Asynchronous Programming

Review: JavaScript, Events, and Fetch

- JavaScript is a multi-paradigm language with prototypical objects, functional features, and imperative features
- **Anonymous functions** are extremely common in JavaScript
  - `let myDoubledList = [1, 2, 3].map( (x) => x*2 );`
- **Functional features** require a different mode of thought
  - `let myWeirdList = [1,2,3].filter((x)=>x > 0).forEach( (x)=>x+1 )` ???
  - `let myWeirdList = [1,2,3].filter((x)=>x > 0).    map( (x)=>x+1 )`
- The **document** API lets you modify the DOM manually
  - `let newp = document.createElement("p"); newp.setAttribute("id", "derp");`
  - `otherElement.appendChild(newp);`

Review: JavaScript Fetch

- The **fetch** function in JavaScript allows dynamically creating HTTP requests in the browser
  - let myResponse = fetch(“https://kjleach.eecs.umich.edu”);
- Fetch can support all sorts of methods and headers:

```
Let myResponse = fetch(“https://derp.com/api/p/1”,
  { method: ‘POST’,
    headers: {‘Content-Type’: ‘application/json’},
    body: JSON.stringify( {‘comment’: ‘Hello, world’} )
  }).then( someCoolResponseHandler ).then( someOtherChainedFunction )… ;
```

Review: Using Fetch

- Using **fetch** creates a **Promise** object
  - *Asynchronous event* – you don't want the browser to wait while you fetch something
  - Makes a *promise* that it will *fulfill* or *reject* the fetch later
    - (then it calls your 'then' chain)
- Consider: what happens if you fetch alone?

```
Let myResponse = fetch("https://derp.com/api/p/1",
  { method: 'POST',
    headers: {'Content-Type': 'application/json'},
    body: JSON.stringify( {'comment': 'Hello, world'} )
  });
```

One-Slide Summary: Closures, Asynchronous Programming, and React

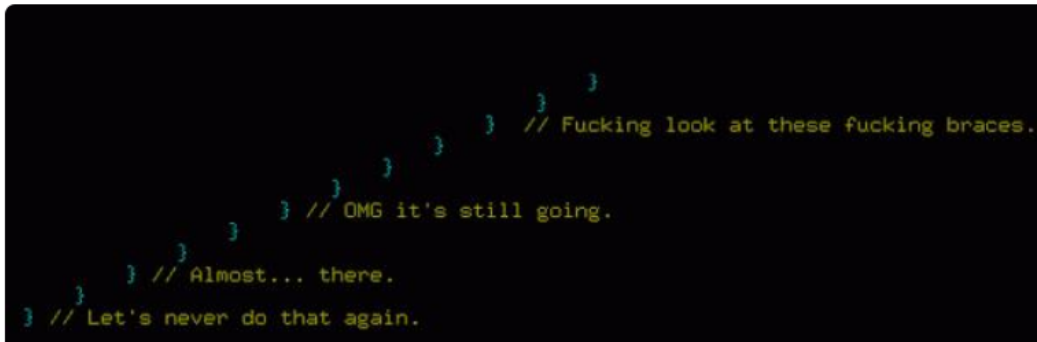
- A **closure** is a JavaScript feature that allows passing along *lexical bindings* even after the *end of a function's lifetime*.
  - Closures are critical in event-driven programming
  - Consider: if you **define function x() inside function y()**, *how do you have access to x() after y() finishes executing?* (Hint: it's closures)
- **Asynchronous Programming** is the common pattern for writing JavaScript-driven web services
  - **Promise** objects are created indicating a "promise to run something later"
    - When a Promise is made, an event handler is setup to fulfill or reject the Promise later
- **React** is a JavaScript framework for creating front-ends
  - You build reusable **components** that contain DOM information (HTML) and state (passed to and from a REST API)
    - e.g., a "Comment" component could have "{ 'author': 'kevin', 'content': 'hello' }" as its state

