Intro to Asynchronous Javascript

Garrett Welson

About me

- Graduate from **Hack Reactor**, a software engineering immersive program located at Galvanize in downtown ATX

- Work at Hack Reactor as a **Resident**, where I mentor junior engineers, teach concepts + best practices

- My first tech conference!

Goals

- Understand what asynchronous code is and why it works differently from synchronous code
    - The event loop
- Understand how to work with asynchronous code
    - Callbacks 😕 —> Promises 🙂 —> Async/Await 🤯
- Understand what's happening under the hood with these newer methods, and how to refactor old code

What is the event loop?

Javascript 101

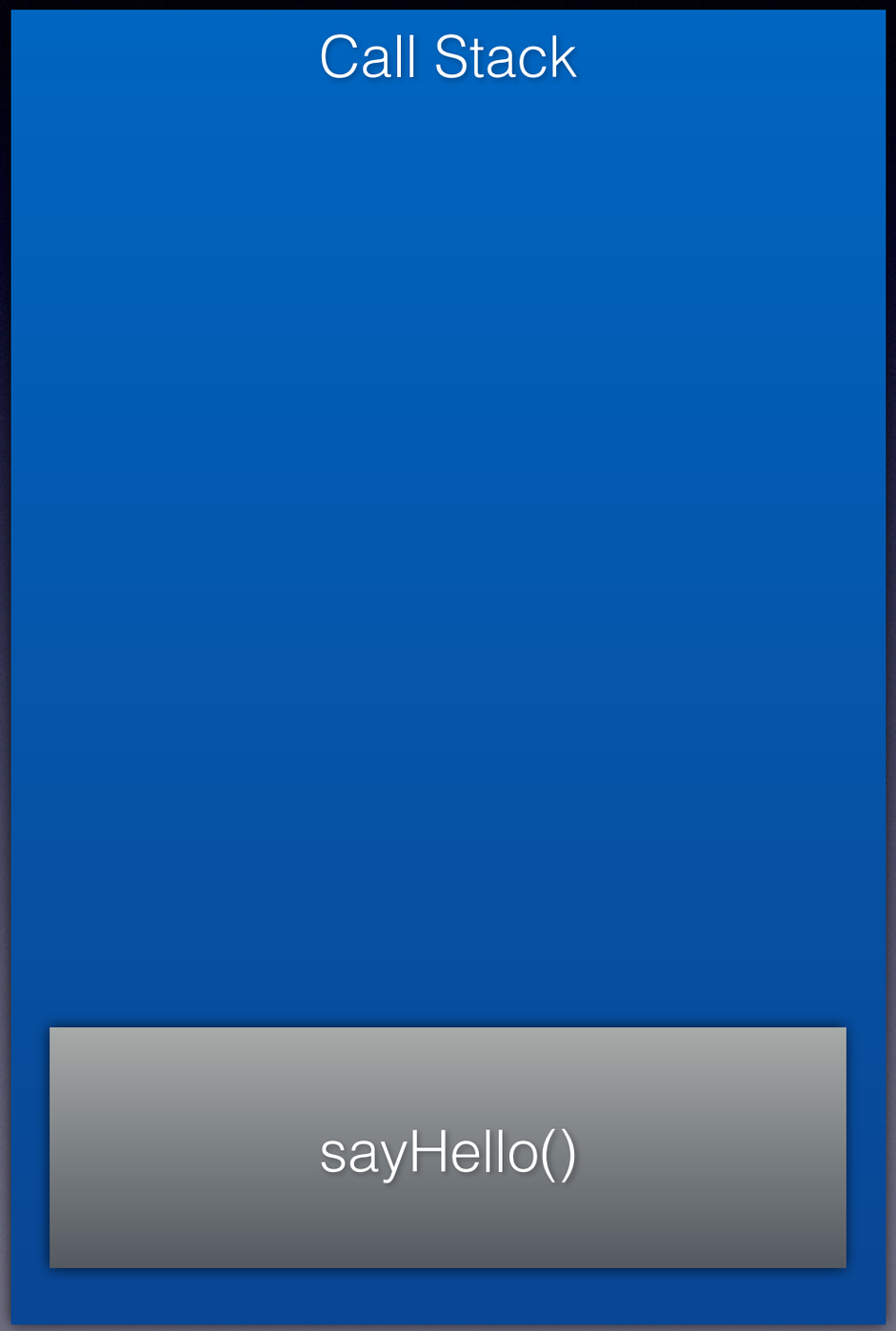
- Javascript is single-threaded
- Code is generally **synchronous**
- Javascript interprets each line **right away** before moving on to what's next
- Code is expected to be **non-blocking**

