# CS193X:
# Web Programming Fundamentals

Spring 2017

Victoria Kirst
(vrk@stanford.edu)

# Schedule

**Today:**

- Functional JavaScript
    - Anonymous functions
    - Currying
    - Closures
- Loading data from files
    - Fetch API
    - Promises - High-level!
    - JSON


- **HW3 due tonight**
- **HW4 out tonight** (will be posted end of day)

# Prereq: Command line

We will start using the command line in class today.

**We will not be teaching how to use a command line interface.** This was a prerequisite for the class through CS1U.

Please make sure you know how to:
- Navigate between directories in a command line
- Open / edit files via command-line
- Execute scripts via command-line

# Practical Functional JavaScript

# Example: `findIndex`

*list*.findIndex(***callback, thisArg***):
    Returns the index of an element.

   ***callback*** is a function with the following parameters:
   - **element:** The current element being processed.
   - **index:** The index of the current element being processed in the array.
   - **array**: the array `findIndex` was called upon.

***callback*** is called for every element in the array, and returns true if found, false otherwise.
***thisArg*** is the value of `this` in ***callback***

# findIndex

```
const flavors =
    ['vanilla', 'chocolate', 'strawberry', 'green tea'];
```

Q: How can we use `findIndex` to see whether or not `'strawberry'` is in the `flavors` list?

# findIndex

```
const flavors =
    ['vanilla', 'chocolate', 'strawberry', 'green tea'];
```

1. Define a **testing function**, to be called on each element in the list. (Returns `true` if it passes the test.)

```
function isStrawberry(element) {
  return element === 'strawberry';
}
```

The **testing function** can take `element`, `index`, and `array` as parameters, but we are only using `element`.

# findIndex

```
const flavors =
    ['vanilla', 'chocolate', 'strawberry', 'green tea'];
```

2. Pass the testing function into `findIndex`.

```
function isStrawberry(element) {
  return element === 'strawberry';
}

const indexOfStrawberry = flavors.findIndex(isStrawberry);
```

# findIndex

```
const flavors =
    ['vanilla', 'chocolate', 'strawberry', 'green tea'];


function isStrawberry(element) {
  return element === 'strawberry';
}

const indexOfStrawberry = flavors.findIndex(isStrawberry);
```

The **isStrawberry** function will fire for each element in the array.

# findIndex

```javascript
const flavors =
    ['vanilla', 'chocolate', 'strawberry', 'green tea'];


function isStrawberry(element) {
  return element === 'strawberry';
}

const indexOfStrawberry = flavors.findIndex(isStrawberry);
```

# findIndex

```
const flavors =
    ['vanilla', 'chocolate', 'strawberry', 'green tea'];


function isStrawberry(element) {
  return element === 'strawberry';
}

const indexOfStrawberry = flavors.findIndex(isStrawberry);
```

Returns **false**, so keep searching.

# findIndex

```
const flavors =
    ['vanilla', 'chocolate', 'strawberry', 'green tea'];


function isStrawberry(element) {
  return element === 'strawberry';
}

const indexOfStrawberry = flavors.findIndex(isStrawberry);
```

# findIndex

```
const flavors =
    ['vanilla', 'chocolate', 'strawberry', 'green tea'];


function isStrawberry(element) {
  return element === 'strawberry';
}

const indexOfStrawberry = flavors.findIndex(isStrawberry);
```

Returns **false**, so keep searching.

# findIndex

```
const flavors =
    ['vanilla', 'chocolate', 'strawberry', 'green tea'];


function isStrawberry(element) {
  return element === 'strawberry';
}

const indexOfStrawberry = flavors.findIndex(isStrawberry);
```

# findIndex

```
const flavors =
    ['vanilla', 'chocolate', 'strawberry', 'green tea'];


function isStrawberry(element) {
  return element === 'strawberry';
}

const indexOfStrawberry = flavors.findIndex(isStrawberry);
```

Returns **true**, so stop searching.