A problem with raw JavaScript

- Large JavaScript applications quickly become unwieldly
- All functions act on the DOM, DOM acts like a giant global variable
- Difficult to decompose program into abstractions

```
function showUser() {
  fetch()
  .then(function(data) {
      const users = data.results;
      users.forEach((user) => {
        const node = document.createElement('p');
        //...
        node.appendChild(textnode);
        entry.appendChild(node);
      });
    })
//...
```

All JavaScript developers will understand :|



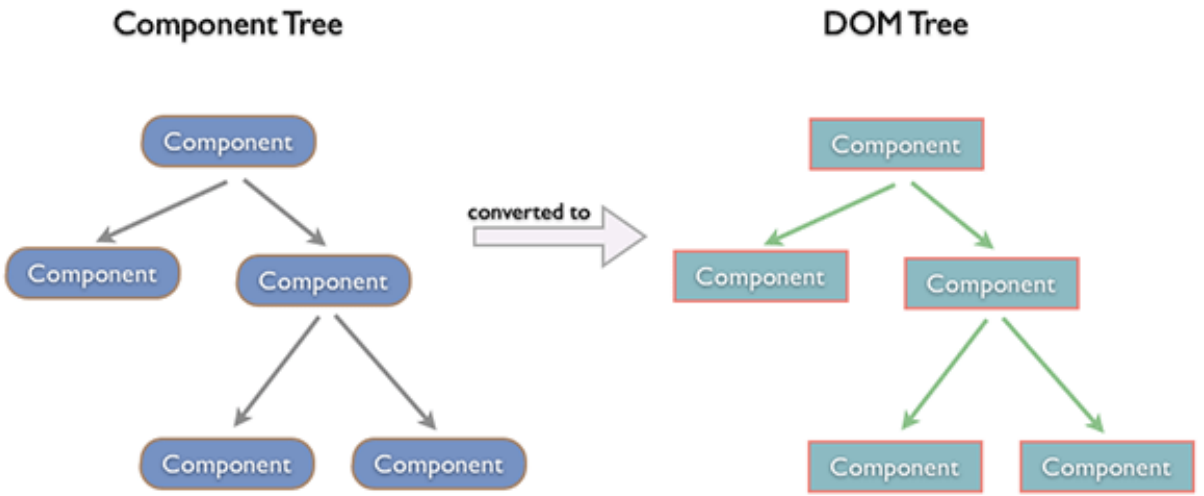
React

- React is a framework built by Facebook
- Build encapsulated components that manage their own state
  - Compose them to make complex UIs
- Efficient updates to the DOM

- https://reactjs.org/

React

- Components
  - Functional: usually stateless
  - Class-type: usually stateful
- Tree of composable components -> DOM



