Web Development Fundamentals

March 2019

Slides are adopted from cs193X http://web.stanford.edu/class/cs193x/lectures/

Today: Open your mind

Open your mind

The next few lectures are probably the most conceptually difficult in the entire quarter.

We are going to be exploring a few ways in which JavaScript is **very, very different** from other programming languages you know.

We will likely push on your understanding of how programming languages work!

Today's schedule

Today:

- Keyboard events
- Mobile events
- Simple CSS animations
- Functional Programming
- First-class functions
- Loading data from files
  - Fetch API
  - Promises - High-level!
  - JSON
-

Other JavaScript events?

We've been doing a ton of JavaScript examples that involve `click` events...

Aren't there other types of events?

Other JavaScript events?

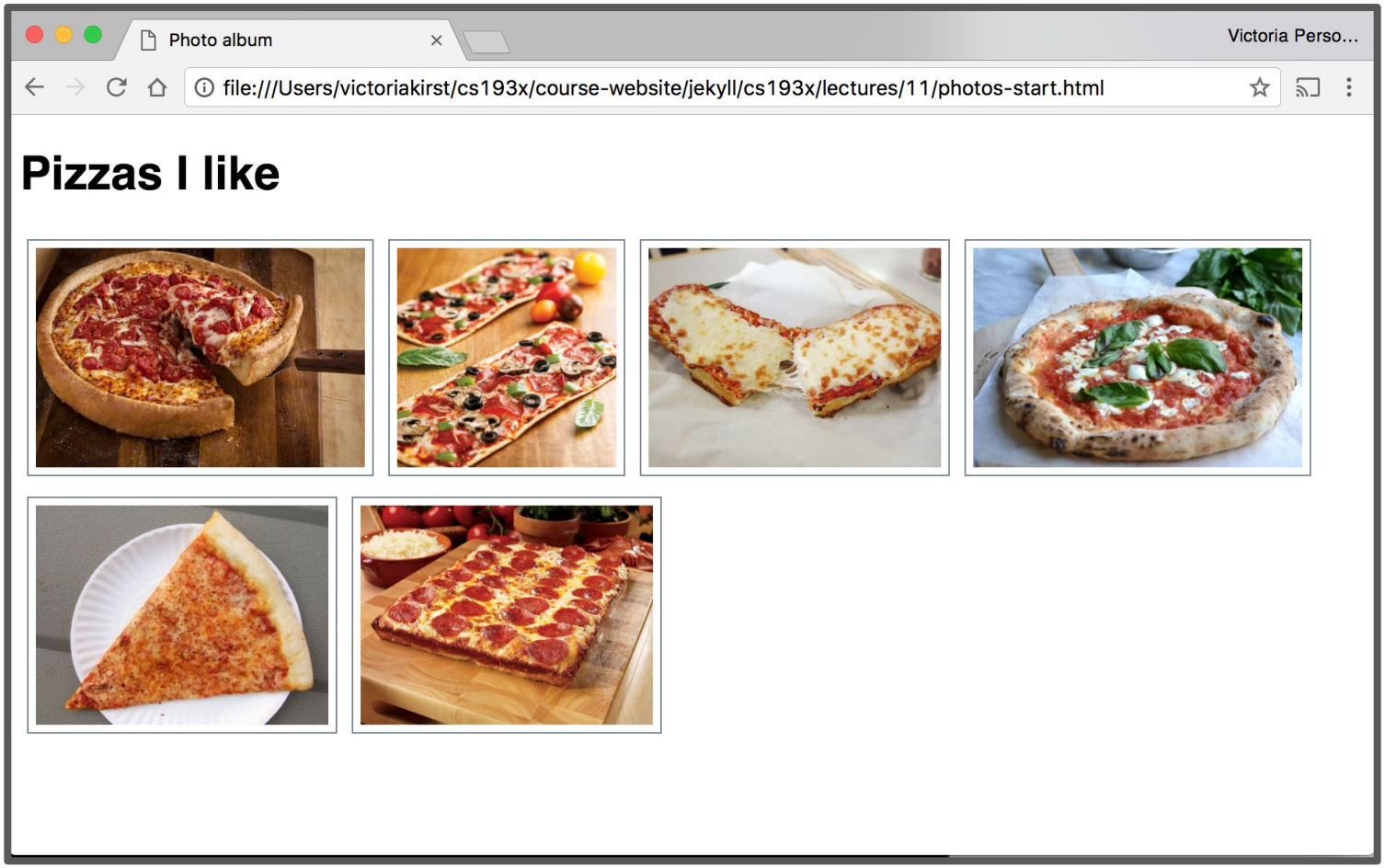
We've been doing a ton of JavaScript examples that involve `click` events...

Aren't there other types of events?

- Of course!
- Today we'll talk about:
  - **Keyboard events**
  - **Pointer / mobile events**
  - (possibly) **Animation events**

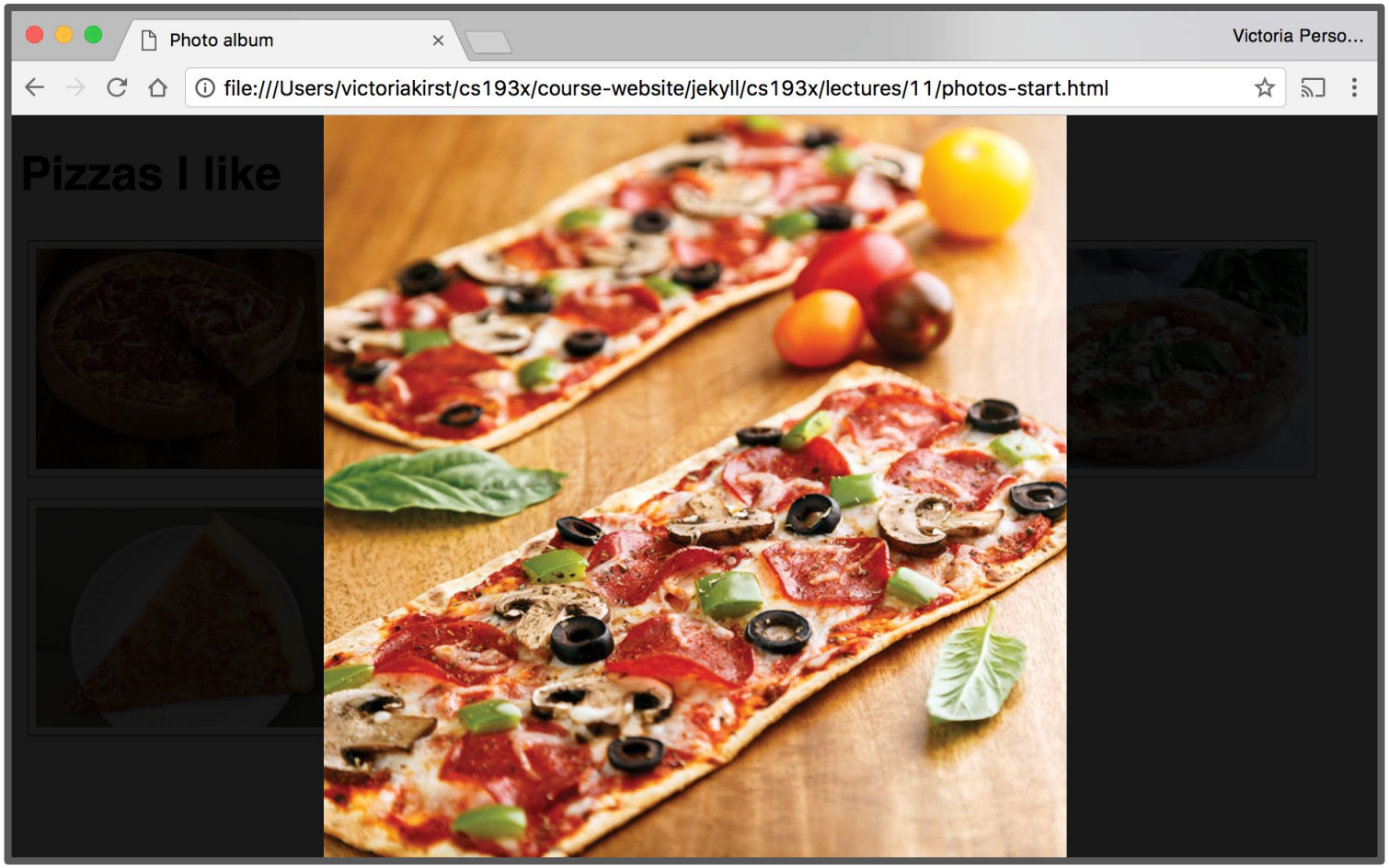
Example: Photo Album

We're going to add a few features to this photo album:



Example: Photo Album

We're going to add a few features to this photo album:



Code walkthrough:

photo-start.html

photo.js

photo.css

General setup

```
<body>
  <h1>Pizzas I like</h1>
  <section id="album-view">
  </section>

  <section id="modal-view" class="hidden">
  </section>
</body>
```

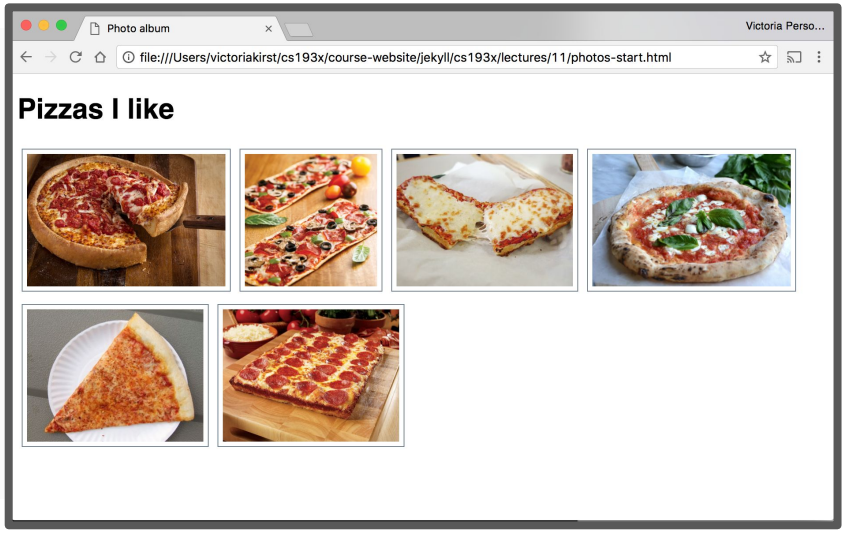
photo.html contains both "screens":

- The album view: Thumbnails of every photo
- The "modal" view: A single photo against a semi-transparent black background
  - Hidden by default

CSS: Album

photo.css: The album view CSS is pretty straightforward:

```
#album-view img {
  border: 1px solid slategray;
  margin: 5px;
  padding: 5px;
  height: 150px;
}
```



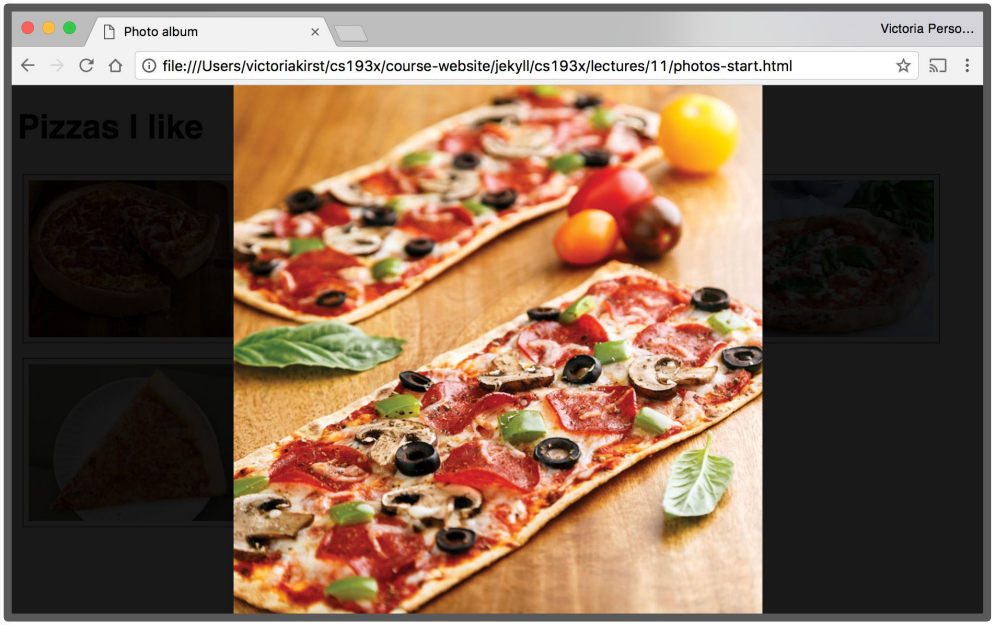
CSS: Modal

Modal view is a little more involved, but all stuff we've learned:

```
#modal-view {
  position: absolute;
  top: 0;
  left: 0;
  height: 100vh;
  width: 100vw;

  background-color: rgba(0, 0, 0, 0.9);
  z-index: 2;

  display: flex;
  justify-content: center;
  align-items: center;
}
```



CSS: Modal image

```css
#modal-view img {
  max-height: 100%;
  max-width: 100%;
}
```

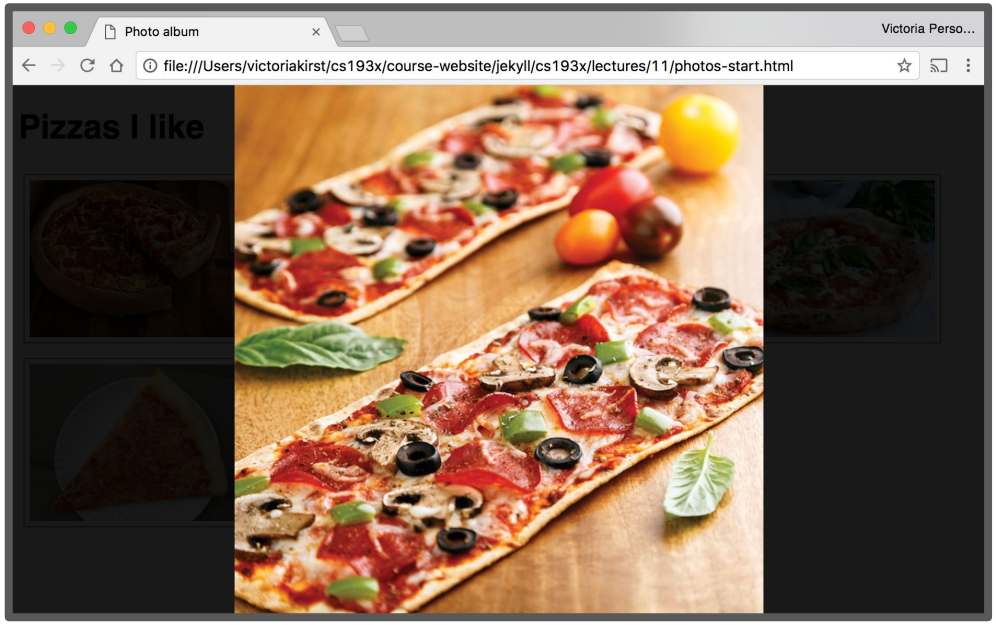


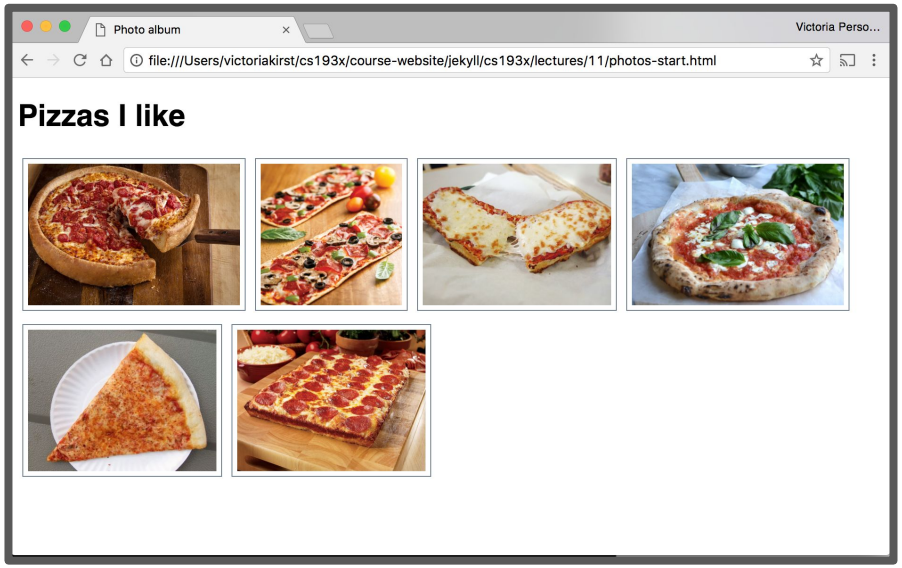
Image sizes are constrained to the height and width of the parent, `#modal-view` (whose height and width are set to the size of the viewport)

CSS: Hidden modal

```
<body>
  <h1>Pizzas I like</h1>
  <section id="album-view">
  </section>

  <section id="modal-view" class="hidden">
  </section>
</body>
```

```
#modal-view.hidden {
  display: none;
}
```



Even though both the album view and modal view are in the HTML, the model view is set to `display: none;` so it does not show up.