# findIndex

```
const flavors =
    ['vanilla', 'chocolate', 'strawberry', 'green tea'];


function isStrawberry(element) {
  return element === 'strawberry';
}

const indexOfStrawberry = flavors.findIndex(isStrawberry);
```

findIndex returns 2, since the first element to pass the **testing function** was found at index 2. ([CodePen](#))

# findIndex

```
const flavors =
    ['vanilla', 'chocolate', 'strawberry', 'green tea'];
                                  ⬆

function isStrawberry(element) {
  return element === 'strawberry';
}

const indexOfStrawberry = flavors.findIndex(isStrawberry);
```

## Let's clean this up a little bit...

# Anonymous functions

# Anonymous functions

We do not need to give an identifier to functions.

When we define a function without an identifier, we call it an **anonymous function**

- Also known as a **function literal**, or a **lambda function**

**We can define our test function directly in `findIndex`:**

```
function isStrawberry(element) {
  return element === 'strawberry';
}

const index = flavors.findIndex(isStrawberry);
```

# Anonymous functions

We do not need to give an identifier to functions.
When we define a function without an identifier, we call it
an **anonymous function**

- Also known as a **function literal**, or a **lambda function**

**We can define our test function directly in `findIndex`:**

```
const index = flavors.findIndex(
  function(element) { return element === 'strawberry'; });
```

# Arrow functions

We can use the **arrow function** syntax for defining functions:

```
const index = flavors.findIndex(
  function(element) { return element === 'strawberry'; });
```

# Arrow functions

We can use the **arrow function** syntax for defining functions:

```
const index = flavors.findIndex(
  (element) => { return element === 'strawberry'; });
```

# Concise arrow functions

We can use the **concise version** of the [arrow function](arrow function):

- You can omit the parentheses if there is only one parameter
- You can omit the curly braces if there's only one statement in the function, and it's a return statement

```
const index = flavors.findIndex(
  (element) => { return element === 'strawberry'; });
```

# Concise arrow functions

We can use the **concise version** of the [arrow function](arrow function):
- You can omit the parentheses if there is only one parameter
- You can omit the curly braces if there's only one statement in the function, and it's a return statement

```
const index = flavors.findIndex(
    element => element === 'strawberry');
```

# Case-insensitive search

If we wanted to make this case insensitive, we could do:

```
const index = flavors.findIndex(
    element => element.toLowerCase() === 'strawberry');
```

# Case-insensitive search

If we wanted to make this case insensitive, we could do:

```
const index = flavors.findIndex(
    element => element.toLowerCase() === 'strawberry');
```

This is a lot more elegant than the for-loop approach!

```
for (let i = 0; i < flavors.length; i++) {
  if (flavors[i].toLowerCase() === 'strawberry') {
    break;
  }
}
const index = i;
```

# Currying

# isFlavor

What if instead of checking specifically for `strawberry`...

```
function isStrawberry(element) {
  return element === 'strawberry';
}
```

# isFlavor

...we wanted to create a generic `isFlavor` checker?

```
function isFlavor(flavor, element) {
  return element === flavor;
}
```

# isFlavor

```
const flavors =
    ['vanilla', 'chocolate', 'strawberry', 'green tea'];


function isFlavor(element) {
  // ERROR: flavor is undefined!
  return element === flavor;
}

const indexOfFlavor = flavors.findIndex(isFlavor);
```

The problem is there's no way to pass in the `flavor` parameter in the callback for `findIndex`...

# Currying

```
const flavors =
    ['vanilla', 'chocolate', 'strawberry', 'green tea'];

function createFlavorTest(flavor) {
  function isFlavor(element) {
    return element === flavor;
  }
  return isFlavor;
}
const isStrawberry = createFlavorTest('strawberry');
const indexOfFlavor = flavors.findIndex(isStrawberry);
```

**Solution:** Create a function that takes a `flavor` parameter and creates a testing function for that parameter. ([CodePen](#))

# *Aside*: closure

```
const flavors =
    ['vanilla', 'chocolate', 'strawberry', 'green tea'];

function createFlavorTest(flavor) {
  function isFlavor(element) {
    return element === flavor;
  }
  return isFlavor;
}
const isStrawberry = createFlavorTest('strawberry');
const indexOfFlavor = flavors.findIndex(isStrawberry);
```

*Aside*: Any function that is declared within another function is called a **closure**. Closures can refer to variables in the outer function (`flavor` in this case).

