writing Asynchronous Tasks

Transcript Server: Calculating statistics

- From an array of StudentIDs:
    - Request each student’s transcript
    - Then for each transcript, save it to disk so that we have a copy
    - Then once all of the pages are downloaded and saved, print out the total size of all of the files that were saved

Example: Writing Asynchronous Tasks

Transcript Server: Calculating statistics

- From an array of StudentIDs:
  - Request each student's transcript
  - Then for each transcript, save it to disk so that we have a copy
  - Then once all of the pages are downloaded and saved, print out the total size of all of the files that were saved

```
const studentIDs = [1, 2, 3, 4];
const promisesForTranscripts = studentIDs.map(
  studentID => axios.get(`https://rest-example.covey.town/transcripts/${studentID}`));
```

Functional magic: map will apply the function specified to each element in the array and return a new array containing the result of each of those functions

The function that is applied to each studentID: axios.get, which will return a promise!

Example: Writing Asynchronous Tasks

Transcript Server: Calculating statistics

- From an array of StudentIDs:
  - Request each student’s transcript
  - Then for each transcript, save it to disk so that we have a copy
  - Then once all of the pages are downloaded and saved, print out the total size of all of the files that were saved

```
const studentIDs = [1, 2, 3, 4];
const promisesForTranscripts = studentIDs.map(
  studentID => axios.get(`https://rest-example.covey.town/transcripts/${studentID}`)
    .then((response) =>
      fsPromises.writeFile(`transcript-${response.data.student.studentID}.json`, JSON.stringify(response.data))
    ));
```

Don’t return the axios promise: return a NEW promise, which will be complete when the request arrives… to save the file!

Example: Writing Asynchronous Tasks

Transcript Server: Calculating statistics

- From an array of StudentIDs:
  - Request each student’s transcript
  - Then for each transcript, save it to disk so that we have a copy
  - Then once all of the pages are downloaded and saved, print out the total size of all of the files that were saved

```
const studentIDs = [1, 2, 3, 4];
const promisesForTranscripts = studentIDs.map(
  studentID => axios.get(`https://rest-example.covey.town/transcripts/${studentID}`)
    .then((response) =>
      fsPromises.writeFile(`transcript-${response.data.student.studentID}.json`, JSON.stringify(response.data))
    ));
return Promise.all(promisesForTranscripts).then(results => {
  const statsPromises = studentIDs.map(studentID => fsPromises.stat(`transcript-${studentID}.json`));
});
```

New trick: Promise.all returns a new promise that completes when *all* of the promises passed are complete, it resolves with an array that contains each resolved promise value

Make an array of Promises for file statistics

Example: Writing Asynchronous Tasks

Transcript Server: Calculating statistics

- From an array of StudentIDs:
  - Request each student's transcript
  - Then for each transcript, save it to disk so that we have a copy
  - Then once all of the pages are downloaded and saved, print out the total size of all of the files that were saved

```
const studentIDs = [1, 2, 3, 4];
const promisesForTranscripts = studentIDs.map(
  studentID => axios.get(`https://rest-example.covey.town/transcripts/${studentID}`)
    .then((response) =>
      fsPromises.writeFile(`transcript-${response.data.student.studentID}.json`, JSON.stringify(response.data))
    ));
return Promise.all(promisesForTranscripts).then(results => {
  const statsPromises = studentIDs.map(studentID => fsPromises.stat(`transcript-${studentID}.json`));
  return Promise.all(statsPromises).then(stats => {
    const totalSize = stats.reduce((runningTotal, val) => runningTotal + val.size, 0);
    console.log(`Finished calculating size: ${totalSize}`);
  });
});
```