Global List of Photos

```
<head>
  <meta charset="utf-8">
  <title>Photo album</title>
  <link rel="stylesheet" href="css/photo.css">
  <script src="js/photo-list.js" defer></script>
  <script src="js/photo.js" defer></script>
</head>
```

```
const PHOTO_LIST = [
  'images/deepdish.jpg',
  'images/flatbread.jpg',
  'images/frenchbread.jpg',
  'images/neapolitan.jpg',
  'images/nypizza.jpg',
  'images/squarepan.jpg'
];
```

photo-list.js: There is a global array with the list of string photo sources called PHOTO_LIST.

Photo thumbnails

```
function createImage(src) {
  const image = document.createElement('img');
  image.src = src;
  return image;
}
```

```
const albumView = document.querySelector('#album-view');
for (let i = 0; i < PHOTO_LIST.length; i++) {
  const photoSrc = PHOTO_LIST[i];
  const image = createImage(photoSrc);
  image.addEventListener('click', onThumbnailClick);
  albumView.appendChild(image);
}
```

photo.js: We populate the initial album view by looping over PHOTO_LIST and appending <img>s to the #album-view.

Clicking a photo

```
function onThumbnailClick(event) {
  const image = createImage(event.currentTarget.src);
  modalView.appendChild(image);
  modalView.classList.remove('hidden');
}
```

When the user clicks a thumbnail:

- We create another `<img>` tag with the same src
- We append this new `<img>` to the `#modal-view`
- We unhide the `#modal-view`

Positioning the modal

```
function onThumbnailClick(event) {
  const image = createImage(event.currentTarget.src);
  modalView.style.top = window.pageYOffset + 'px';
  modalView.appendChild(image);
  modalView.classList.remove('hidden');
}
```

We'll add another line of JavaScript to anchor our modal dialog to the top of the viewport, not the top of the screen:

`modalView.style.top = window.pageYOffset + 'px';`

(See window.pageYOffset mdn. It is the same as window.scrollY.)

Aside: `style` attribute

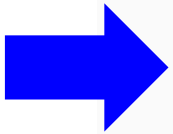
Every `HTMLElement` has a `style` attribute that lets you set a style directly on the element:

```
element.style.top = window.pageYOffset + 'px';
```

Generally **you should not use the `style` property**, as adding and removing classes via `classList` is a better way to change the style of an element via JavaScript

But when we are setting a CSS property based on JavaScript values, we must set the `style` attribute directly.

No scroll on page



```
function onThumbnailClick(event) {
  const image = createImage(event.currentTarget.src);
  document.body.classList.add('no-scroll');
  modalView.style.top = window.pageYOffset + 'px';
  modalView.appendChild(image);
  modalView.classList.remove('hidden');
}
```

```
.no-scroll {
  overflow: hidden;
}
```

And we'll also set `body { overflow: hidden; }` as a way to disable scroll on the page.

Closing the modal dialog

```
function onModalClick() {
  document.body.classList.remove('no-scroll');
  modalView.classList.add('hidden');
  modalView.innerHTML = '';
}
```

```
const modalView = document.querySelector('#modal-view');
modalView.addEventListener('click', onModalClick);
```

When the user clicks the modal view:

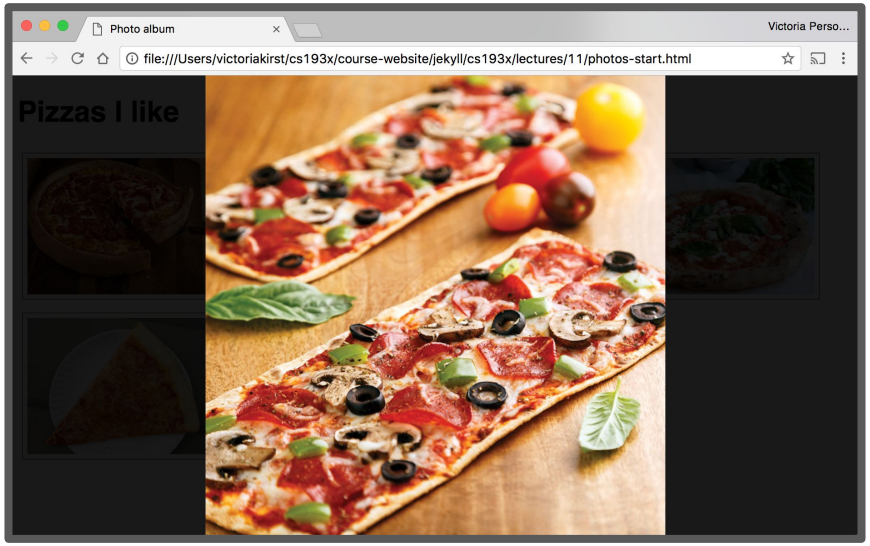
- We hide the modal view again
- We enable scroll on the page again
- We clear the image we appended to it by setting `innerHTML = '';`

Adding keyboard navigation

Navigating photos

Let's add some keyboard events to navigate between photos in the Modal View:

- Left arrow: Show the "i - 1"th picture
- Right arrow: Show the "i + 1"th picture
- Escape key: Close dialog



How do we listen
to keyboard events?

Keyboard events

Event name	Description
`keydown`	Fires when any key is pressed. Continues firing if you hold down the key. (mdn)
`keypress`	Fires when any **character** key is pressed, such as a letter or number. Continues firing if you hold down the key. (mdn)
`keyup`	Fires when you stop pressing a key. (mdn)

You can listen for keyboard events by adding the event listener to `document`:

```
document.addEventListener('keyup', onKeyUp);
```

KeyboardEvent.key

```
function onKeyUp(event) {
  console.log('onKeyUp:' + event.key);
}
document.addEventListener('keyup', onKeyUp);
```

Functions listening to a key-related event receive a parameter of `KeyboardEvent` type.

The `KeyboardEvent` object has a `key` property, which stores the string value of the key, such as `"Escape"`

- List of key values

Useful key values

Key string value	Description
`"Escape"`	The Escape key
`"ArrowRight"`	The right arrow key
`"ArrowLeft"`	The left arrow key

Example: key-events.html

Let's finish the feature!

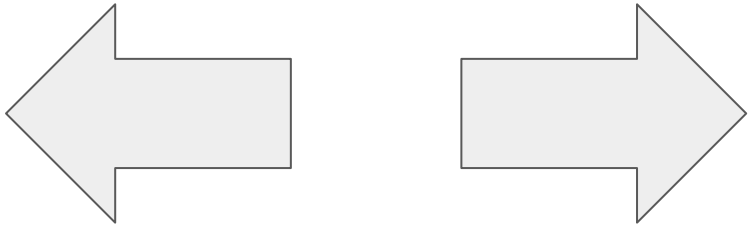
Finished result:
photo-desktop-finished.html

Mobile?

Keyboard events work well on desktop, but keyboard navigation doesn't work well for mobile.

On your phone, you can usually navigate photo albums using **gestures**:

- **Left swipe** reveals the next photo
- **Right swipe** reveals the previous photo



Next Previous

Mobile?

Keyboard events work well on desktop, but keyboard navigation doesn't work well for mobile.

On your phone, you can usually navigate photo albums using **gestures**:

- **Left swipe** reveals the next photo
- **Right swipe** reveals the previous photo

How do we implement the swipe gesture on the web?

Custom swipe events

- There are no gesture events in JavaScript (yet).
- That means there is no "Left Swipe" or "Right Swipe" event we can listen to. (Note that `drag` does not do what we want, nor does it work on mobile)

To get this behavior, we must implement it ourselves.

To do this, it's helpful to learn about a few more JS events:

- MouseEvent
- TouchEvent
- PointerEvent

MouseEvent

Event name	Description
`click`	Fired when you click and release (mdn)
`mousedown`	Fired when you click down (mdn)
`mouseup`	Fired when when you release from clicking (mdn)
`mousemove`	Fired repeatedly as your mouse moves (mdn)

*__`mousemove`__ only works on desktop, since there's no concept of a mouse on mobile.

TouchEvent

Event name	Description
`touchstart`	Fired when you touch the screen (mdn)
`touchend`	Fired when you lift your finger off the screen (mdn)
`touchmove`	Fired repeatedly while you drag your finger on the screen (mdn)
`touchcancel`	Fired when a touch point is "disrupted" (e.g. if the browser isn't totally sure what happened) (mdn)

*__`touchmove`__ only works on mobile (example)

`clientX` and `clientY`

```
function onClick(event) {
  console.log('x' + event.clientX);
  console.log('y' + event.clientY);
}
element.addEventListener('click', onClick);
```

MouseEvents have a `clientX` and `clientY` :

- `clientX`: x-axis position relative to the left edge of the browser viewport
- `clientY`: y-axis position relative to the top edge of the browser viewport

Implementing drag

When a user clicks down/touches an element…

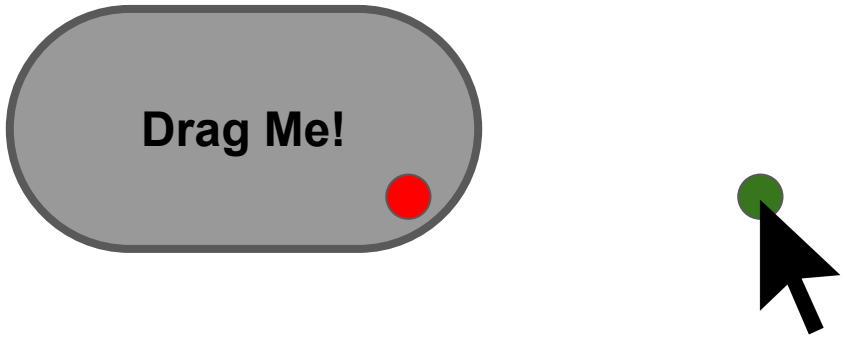
Implementing drag

```
originX = 100;
```

Take note of the starting position.

Implementing drag

```
originX = 100;
newX = 150;
```



Then on `mousemove` / `touchmove`, make note of the new mouse position

Implementing drag

```
originX = 100;
newX = 150;
```

Move the element by the difference between the old and new positions.

Implementing drag

Then on release...

Implementing drag

… stop listening to `mousemove` / `touchmove`.

Dragging on mobile and desktop

Wouldn't it be nice if we didn't have to listen to different events for mobile and desktop?

PointerEvent

`PointerEvent`: "pointer" events that work the same with for both mouse and touch

- Not to be confused with `pointer-events` CSS property (completely unrelated)
- **Note:** In this case, Mozilla's documentation on `PointerEvent` is not great.
  - A Google blog post on `PointerEvent`

`PointerEvent` inherits from `MouseEvent`, and therefore has `clientX` and `clientY`

PointerEvent

Event name	Description
`pointerdown`	Fired when a "pointer becomes active" (touch screen or click mouse down) (mdn)
`pointerup`	Fired when a pointer is no longer active (mdn)
`pointermove`	Fired repeatedly while the pointer moves (mouse move or touch drag) (mdn)
`pointercancel`	Fired when a pointer is "interrupted" (mdn)

*__`pointermove`__ works on mobile and desktop!

… Except…

Our first controversial feature!

`PointerEvent` is **not** implemented on all browsers yet:

- Firefox implementation is in progress
- Safari outright opposes this API… since 2012.

Argh!!! Does this mean we can't use it?

Polyfill library

A polyfill library is code that implements support for browsers that do not natively implement a web API.

Luckily there is a polyfill library for `PointerEvent`: https://github.com/jquery/PEP

PointerEvent Polyfill

To use the PEP polyfill library, we add this `script` tag to our HTML:

```
<script src="https://code.jquery.com/pep/0.4.1/pep.js"></script>
```

And we'll add need to add `touch-action="none"` to the area where we want `PointerEvents` to be recognized*:

```
<section id="photo-view" class="hidden" touch-action="none">
</section>
```

*Technically what this is doing is it is telling the browser that we do not want the default touch behavior for children of this element, i.e. on a mobile phone, we don't want to recognize the usual "pinch to zoom" type of events because we will be intercepting them via PointerEvent. This is normally a CSS property, but the limitations of the polyfill library requires this to be an HTML attribute instead.

Moving an element

We are going to use the `transform` CSS property to move the element we are dragging from its original position:

```
originX = 100;
newX = 150;
delta = newX - originX;
```


```
element.style.transform = 'translateX(' + delta + 'px)';
```

transform

transform is a strange but powerful CSS property that allow you to translate, rotate, scale, or skew an element.

`transform: translate(`***x***`, `***y***`)`	Moves element relative to its natural position by ***x*** and ***y***
`transform: translateX(`***x***`)`	Moves element relative to its natural position horizontally by ***x***
`transform: translateY(`***y***`)`	Moves element relative to its natural position vertically by ***y***
`transform: rotate(`***deg***`)`	Rotates the element clockwise by ***deg***
`transform: rotate(10deg) translate(5px, 10px);`	Rotates an element 10 degrees clockwise, moves it 5px down, 10px right

Examples

`translate` vs `position`

Can't you use `relative` or `absolute` positioning to get the same effect as `translate`? What's the difference?