# Currying

```
function isFlavor(flavor, element) {
  return element === flavor;
}
```



```
function createFlavorTest(flavor) {
  function isFlavor(element) {
    return element === flavor;
  }
  return isFlavor;
}
```



```
flavors.findIndex(isFlavor);
```

This idea is called **currying**: breaking down a function with multiple arguments by applying one at a time in a sequence of created functions.

# Review: ES6 classes

- ES6 classes mostly work the way you expect

- **`this` in a constructor:** refers to the new object being created
- **`this` outside a constructor:** refers to a different value depending on how the function is called
  - In response to a **DOM event**, `this` is the element that the event handler was tied to
  - When called in a **method**, `this` is the object that the method is called from
- **`bind`:** sets the value of `this` for a function so it does not change depending on the context

# Review: Functional JavaScript

- Functions in JavaScript are **first-class citizens**:
    - Objects that can be passed as parameters
    - Can be created within functions:
        - Inner functions are called **closures**
    - Can be created without being saved to a variable
        - These are called **anonymous functions**, or function literals, or lambdas
    - Can be created and returned from functions
        - Constructing a new function that references part of the outer function's parameters is called **currying**

# More Array functions

| Function name | Description |
|---|---|
| *list*.forEach(*function*) | Executes the provided function once for each array element. ([mdn](#)) |
| *list*.filter(*function*) | Creates a new array with all elements that pass the test implemented by the provided function. ([mdn](#)) |
| *list*.every(*function*) | Tests whether all elements in the array pass the test implemented by the provided function. ([mdn](#)) |

[All Array functions](#)

# Loading data from files

# Loading data from a file

What if you had a list of images in a text file that you wanted to load in your web page?

```
1   https://media1.giphy.com/media/xNT2CcLjhbI0U/200.gif
2   https://media2.giphy.com/media/3o7btM3VVVNtssGReo/200.gif
3   https://media1.giphy.com/media/l3q2uxEzLIE8cWMq4/200.gif
4   https://media2.giphy.com/media/LDwL3ao61wfHa/200.gif
5   https://media1.giphy.com/media/3o7TKMt1VVNkHV2PaE/200.gif
6   https://media3.giphy.com/media/DNQFjMJbbsNmU/200.gif
7   https://media1.giphy.com/media/26FKTsKMKtUSomuNq/200.gif
8   https://media1.giphy.com/media/xThuW5Hf2N8idJHFVS/200.gif
9   https://media1.giphy.com/media/XlFfSD0CiyGLC/200.gif
10  https://media3.giphy.com/media/ZaBHSbiLQTmFi/200.gif
11  https://media3.giphy.com/media/JPbZwjMcxJYic/200.gif
12  https://media1.giphy.com/media/FArgGzk7KO14k/200.gif
13  https://media1.giphy.com/media/UFoLN1EyKjLbi/200.gif
14  https://media1.giphy.com/media/11zXBCAb9soCQM/200.gif
15  https://media4.giphy.com/media/xUPGcHeIeZMmTcDQJy/200.gif
16  https://media2.giphy.com/media/apZwWJInOBvos/200.gif
17  https://media2.giphy.com/media/sB4nvt5xIiNig/200.gif
18  https://media0.giphy.com/media/Y8Bi9lCOzXRkY/200.gif
19  https://media1.giphy.com/media/12wUXjm6f8Hhcc/200.gif
20  https://media4.giphy.com/media/26gsuVyk5fKB1YAAE/200.gif
21  https://media3.giphy.com/media/l2SpMU9sWIvT2nrCo/200.gif
22  https://media2.giphy.com/media/kR1vWazNc7972/200.gif
23  https://media4.giphy.com/media/Tv3m2GAAl2Re8/200.gif
24  https://media2.giphy.com/media/9nujydsBLz2dq/200.gif
25  https://media3.giphy.com/media/AG39l0rHgkRLa/200.gif
```

# Intuition: `loadFromFile`

If we wanted to have an API to load external files in JavaScript, it might look something like this:

```
// FAKE HYPOTHETICAL API.
// This is not real a JavaScript function!
const contents = loadFromFile('images.txt');
```

# Intuition: `loadFromFile`

```
// FAKE HYPOTHETICAL API.
// This is not real a JavaScript function!
const contents = loadFromFile('images.txt');
```

A few problems with this hypothetical fake API:

- We want to load the file **asynchronously**: the JavaScript should not block while we're loading the file

- There's no way to check the status of the request. What if the resource didn't exist? What if we're not allowed to access the resource?