Virtual DOM

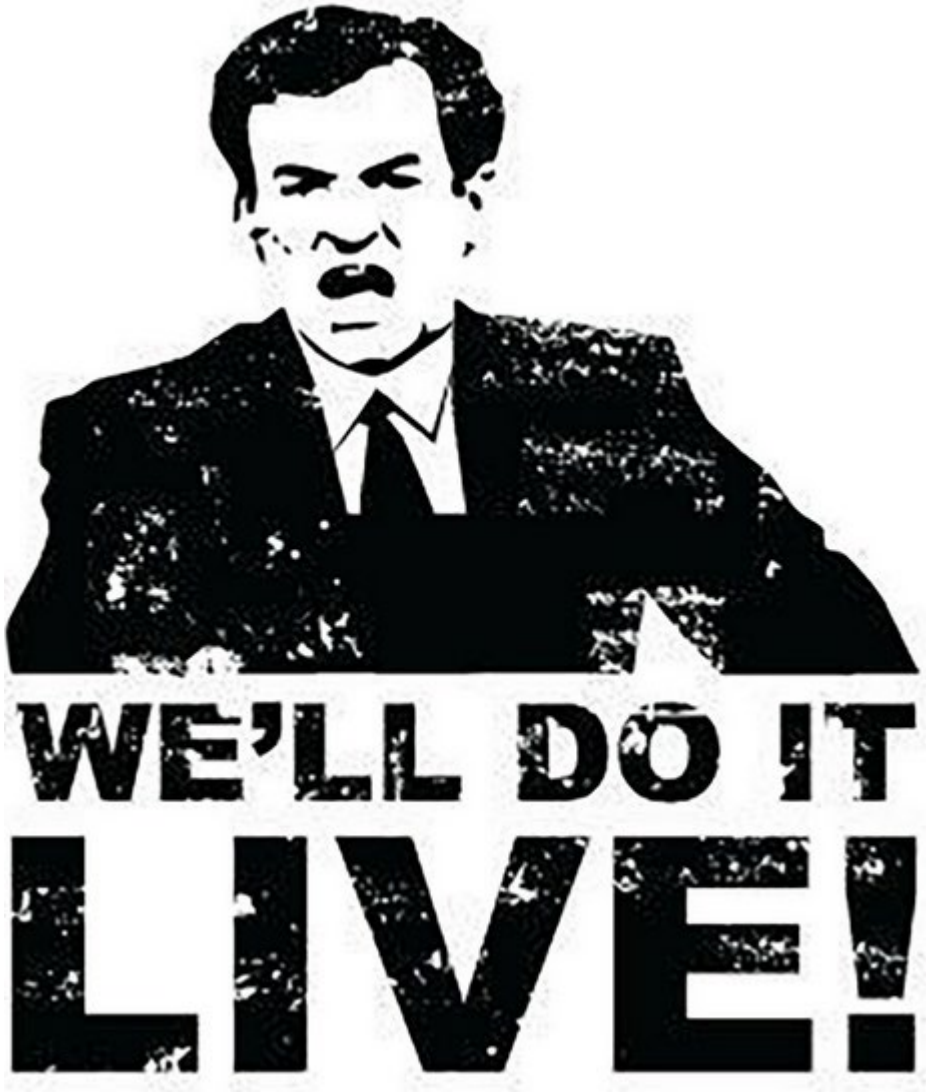
- Components rendering cause other components to render
- This would cause lots of DOM updates, which are slow
  - Because the actual screen changes
- Solution: a Virtual DOM

- Periodically reconcile virtual DOM with real DOM
  - Avoids unnecessary changes

- For lots of details: https://reactjs.org/docs/reconciliation.html

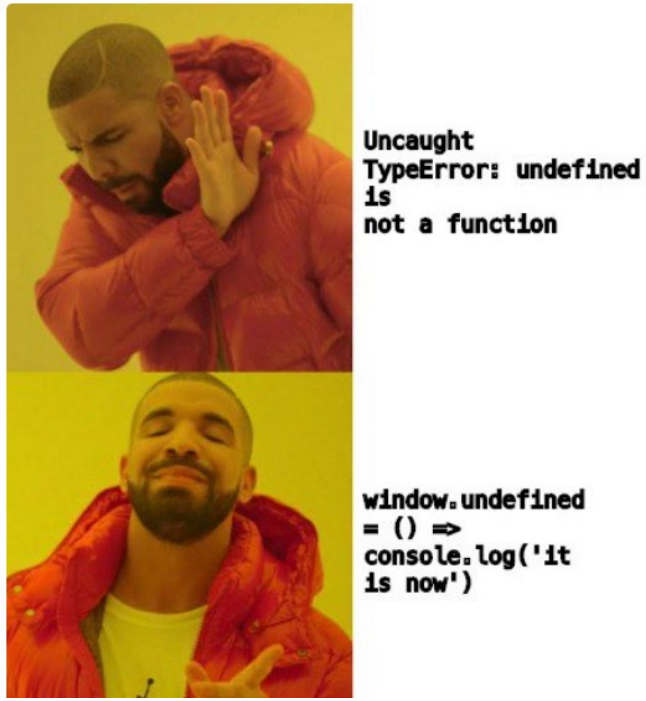
React documentation

- Required reading for project 3: https://reactjs.org/docs/hello-world.html
- Live example https://codepen.io/awdeorio/pen/yzXjzZ?editors=1010
- Let's take a look at React