An early mistake...

```
let data = fetch('https://jsonplaceholder.typicode.com/posts')
console.log(data)
```

An early mistake...

```
let data = fetch('https://jsonplaceholder.typicode.com/posts')
console.log(data) // logs: Promise {<pending>}
```




```
const myName = "Garrett";

function sayHello(name) {
  console.log(`Hello, ${name}
`);
}

sayHello(myName);
```

Call Stack

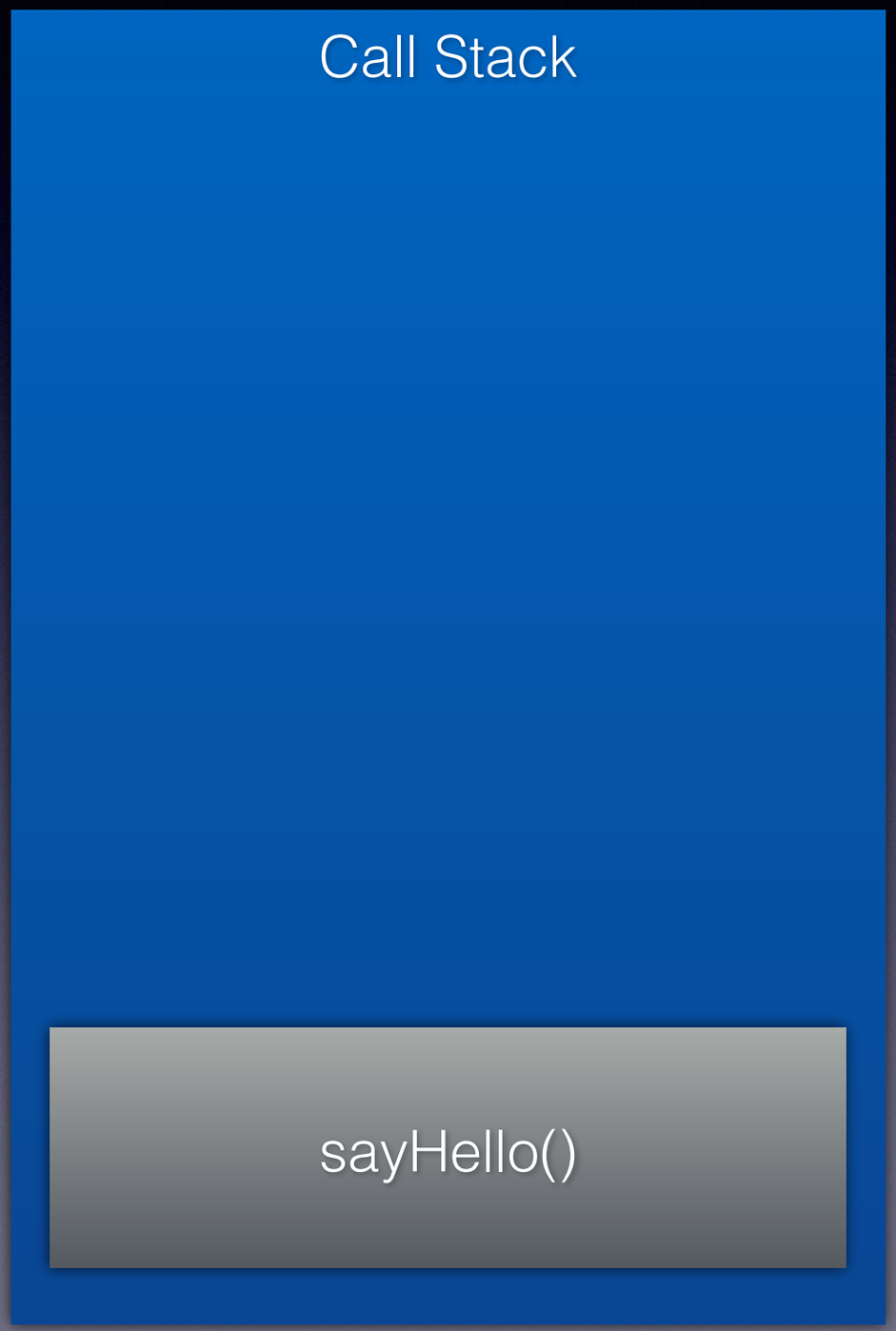
console.log(“Garrett”)

sayHello()

```
const myName = "Garrett";

function sayHello(name) {
  console.log(`Hello, ${name}
`);
}

sayHello(myName);
```




```
const myName = "Garrett";

function sayHello(name) {
  console.log(`Hello, ${name}
`);
}

sayHello(myName);
```

Call Stack

```
const myName = "Garrett";

function sayHello(name) {
  console.log(`Hello, ${name}
`);
}

sayHello(myName);
```

What about asynchronous code?