# Intuition: `loadFromFile`

An asynchronous version of this API might look like this:

```
// FAKE HYPOTHETICAL API.
// This is not real a JavaScript function!
function onSuccess(response) {
  const body = response.text;
  ...
}
loadFromFile('images.txt', onSuccess, onFail);
```

Where `onSuccess` and `onFail` are callback functions that should fire if the request succeeded or failed, respectively.

# Fetch API

# Fetch API: `fetch()`

The [Fetch API](#) is the API to use to load external resources (text, JSON, etc) in the browser.

The Fetch API is made up of one function, and its syntax is is concise and easy to use:

```
fetch('images.txt');
```

**Note:** [XMLHttpRequest](#) ("XHR") is the old API for loading resources from the browser. XHR still works, but is clunky and harder to use.

# Fetch API: `fetch()`

The [Fetch API](#) is the API to use to load external resources (text, JSON, etc) in the browser.

The Fetch API is made up of one function, and its syntax is is concise and easy to use:

## `fetch('images.txt');`

- The `fetch()` method takes the string path to the resource you want to fetch as a parameter
- It returns a `Promise`

# Fetch API: `fetch()`

The [Fetch API](#) is the API to use to load external resources (text, JSON, etc) in the browser.

The Fetch API is made up of one function, and its syntax is is concise and easy to use:

```
fetch('images.txt');
```

- The `fetch()` method takes the string path to the resource you want to fetch as a parameter
- It returns a `Promise`
  - **What the heck is a `Promise`?**

# Promises:
# Another conceptual odyssey

# Promises and .then()

A Promise:

- An object used to manage asynchronous results
- Has a then() method that lets you attach functions to execute onSuccess or onError
- Allows you to build **chains** of asynchronous results.

Promises are easier to use than to define…

# Simple example: getUserMedia

There is an API called `getUserMedia` that allows you get the media stream from your webcam.

There are two versions of getUserMedia:
- `navigator.`[`getUserMedia`](#) **(deprecated)**
  - Uses callbacks

- `navigator.mediaDevices.`[`getUserMedia`](#)
  - Returns a `Promise`

# getUserMedia with callbacks

```javascript
const video = document.querySelector('video');

function onCameraOpen(stream) {
  video.srcObject = stream;
}
function onError(error) {
  console.log(error);
}

navigator.getUserMedia({ video: true },
  onCameraOpen, onError);
```

[CodePen](#)

# getUserMedia with Promise

```javascript
const video = document.querySelector('video');

function onCameraOpen(stream) {
  video.srcObject = stream;
}
function onError(error) {
  console.log(error);
}

navigator.mediaDevices.getUserMedia({ video: true })
  .then(onCameraOpen, onError);
```

[CodePen](CodePen)

# Hypothetical Fetch API

```
// FAKE HYPOTHETICAL API.
// This is not how fetch is called!
function onSuccess(response) {
  ...
}
function onFail(response) {
  ...
}
fetch('images.txt', onSuccess, onFail);
```

# Real Fetch API

```
function onSuccess(response) {
  ...
}
function onFail(response) {
  ...
}
fetch('images.txt').then(onSuccess, onFail);
```

# Promise syntax

**Q: How does this syntax work?**

```
fetch('images.txt').then(onSuccess, onFail);
```

# Promise syntax

**Q: How does this syntax work?**

```
fetch('images.txt').then(onSuccess, onFail);
```

**The syntax above is the same as:**

```
const promise = fetch('images.txt');
promise.then(onSuccess, onFail);
```

# Promise syntax

```
const promise = fetch('images.txt');
promise.then(onSuccess, onFail);
```

The object `fetch` returns is of type [Promise](#).

A promise is in one of three states:
- **pending**: initial state, not fulfilled or rejected.
- **fulfilled**: the operation completed successfully.
- **rejected**: the operation failed.

You attach handlers to the promise via `.then()`

# Promise syntax

```
const promise = fetch('images.txt');
promise.t
```

The obje

A promis
- pendi
- fulfill
- reject

(We'll think about this more
deeply in a later lecture.)

(Right now we will just use
Promises.)

