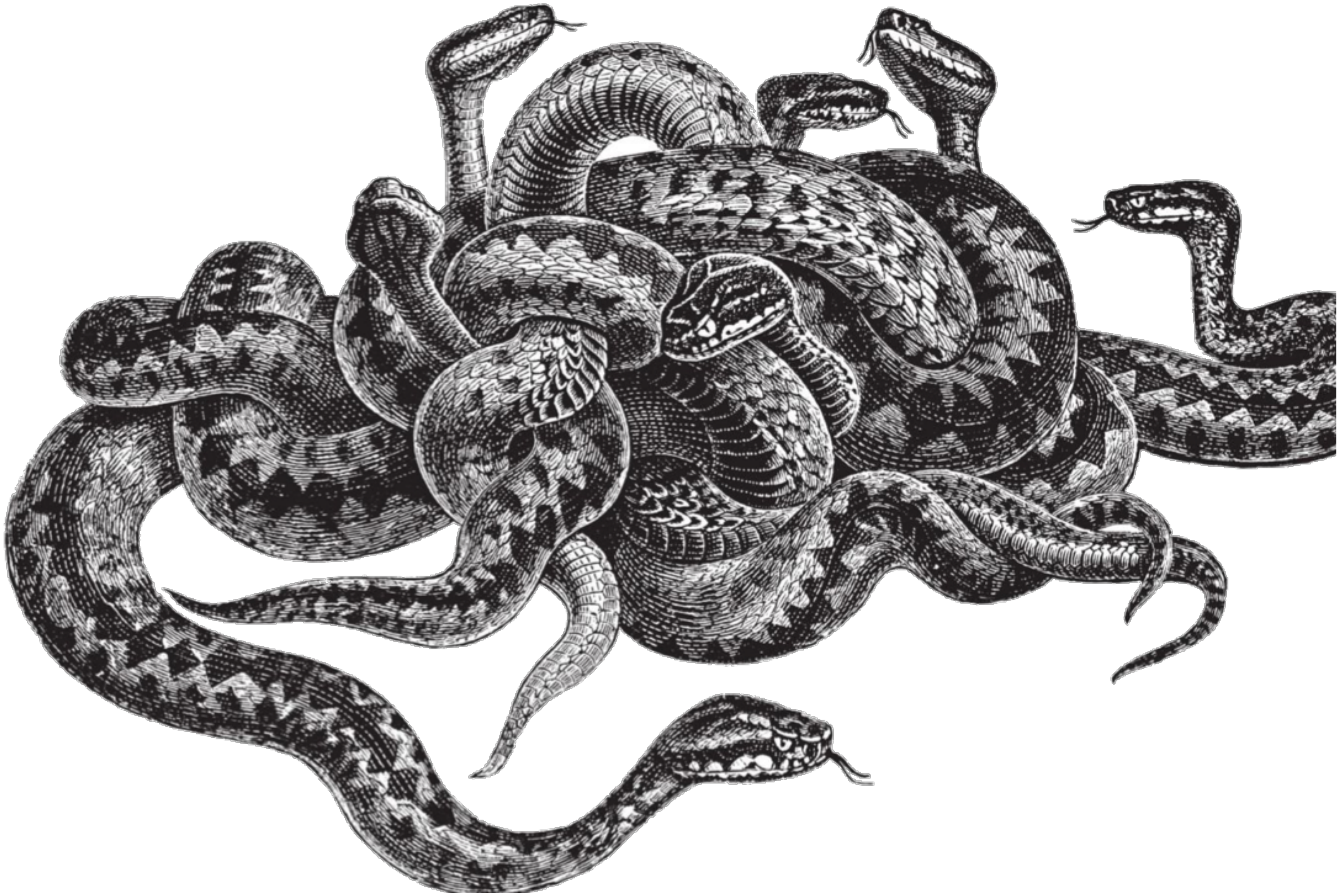
Asynchronous Programming



Andrew DeOrio 2017

Image credit: "Twisted - Network Programming Essentials" by Abe Fettig

Agenda

- Asynchronous programming introduction
- Review: JavaScript event table, event loop and event queue
- AJAX
- Using Promises
- Creating Promises
- Asynchronous, event-driven and ES7

Asynchronous is not ...

- Asynchronous programming is not a single-thread blocking program
- Blocking: wait for one task to finish before executing the next

- Examples of tasks:
  1. fetch(): a GET request to a REST API
  2. json(): parse JSON string
  3. Respond to user clicking a button on UI and update UI

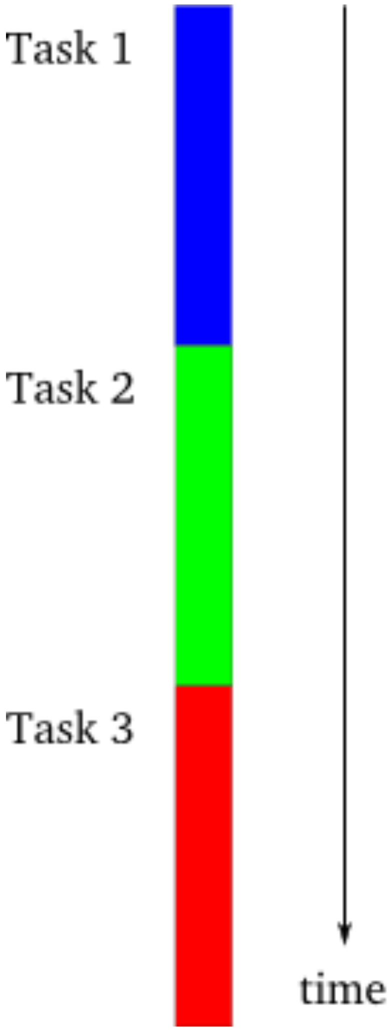


Image credit: http://krondo.com/

Asynchronous is not ...

- Asynchronous programming is not a multi-thread blocking program
- Modern OS threads "take turns" on one processor

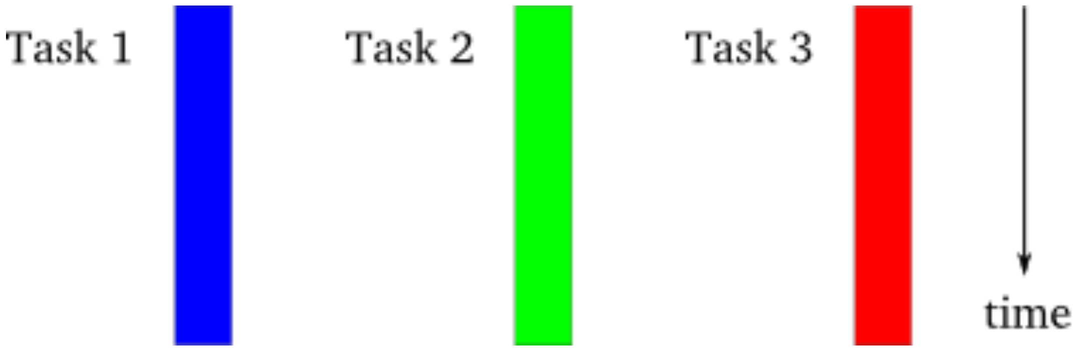


Image credit: http://krondo.com/

Asynchronous is ...