Some examples…

- Code that runs on a delay (setTimeout, setInterval, etc.) 🕐
- Code that requests data from a webpage/API (Ajax, Fetch, etc.) 🌐
- Code that requests data from a database or filesystem (typically run server-side) 🗂

Some callback-based code

```
function sayHello(name) {
  console.log(`Hello, ${name}`);
}

setTimeout(() => {
  sayHello("Garrett")
}, 2000)
```

Call Stack

setTimeout()

Browser
APIs

Callback Queue

Call Stack

Browser
APIs

setTimeout()

Callback Queue

Call Stack

Browser
APIs

Callback Queue

() => {}

Call Stack

() => {}

Browser
APIs

Callback Queue

Call Stack

sayHello()

() => {}

Browser
APIs

Callback Queue

Call Stack

console.log()

sayHello()

() => {}

Browser
APIs

Callback Queue

Call Stack

sayHello()

() => {}

Browser
APIs

Callback Queue

Call Stack

() => {}

Browser
APIs

Callback Queue

Call Stack

Browser
APIs

Callback Queue

A few choices

- Callbacks
- Promises
- Async/Await

Callbacks

Callbacks

- Just functions!
- Not any kind of special part of Javascript
- Function passed into another function to be run on the result

Synchronous Example

```
function sayHello(person, response) {
  console.log(`Hello, ${person}!`)
  response()
}

function sayHiBack() {
  console.log('Hi! Good to see you!')
}

sayHello('DevWeek', sayHiBack)
```

Synchronous Example

```
function sayHello(person, response) {
  console.log(`Hello, ${person}!`)
  response()
}

function sayHiBack() {
  console.log('Hi! Good to see you!')
}

sayHello('DevWeek', sayHiBack)

// 'Hello, DevWeek!'
// 'Hi! Good to see you!
```

Some common examples

- Javascript APIs like setTimeout, setInterval, etc.
- jQuery AJAX requests
- Node-style callbacks
    - Often in the form of (err, data) => { … }

The AJAX Way

```
$.ajax({
  url: 'https://jsonplaceholder.typicode.com/users',
  success: function(data) {
    console.log(data);
  },
  error: function(request, error) {
    console.log(error);
  },
  dataType: ‘json’
});
```

The AJAX Way


```
[
  {
    "id": 1,
    "name": "Leanne Graham",
    "username": "Bret",
    "email": "Sincere@april.biz",
    "address": {
      "street": "Kulas Light",
      "suite": "Apt. 556",
      "city": "Gwenborough",
      "zipcode": "92998-3874",
      "geo": {
        "lat": "-37.3159",
        "lng": "81.1496"
      }
  },…
]
```

Advantages

- Straightforward (at first)
- Many examples
- Comfortable

Disadvantages

- Debugging
- Readability
- Complexity

Callback Hell

```
$.ajax({
  url: 'https://jsonplaceholder.typicode.com/users',
  success: function(data) {
    const target = data[4];
    const id = target.id;
    $.ajax({
      url: `https://jsonplaceholder.typicode.com/posts?userId=${id}`,
      success: function(data) {
        console.log(data)
      },
      error: function(request, error) {
        console.log(error)
      },
      dataType: 'json'
    });
  },
  error: function(request, error) {
    console.log(error)
  },
  dataType: 'json'
});
```

Promises

What is a promise?

- “An object that represents the eventual completion or failure of an asynchronous operation”
- A much easier way to work with async code

Let’s look inside

```
> console.log(Promise.prototype)
                                                        VM225:1
  Promise {Symbol(Symbol.toStringTag): "Promise",
  constructor: ƒ, then: ƒ, catch: ƒ, finally: ƒ}
    i
    ▶ catch: ƒ catch()
    ▶ constructor: ƒ Promise()
    ▶ finally: ƒ finally()
    ▶ then: ƒ then()
      Symbol(Symbol.toStringTag): "Promise"
    ▶ __proto__: Object
```

The Promise Constructor

```
const newPromise = new Promise((resolve, reject) => {
  // asynchronous operation here
  if (error) {
    reject(error); // rejected
  }
  resolve(someValue); // fulfilled
});
```

Why are they useful?