Now wait for the stats…

More functional magic: Take the array of stats, accumulate the size of each file

Problems with Promises

The order of operations is not intuitive from the code

```
console.log('Making a requests');
const studentIDs = [1, 2, 3, 4];
const promisesForTranscripts = studentIDs.map(
  studentID => axios.get(`https://rest-example.covey.town/transcripts/${studentID}`)
    .then((response) =>
      fsPromises.writeFile(`transcript-${response.data.student.studentID}.json`, JSON.stringify(response.data))
    ));
return Promise.all(promisesForTranscripts).then(results => {
  const statsPromises = studentIDs.map(studentID => fsPromises.stat(`transcript-${studentID}.json`));
  return Promise.all(statsPromises).then(stats => {
    const totalSize = stats.reduce((runningTotal, val) => runningTotal + val.size, 0);
    console.log(`Finished calculating size: ${totalSize}`);
  });
}).then(()=>{
  console.log('Done');
});
```

Async/Await

Your asynchronous friend

- 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 (this is what we want it for)!
  - Handle errors using try/catch

```
axios.get('https://rest-example.covey.town/').then(response => {
  console.log('Heard back from server');
  console.log(response.data);
}).catch(err => {
  console.log("Uh oh!");
  console.trace(err);
});
```

```
async function axiosAwaitExample() {
  try{
    const response = await axios.get('https://rest-example.covey.town/')
    console.log('Heard back from server');
    console.log(response.data);
  } catch(err){
    console.log("Uh oh!");
    console.trace(err);
  }
}
```

Example: Writing Asynchronous Tasks

Transcript Server: Calculating statistics (`async/await`)

- From an array of StudentIDs:
  - Request each student's transcript
  - Then for each transcript, save it to disk so that we have a copy
  - Then once all of the pages are downloaded and saved, print out the total size of all of the files that were saved

```
async function runClientAsync() {
  console.log('Making a requests');
  const studentIDs = [1, 2, 3, 4];
  const promisesForTranscripts = studentIDs.map(
    async (studentID) => {
      const response = await axios.get(`https://rest-example.covey.town/transcripts/${studentID}`)
   });
  console.log('Req
}
```

async: this function will automatically return a promise

Functional magic: map will apply the function specified to each element in the array and return a new array containing the result of each of those functions

await: wait for promise to resolve, then get its resolved value

Example: Writing Asynchronous Tasks

Transcript Server: Calculating statistics (`async/await`)

- From an array of StudentIDs:
  - Request each student's transcript
  - Then for each transcript, save it to disk so that we have a copy
  - Then once all of the pages are downloaded and saved, print out the total size of all of the files that were saved

```
unction runClientAsync() {
le.log('Making a requests');
st studentIDs = [1, 2, 3, 4];
const promisesForTranscripts = studentIDs.map(
  async (studentID) => {
    const response = await axios.get(`https://rest-example.covey.town/transcripts/${studentID}`)
    await fsPromises.writeFile(`transcript-${response.data.student.studentID}.json`, JSON.stringify(response.data))
  });
ts sent!');
}
```

async: the Promise we return won't be resolved until everything we await is

await: wait for promise to resolve, then get its resolved value

Example: Writing Asynchronous Tasks

Transcript Server: Calculating statistics (`async/await`)

- From an array of StudentIDs:
  - Request each student's transcript
  - Then for each transcript, save it to disk so that we have a copy
  - Then once all of the pages are downloaded and saved, print out the total size of all of the files that were saved

```
async function runClientAsync() {
  console.log('Making a requests');
  const studentIDs = [1, 2, 3, 4];
  const promisesForTranscripts = studentIDs.map(
    async (studentID) => {
      const response = await axios.get(`https://rest-example.covey.town/transcripts/${studentID}`)
      await fsPromises.writeFile(`transcript-${response.data.student.studentID}.json`, JSON.stringify(response.data))

  console.log('Requests sent!');
  await Promise.all(promisesForTranscripts);
  const stats = await Promise.all(studentIDs.map(studentID => fsPromises.stat(`transcript-${studentID}.json`)));
  const totalSize = stats.reduce((runningTotal, val) => runningTotal + val.size, 0);
  console.log(`Finished calculating size: ${totalSize}`);
  console.log('Done');
}
```