- Asynchronous programming is tasks interleaved with one another, in a single thread of control
- Programmer controls when tasks "take turns"

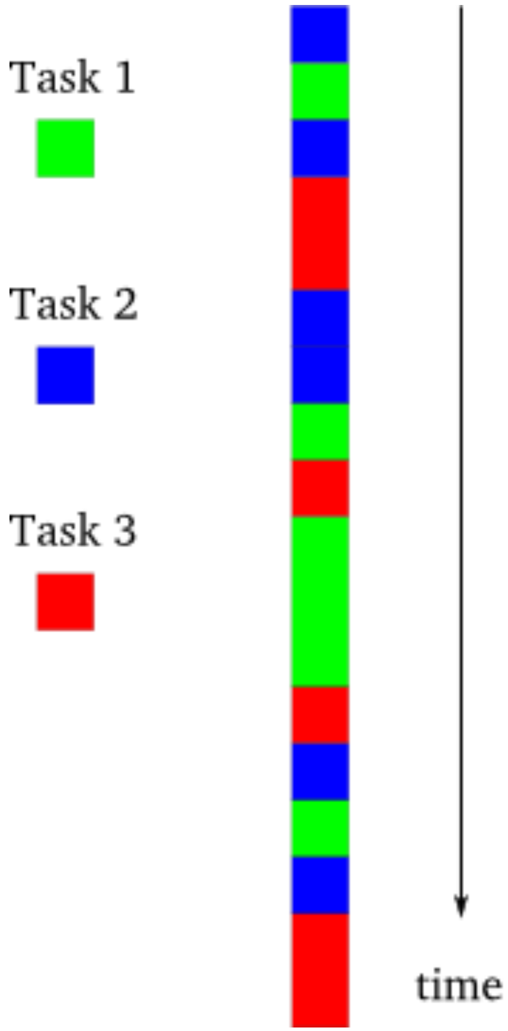


Image credit: http://krondo.com/

Why asynchronous?

- Why use asynchronous programming?
- UIs: by interleaving the tasks, system is responsive to user input while still performing other work in the "background”
- Waiting for I/O: do "other useful things" while waiting for I/O, like a network or disk
  - Synchronous programs are bad at this

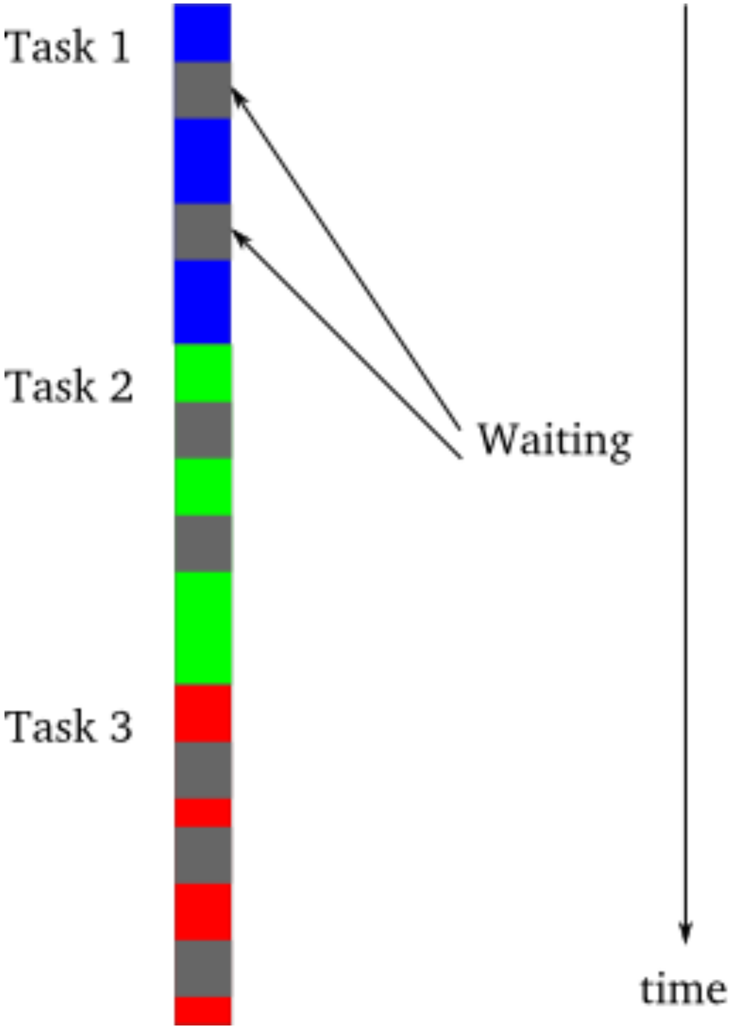


Image credit: http://krondo.com/

Why asynchronous?

- What are "other useful things" to do while waiting in a web app?
  - Respond to user mouse hover event
  - Respond to user clicking a radio button
  - Respond to use filling in a form, e.g., validate input
  - Check for new mail (Gmail)
  - Check for new posts (Facebook)

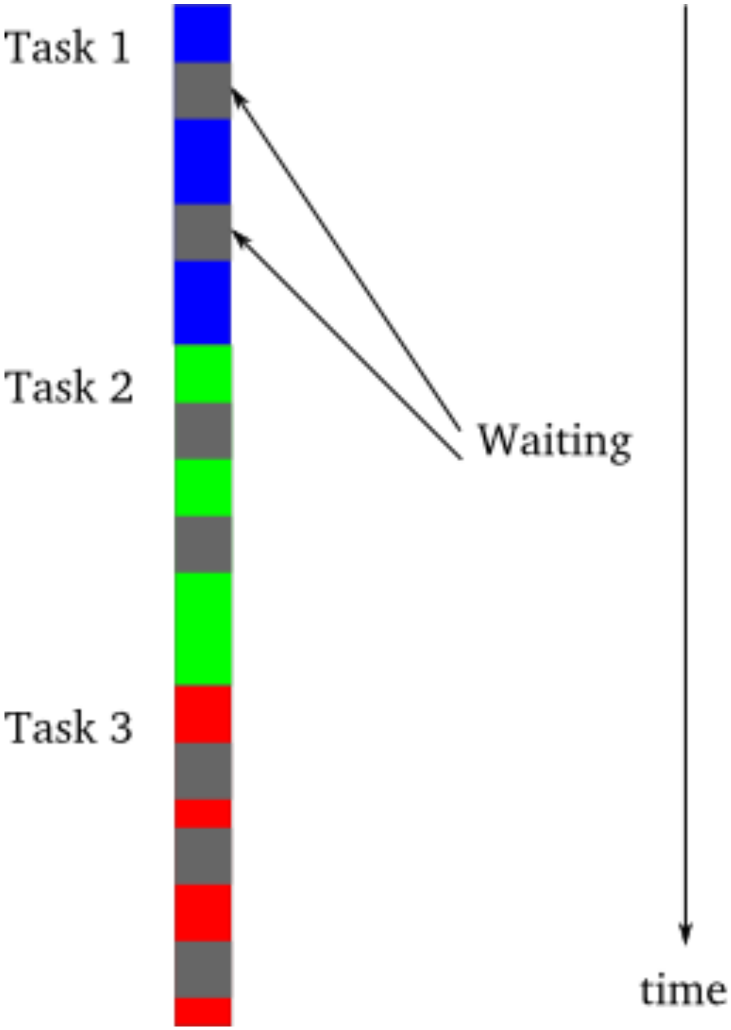


Image credit: http://krondo.com/

When asynchronous?

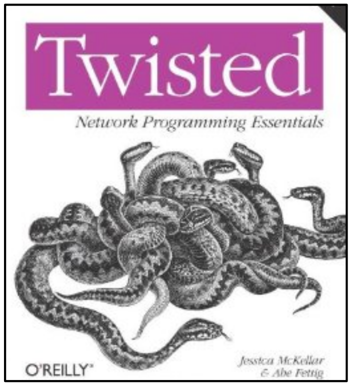
- When to use asynchronous programming?