You attach handlers to the promise via .then()

# Using Fetch

```
function onSuccess(response) {
  console.log(response.status);
}
fetch('images.txt').then(onSuccess);
```

The success function for Fetch gets a response parameter:
- `response.status`: Contains the status code for the request, e.g. 200 for HTTP success
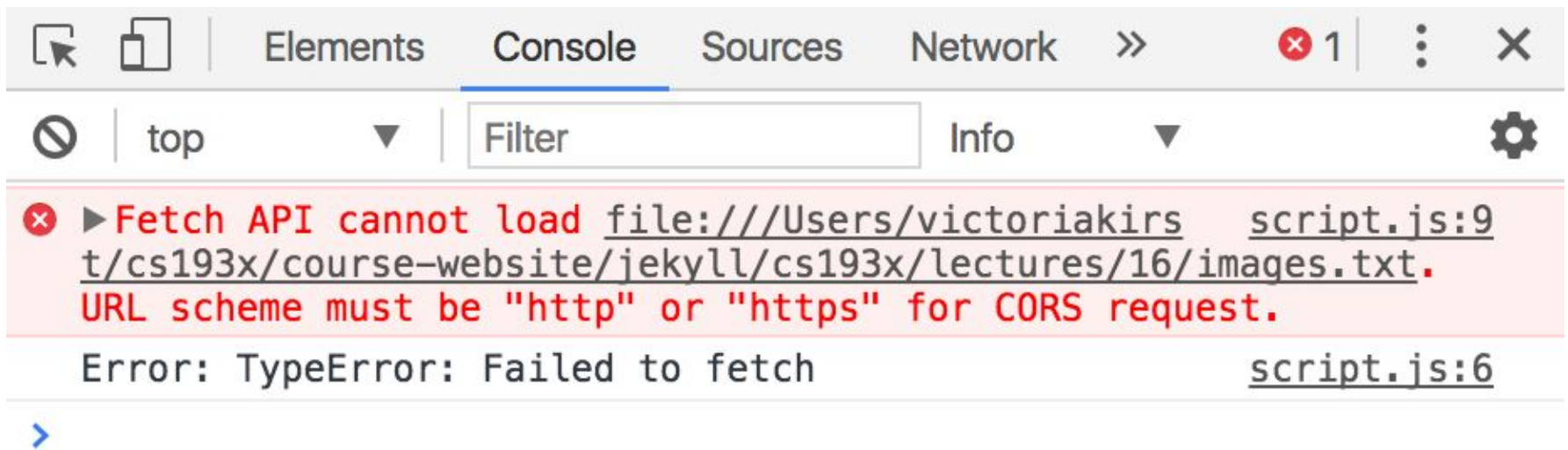  - HTTP status codes

# Fetch attempt

```javascript
function onSuccess(response) {
  console.log(response.status);
}

function onError(error) {
  console.log('Error: ' + error);
}

fetch('images.txt')
    .then(onSuccess, onError);
```

# Fetch error

If we try to load this in the browser, we get the following JavaScript error:



Notice that our onError function was also called.

# Local files

When we load a web page in the browser that is saved on our computer, it is served via `file://` protocol:



We are **not allowed** to load files in JavaScript
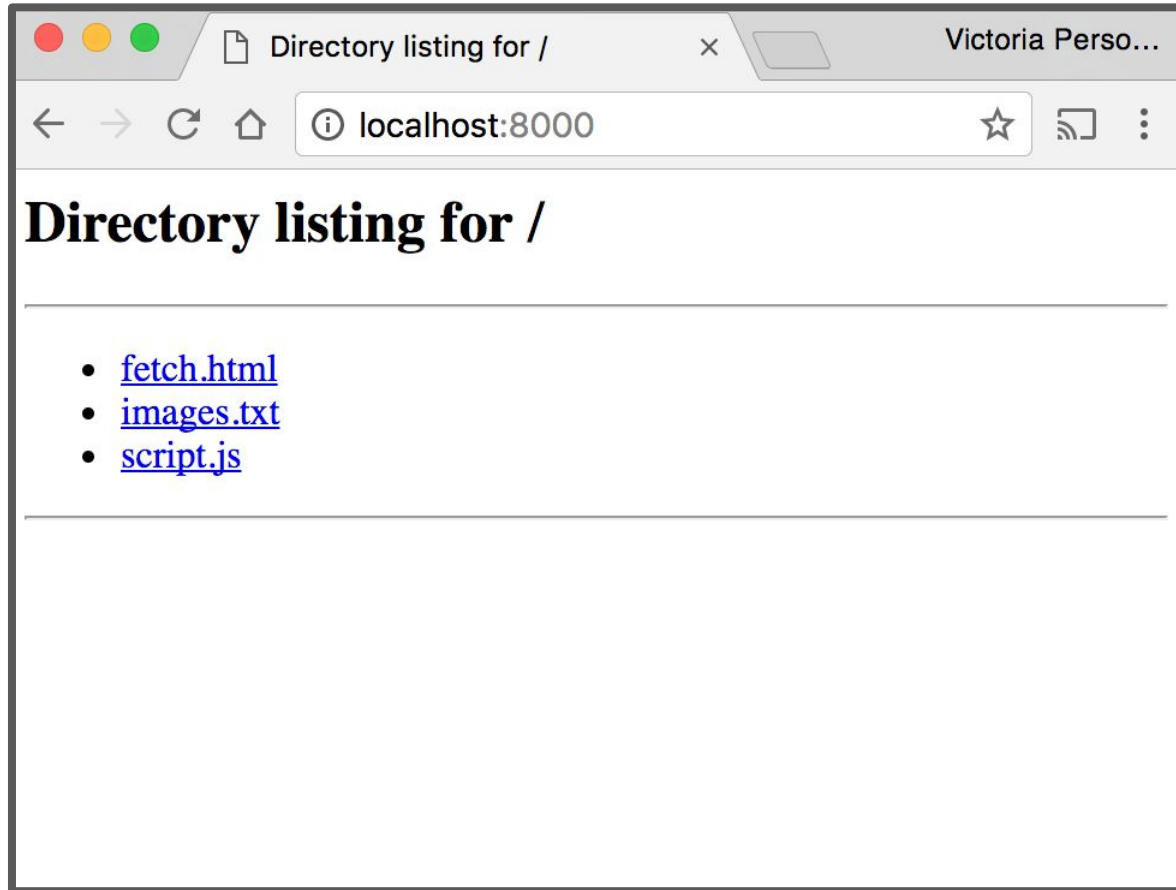from the `file://` protocol, which is why we got the error.