- `translate` is much faster
- `translate` is optimized for animations

See comparison (article):

- Absolute positioning (click "10 more macbooks")
- transform: translate (click "10 more macbooks")

Finally, let's code!

preventDefault()

On desktop, there's a default behavior for dragging an image, which we need to disable with `event.preventDefault()`:

```
function startDrag(event) {
  event.preventDefault();
```

setPointerCapture()

To listen to pointer events that occur when the pointer goes offscreen, call `setPointerCapture` on the target you want to keep tracking:

```
event.target.setPointerCapture(event.pointerId);
```

`style` attribute

Every `HTMLElement` also has a `style` attribute that lets you set a style directly on the element:

```
element.style.transform =
        'translateX(' + value + ')';
```

Generally **you should not use the `style` property**, as adding and removing classes via `classList` is a better way to change the style of an element via JavaScript

But when we are dynamically calculating the value of a CSS property, we have to use the `style` attribute.

`style` attribute

The `style` attribute has **higher precedence** than any CSS property.

To undo a style set via the `style` attribute, you can set it to the empty string:

```
element.style.transform = '';
```

Now the element will be styled according to any rules in the CSS file(s).

(`requestAnimationFrame`)

(We are missing one key piece of getting smooth dragging motion, which is: `requestAnimationFrame`

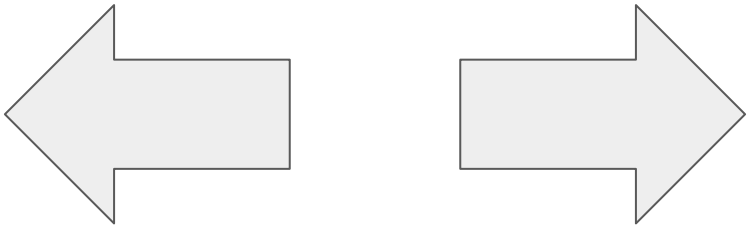
However, using `requestAnimationFrame` well requires us to know a little bit more about the JavaScript event loop. Functional programming also helps. We'll get there next week!)

CSS animations

Softening the edges

Our photo album feels a little jerky still. We can make the UI feel a little smoother if we added some animations.

- The image should **slide in from the left** if we are going to the previous picture
- The image should **slide in from the right** if we are going to the next picture



CSS animations syntax

```
@keyframes animation-name {
   from {
      CSS styles
   }
   to {
      CSS styles
   }
}
```

Examples

Then set the following CSS property:

`animation: animation-name duration;`

Easier example: Fade in

```
#album-view img {
  animation: fadein 0.5s;
}

@keyframes fadein {
  from {
    opacity: 0;
  }
  to {
    opacity: 1;
  }
}
```

Finished result:
photo-mobile-finished.html

One strategy for doing this: Custom events

Custom Events

You can listen to and dispatch Custom Events to communicate between classes (mdn):

```
const event = new CustomEvent(
    eventNameString, optionalParameterObject);
element.addEventListener(eventNameString,
functionName);
element.dispatchEvent(eventNameString);
```

Custom Events on `document`

CustomEvent **can only be listened to / dispatched on HTML elements,** and not on arbitrary class instances.

Therefore we are going to be adding/dispatching events on the `document` object, so that events can be globally listened to/dispatched.

```
document.addEventListener(eventNameString, functionName);
document.dispatchEvent(eventNameString);
```

Define a custom event

We'll define a custom event called `'button-click'`:

Menu will listen for the event:

```
document.addEventListener(
    'button-click', this.showButtonClicked);
```

Button will dispatch the event:

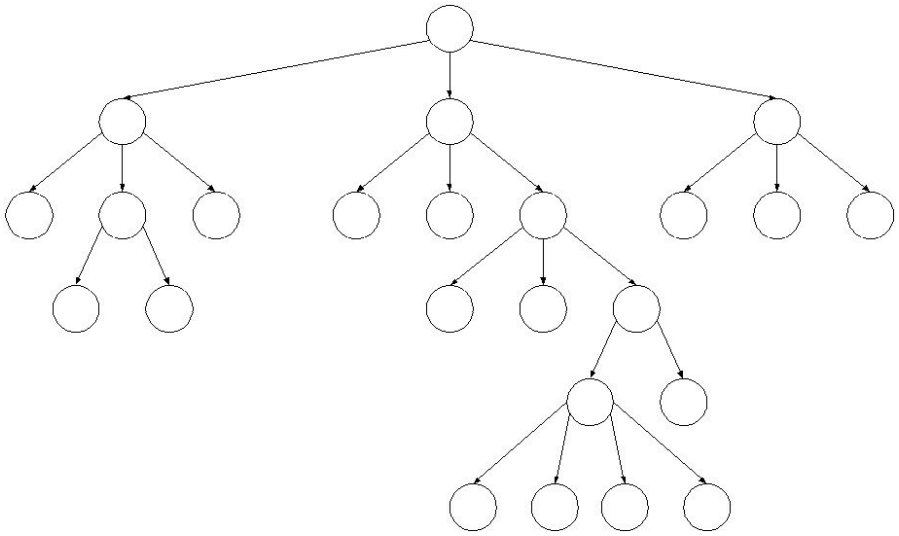
```
document.dispatchEvent(
    new CustomEvent('button-click'));
```

Understanding the DOM

DOM Nodes

If the DOM is a tree composed of Nodes…

Q: Does that mean a Node in the DOM has child pointers like the trees we learned about in 106B?

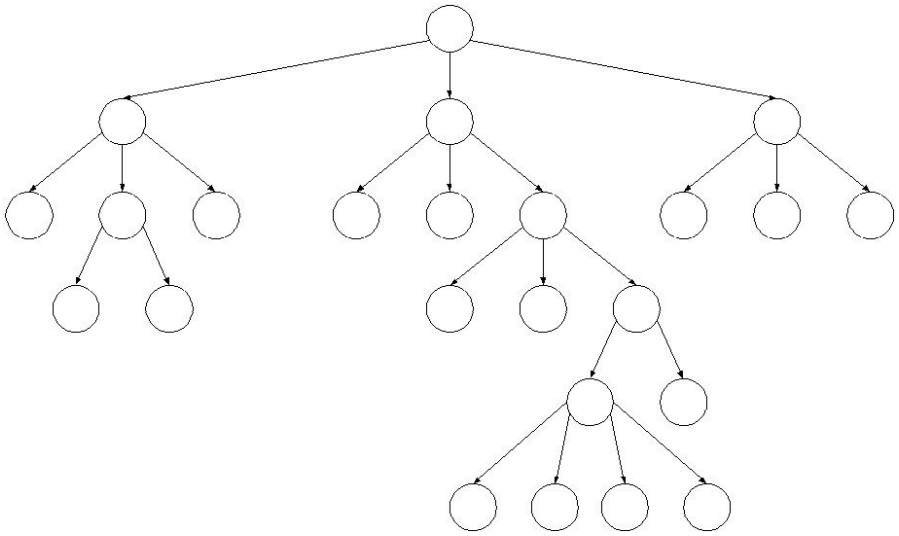


DOM Nodes

If the DOM is a tree composed of Nodes…

Q: Does that mean a Node in the DOM has child pointers like the trees we learned about in 106B?

A: Yes!



Node properties

Property	Description
textContent	The text content of a node and its descendants. (This property is writeable)
childNodes	An array of this node's children (empty if a leaf)
parentNode	A reference to this node's parent Node

```
<body>
    <h1>My favorites</h1>
    <section>
        <p>Strawberries</p>
        <p>Chocolate</p>
    </section>
</body>
```

What's the **parentNode** of **<section>**?

parentNode

```
> section = document.querySelector('section');
<· ▶<section>…</section>
> section.parentNode
<· ▶<body>…</body>
```

```
<body>
    <h1>My favorites</h1>
    <section>
        <p>Strawberries</p>
        <p>Chocolate</p>
    </section>
</body>
```

The **parentNode** of **<section>** is **<body>**.

What are the **childNodes** of **<section>**?

childNodes

```
> section = document.querySelector('section');
<- ▶<section>…</section>
> section.childNodes
<- ▶[text, p, text, p, text]
> section.childNodes.length
<- 5
```

???

```
<body>
    <h1>My favorites</h1>
    <section>
        <p>Strawberries</p>
        <p>Chocolate</p>
    </section>
</body>
```

Why does `section` have 5 children, not 2?!

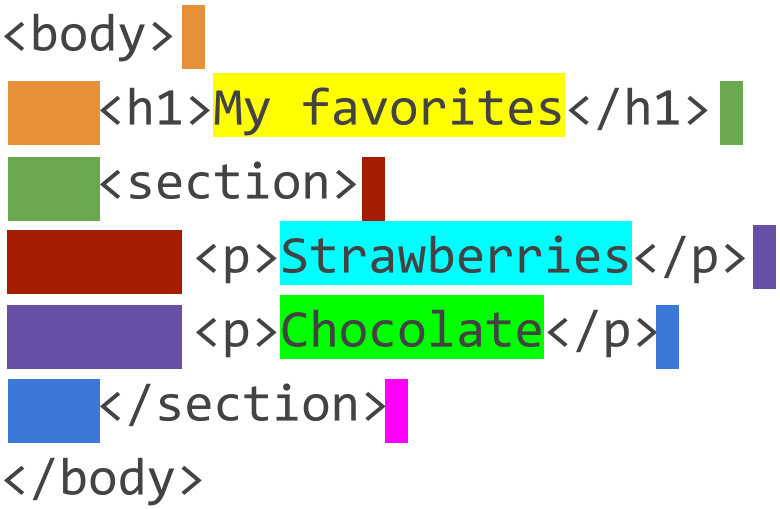
TextNode

In addition to `Element` nodes, the DOM also contains `Text` nodes. All text present in the HTML, **including whitespace**, is contained in a text node:

```
<body>
    <h1>My favorites</h1>
    <section>
        <p>Strawberries</p>
        <p>Chocolate</p>
    </section>
</body>
```

TextNode

All text present in the HTML, **including whitespace**, is contained in a `Text` node:



DOM and Text nodes

The DOM is composed of `Node`s, and there are several subtypes of `Node`.

- `Element`: HTML (or SVG) elements in the DOM
- `Text`: Text content in the DOM, including whitespace
  - `Text` nodes cannot contain children (are always leafs)
- `Comment`: HTML comments
- (more / DOM visualizer)

The type of a node is stored in the `nodeType` property

Traversing the DOM

Q: How would we print out all nodes in the DOM?

Traversing the DOM

Q: How would we print out all nodes in the DOM?

A: Recursively walk the DOM tree:

```
function walkTree(root, level) {
  if (root.nodeType == Node.TEXT_NODE) {
    console.log(level + 'text:' + root.textContent);
  } else {
    console.log(level + root.nodeName);
  }
  for (const child of root.childNodes) {
    walkTree(child, level + "    ");
  }
}
walkTree(document.querySelector('html'), "");
```

What's the point?

- If we have document.querySelector that lets us get elements in the DOM…
- And if we can change the HTML as necessary to add classes/ids/elements/etc to select the right things…

Q: When would we ever want to traverse the DOM?

What's the point?

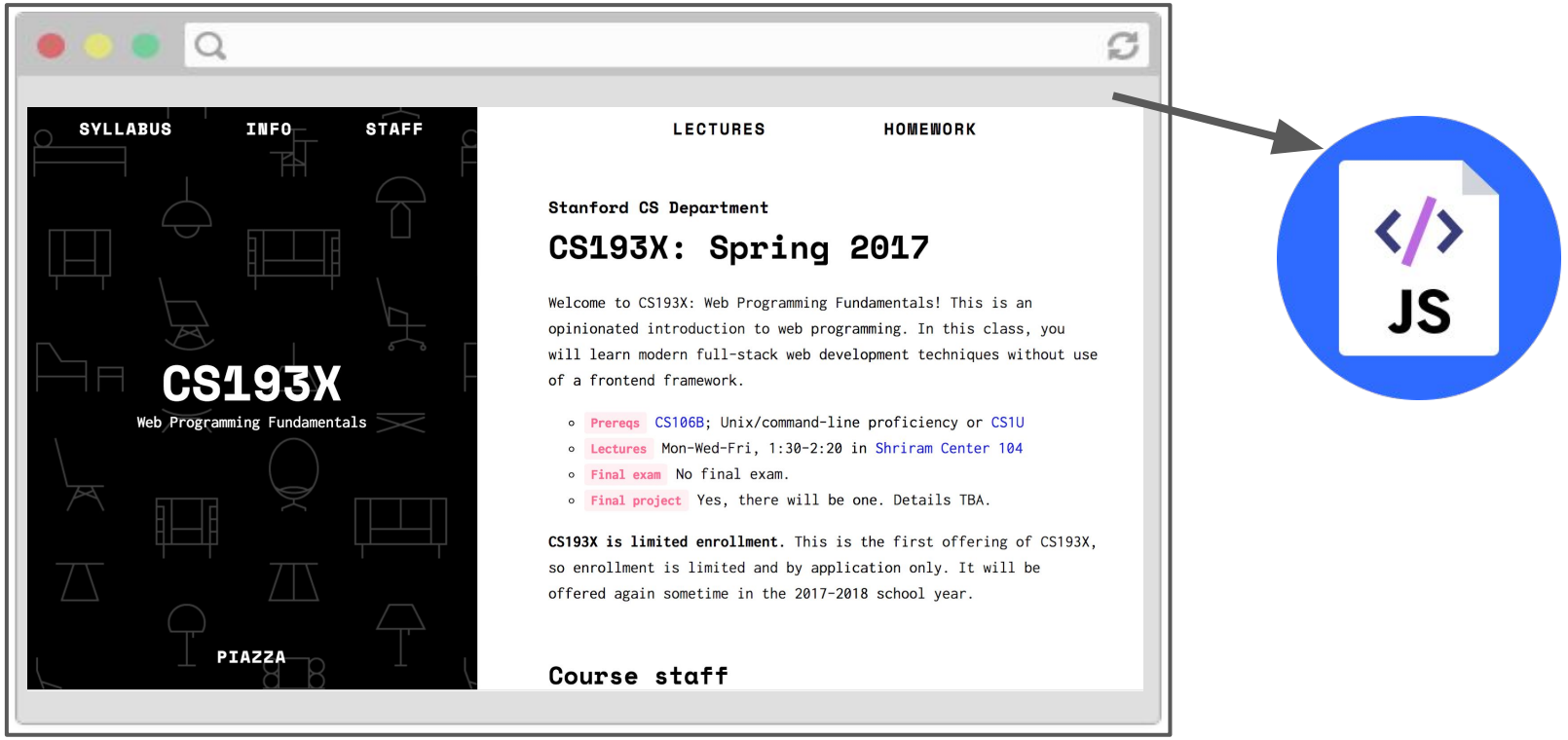
- If we have document.querySelector that lets us get elements in the DOM…
- And if we can change the HTML as necessary to add classes/ids/elements/etc to select the right things…

Q: When would we ever want to traverse the DOM?