Closures

```
function outer() {
    let x = 0;
    function inner() {
        x++;
        console.log(x);
    }
    return inner;
}

let f = outer();
f();
f();
```



Notice: *f* is a function pointer to “inner” (the thing returned by “outer”)

Closures

- A **Closure** is the combination of a function and its *lexical environment*
  - All the variables (local, global, outer scopes, etc.)
  - Every function invocation creates a corresponding Closure

(below) note that “handleData” has access to “entry”

even though showUsers() will have finished executing

```
function showUsers() {
  const entry = document.getElementById('JSEntry');
  //...
  function handleData(data) {
    //...
    entry.appendChild(node);
  }
  fetch(/* ... */) // ...
}
```

Closures

```
function showUsers() {
  const entry = document.getElementById('JSEntry');

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

  function handleData(data) {
    // ...
    entry.appendChild(node);
  }

  fetch(/*...*/)
    .then(handleResponse)
    .then(handleData)
    .catch(error => console.log(error));
}
```

- The inner function has a longer lifetime than the outer function
- `handleData()` has access to `entry` even though `showUsers()` has already returned!

Closures in the interpreter

```
function showUsers() {
  const entry = /*...*/;

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

  function handleData(data) {
    // ...
    entry.appendChild(node);
  }

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

1. Objects created for `entry`, `handleResponse`, `handleData`
2. `fetch` function executes
   1. Enqueue callbacks `handleResponse`, `handleData`
   2. `fetch` returns before response arrives
3. .... wait for response
4. Later, response arrives and `handleResponse` executes
5. Later, JSON data is ready and `handleData` executes

Why closures important in web development

- Callback Functions are everywhere
  - Event handlers: click
  - Promise handlers: .then()

- These functions need to remember their context
  - Remember that execution keeps going past a fetch() call

- See https://developer.mozilla.org/en-US/docs/Web/JavaScript/Closures for more examples

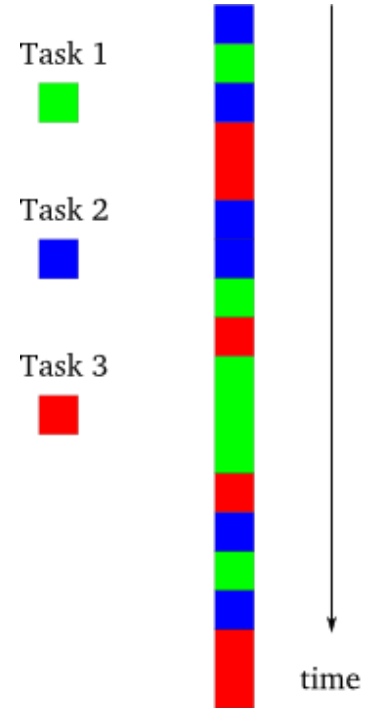
Asynchronous Programming

- Synchronous
  - sync
  - Happens right away