- Quicker to write
- Easily “chainable”
- Better at dealing with complex situations

Callback Hell

```
$.ajax({
  url: 'https://jsonplaceholder.typicode.com/users',
  success: function(data) {
    const target = data[4];
    const id = target.id;
    $.ajax({
      url: `https://jsonplaceholder.typicode.com/posts?userId=${id}`,
      success: function(data) {
        console.log(data)
      },
      error: function(request, error) {
        console.log(error)
      },
      dataType: 'json'
    });
  },
  error: function(request, error) {
    console.log(error)
  },
  dataType: 'json'
});
```

With Promises...

```
fetch("https://jsonplaceholder.typicode.com/users")
.then(result => result.json())
.then(data => {
  let id = data[4].id;
  return fetch(`https://jsonplaceholder.typicode.com/
posts?userId=${id}`);
})
.then(result => result.json())
.then(posts => console.log(posts))
```

Building blocks

- .then()
- .catch()
- .finally()

Making a promise

- Native ES6 constructor
- Node's util.promisify()
- Third-party libraries like Bluebird

Example

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

```
wait(2000, "hello").then(result =>
{console.log(result)})
```

What about the event loop?

Not quite the same

- **Micro task** queue
- .then()/.catch() statements take priority over **macro-tasks** (like setTimeout or browser events) that move into the normal event queue

Example

```
console.log('start');

setTimeout(function() {
  console.log('end of timeout')
},0);

Promise.resolve("result of promise")
.then(value => console.log(value));
```

Example

```
console.log('start');

setTimeout(function() {
  console.log('end of timeout')
},0);

Promise.resolve("result of promise")
.then(value => console.log(value));

// "start"
// "result of promise"
// "end of timeout"
```

Why is this useful?

- Granular control over the order of async actions
- Control over the state of the DOM/events

A few additional methods

- Promise.all(…)
  - Array of results once all promises have resolved
- Promise.race(…)
  - Result of first promise to resolve (or reason if promise rejects)
- Promise.allSettled(…)
  - Array of statuses of all promises (not results)

Promise.all()

```
fetch("https://jsonplaceholder.typicode.com/users")
.then(result => result.json())
.then(data => {
  let id = data[4].id;
  return fetch(`https://jsonplaceholder.typicode.com/posts?userId=${id}`);
})
.then(result => result.json())
.then(posts => posts.map(post => fetch(`https://jsonplaceholder.typicode.com/
comments?postId=${post.id}`)))
.then(promiseArr => Promise.all(promiseArr))
.then(responses => Promise.all(responses.map(response => response.json())))
.then(results => console.log(results))
```

Async/Await

Background

- Introduced in ES8
- Similar to generators
- Allows async code to be written more like synchronous code
- **Syntactic sugar on top of promises**

Example

```
async function sayHello() {
  return "Hello!"
}
```

```
sayHello().then(response => console.log(response))
```

Example

```
async function sayHello() {
  return "Hello!"
}
```

```
sayHello().then(response => console.log(response))

// logs "Hello!"
```

Async Functions

- Must be declared with the **async** keyword
- Always return a promise
- Pause their execution when they hit the **await** keyword

Refactoring

```
async function getComments() {
  let users = await fetch("https://jsonplaceholder.typicode.com/users");
  users = await users.json();
  let id = users[4].id;
  let posts = await fetch(`https://jsonplaceholder.typicode.com/posts?userId=${id}`);
  posts = await posts.json();
  let promiseArr = posts.map(post => await fetch(  `https://jsonplaceholder.typicode.com/
comments?postId=${post.id}`));
  let comments = await Promise.all(promiseArr)
  comments = await Promise.all(comments.map(comment => comment.json()))
  console.log(comments)
}

getComments()
```

Even cleaner

```
async function getComments() {
  let users = await (await fetch("https://jsonplaceholder.typicode.com/users")).json();
  let id = users[4].id;
  let posts = await (await fetch(`https://jsonplaceholder.typicode.com/posts?userId=${id}`)).json()
  let comments = await Promise.all(
    posts.map(async (post) => await (await fetch( `https://jsonplaceholder.typicode.com/comments?postId=${post.id}`)).json())
  )
  console.log(comments)
}

getComments()
```

Pros/Cons

- Lets us write code that looks and feels more synchronous
- Allows us to easily use try/catch blocks inside of async functions
- Sequential nature of code can make successive async actions take longer
    - Can use Promise.all() to avoid this

Wrapping Up

Takeaways

- Understand the event loop and how asynchronous code is executed in Javascript
- Understand callbacks and their pros & cons
- Understand the pros & cons of promises and async/await
- Understand how to refactor old code to utilize newer methods

Thanks!