A: Pretty much only in browser extensions or the Web Console (i.e. manipulating someone else's page)

Browser extensions

- Add-on that extends the functionality of the browser
- A piece of JavaScript that is injected into the webpage before or after it has loaded



Content scripts

- A type of extension that runs in the context of a web browser, meaning it can access the DOM of the page on which it's loaded
  - Concept is the same in Chrome, Firefox, probably other browsers

Usually composed of:

- `manifest.json`: Contains title, settings, etc for the script
- `page.js`: The extension file

We are using a **content script** in HW2, Part 2

Example `manifest.json`

```
{
  "name": "Empty Chrome Extension",
  "version": "1.0",
  "description": "",
  "content_scripts": [
    {
      "matches": ["<all_urls>"],
      "js": ["page.js"]
    }
  ],
  "icons": {
    "16": "pizza.png",
    "48": "pizza.png",
    "128": "pizza.png"
  },
  "manifest_version": 2
}
```

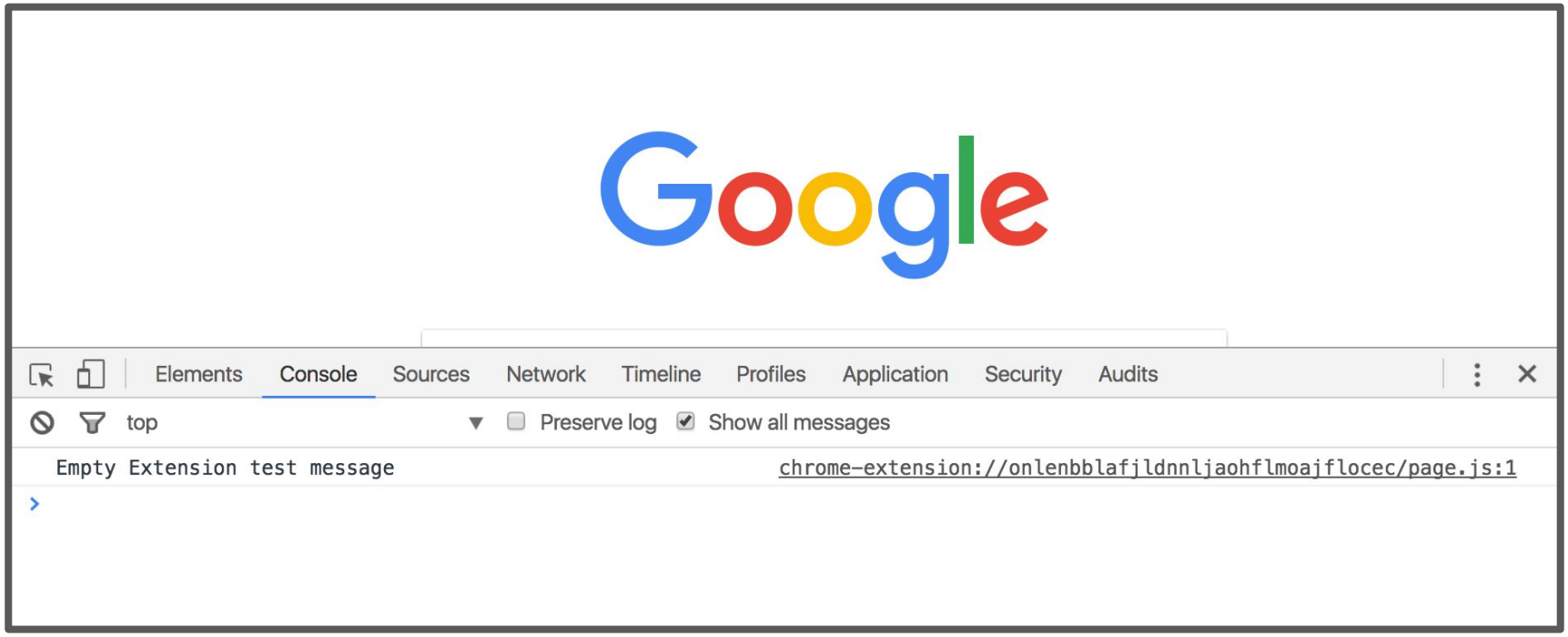
Example `page.js`

page.js

```
1  console.log('Empty Extension test message');
2
```

Result

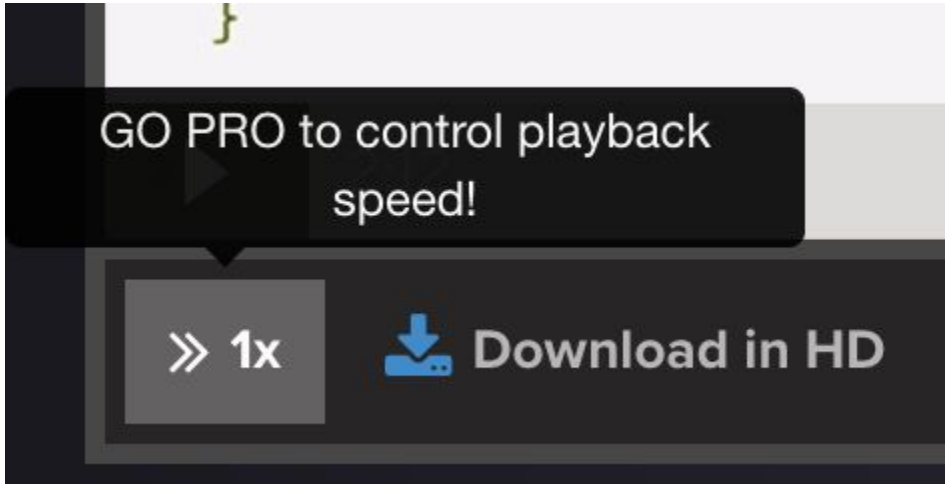
When this extension is loaded, the console message appears in the Web Inspector for every page:

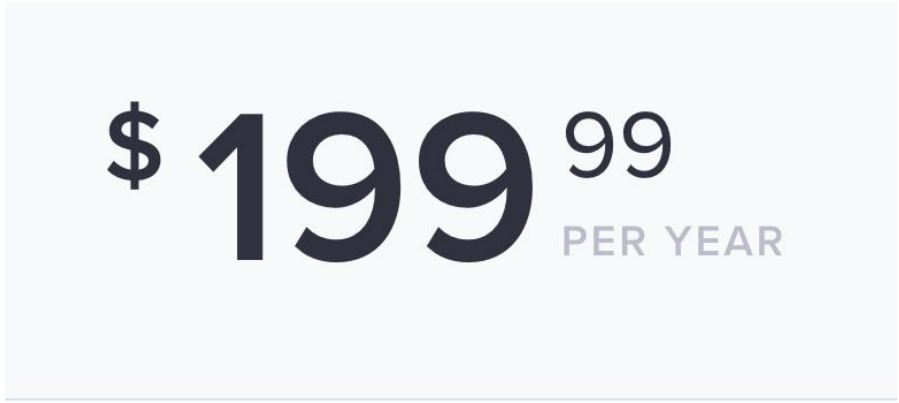


Hacks and Mischief

Example: Egghead

[Dan Abromov's Redux videos]





Example: Egghead

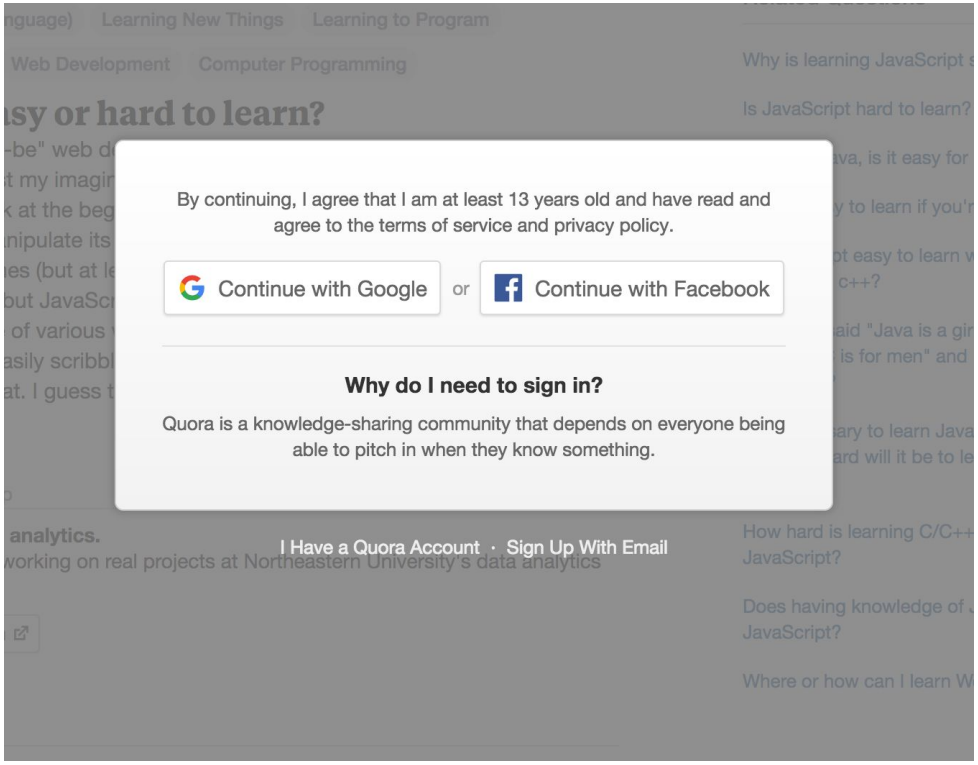
```
> document.querySelector('video')
<- ▶<video id="wistia_simple_video_25" crossorigin="anonymous" preload="me
   "background: transparent; display: block; height: 100%; position: static
> document.querySelector('video').playbackRate = 2
<- 2
> document.querySelector('video').playbackRate = 1.5
<- 1.5
>
```

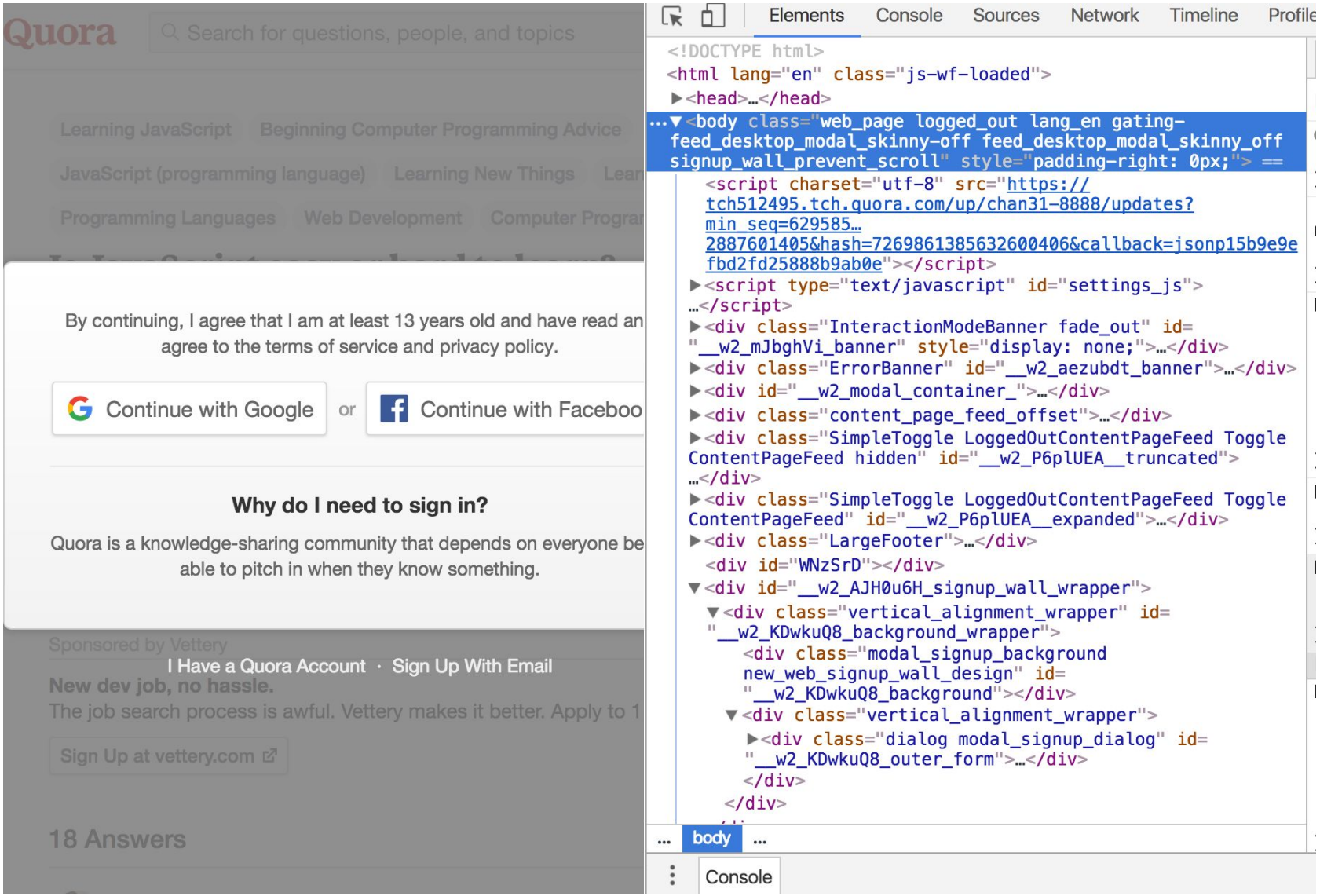
We can access the DOM via `querySelector` in the Web Inspector. Useful for debugging, and for speeding up videos.

(MDN for `playbackRate`)

Example: Quora signin wall

https://www.quora.com/Why-is-learning-JavaScript-so-hard




Quora
Search for questions, people, and topics
Learning JavaScript
Beginning Computer Programming Advice
JavaScript (programming language)
Learning New Things
Programming Languages
Web Development
By continuing, I agree that I am at least 13 years old and have read an
agree to the terms of service and privacy policy.
Continue with Google
or
Continue with Faceboo
Why do I need to sign in?
Quora is a knowledge-sharing community that depends on everyone be
able to pitch in when they know something.
Sponsored by Vettery
I Have a Quora Account · Sign Up With Email
New dev job, no hassle.
The job search process is awful. Vettery makes it better. Apply to 1
Sign Up at vettery.com
18 Answers
Elements
Console
Sources
Network
Timeline
Profile
<!DOCTYPE html>
<html lang="en" class="js-wf-loaded">
▶<head>…</head>
…▼<body class="web_page logged_out lang_en gating-
feed_desktop_modal_skinny-off feed_desktop_modal_skinny_off
signup_wall_prevent_scroll" style="padding-right: 0px;"> ==
<script charset="utf-8" src="https://
tch512495.tch.quora.com/up/chan31-8888/updates?
min_seq=629585…
2887601405&hash=726986138563260040 6&callback=jsonp15b9e9e
fbd2fd25888b9ab0e"></script>
▶<script type="text/javascript" id="settings_js">
…</script>
▶<div class="InteractionModeBanner fade_out" id=
"__w2_mJbghVi_banner" style="display: none;">…
▶ …
▶ …
▶ …
▶<div class="SimpleToggle LoggedOutContentPageFeed Toggle
ContentPageFeed hidden" id="__w2_P6plUEA__truncated">
…
▶<div class="SimpleToggle LoggedOutContentPageFeed Toggle
ContentPageFeed" id="__w2_P6plUEA__expanded">…
▶ …

▼
▼<div class="vertical_alignment_wrapper" id=
"__w2_KDwkuQ8_background_wrapper">
<div class="modal_signup_background
new_web_signup_wall_design" id=
"__w2_KDwkuQ8_background">
▼
▶<div class="dialog modal_signup_dialog" id=
"__w2_KDwkuQ8_outer_form">…

…
body
…
Console

Quora

Search for questions, people, and topics

Sign In

Learning JavaScript | Beginning Computer Programming Advice | JavaScript (programming language) | Learning New Things | Learning to Program | Programming Languages | Web Development | Computer Programming

Is JavaScript easy or hard to learn?