  - Example: function call
  - C++ programs in 183, 280, 281

- Asynchronous
  - async
  - "doesn't run right away"
  - Example: event-driven programs, fetch API
  - Promises

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 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

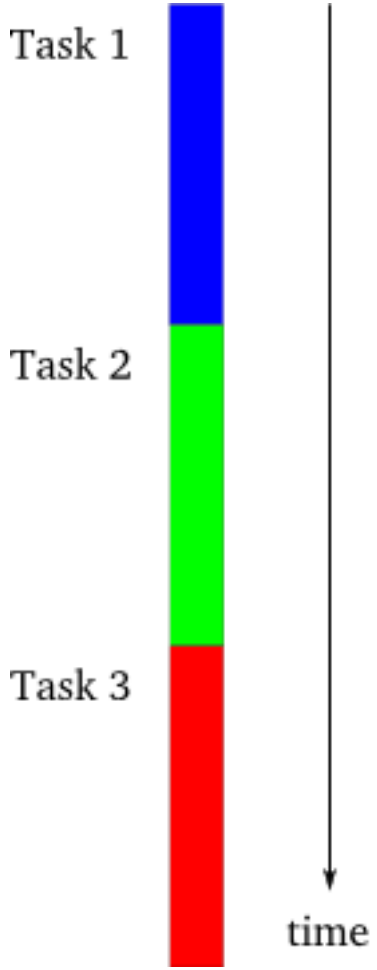


Image credit: http://krondo.com/

Asynchronous is not ...

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

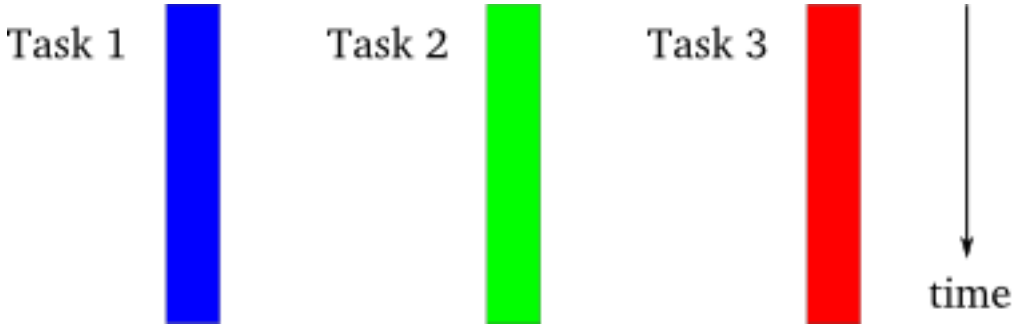


Image credit: http://krondo.com/

Asynchronous is ...

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

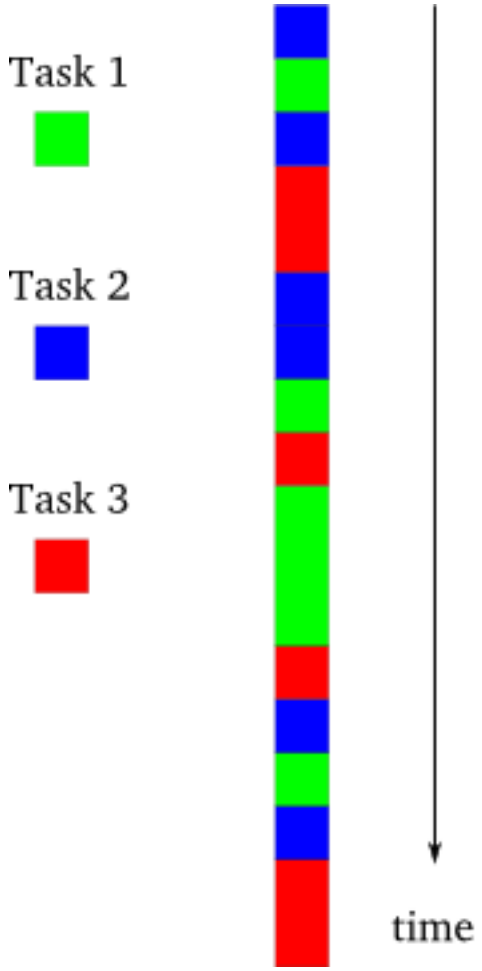


Image credit: http://krondo.com/

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

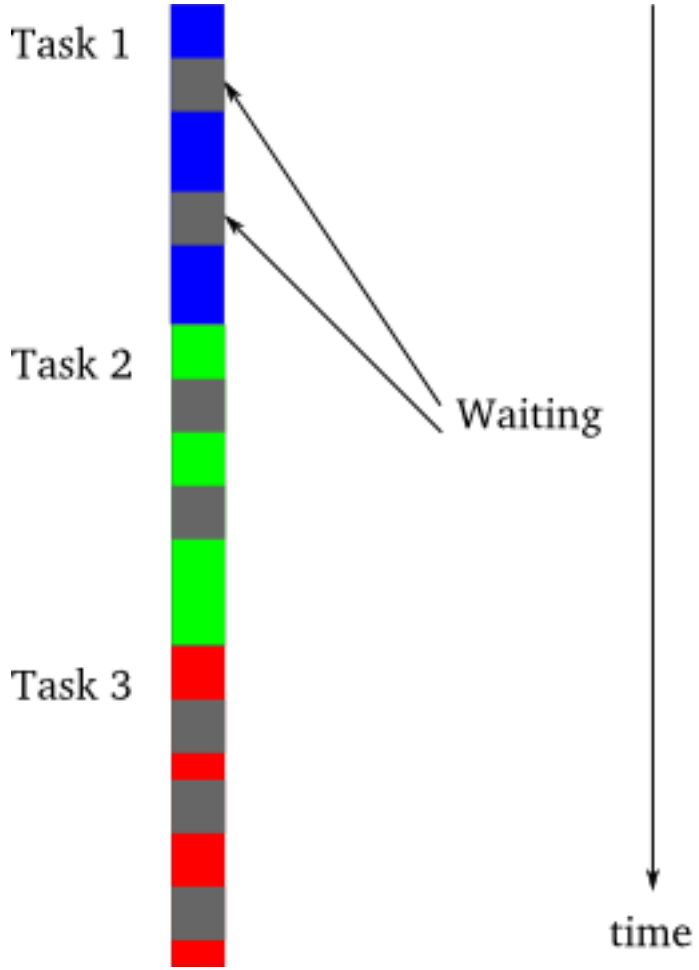


Image credit: http://krondo.com/

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)

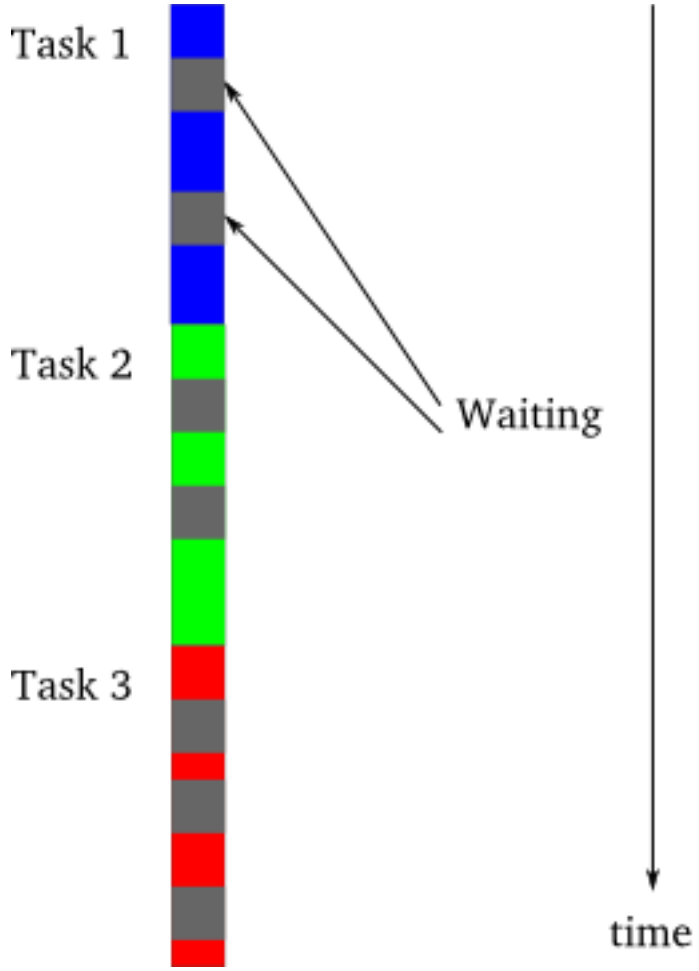


Image credit: http://krondo.com/

Olde: Asynchronous JavaScript And XML

- AJAX was an early paradigm for enabling client-side dynamic pages
- *Asynchronously* use *JavaScript* to communicate *XML* documents
  - We use JSON today