- There are a large number of tasks so there is likely always at least one task that can make progress
- The tasks perform lots of I/O, causing a synchronous program to waste lots of time blocking when other tasks could be running
- The tasks are largely independent from one another so there is little need for inter-task communication (and thus for one task to wait upon another)

- These conditions are common in web systems!

What is asynchronous?

- Examples of existing web technology using asynchronous programming



- Twisted
  - A networking library written in Python
- NGINX
  - A web server
- AJAX
  - Asynchronous JavaScript and XML

Agenda

- Asynchronous programming introduction
- **Review: JavaScript event table, event loop and event queue**
- AJAX
- Using Promises
- Creating Promises
- Asynchronous, event-driven and ES7

Review: the event queue

- In JavaScript, function calls live on the stack, objects live on the heap, and *messages live on the queue*
- The function on the top of the stack executes.
- *When the stack is empty, a message is taken out of the queue and processed.*
- Each message is a function

- An event adds a message to the queue

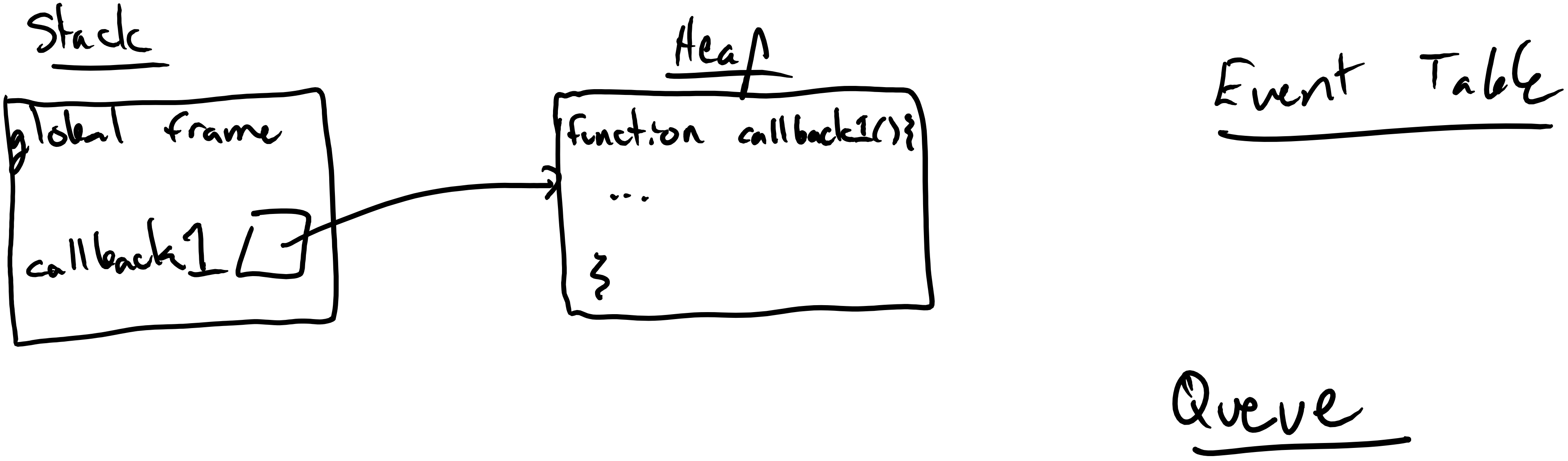
Review: adding events to the queue

- Example: You can schedule an event on the queue for a later time
- This function will run approximately 1s in the future
- `callback1` is added to the *event table*, which maps events to callbacks

```
function callback1() {
    console.log('this is a msg from callback1');
}
setTimeout(callback1, 1000);
```

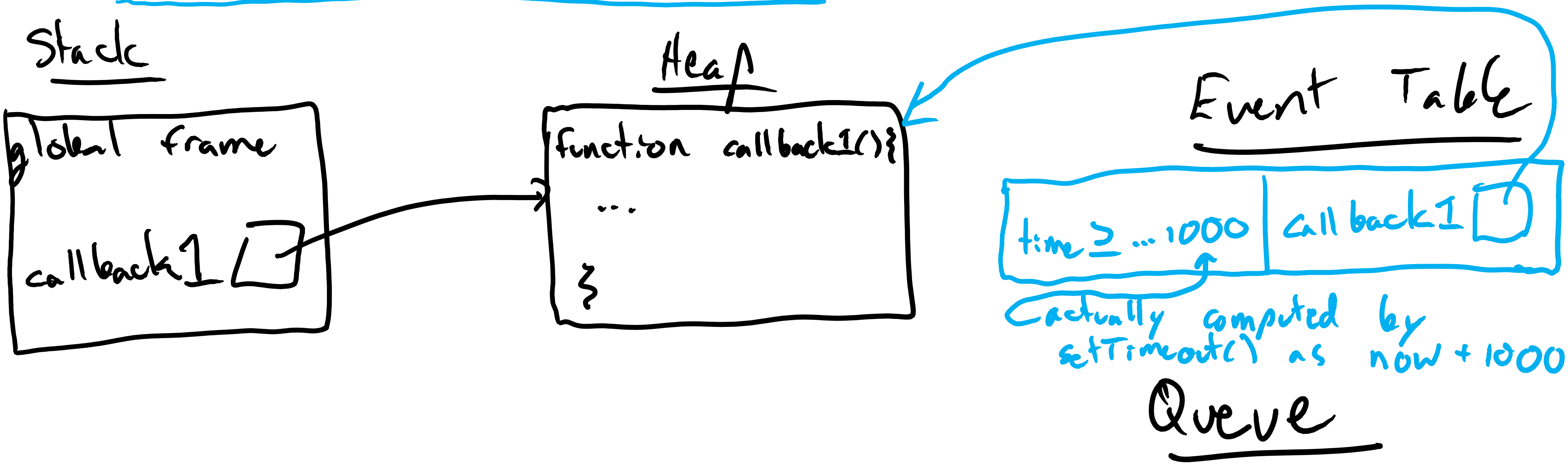
Review: adding events to the queue

```
function callback1() {
    console.log('this is a msg from callback1');
}
setTimeout(callback1, 1000);
```



Review: adding events to the queue

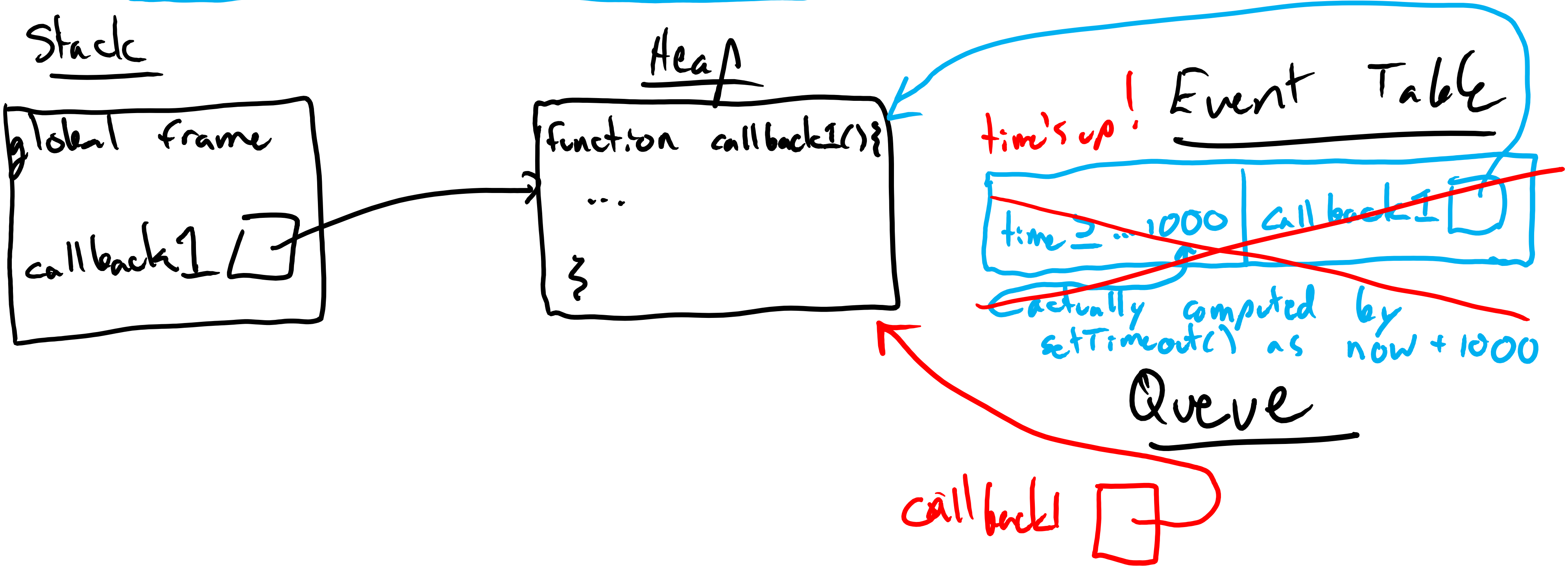
```
function callback1() {
    console.log('this is a msg from callback1');
}
setTimeout(callback1, 1000);
```