I'm a 15 year old "want-to-be" web developer and I have dreams of using a plethora of web languages to manifest my imagination onto the Internet. They're big dreams for me, but unfortunately I'm stuck at the beginning mulling over the semantic structure of JavaScript and how to manipulate its use for my whim. I've found it at sometimes frustrating and at sometimes (but at less times) absolutely magical. I've gotten over HTML quickly, and CSS as well; but JavaScript is by far obviously the hardest of all three. I have looked at the source code of various websites and become boggled at how developers for that specific website easily scribble down all that code. I tell myself sometimes that I will never be able to do that. I guess the essence of my question is: Is JavaScript easy or hard to learn?

18 Answers

Glen Murphy, JS Engineer@Google (no longer)

Related Questions

- Why is learning JavaScript so hard?
- Is JavaScript hard to learn?
- Is Javascript easy to learn when I already can program in c++?
- If I know Java, is it easy for me to learn JavaScript?
- Is PHP easy to learn if you're used to JavaScript?
- My friend said "Java is a girl's programming language and C++/C is for men" and I feel offended. How can I get over it?
- Is it necessary to learn JavaScript if I know Ruby? And how hard will it be to learn JavaScript if I know Ruby?
- How hard is learning C/C++ considering you know JavaScript?
- What is the best approach to learn Hybris?
- Does having knowledge of Java make it easy to learn JavaScript?

Quora Chrome Extension

You can make a Chrome extension to automate this:

manifest.json

```
  "matches": ["*://www.quora.com/*"],
  "js": ["page.js"]
}
```

page.js

```
document.body.classList.remove('signup_wall_prevent_scroll');
const nagScreen = document.querySelector('.vertical_alignment_wrapper');
if (nagScreen) {
  nagScreen.style.display = 'none';
}
```

Aside: `style` attribute

Every `HTMLElement` also has a `style` attribute that lets you set a style directly on the element:

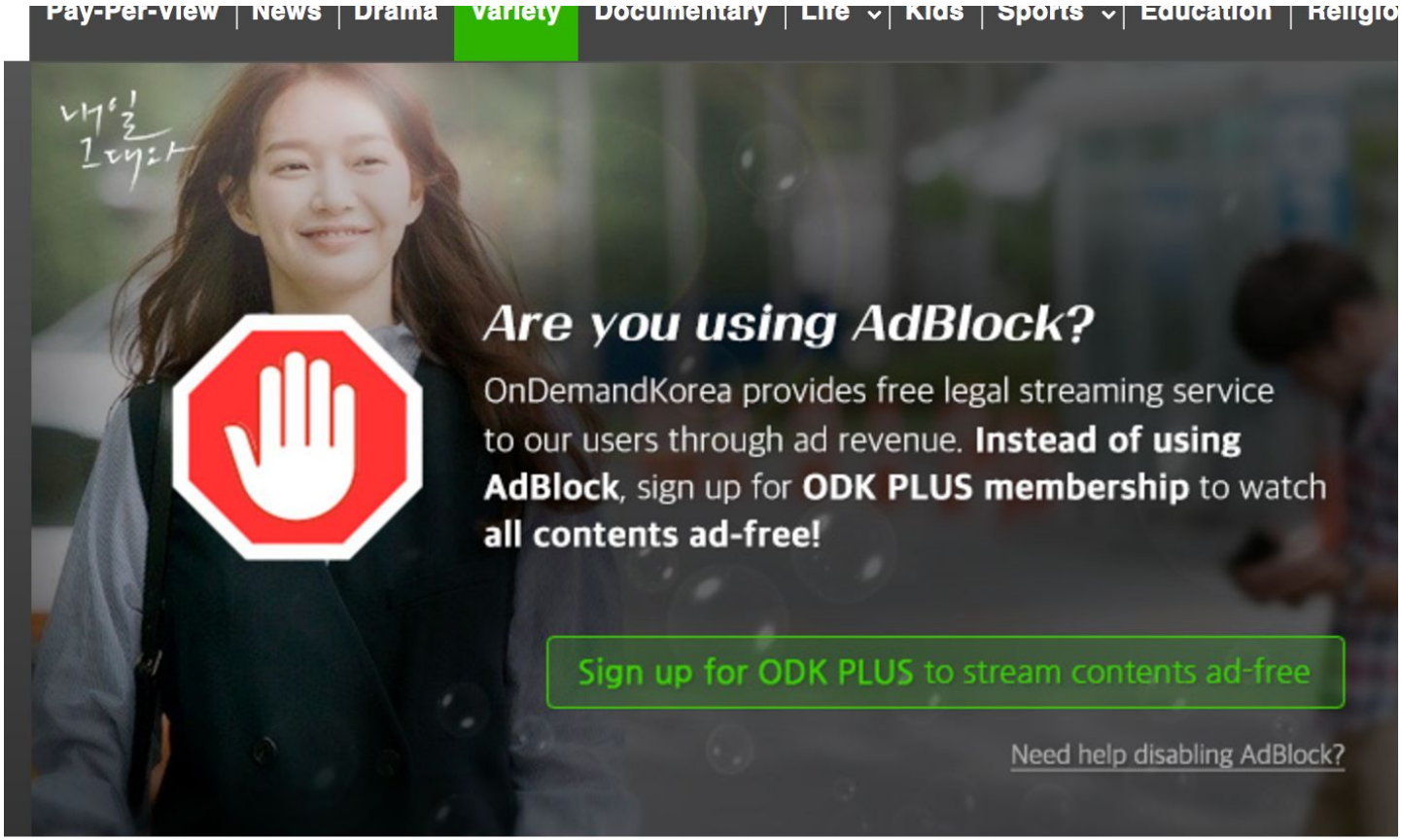
```
nagScreen.style.display = 'none';
```

Generally **you should not use this property**, as adding and removing classes via `classList` is a better way to change the style of an element via JavaScript

(But for extensions/hacking, `style` can be useful!)

Example: Adblock block

http://www.ondemandkorea.com/kpop-star-season-6-seoul-qualifier.html



View-source, search for "adblock"...

view-source:www.ondemandkorea.com/kpop-star-season-6-e20161120.html

adblock 2 of 10

```
        });

        var view_seconds = 0;
        var last_watched_position = 0;
        var view_timer = setInterval(function() {
            if(jwplayer().getState() == 'playing')
            {
                if(jwplayer().getPosition() != last_watched_position)
                {
                    view_seconds += 1;
                    if(view_seconds > 0 && view_seconds%300 == 0)
                    {
                        $p("send","watched_5_min");
                        ga('set', 'dimension5', 'sbs_sub');
                        ga('send', 'event', 'Video/variety', 'ViewhourLog', 'kpop-star-season-6-e20161120', 300);
                    }
                }
            }
            last_watched_position = jwplayer().getPosition();
        }, 1000);

                            jwplayer().on('adBlock',function(event) {
                        jwplayer().remove();
                        $("#odk_player").html('<a id="adblock_signup_in_player" href="https://www.ondemandkorea.com/odk-plus-benefits" target="_blank"
style="position: absolute;margin-top: 291px;margin-left: 262px;width: 406px;height: 38px;"></a><a href="http://www.ondemandkorea.com/blog/?p=2085"
target="_blank" style="position: absolute;margin-top: 341px;margin-left: 497px;width: 172px;height: 20px;"></a><img
src="http://www.ondemandkorea.com/images/player_adblock_0_en.jpg">');
                    });
                            if(offset != false)
        {
            jwplayer().on('firstFrame',function(){
                jwplayer().seek(offset);
            });
        }
                        var overlay_shown = true;
                var overlay_showing_now = false;
                var ad_playing = false;

                jwplayer("odk_player").on('time',function(e){
                    if(overlay_shown == false && ad_playing == false)
                    {
                        if(overlay_showing_now == false && (e.position > 0.5 && e.position < 4.5))
                        {
```

Lol global scope

```
jwplayer().on('adBlock', function(event) {
   jwplayer().remove();
   $("#odk_player").html('<a id="adblock_signup_in_player"
href="https://www.ondemandkorea.com/odk-plus-benefits" target="_blank"
style="position: absolute;margin-top: 291px;margin-left: 262px;width:
406px;height: 38px;"></a><a
href="http://www.ondemandkorea.com/blog/?p=2085" target="_blank"
style="position: absolute;margin-top: 341px;margin-left: 497px;width:
172px;height: 20px;"></a><img
src="http://www.ondemandkorea.com/images/player_adblock_0_en.jpg">');
});
```

Note that this is using the jQuery library, which I **do not** recommend you use

jQuery on?

Again, I **do not** recommend you use jQuery. But if we're trying to understand someone else's jQuery code...

.on()

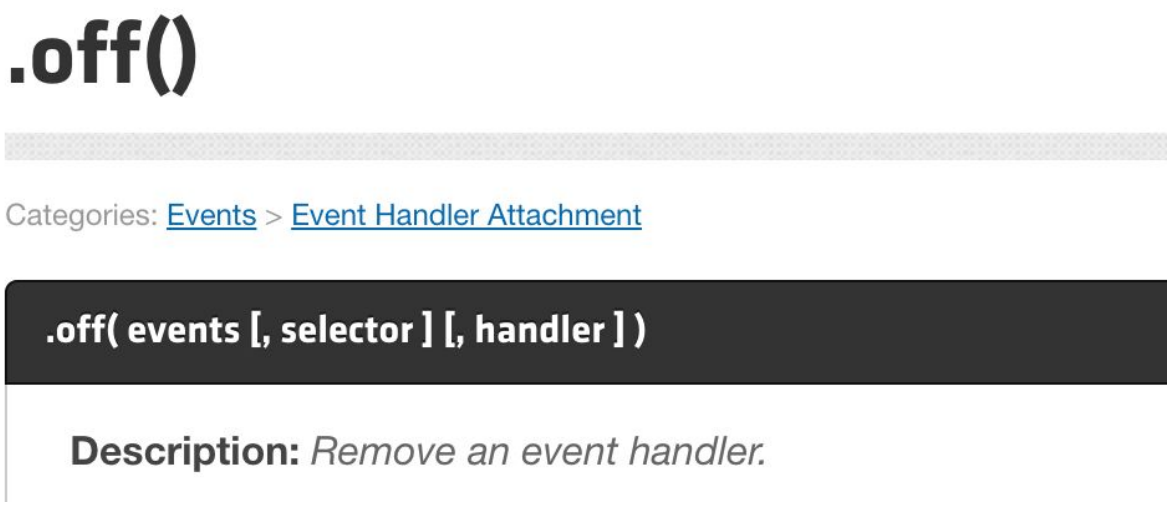
Categories: Events > Event Handler Attachment

.on(events [, selector] [, data], handler)

Description: *Attach an event handler function for one or more events to the selected elements.*

We see that `.on()` is essentially `addEventListener`

jQuery off



And `.off()` is essentially `removeEventListener`

Lol global scope

What if we try this...

page.js

```
1  const child = document.createElement('script');
2  child.textContent += 'jwplayer().off(\'adBlock\');';
3  document.body.appendChild(child);
4
```

Aside: inline `<script>`

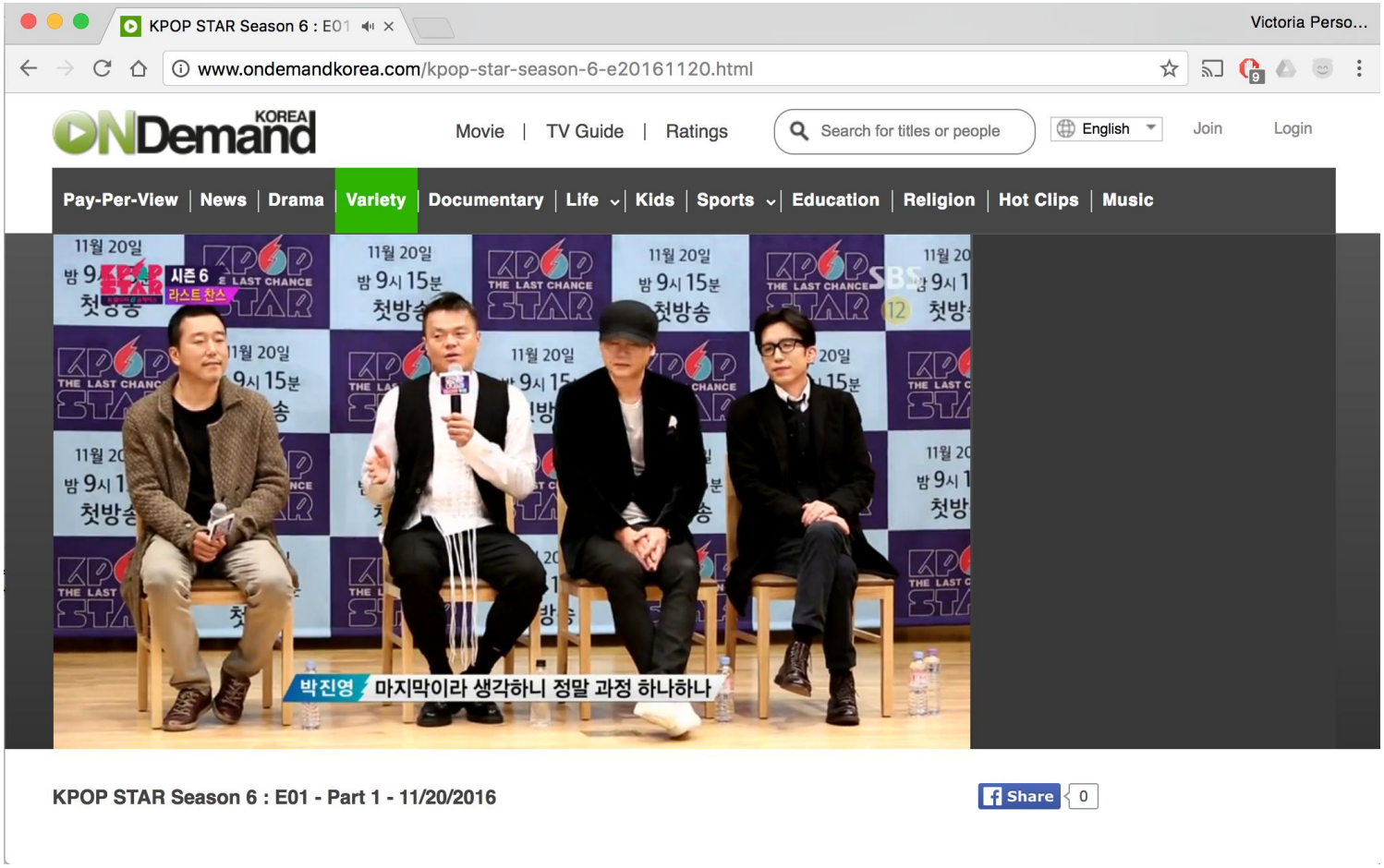
You can put JavaScript directly in your HTML by adding it to the body of a <script> tag:

```
<script>console.log('hello');</script>
```

Generally **you should not use do this**, as it's a violation of Separation of Concerns, mixing behavior (JS) in content (HTML)

(But for extensions/hacking, adding JavaScript directly in a `<script>` tag can be useful!)