# Serve over HTTP

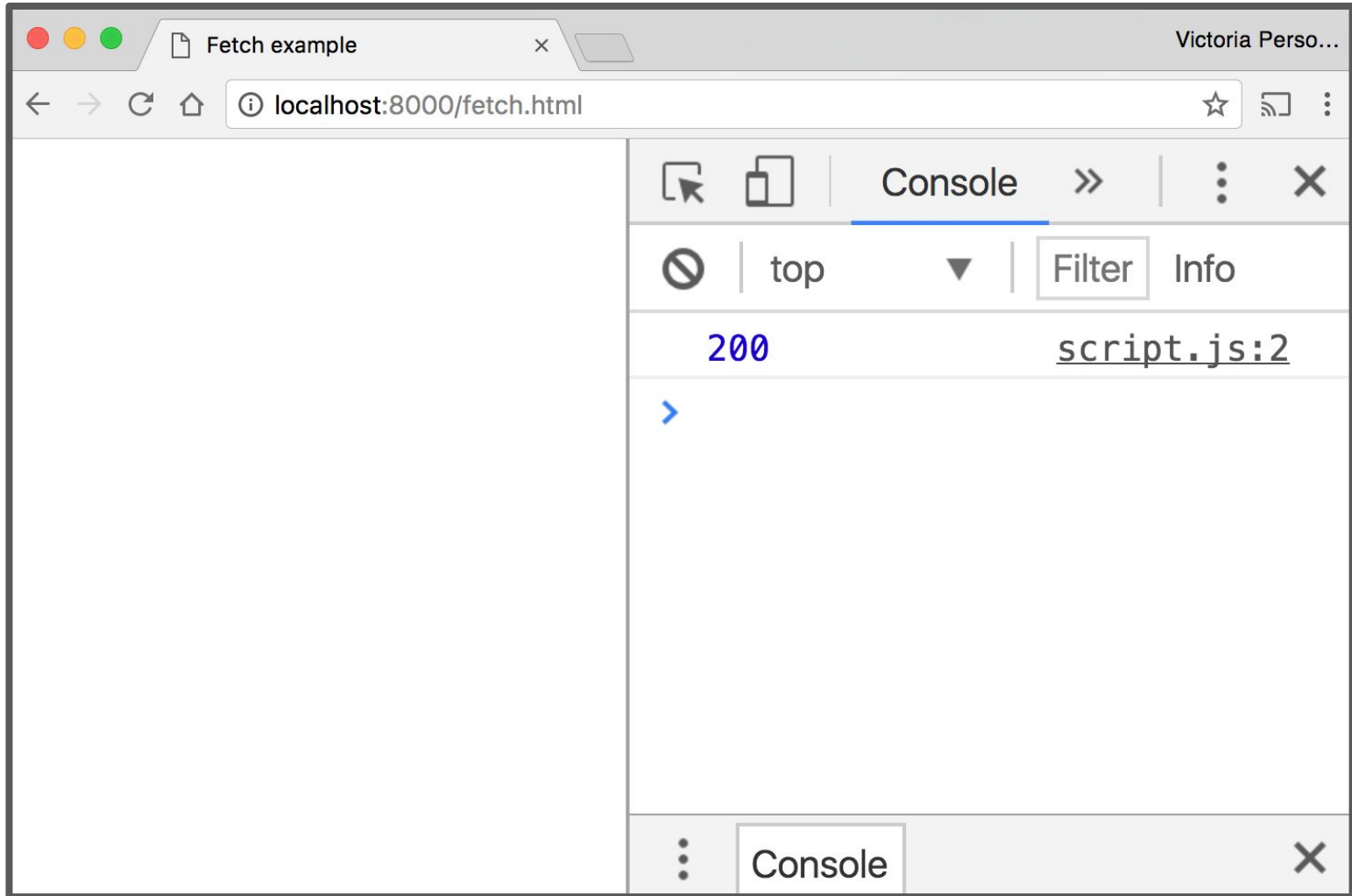We can run a program to serve our local files over HTTP:

```
$ python -m SimpleHTTPServer
Serving HTTP on 0.0.0.0 port 8000 ...
```

This now starts up a **server** that can load the files in the current directory over HTTP.

- We can access this server by navigating to: http://localhost:8000/

```
$ python -m SimpleHTTPServer
Serving HTTP on 0.0.0.0 port 8000 ...
```



**Directory listing for /**

- fetch.html
- images.txt
- script.js

We got HTTP response 200, which is success! ([codes](#))

# How do we get the data from `fetch()`?

# Using Fetch

```
function onSuccess(response) {
  ..
}
fetch('images.txt').then(onSuccess);
```

- `response.status`: Status code for the request
- `response.`**`text()`**:
    - Asynchronously reads the response stream
    - **Returns a `Promise`** that resolves with the string containing the response stream data.

# text() Promise

**Q: How do we change the following code to print out the response body?**

```javascript
function onSuccess(response) {
  console.log(response.status);
}

function onError(error) {
  console.log('Error: ' + error);
}

fetch('images.txt')
    .then(onSuccess, onError);
```

```javascript
function onStreamProcessed(text) {
  console.log(text);
}

function onResponse(response) {
  console.log(response.status);
  response.text().then(onStreamProcessed);
}

function onError(error) {
  console.log('Error: ' + error);
}

fetch('images.txt').then(onResponse, onError);
```

# Chaining Promises

We want the following asynchronous actions to be completed in this order:

1. When the `fetch` completes, run `onResponse`
2. When `response.text()` completes, run `onStreamProcessed`

```
function onStreamProcessed(text) { … }
function onResponse(response) {
  response.text().then(onStreamProcessed);
}
fetch('images.txt').then(onResponse, onError);
```

**We can rewrite this:**

```
function onStreamProcessed(text) {
  console.log(text);
}

function onResponse(response) {
  response.text().then(onStreamProcessed);
}

function onError(error) {
  console.log('Error: ' + error);
}

fetch('images.txt').then(onResponse, onError);
```