Review: adding events to the queue

1000 ms later...

```
function callback1() {
    console.log('this is a msg from callback1');
}
setTimeout(callback1, 1000);
```

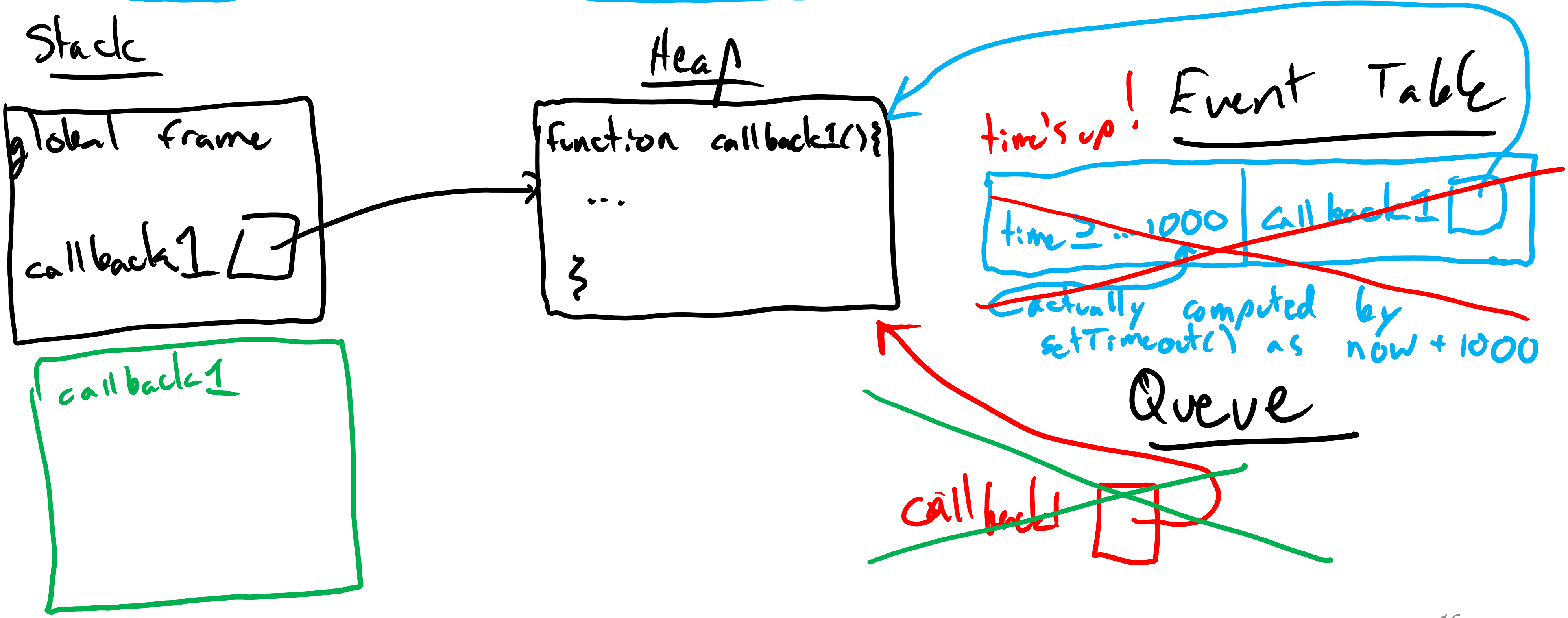


Output: 'this is a msg from callback1'

Review: adding events to the queue

1000 ms later...

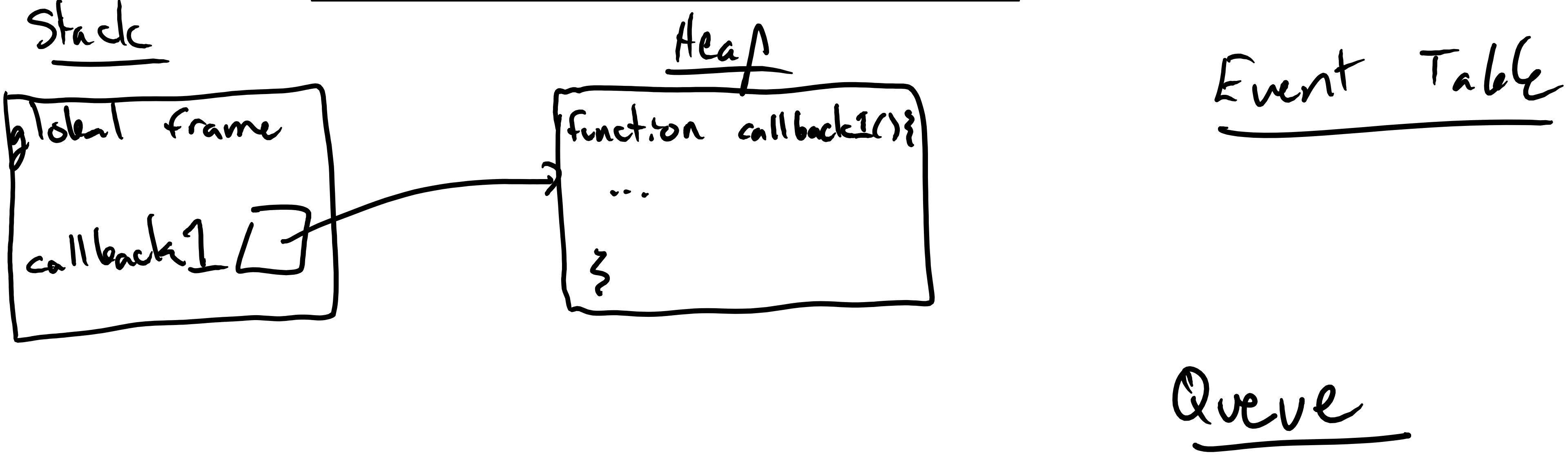
```
function callback1() {
    console.log('this is a msg from callback1');
}
setTimeout(callback1, 1000);
```



Review: adding events to the queue

```
function callback1() {
    console.log('this is a msg from callback1');
}
setTimeout(callback1, 1000);
slow();
```

<- how would this example change?