Success!


KPOP STAR Season 6 : E01
Victoria Perso...
www.ondemandkorea.com/kpop-star-season-6-e20161120.html
ONDemand KOREA
Movie | TV Guide | Ratings
Search for titles or people
English
Join
Login
Pay-Per-View | News | Drama | Variety | Documentary | Life | Kids | Sports | Education | Religion | Hot Clips | Music
박진영 마지막이라 생각하니 정말 과정 하나하나
KPOP STAR Season 6 : E01 - Part 1 - 11/20/2016
Share 0

Callbacks

A real example: Callbacks

Another way we can communicate between classes is through callback functions:

- **Callback**: A function that's passed as a parameter to another function, usually in response to something.

First-class functions

Recall: `addEventListener`

Over the last few weeks, we've been using **functions** as a parameter to `addEventListener`:

```
dragon.addEventListener(
    'pointerdown', onDragStart);

image.addEventListener(
    'click', this._openPresent);
```

Q: How does this actually work?

First-class functions

Functions in JavaScript are objects.

- They can be saved in variables
- They can be passed as parameters
- They have properties, like other objects
- They can be defined without an identifier

(This is also called having **first-class functions**, i.e. functions in JavaScript are "first-class" because they are treated like any other variable/object.)

First-class functions

Functions in JavaScript are objects.

- They can be saved in variables
- They can be passed as parameters
- They have properties, like other objects
- They can be defined without an identifier

(This is also called having **first-class functions**, i.e. functions in JavaScript are "first-class" because they are treated like any other variable/object.)

???

First-class functions

Functions in JavaScript are objects.

- They can be saved in variables
- They can be passed as parameters
- They have properties, like other objects
- They can be defined without an identifier

(This is also called having **first-class functions**, i.e. functions in JavaScript are "first-class" because they are treated like any other variable/object.)

???

Isn't there like… a fundamental difference between "code" and "data"?

Be prepared to let go of some assumptions
you had about programming languages.

Let's take it all the way
back to first principles...

Back to the veeeeery basics

What is code?

- A list of instructions your computer can execute
- Each line of code is a **statement**

What is a function?

- A labeled group of **statements**
- The statements in a function are executed when the function is invoked

What is a variable?

- A labeled piece of **data**

Recall: Objects in JS

Objects in JavaScript are sets of property-value pairs:

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

- Like any other value, Objects can be saved in **variables.**
- Objects can be passed as parameters to functions

Back to the veeeeery basics

What is code?

- A list of instructions your computer can execute
- Each line of code is a **statement**

What is a function?

- A labeled group of **statements**
- The statements in a function are executed when the function is invoked

What is a variable?

- A labeled piece of **data**

What could it mean for a function to be an object, i.e. a kind of data?

Function variables

You can declare a function in several ways:

```
function myFunction(params) {
}

const myFunction = function(params) {
};

const myFunction = (params) => {
};
```

Function variables

```
function myFunction(params) {
}

const myFunction = function(params) {
};

const myFunction = (params) => {
};
```

Functions are invoked in the same way, regardless of how they were declared:

```
myFunction();
```

```
const x = 15;
let y = true;

const greeting = function() {
  console.log('hello, world');
}
```

"A function in JavaScript is an object of type `Function`"

In the interpreter's memory:

```
const x = 15;
let y = true;

const greeting = function() {
  console.log('hello, world');
}
```

"A function in JavaScript is an object of type `Function`"

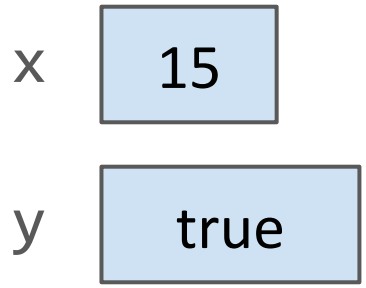
In the interpreter's memory:


```
const x = 15;
let y = true;

const greeting = function() {
  console.log('hello, world');
}
```

"A function in JavaScript is an object of type `Function`"

In the interpreter's memory:

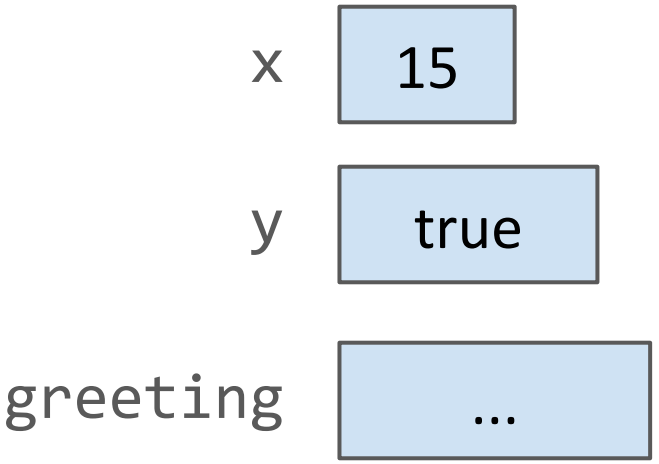



```
const x = 15;
let y = true;

const greeting = function() {
  console.log('hello, world');
}
```

"A function in JavaScript is an object of type `Function`"

In the interpreter's memory:




```
const x = 15;
let y = true;

const greeting = function() {
  console.log('hello, world');
}
```

"A function in JavaScript is an object of type `Function`"

What this really means:

- When you declare a function, there is an object of type `Function` that gets created alongside the labeled block of executable code.

Function properties

```
const greeting = function() {
  console.log('hello, world');
}

console.log(greeting.name);
console.log(greeting.toString());
```

When you declare a function, you create an object of type Function, which has properties like:

- name
- toString

CodePen

Function properties

```
const greeting = function() {
  console.log('hello, world');
}

greeting.call();
```

Function objects also have a call method, which invokes the underlying executable code associated with this function object.

CodePen

Function properties

```
const greeting = function() {
  console.log('hello, world');
}

greeting.call();
greeting();
```

() is an operation on the Function object (spec)

- When you use the () operator on a Function object, it is calling the object's call() method, which in turn executes the function's underlying code

Code vs `Functions`

Important distinction:

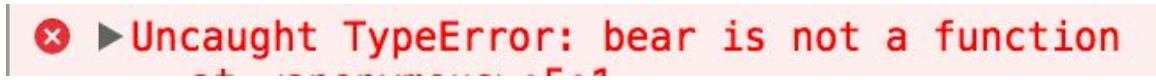
- Function, the executable code
  - A group of instructions to the computer
- `Function`, the object
  - A JavaScript object, i.e. a set of property-value pairs
  - `Function` objects have executable code associated with them
  - This executable code can be invoked by
    - *functionName*`();` *or*
    - *functionName*`.call();`

Note: `Function` is special

Only `Function` objects have executable code associated with them.

- Regular JS objects **cannot** be invoked
- Regular JS objects **cannot** be given executable code
  - I.e. you can't make a regular JS object into a callable function

```
const bear = {
  name: 'Ice Bear',
  hobbies: ['knitting', 'cooking', 'dancing']
};
bear(); // error!
```



Function Objects vs Objects

```
function sayHello() {
  console.log('Ice Bear says hello');
}

const bear = {
  name: 'Ice Bear',
  hobbies: ['knitting', 'cooking', 'dancing'],
  greeting: sayHello
};
bear.greeting();
```

CodePen

But you can give your object `Function` properties and then invoke those properties.

Function Objects vs Objects

```
function sayHello() {
  console.log('Ice Bear says hello');
}

const bear = {
  name: 'Ice Bear',
  hobbies: ['knitting', 'cooking', 'dancing'],
  greeting: sayHello
};
bear.greeting();
```

CodePen

The `greeting` property is an object of `Function` type.

Why do we have `Function` objects?!

Callbacks

`Function` objects **really** come in handy for event-driven programming!

```
function onDragStart(event) {
    ...
}
dragon.addEventListener('pointerdown', onDragStart);
```

Because every function declaration creates a `Function` object, we can pass `Functions` as parameters to other functions.

Simple, contrived example

```
function greetings(greeterFunction) {
  greeterFunction();
}

const worldGreeting = function() {
  console.log('hello world');
};

const hawaiianGreeting = () => {
  console.log('aloha');
};

greetings(worldGreeting);
greetings(hawaiianGreeting);
```

CodePen

```
function greetings(greeterFunction) {
  greeterFunction();
}

const worldGreeting = function() {
  console.log('hello world');
};

const hawaiianGreeting = () => {
  console.log('aloha');
};

greetings(worldGreeting);
greetings(hawaiianGreeting);
```

CodePen

This example is really contrived!

Aside from `addEventListener`, when would you ever want to pass a `Function` as a parameter?

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**

```
function makeHelloFunction(name) {
  return function() {
    console.log('Hello, ' + name);
  };
}
```

CodePen

Practical Functional JavaScript

Functional programming

We are going to cover some topics that are fundamental to a programming paradigm called **functional programming**.

Pure functional programming is pretty extreme:

- Everything in your code is either a function or an expression
- There are no statements
- There is no state:
  - No variables, fields, objects, etc

Comes from the idea of treating a computer program as a mathematical function

Functional programming

This is a code snippet from Scheme, a functional programming language:

```
(define (sum row)
  (let loop ((row row) (result '()))
    (if (= (length row) 1)
        (reverse result)
        (loop (cdr row)
              (cons (+ (first row) (second row))
                    result)))))
```

Everything is a function or the result of a function call.

Practical FP in JS

Most software is **not** built using a pure functional programming paradigm, so we won't be covering it.

But there are some ideas from functional programming that are immensely useful:

- First-class functions (functions as objects)
- **Currying**
- **Closures**
- **Anonymous functions** / lambdas / function literals

Why FP matters

Why should we learn about this other programming paradigm?

- There are **ideas you can express more clearly** and concisely with functional programming.
- There are **problems you can solve much more easily** with functional programming.
- *(very practically)* You will see JavaScript code in the wild that uses functional programing and the code will be indecipherable if you don't learn it.
- *(very practically)* Functional programming is trendy and so useful that C++ and Java added support for a few critical FP concepts (lambdas/closures) in the past few years.

First-class functions

Functions in JavaScript are objects.

- They can be saved in variables
- They can be passed as parameters
- They have properties, like other objects
- They can be defined without an identifier

(This is also called having **first-class functions**, i.e. functions in JavaScript are "first-class" because they are treated like any other variable/object.)

Recall: Functions as parameters

We know that we can pass functions as parameters to other functions. We've already done this multiple times:

- The event handler parameter to `addEventListener`
- As a parameter for a constructor of a new object

`Array` objects also have several methods that take functions as parameters.

Closure & Currying

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

```
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