```
<script>
function loadDoc() {
  var xhttp = new XMLHttpRequest();
  xhttp.onreadystatechange = function() {
    if (this.readyState == 4 && this.status == 200) {
      document.getElementById("demo").innerHTML =
      this.responseText;
    }
  };
  xhttp.open("GET", "ajax_info.txt", true);
  xhttp.send();
}
</script>
```

Async example: fetch

```
fetch("/api/v1/u/hello")
    .then(handleResponse)
    .then(handleData);
```

Calling fetch makes a Promise object.

Calling “then” on a Promise schedules an event to call “handleResponse” later

(when the Promise is fulfilled)

Considerations for asynchronous

- Large number of tasks
  - Regularly at least one task that can make progress
- I/O-bound tasks
  - Synchronous would be wasteful (remember the UI freezing?)
  - Asynchronous means other work can be completed while waiting for I/O
- Embarrassingly parallel
  - Independent tasks means no need for synchronization or communication
  - (no need to block on other tasks)

- These conditions are common in web systems!

Promises

- Promise represents a value that is not available right away
- Creating a Promise sets up a pipeline for what will happen when the value is available
- Real-life example
  - Your friend promises to give you $1 in the future
  - You can start to plan what to do with it (buy a coffee?)
  - But you can't actually do it yet
    - Will they fulfill their promise?

Promise picture - fetch

```
fetch("/api/v1/u/jklooste")
    .then(handleResponse)
    .then(handleData);
```

Promises

- 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

Creating Promises

```
let p = new Promise((resolve, reject) => {

   // do some asynchronous work
   // in "pending" state

   // call reject if there's an error
   if (error happens) {
      // enter "rejected" state
      reject("Error");
   }

   // call resolve when promise complete
   // enter "fulfilled" state
   resolve("All finished");

});
```

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)
}
```

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

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);
    })
}
```

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()`

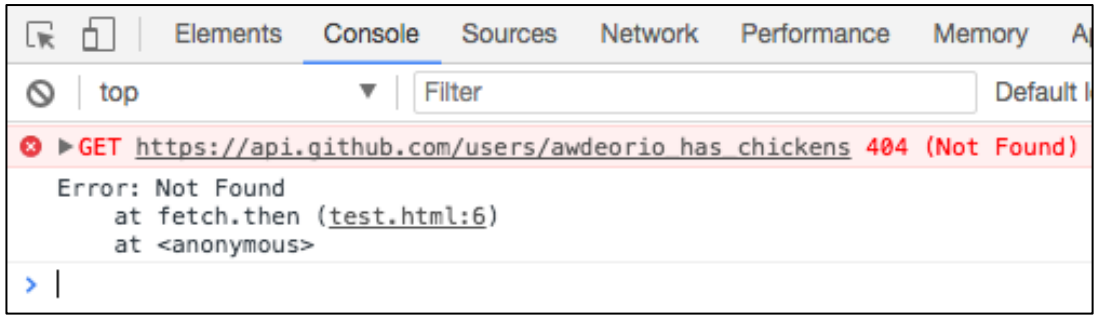
```
function showUser() {
  fetch('https://api.github.com/users/awdeorio')
    .then((response) => {
      if (!response.ok) throw Error(response.statusText);
      return response.json();
    })
    .then((data) => {
      console.log(data);
    })
    .catch(error => console.log(error))
}
```

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



- 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))
}
```

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
```