Agenda

- Asynchronous programming introduction
- Review: JavaScript event table, event loop and event queue
- **AJAX**
- Using Promises
- Creating Promises
- Asynchronous, event-driven and ES7

AJAX

```
<!-- index.html -->
<html>
  <head></head>
  <body>
    <div id="JSEntry"></div>
    <script src="users.js"></script>
  </body>
</html>
```

- AJAX: Asynchronous JavaScript and XML
  - XML is a misnomer these days, we use JSON
- We implemented an AJAX app last time

```
//users.js
function showUser() {
  function handleResponse(response) {/*... */}

  function handleData(data) {/*...*/}

  fetch('https://api.github.com/users/awdeorio')
    .then(handleResponse)
    .then(handleData)
}
showUser();
```

AJAX

- We implemented an AJAX app last time
  - `handleResponse()` runs asynchronously, after server response arrives
  - `handleData()` runs asynchronously, after JSON parsing is finished

```
function showUser() {
  function handleResponse(response) {/*... */}

  function handleData(data) {/*...*/}

  fetch('https://api.github.com/users/awdeorio')
    .then(handleResponse)
    .then(handleData)
}
showUser();
```

GitHub API

- We'll use the GitHub API for our examples today
- Example:

```
$ curl -s https://api.github.com/users/awdeorio
{
  "login": "awdeorio",
  "id": 7503005,
  "avatar_url": "https://avatars3.githubusercontent.com/u/7503005?v=4",
  ...
  "url": "https://api.github.com/users/awdeorio",
  ...
}
```

Review: fetch API

- The `fetch` API provides an interface for HTTP requests
- Call a function when the response arrives
    - Parse JSON into JavaScript object
- Call another function when JSON parsing is finished
    - Add DOM nodes using JavaScript object

```
function showUser() {
  fetch('https://api.github.com/users/awdeorio')
    .then(/* handle response and parse JSON */)
    .then(/* handle data and add DOM nodes */)
}
```

Review: fetch API

- Function to parse JSON from HTTP response
- `fetch` calls this function when response arrives

```
function showUser() {
  function handleResponse(response) {
    return response.json();
  }

  fetch('https://api.github.com/users/awdeorio')
    .then(handleResponse)
    .then(/* handle data and add DOM nodes */)
}
```

Review: fetch API

- Add a function to process the data parsed from the JSON response

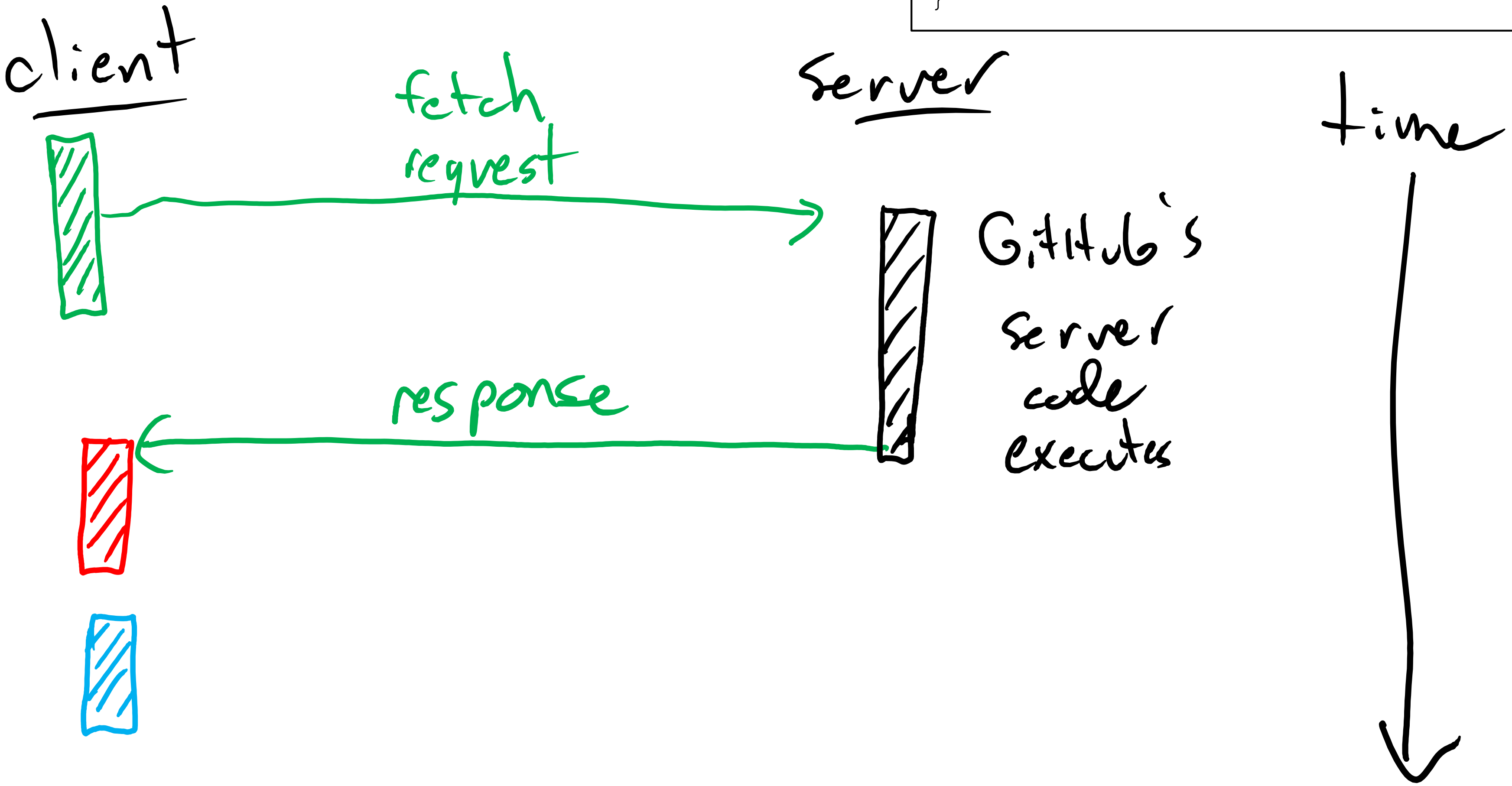
```
function showUser() {
  //...
  function handleResponse(response) {
    return response.json();
  }

  function handleData(data) {
    // just print to console for today's examples
    console.log(data);
  }
  fetch('https://api.github.com/users/awdeorio')
    .then(handleResponse)
    .then(handleData)
}
```

Fetch API timing diagram

```
function showUser() {
  function handleResponse(response)
  { /*...*/ }

  function handleData(data)
  { /*...*/ }

  fetch(/*...*/)
    .then(handleResponse)
    .then(handleData)
}
```

Fetch API timing diagram

```
function showUser() {
  function handleResponse(response)
  { /*...*/ }

  function handleData(data)
  { /*...*/ }

  fetch(/*...*/)
    .then(handleResponse)
    .then(handleData)
}
```