```
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

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 objec

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

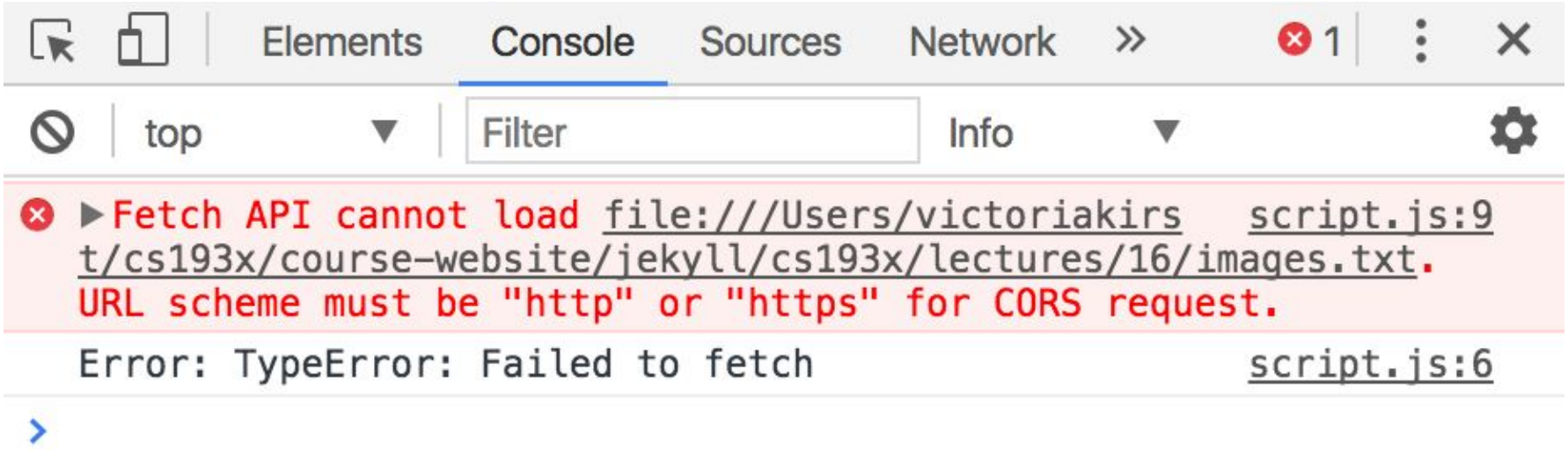
```
function onSuccess(response) {
  console.log(response.status);
}

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

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

Fetch error

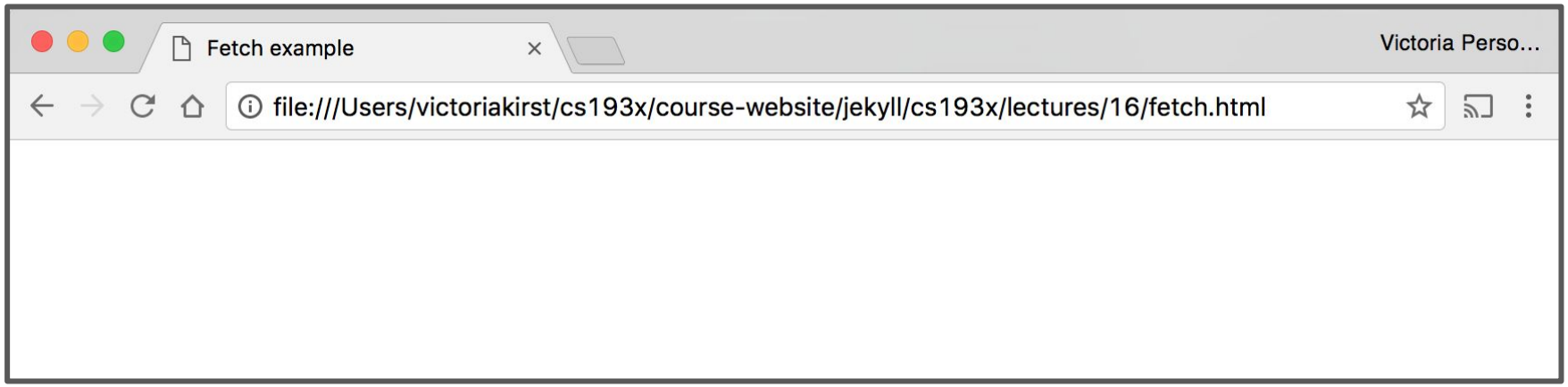
If we try to load this in the browser, we get the following JavaScript error:



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

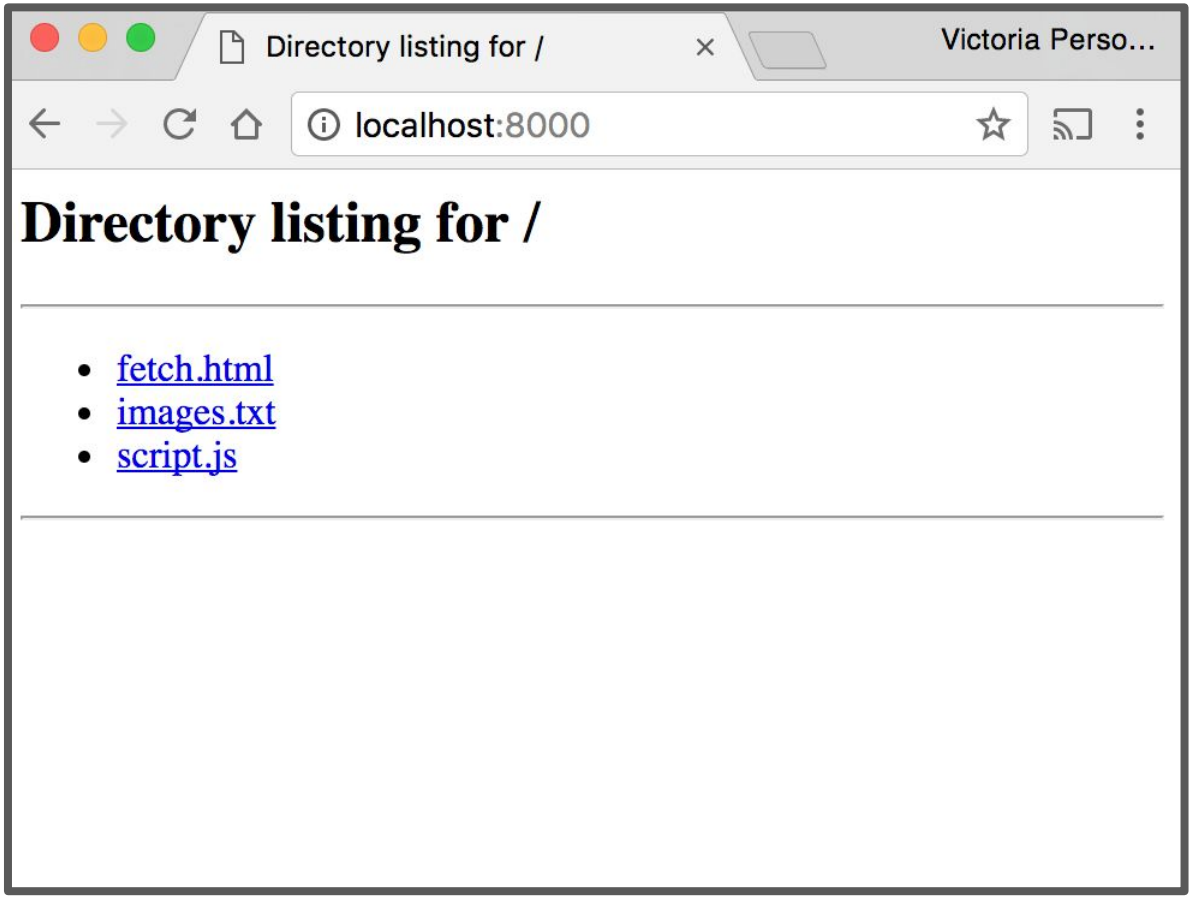
We can run a program to serve our local files over HTTP:

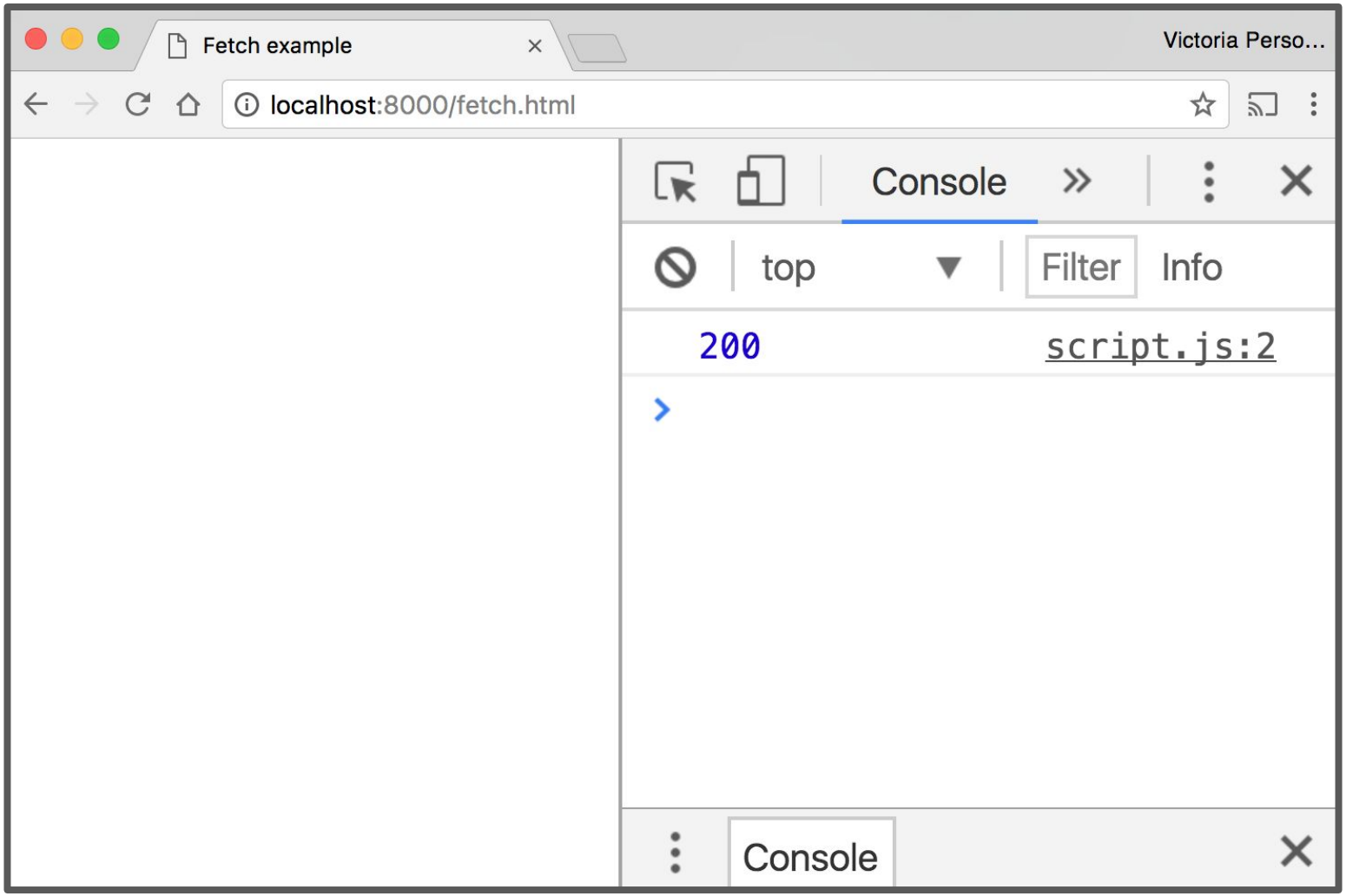
```
$ python -m SimpleHTTPServer
Serving HTTP on 0.0.0.0 port 8000 ...
```

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 ...
```





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?

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

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

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

```
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:

```
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

JSON.parse()