await for all transcripts to be downloaded and saved

await for all file statistics to be collected

Example: Writing Asynchronous Tasks

Transcript Server: Calculating statistics (`async/await vs Promise`)

```
function runClientPromises() {
  console.log('Making a requests');
  const studentIDs = [1, 2, 3, 4];
  const promisesForTranscripts = studentIDs.map(
    studentID => axios.get(`https://rest-example.covey.town/transcripts/${studentID}`)
      .then((response) =>
        fsPromises.writeFile(`transcript-${response.data.student.studentID}.json`, JSON.stringify(response.data))
      ));
  return Promise.all(promisesForTranscripts).then(results => {
    const statsPromises = studentIDs.map(studentID => fsPromises.stat(`transcript-${studentID}.json`));
    return Promise.all(statsPromises).then(stats => {
      const totalSize = stats.reduce((runningTotal, val) => runningTotal + val.size, 0);
      console.log(`Finished calculating size: ${totalSize}`);
    });
  }).then(() => {
    console.log('Done');
  });
  console.log('Requests sent!');
}
```

```
async function runClientAsync() {
  console.log('Making a requests');
  const studentIDs = [1, 2, 3, 4];
  const promisesForTranscripts = studentIDs.map(
    async (studentID) => {
      const response = await axios.get(`https://rest-example.covey.town/transcripts/${studentID}`)
      await fsPromises.writeFile(`transcript-${response.data.student.studentID}.json`, JSON.stringify(response.data))
    });
  console.log('Requests sent!');
  await Promise.all(promisesForTranscripts);
  const stats = await Promise.all(studentIDs.map(studentID => fsPromises.stat(`transcript-${studentID}.json`)));
  const totalSize = stats.reduce((runningTotal, val) => runningTotal + val.size, 0);
  console.log(`Finished calculating size: ${totalSize}`);
  console.log('Done');
}
```

Async/Await gone mad

Where you place awaits can make a big difference!

The code we've seen on past slides:

For each student: make an async handler to fetch their transcript and save it

```
on runClientAsync() {
g('Making a requests');
const studentIDs = [1, 2, 3, 4];
const promisesForTranscripts = studentIDs.map(
  async (studentID) => {
    const response = await axios.get(`https://rest-example.covey.town/transcripts/${studentID}`)
    await fsPromises.writeFile(`transcript-${response.data.student.studentID}.json`, JSON.stringify(response.data))
  });
console.log('Requests sent!');
await Promise.all(promisesForTranscripts);
const stats = await Promise.all(studentIDs.map(studentID => fsPromises.stat(`transcript-${studentID}.json`)));
const totalSize = stats.reduce((runningTotal, val) => runningTotal + val.size, 0);
console.log(`Finished calculating size: ${totalSize}`);
```

✓

Running time: 1.5 sec

For each student: wait to fetch their transcript, then wait to write it, then go on to the next student

This does something different:

```
async function runClientAsyncSerially() {
  console.log('Making a requests');
  const studentIDs = [1, 2, 3, 4];
  for(let studentID of studentIDs){
    const response = await axios.get(`https://rest-example.covey.town/transcripts/${studentID}`);
    await fsPromises.writeFile(`transcript-${response.data.student.studentID}.json`, JSON.stringify(response.data))
  }
  let totalSize = 0;
  for(let studentID of studentIDs){
    const stats = await fsPromises.stat(`transcript-${studentID}.json`);
    totalSize += stats.size;
  }
  console.log(`Finished calculating size: ${totalSize}`);
}
```

✗

Running time: 2.2 sec

This is what we mean by "your code can become synchronous"

Async/Await Programming Activity

Transcript Server: Create a student, then update their

1. Create a new student in the transcript server

```
axios.post('https://rest-example.covey.town/transcripts', {name: ‘Breakout Group 0’})
```

then…

2. Assign several grades for that student

```
axios.post(`https://rest-example.covey.town/transcripts/${studentID}/${course}`,{grade: theGrade}))
```

then…

3. Fetch the transcript for that student

```
axios.get(`https://rest-example.covey.town/transcripts/${studentID}`)
```

If you finish with time to spare, try to make different variants: make a lot of requests concurrently vs making the requests synchronously (waiting between each request)

This work is licensed under a Creative Commons Attribution-ShareAlike license