**We can rewrite this:**

```javascript
function onStreamProcessed(text) {
  console.log(text);
}


function onResponse(response) {
  return response.text();
}


function onError(error) {
  console.log('Error: ' + error);
}


fetch('images.txt')
    .then(onResponse, onError)
    .then(onStreamProcessed);
```

# Chaining Promises

```
function onStreamProcessed(text) {
  console.log(text);
}

function onResponse(response) {
  return response.text();
}

fetch('images.txt')
    .then(onResponse, onError)
    .then(onStreamProcessed);
```

# Chaining Promises

```
function onStreamProcessed(text) {
  console.log(text);
}


function onResponse(response) {
  return response.text();
}


const responsePromise = fetch('images.txt')
    .then(onResponse, onError)
responsePromise.then(onStreamProcessed);
```

**The Promise returned by** onResponse **is effectively\* the Promise returned by** `fetch`. (\*Not actually what's happening, but that's how we'll think about it for right now.)

# Chaining Promises

```
function onStreamProcessed(text) {
  console.log(text);
}

function onResponse(response) {
  return response.text();
}

fetch('images.txt')
    .then(onResponse, onError)
    .then(onStreamProcessed);
```

**If we don't think about it too hard, the syntax is fairly intuitive.**
We'll think about this more deeply later!

# Completed example

```
function onStreamProcessed(text) {
  const urls = text.split('\n');
  for (const url of urls) {
    const image = document.createElement('img');
    image.src = url;
    document.body.append(image);
  }
}
function onSuccess(response) {
  response.text().then(onStreamProcessed)
}
function onError(error) {
  console.log('Error: ' + error);
}

fetch('images.txt').then(onSuccess, onError);
```

# JSON

# JavaScript Object Notation

**JSON**: Stands for **J**ava**S**cript **O**bject **N**otation

- Created by Douglas Crockford
- Defines a way of **serializing** JavaScript objects
  - **to serialize**: to turn an object into a string that can be deserialized
  - **to deserialize**: to turn a serialized string into an object
- `JSON.stringify(`*object*`)` returns a string representing *object* serialized in JSON format
- `JSON.parse(`*jsonString*`)` returns a JS object from the *jsonString* serialized in JSON format

# JSON.stringify()

We can use the `JSON.stringify()` function to seralize a JavaScript object:

```
const bear = {
  name: 'Ice Bear',
  hobbies: ['knitting', 'cooking', 'dancing']
};

const serializedBear = JSON.stringify(bear);
console.log(serializedBear);
```

[CodePen](CodePen)

# JSON.parse()

We can use the `JSON.parse()` function to deseralize a JavaScript object:

```
const bearString = '{"name":"Ice
Bear","hobbies":["knitting","cooking","danci
ng"]}';
```

```
const bear = JSON.parse(bearString);
console.log(bear);
```

[CodePen](CodePen)

# Why JSON?

JSON is a useful format for storing data that we can load into a JavaScript API via fetch().

Let's say we had a list of Songs and Titles.
- If we stored it as a text file, we would have to know how we are separating song name vs title, etc
- If we stored it as a JSON file, we can just deserialize the object.

# JSON

songs.json

```json
{
  "cranes": {
    "fileName": "solange-cranes-kaytranada.mp3",
    "artist": "Solange",
    "title": "Cranes in the Sky [KAYTRANADA Remix]"
  },
  "timeless": {
    "fileName": "james-blake-timeless.mp3",
    "artist": "James Blake",
    "title": "Timeless"
  },
  "knock": {
    "fileName": "knockknock.mp4",
    "artist": "Twice",
    "title": "Knock Knock"
  },
  "deep": {
    "fileName": "janet-jackson-go-deep.mp3",
    "artist": "Janet Jackson",
    "title": "Go Deep [Alesia Remix]"
  },
  "discretion": {
    "fileName": "mitis-innocent-discretion.mp3",
    "artist": "MitiS",
    "title": "Innocent Discretion"
  },
  "spear": {
    "fileName": "toby-fox-spear-of-justice.mp3",
    "artist": "Toby Fox",
    "title": "Spear of Justice"
  }
}
```

# Fetch API and JSON

The Fetch API also has built-in support for json:

```
function onStreamProcessed(json) {
  console.log(json);
}


function onResponse(response) {
  return response.json();
}

fetch('songs.json')
    .then(onResponse, onError)
    .then(onStreamProcessed);
```