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

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

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

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);
```

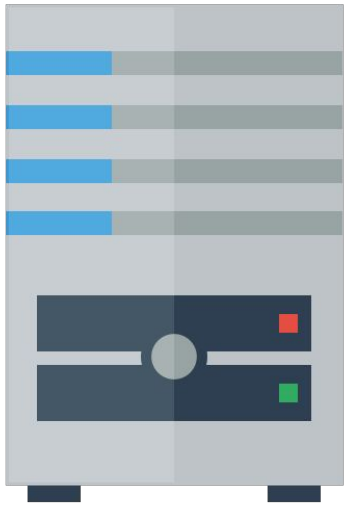
Querying REST APIs

First: Servers again

Servers

Sometimes when you type a URL in your browser, the URL is a **path to a file** on the internet:

- Your browser connects to the host address and requests the given file over **HTTP**
- The web server software (e.g. Apache) grabs that file from the server's local file system, and sends back its contents to you



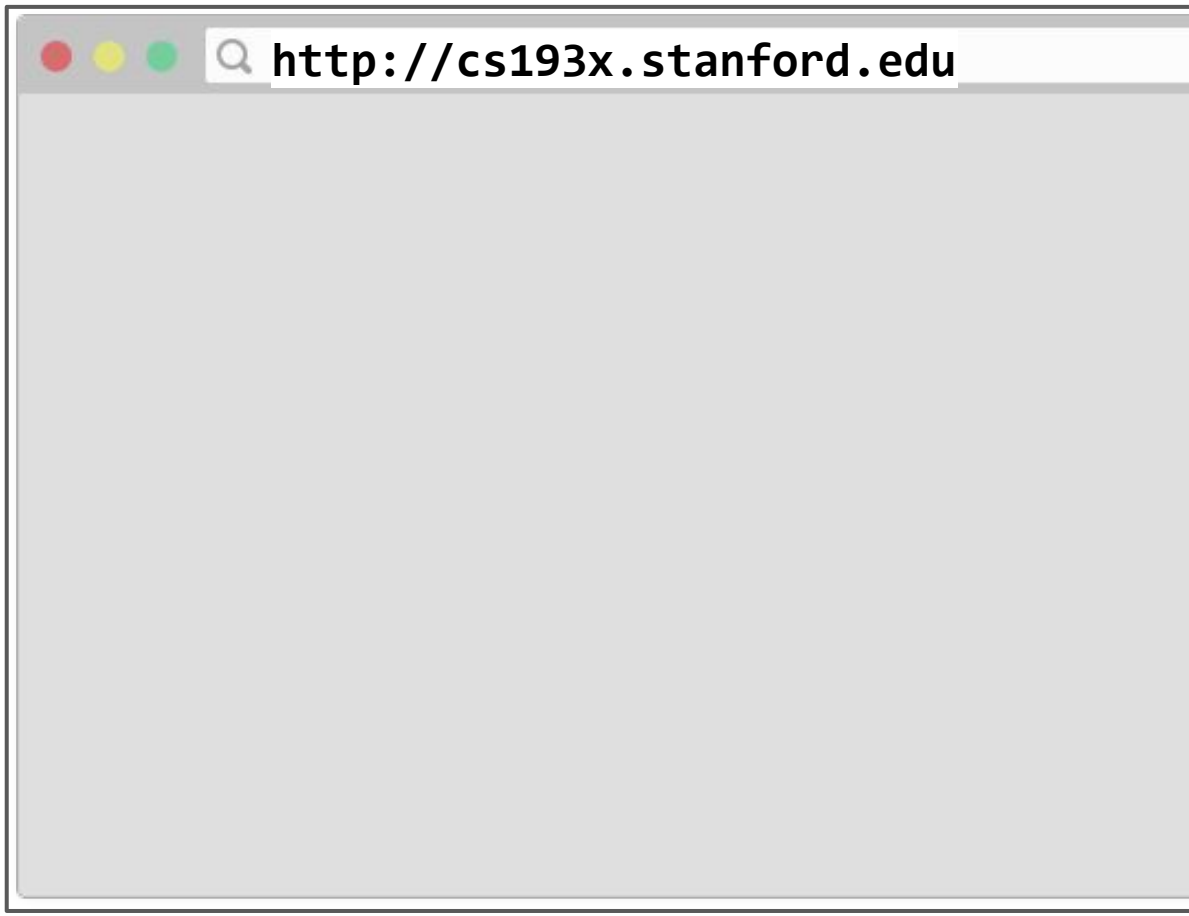
HTTP: Hypertext Transfer Protocol, the protocol for sending files and messages through the web

HTTP methods

HTTP Methods: the set of commands understood by a web server and sent from a browser

- **GET**: request/retrieve data
  This is request sent by the browser automatically whenever you navigate to a URL!
- **POST**: send/submit data
- **PUT**: upload file
- **PATCH**: updates data
- **DELETE**: delete data
- More HTTP methods

You type a URL in
the address bar and
hit "enter"



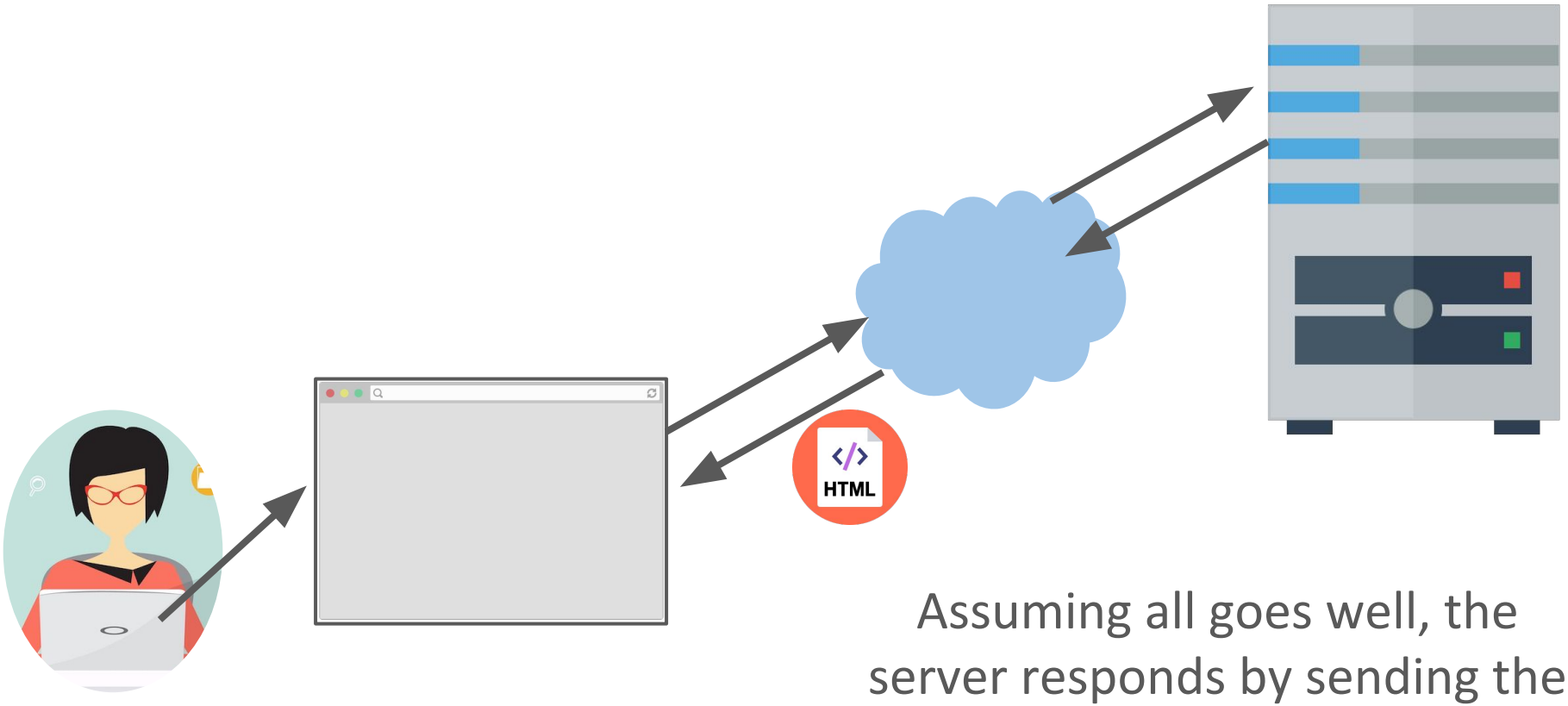
Server at
http://cs193x.stanford.edu

Browser sends an HTTP GET request saying "Please GET me the index.html file at http://cs193x.stanford.edu"

(**Warning:** Somewhat inaccurate, massive hand-waving begins now.

See this Quora answer for slightly more detailed/accurate handwaving)

Server at
http://cs193x.stanford.edu

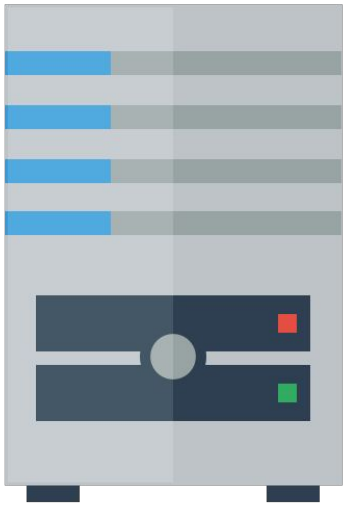


Assuming all goes well, the server responds by sending the HTML file through the internet back to the browser to display.

Servers

Sometimes when you type a URL in your browser, the URL is a **path to a file** on the internet:

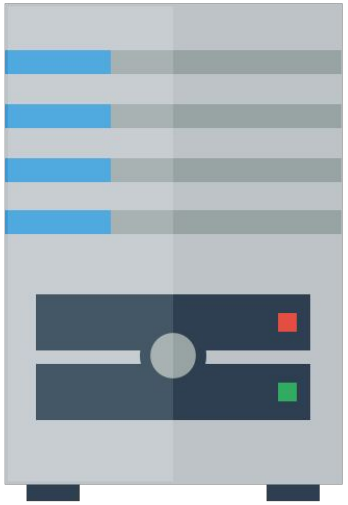
- Your browser connects to the host address and requests the given file over **HTTP**
- The web server software (e.g. Apache) grabs that file from the server's local file system, and sends back its contents to you



But that's not always the case.

Web Services

Other times when you type a URL into your browser, the URL represents **an API endpoint**, and not a path to a file.



That is:

- The web server does **not** grab a file from the local file system, and the URL is **not** specifying where a file is located.
- Rather, the URL represents **a parameterized request**, and the web server dynamically generates a response to that request.

API endpoint example

Look at the URL for this Google slide deck:

`https://docs.google.com/presentation/d/1Rim3-IXt6yN7yny_SBv7B5NMBiYbaQEiRMUD5s66uN8`

API endpoint example

Look at the URL for this Google slide deck:

`https://docs.google.com/presentation/d/1Rim3-IXt6yN7yny_SBv7B5NMBiYbaQEiRMUD5s66uN8`

- **presentation**: Tells the server that we are requesting a doc of type "presentation"
- **d/1Rim3-IXt6yN7yny_SBv7B5NMBiYbaQEiRMUD5s66uN8**: Tells the server to request a doc ("d") with the document id of "1Rim3-IXt6yN7yny_SBv7B5NMBiYbaQEiRMUD5s66uN8"

RESTful API

RESTful API: URL-based API that has these properties:

- Requests are sent as an **HTTP request**:
  - HTTP Methods: GET, PUT, POST, DELETE, etc
- Requests are sent to **base URL**, also known as an "**API Endpoint**"
- Requests are sent with a specified **MIME/content type**, such as HTML, CSS, JavaScript, plaintext, JSON, etc.

RESTful API

Almost every website on the internet uses RESTful URLs / RESTful APIs to handle requests to its servers.

Notable alternatives to REST:

- GraphQL,
  - Used by Facebook since 2012
  - Open-sourced by Facebook since 2015
  - Still early but some big clients: GitHub, Pinterest
- Falcor?
  - Netflix's REST alternative, introduced ~2015
  - Probably cool but never hear of anyone using it
  - Doesn't even have a Wikipedia page

Using REST APIs

3rd-Party APIs

Many websites expose REST APIs to outside developers.
These are often called "**3rd-party API**s" or "**Developer APIs**"

Examples:

- Spotify
- Giphy
- GitHub
- Hoards of Google APIs
- Facebook
- Instagram
- Twitter
- etc...

Try Googling
"<product name> API"
to see if one exists for
a given company!

Example: Spotify

Spotify has a REST API that external developers (i.e. people who aren't Spotify employees) can query:

Our Web API endpoints give external applications access to Spotify catalog and user data.

Web API Base URL: `https://api.spotify.com` User Guide | Tutorial | Code Examples

Search:

METHOD	ENDPOINT	USAGE	RETURNS
GET	/v1/albums/{id}	Get an album	album
GET	/v1/albums?ids={ids}	Get several albums	albums
GET	/v1/albums/{id}/tracks	Get an album's tracks	tracks*
GET	/v1/artists/{id}	Get an artist	artist
GET	/v1/artists?ids={ids}	Get several artists	artists
GET	/v1/artists/{id}/albums	Get an artist's albums	albums*

Example: Spotify

REST API structure (details):

- The **Base URL** is `https://api.spotify.com`
- The **HTTP method** is GET
- The **API endpoint** to query is:
  `https://api.spotify.com/v1/albums/`***<spotify_id>***
- It returns **JSON data** about the album that's requested

Web API Base URL: `https://api.spotify.com`

METHOD	ENDPOINT
GET	/v1/albums/{id}

Example: Spotify

If we had Spotify Album ID 7aDBFWp72Pz4NZEtVBANi9, how would we make a GET request for the album information?

REST API structure (details):

- The **Base URL** is `https://api.spotify.com`
- The **HTTP method** is GET
- The **API endpoint** to query is: `https://api.spotify.com/v1/albums/`***<spotify_id>***
- It returns **JSON data** about the album that's requested

GET request: Browse to URL

Loading a URL in a browser issues an HTTP GET request for that resource.

So if we just piece together this URL:

- **API Endpoint:** `https://api.spotify.com/v1/albums/`***<spotify_id>***
- **Album ID:** `7aDBFWp72Pz4NZEtVBANi9`
- **Request URL:** https://api.spotify.com/v1/albums/7aDBFWp72Pz4NZEtVBANi9

If you click on the link, you see it returns a JSON object.

GET request: `fetch()`

Actually, the `fetch()` API also issues an HTTP GET request by default.

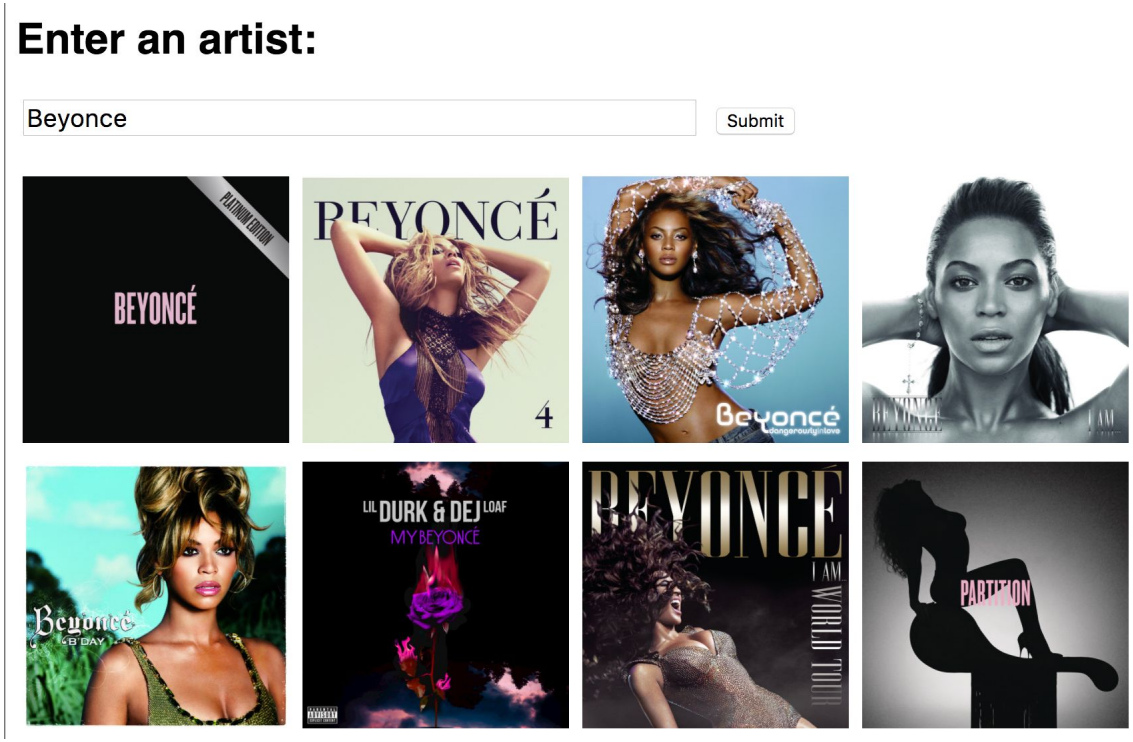
So if we do:

```
fetch('https://api.spotify.com/v1/albums/7aDBFWp72Pz4
NZEtVBANi9')
    .then(onResponse)
    .then(onTextReady);
```

...we can load the JSON data as a JavaScript object, as we did with our .json files!
(CodePen / demo)

Album example

Let's write a web page that asks the user to enter an artist's name, then displays the albums of that artist, as provided by the Spotify Search API. (live demo)



Spotify search API

Spotify Search URL:

https://api.spotify.com/v1/search?type=album&q=***query***

E.g.

https://api.spotify.com/v1/search?type=album&q=beyonce

Q: Hey, what's that at the end of the URL?

- ?type=album&q=beyonce

Query parameters

You can pass parameters to HTTP GET requests by adding **query parameters** to the URL:

```
?type=album&q=beyonce
```

- Defined as key-value pairs
  - ***param=value***
- The first query parameter starts with a **?**
- Subsequent query parameters start with **&**

Reminder: HTML elements

Single-line text input:

```
<input type="text" />
```

In JavaScript, you can read and set the input text via `inputElement.value`

Some other input types:

- Select
- Textarea
- Checkbox

Form submit

Q: What if you want the form to submit after you click "enter"?

Form submit

1. Wrap your input elements in a `<form>`

```
<form>
  <input type="text" id="artist-text" />
  <input type="submit" />
</form>
```

You should also use `<input type="submit">` instead of `<button>` for the reason on the next slide...

Form submit

2. Listen for the `'submit'` event on the form element:

```
const form = document.querySelector('form');
form.addEventListener('submit', this._onSubmit);
```

This is why you want to use `<input type="submit">` instead of `<button>` -- the 'submit' event will fire on click for but not `<button>`.

Form submit

3. Prevent the default action before handling the event through `event.preventDefault()`:

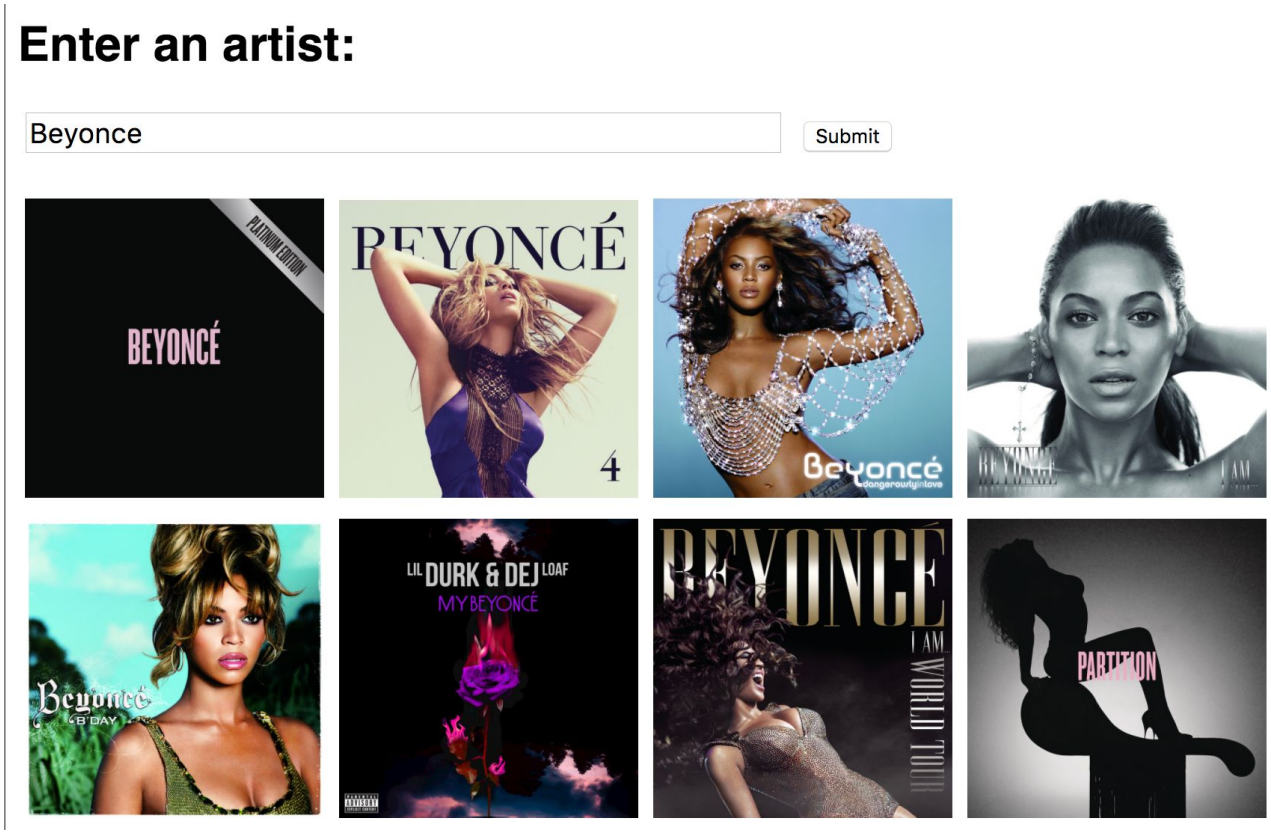
```
_onSubmit(event) {
  event.preventDefault();
  const textInput = document.querySelector('#artist-text');
  const query = encodeURIComponent(textInput.value);

  this.albumUrls = [];
  fetch(SPOTIFY_PATH + query)
      .then(this._onResponse)
      .then(this._onJsonReady);
}
```

The page will refresh on submit unless you explicitly prevent it.

Album example

Solution: [GitHub](#) / [Demo](#)



Chuck Norris API

The Cats API

Musixmatch lyrics API

https://api.chucknorris.io/jokes/random

Chuck Norris API

The Cats API

Musixmatch lyrics API

Chuck Norris API
The Cats API
Musixmatch lyrics API

Recap

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**