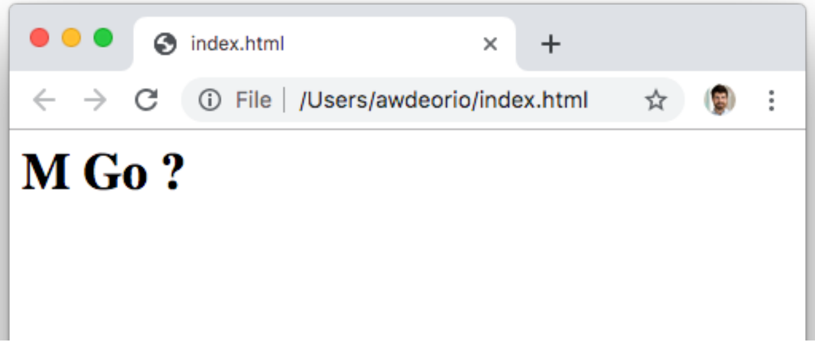


More work to do

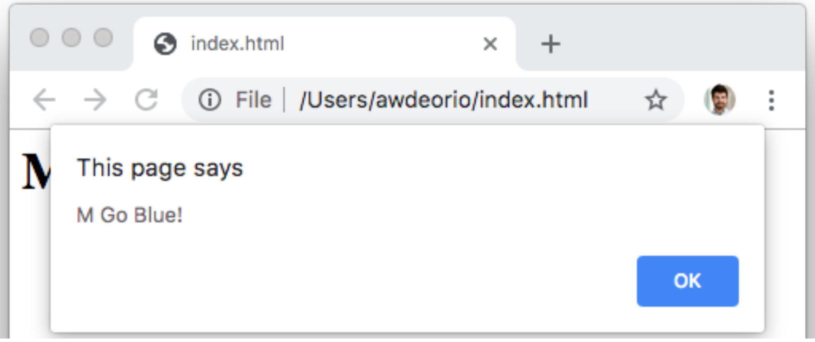
- Let's add an extra user interface feature
- Pop-up when user hovers over the title
- Now, we have 4 tasks:
  1. fetch()
  2. handleData()
  3. handleResponse()
  4. mgoblue()

```
function showUser() {
  function handleResponse(response)
  { /*...*/ }

  function handleData(data)
  { /*...*/ }

  fetch(/*...*/)
    .then(handleResponse)
    .then(handleData)
}
showUser();

function mgoblue() {
  window.alert("M Go Blue!");
}
```

```
<html>
  <body>
    <h1 onmouseover="mgoblue()">M Go ?</h1>
    <script src="test.js"></script>
  </body>
</html>
```



M Go ?



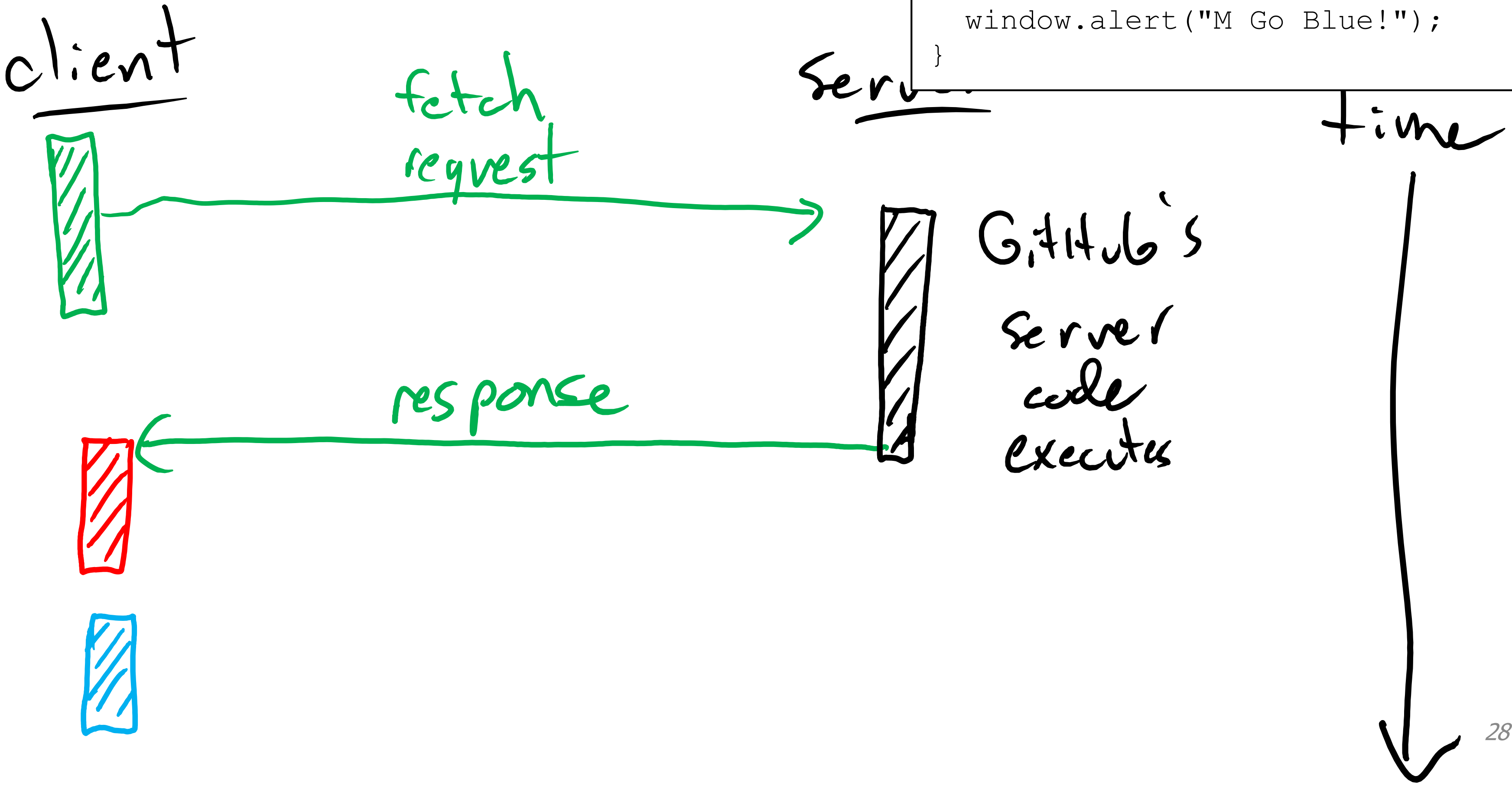
This page says

M Go Blue!

More work to do

- What happens if the user hovers *before* the server response arrives? What about before `handleData()`?

```
function showUser() {
  function handleResponse(response)
  { /*...*/ }

  function handleData(data)
  { /*...*/ }

  fetch(/*...*/)
    .then(handleResponse)
    .then(handleData)
}
showUser();

function mgoblue() {
  window.alert("M Go Blue!");
}
```



Agenda

- Asynchronous programming introduction
- Review: JavaScript event table, event loop and event queue
- AJAX
- **Using Promises**
- Creating Promises
- Asynchronous, event-driven and ES7

Promises

- Control the flow of deferred and asynchronous operations
- First class representation of a value that may be made asynchronously and be available in the future
- Added to JavaScript in ES6

- Examples of values that will be available in the future
  - The response to a server request: `fetch()`
  - The data from parsing a JSON string: `json()`

Using a `Promise`

- `fetch()` returns a `Promise`
- `response.json()` returns a `Promise`

```
function showUser() {
  function handleResponse(response) {
    return response.json();
  }

  function handleData(data) {
    console.log(data);
  }

  fetch('https://api.github.com/users/awdeorio')
    .then(handleResponse)
    .then(handleData)
}
```

Using a `Promise`

- After the value is available, the `Promise` calls a function provided by `.then()`

```
function showUser() {
  function handleResponse(response) {
    return response.json();
  }

  function handleData(data) {
    console.log(data);
  }

  fetch('https://api.github.com/users/awdeorio')
    .then(handleResponse)
    .then(handleData)
}
```

Using a `Promise`: diagram

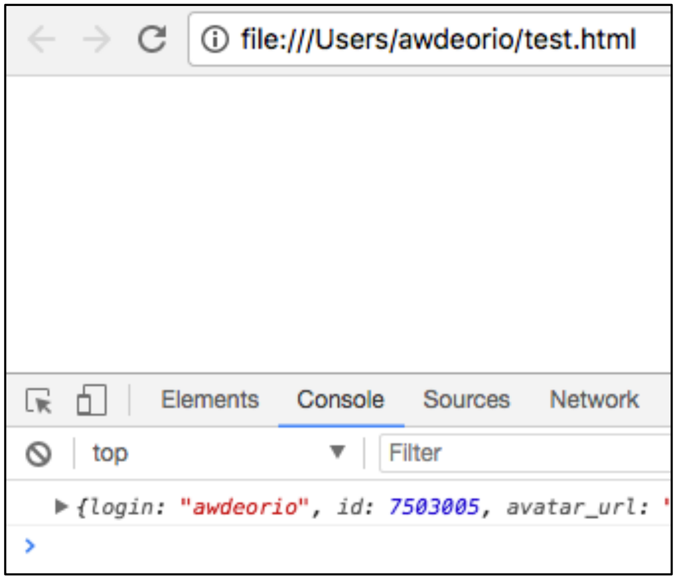
- Imagine a `Promise` as a linked list of function objects

```
fetch('https://api.github.com/users/awdeorio')
     .then(handleResponse)
     .then(handleData)
}
```

Using a `Promise`

- Refactor to use anonymous functions

```
function showUser() {
  fetch('https://api.github.com/users/awdeorio')
    .then((response) => {
      return response.json();
    })
    .then((data) => {
      console.log(data);
    })
}
```



file:///Users/awdeorio/test.html

```
▶{login: "awdeorio", id: 7503005, avatar_url: '
>
```

Promises explained again

- Functions performing asynchronous tasks return a `Promise`
- A `Promise` is an object to which you can attach a callback
  - Using `.then()`

```
function showUser() {
  fetch('https://api.github.com/users/awdeorio')
    .then((response) => {
      return response.json();
    })
    .then((data) => {
      console.log(data);
    })
}
```

Promise states

- A `Promise` is in one of these states:
  - *pending*: initial state, neither fulfilled nor rejected
  - *fulfilled*: meaning that the operation completed successfully
  - *rejected*: meaning that the operation failed

- On success, the method provided by `.then()` runs

Promises explained again

- We can rewrite this code to use variables instead of chaining

```
//before
function showUser() {
  fetch('https://api.github.com/users/awdeorio')
    .then((response) => {
      return response.json();
    })
    .then((data) => {
      console.log(data);
    })
}


//after
function showUser() {
  let p1 = fetch('https://api.github.com/users/awdeorio');
  let p2 = p1.then(response => response.json());
  let p3 = p2.then(data => console.log(data));

}
```

Diagra

```
function showUser() {
  let p1 = fetch('https://api.github.com/users/awdeorio');
  let p2 = p1.then(response => response.json());
  let p3 = p2.then(data => console.log(data));
}
```

Exercise

- What is the output of this code?

```
function showUser() {
  console.log("hello");
  let p1 = fetch('https://api.github.com/users/awdeorio');
  let p2 = p1.then(response => response.json());
  let p3 = p2.then(data => console.log(data.login));
  console.log("world");
}
```

Exercise

- What is the output of this code?

```
function showUser() {
  console.log("hello");
  let p1 = fetch('https://api.github.com/users/awdeorio');
  let p2 = p1.then(response => response.json());
  let p3 = p2.then(data => console.log(data.login));
  console.log("world");
}
```

```
// hello
// world
// awdeorio
```

Chaining promises

- A common need is to execute two or more asynchronous operations back-to-back, where each subsequent operation starts when the previous operation succeeds, with the result from the previous step.
- Example:
  1. Request
  2. Parse JSON
- We accomplish this by creating a *promise chain*

```
function showUser() {
  fetch('https://api.github.com/users/awdeorio')
    .then((response) => {
      return response.json();
    })
    .then((data) => {
      console.log(data);
    })
}
```

Chaining promises

- The output (resolved value) of one Promise is the input to the next

```
function showUser() {
  fetch('https://api.github.com/users/awdeorio')
    .then((response) => {
      return response.json();
    })
    .then((data) => {
      console.log(data);
    })
}
```

Error handling

- We can also provide a callback for handling a errors
- A Promise will call one of the two callbacks provided by
  - `.then()`
  - `.catch()`

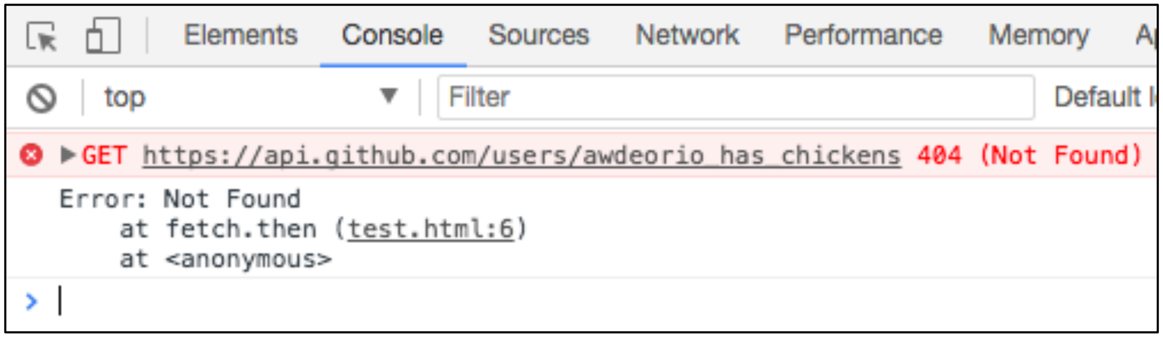
```
function showUser() {
  fetch('https://api.github.com/users/awdeorio')
    .then((response) => {
      if (!response.ok) throw Error(response.statusText);
      return response.json();
    })
    .then((data) => {
      console.log(data);
    })
    .catch(error => console.log(error))
}
```

Error example

- REST APIs typically return errors in JSON format instead of HTML

```
$ http
https://api.github.com/users/awdeorio_has_chickens
HTTP/1.1 404 Not Found

{

  "documentation_url":
"https://developer.github.com/v3/users/#get-a-single-
user",

  "message": "Not Found"

}
```

Error propagation

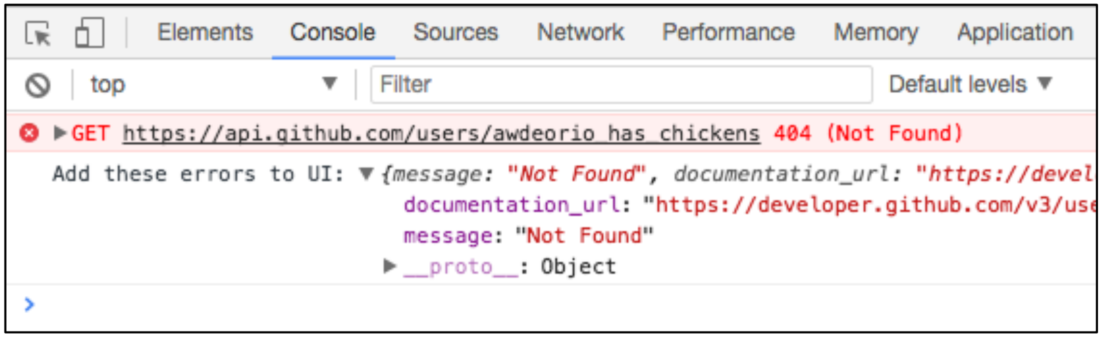


```
▶GET https://api.github.com/users/awdeorio_has_chickens 404 (Not Found)
Error: Not Found
    at fetch.then (test.html:6)
    at <anonymous>
> |
```

- A promise chain stops if there's an exception, looking down the chain for catch handlers instead
- REST API returned 4xx will trigger error
- Similar to `try/catch` in synchronous code

```
function showUser() {
  fetch('https://api.github.com/users/awdeorio_has_chickens')
    .then((response) => {
      if (!response.ok) throw Error(response.statusText);
      return response.json();
    })
    .then((data) => {
      console.log(data);
    })
    .catch(error => console.log(error))
}
```

Error handling



```
GET https://api.github.com/users/awdeorio_has_chickens 404 (Not Found)
Add these errors to UI: ▼{message: "Not Found", documentation_url: "https://devel
                          documentation_url: "https://developer.github.com/v3/use
                          message: "Not Found"
                         ▶__proto__: Object
>
```

- Chain *after* a failure, i.e. a `catch`, to handle error
- Recall: REST APIs usually return JSON error messages

```
function showUser() {
  fetch('https://api.github.com/users/awdeorio_has_chickens')
    .then((response) => {
      if (!response.ok) throw Error(response.statusText);
      return response.json();
    })
    .then((data) => {
      console.log(data);
    })
    .catch((errorResponse) => {
      errorResponse.json()
        .then(errorData => {
          console.log('Add these errors to UI:', errorData);
        });
    })
}
```

Agenda

- Asynchronous programming introduction
- Review: JavaScript event table, event loop and event queue
- AJAX
- Using Promises
- **Creating Promises**
- Asynchronous, event-driven and ES7

Creating a `Promise`

- So far, we've looked at Promises from the perspective of using a Promise returned by a function that somebody else wrote
- Next, we'll look at them from the perspective of creating a Promise "from scratch"

Creating a `Promise`

- Let's turn `setTimeout` into a function that returns a `Promise`
- Remember, `setTimeout` calls a function after a period of time

```
function callback1() {
  console.log("1 second passed");
}
setTimeout(callback1, 1000);
```

- Refactor using an anonymous function

```
setTimeout(() => console.log("1 second passed"), 1000);
```

Creating a `Promise`

```
function callback1() {
  console.log("1 second passed");
}
setTimeout(callback1, 1000);
```

- Refactor using an anonymous function

```
setTimeout(() => console.log("1 second passed"), 1000);
```

- Refactor to use a `Promise`

```
function wait(ms) {
  return new Promise(resolve => {
    setTimeout(resolve, ms);
  });
}
```

Executor function

- A Promise has an *executor function*
- An executor function normally initiates some asynchronous work, and calls `resolve()` once the work completes

-
  ```
  function wait(ms) {
    return new Promise(resolve => {
      setTimeout(resolve, ms);
    });
  }
  ```
- Executor function is executed immediately
  ```
  wait(1000)
    .then(() => console.log('1 second passed'));
  ```
- Equivalent code that runs immediately:
  ```
  setTimeout(() => console.log('1 second passed'), 1000);
  ```

Success handler

- A Promise allows you to associate handlers with an asynchronous action's eventual success value

```
function wait(ms) {
  return new Promise(resolve => {
    setTimeout(resolve, ms);
  });
}
```

- `.then()` associates the handler for success

```
wait(1000)
  .then(() => console.log('1 second passed'));
```

Failure handler

- A Promise allows you to associate handlers with an asynchronous action's eventual failure reason

```
function wait(ms) {
  return new Promise(resolve => {
    setTimeout(resolve, ms);
  });
}
```

- `.catch()` associates the handler for failure

```
wait(1000)
  .then(() => console.log('1 second passed'))
  .catch(error => console.log(error))
```

Relation to synchronous methods

- Asynchronous methods (like `wait`) return values like synchronous methods
- Instead of immediately returning the final value, the asynchronous method returns a *promise* to supply the value at some point in the future

```
wait(1000)
  .then(() => console.log('1 second passed'))
  .catch(error => console.log(error))
```

Promise states

- Recall: a Promise is in one of these states:
  - *pending*: initial state, neither fulfilled nor rejected
  - *fulfilled*: meaning that the operation completed successfully
  - *rejected*: meaning that the operation failed

- If the executor function succeeds, then the method provided by `.then()` runs
- If the executor function fails, then the method provided by `.catch()` runs

Exercise

- What is the output? How long does this program take?

```
function main() {
  wait(1000).then(() => console.log('1 s passed'));
  wait(0).then(() => console.log('0 s passed'));
  wait(500).then(() => console.log('0.5 s passed'));
}

main();
```

Solution

- What is the output? How long does this program take?

```
function main() {
  wait(1000).then(() => console.log('1 s passed'));
  wait(0).then(() => console.log('0 s passed'));
  wait(500).then(() => console.log('0.5 s passed'));
}

main();
```

Output

```
0 s passed
0.5 s passed
1 s passed
```

Runtime

```
1.0s
```

Exercise

- What is the output? How long does this program take?

```
function main() {
  wait(1000)
  .then(() => {
    console.log('1 s passed');
    return wait(0);
  })
  .then(() => {
    console.log('0 s passed');
    return wait(500);
  })
  .then(() => console.log('0.5 s passed'));
}
main();
```

Solution

- What is the output? How long does this program take?

```
function main() {
  wait(1000)
  .then(() => {
    console.log('1 s passed');
    return wait(0);
  })
  .then(() => {
    console.log('0 s passed');
    return wait(500);
  })
  .then(() => console.log('0.5 s passed'));
}
main();
```

Output

```
1 s passed
0 s passed
0.5 s passed
```

Runtime

```
1.5s
```

Agenda

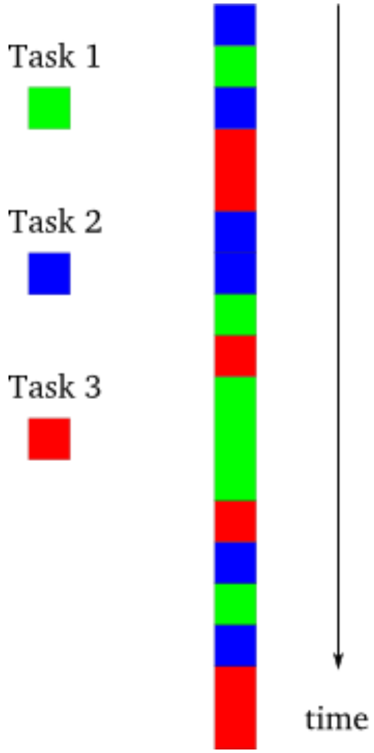
- Asynchronous programming introduction
- Review: JavaScript event table, event loop and event queue
- AJAX
- Using Promises
- Creating Promises
- **Asynchronous, event-driven and ES8**

Asynchronous vs. event-driven

- Asynchronous programming describes the execution

- Event-driven describes the implementation

```
// somewhere in the JS interpreter
while (queue.waitForMessage()) {
  queue.processNextMessage();
}
```



Asynchronous and ES8

- Cool features in ES8
  - `async` and `await` keywords
- Syntactic sugar for a `Promise`
  - `.then()`

```
// ES6-style
function showUser() {
  fetch('https://api.github.com/users/awdeorio')
  .then((response) => {
    if (!response.ok) throw Error(response.statusText);
      return response.json();
    })
  .then((data) => {
    console.log(data);
  })
}
```

Async/await

- `async` functions return a `Promise`
- `async` functions can contain `await` expressions
- `await` pauses the execution of the `async` function and waits for the passed `Promise's` resolution, and then resumes the `async` function's execution and returns the resolved value.

```
// ES8 async/await style
async function showUser() {
  // await response of fetch call
  let response = await fetch('https://api.github.com/users/awdeorio');

  // only proceed once promise is resolved
  let data = await response.JSON();

  // only proceed once second promise is resolved
  // add nodes to DOM here
  console.log(data);
}
```

Further reading: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/async_function

Further reading

- https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise