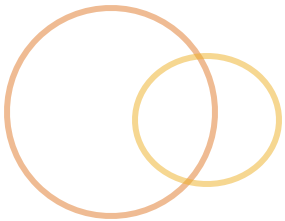
JS Fundamentals

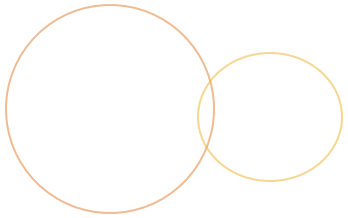
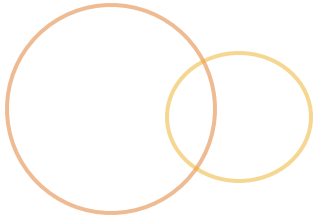
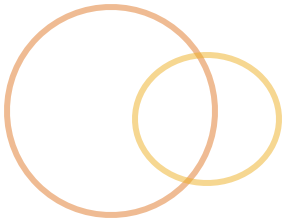
JavaScript in the Web

Ryan Morris
@mrmorris

What we'll cover



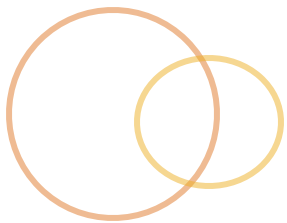
- HTML & CSS refresher
- The DOM
- Events
- Ajax/XHR
- Promises


refresher

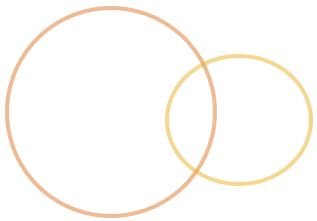
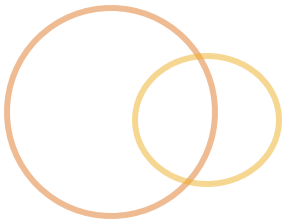
HTML

Wizard check



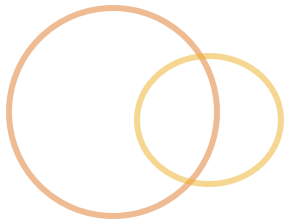
- OK with basic HTML?
- Can write a page in full?
- Write a **<form>** and all necessary input controls?
- Understand the difference between ** ** and ****?
- Understand the usage of **attributes** on elements
- When to use **id** versus **class**?

HTML


- **H**yper**T**ext **M**arkup **L**anguage
- Browsers allow support for all sorts of errors – html is very error tolerant
- Structure of the UI and "view data"
- Tree of element nodes
- HTML5
  - Rich feature set
  - Semantic
  - Cross-device compatibility
  - Easier!

Anatomy of a page



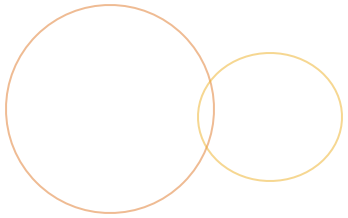
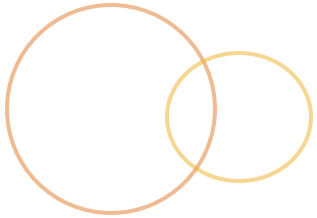
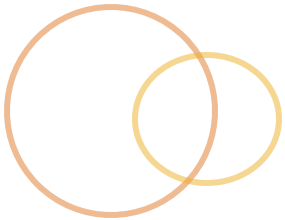
```
<!doctype html>
<html lang="en">
    <head>
        <meta charset="utf-8">
        …document info and includes…
    </head>
    <body>
        <h1>Hello World!</h1>
    </body>
</html>
```

Anatomy of an element

- <element attributeName="attributeValue">
  Content of element
  </element>
- Block vs inline
  -
  -
- Self closing elements
  - <input type="text" name="username" />

HTML Elements refresher

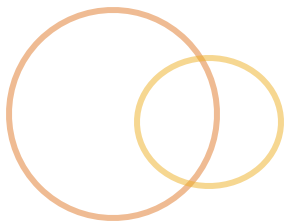
- **Structure**
  -
  -
  - <table>
    - <tr>, <td>, <thead>, <tbody>
  - <form>
    - <fieldset>, <label>, <input>, <select>, <textarea>
- **Content**
  - <h1> through <h6>
  - <p>
  - or <ul> (with<li>)
- **Text modifiers**
  - ,
- A list of elements:
  - https://developer.mozilla.org/en-US/docs/Web/HTML/Element


refresher

CSS

Wizard check



- OK with basic CSS selectors?
- Style a page in full?
- Select an element using CSS?
- Understand specificity?
- Got a few special pseudo-selectors under your belt?

Cascading Style Sheets

- Language for describing the look and formatting of the document
- Separates presentation from content

```
<!-- external resource -->
<link rel="stylesheet" type="text/css" href="theme.css">


<!-- inline block -->
<style type=“text/css”>
      span {color: red;}
</style>


<!-- inline -->
<span style="color:red">RED</span>
```

Anatomy of a css declaration

```
selectors {
    /* declaration block */
    property: value;
    property: value;
    property: val1 val2 val3 val4;
}
div {
    color: #f90;
    border: 1px solid #000;
    padding: 10px;
    margin: 5px 10px 3px 2px;
}
```

CSS Selectors

- By element
  - h1 {color:#f90;} <h1></h1>
- By id
  - #header {}
- By class
  - .main {}
- By attribute
  - div[name="user"] {}
- By relationship to other elements
  - li:nth-child(2) {} ***
***
  - p span {} ****
  - p > span {} ****

CSS Specificity

- Selectors apply styles based on its **specificity**
  - inline, id, pseudo-classes, attributes, class, type, universal
- **`!important`** allows you to override

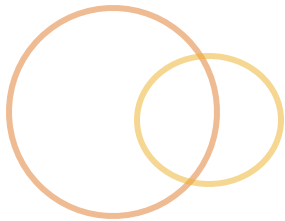
html:

```
<div id="main" class="fancy">
     What color will I be?
</div>
```

css:

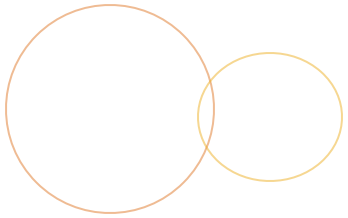
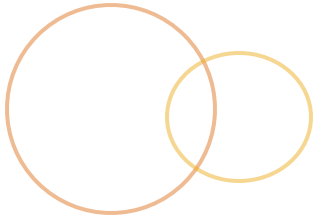
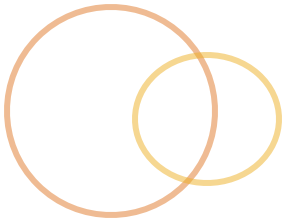
```
#main{
   color: orange;
}
.fancy{
   color: blue;
}
#main.fancy{
   color: red;
}
```

Warm Up



- **[just js] JavaScript Basics**
  - http://jsfiddle.net/mrmorris/a5v1p5by/
- **[dom + js] Input History**
  - http://jsfiddle.net/mrmorris/t2wazjmg/

Solutions:
JavaScript Basics: http://jsfiddle.net/mrmorris/11u4vmkL/
Input History: http://jsfiddle.net/mrmorris/0hvt7d9e/

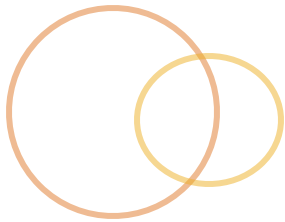

mini-module

LOADING JS IN THE BROWSER

Block and inline

- `Script blocks`
  - `<script>…</script>`
- Script resources
  - `<script src="filename.js"></script>`

Scripts are blocking

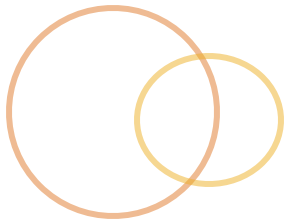


- Browse loads resources top down
- Browser will wait on js+css downloads
- DOM is not parsed until scripts are loaded
- So…
  - Defer your <script> load
  - Include at the bottom of `</body>`
    - It won't block & the DOM is loaded
  - Or leverage the DOMContentLoaded (ie9+) events

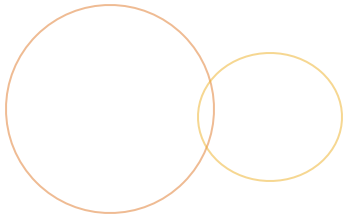
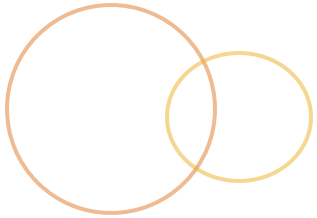
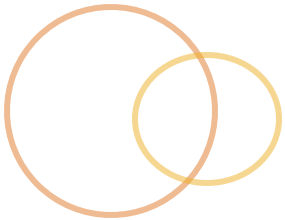
Resource order matters

```
<html>
  <head>
      <!– meta –>
      <!– essential scripts? –>
      <!– essential css/above-the-fold –>
  </head>
 <body>
      <!– all your html –>
      <!– non-essential css –>
      <!– scripts –>
 </body>
</html>
```

Selector Warmup



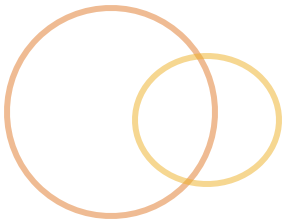
- CSS Diner
  - https://flukeout.github.io/


module

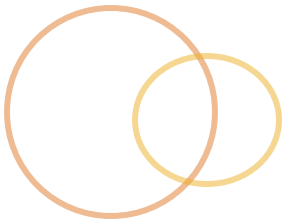
THE DOM

The DOM



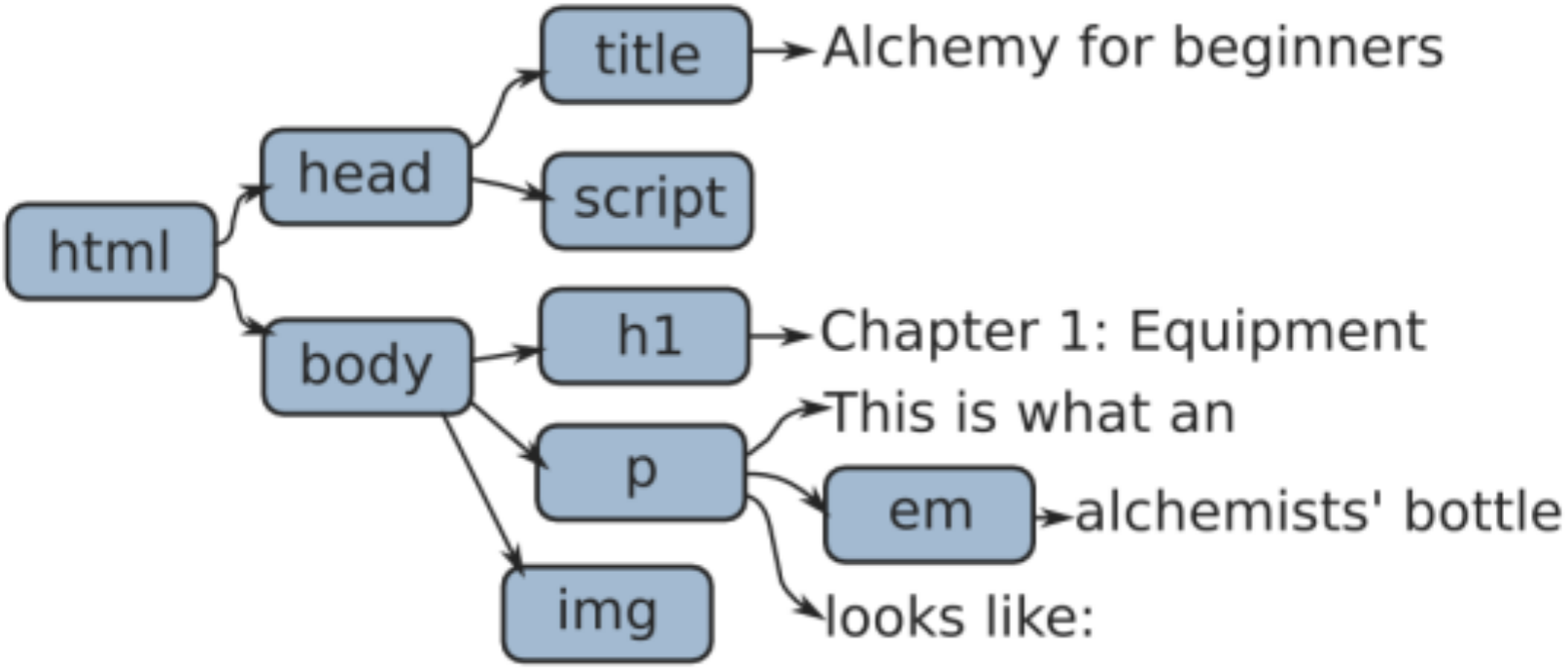
- **D**ocument **O**bject **M**odel
- What most people hate when they say they hate JavaScript
- The browser's API
  - JavaScript interface to the page
  - Browser parses our HTML and builds a model of the structure, then uses the model to draw it on the screen
- "Live" data structure

A simple document



```
<html>
  <head>
    <title>Alchemy for beginners</title>
    <script></script>
  </head>
  <body>
    <h1>Chapter 1: Equipment</h1>
    <p>This is what an <em>alchemist's bottle</em>
looks like:</p>
    <img src="bottle.png">
  </body>
</html>
```

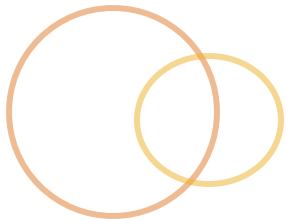
Document Structure



DOM Structure

- Global **`document`** variable gives us programmatic access to the DOM
  - It's a **tree-like** structure
  - Parent-Child relationships between nodes allow **traversal**
- Each node represents an **element** in the page, or **attribute**, or **content** of an element

Document Nodes



- HTML like:

```
<p id="name" class="hi">My text</p>
```

- Maps to an object like:

```
{
    …
    childNodes: NodeList[1],
    id: “name”
     className: "hi",
    innerHTML: "My text",
    id: "name",
    …
}
```

- HTML attributes map **very loosely** to object properties

Working with the DOM

- Access the element(s)
  - Select one
  - Select many
  - Traverse
- Work with the element(s)
  - Text
  - Html
  - Attributes

Accessing individual elements

- Starting at **`document`**

```
// returns first element with given id
.getElementById("main");
// <div id="main">Hi</div>


// returns first matching css selector
.querySelector("p span");
// <p><span>Me!</span><span>Not!</span></p>
```

http://jsfiddle.net/mrmorris/wcff257b/

Accessing element lists

- ... or a previously selected element

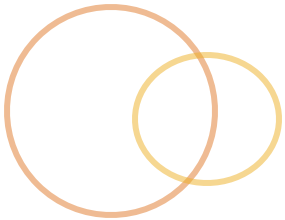
```
.getElementsByTagName("a");
// all <a> elements
.getElementsByClassName("fancy");
// all elements with specified class
// <span class="fancy"></span>
.querySelectorAll("p span");
// all elements that match the css selector
// <p><span>Me!</span><span>Me!</span></p>
```

Node Types

- Nodes can be of different types, we are mostly concerned with element nodes…
  - `anElement.nodeType`

```
// 1 = Element
// 3 = Text node
// 8 = Comment node
// 9 = Document node
```

Node Content



- Text node content
  - `textNode.nodeValue`
- Element node content
  - `el.textContent`
  - `el.innerText`
  - `el.innerHTML`

Node Attributes

- Accessor methods

```
el.getAttribute("title");
el.setAttribute("title", "Hat");
el.hasAttribute("title");
el.removeAttribute("title");
```

- As properties
  - `.href`
  - `.className`
  - `.id`
  - `.checked`

http://jsfiddle.net/mrmorris/duopdjdb/

Traversal

- Move between nodes via their relationships
- Element node relationship properties
  - `.parentNode`
  - `.previousSibling, .nextSibling`
  - `.firstChild, .lastChild`
  - `.childNodes // NodeList`
- But… mind the whitespace!

http://jsfiddle.net/mrmorris/dv13y28m/

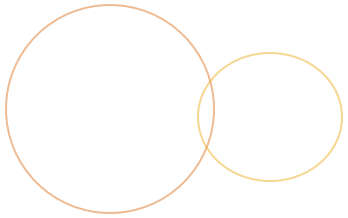
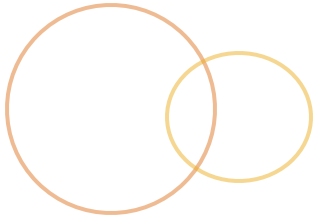
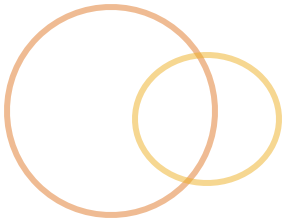
Modern Element Traversal

- Old traversal methods get tripped up by text-nodes, line breaks and whitespace
- New methods avoid that
  - *Supported in ie9+*
- From an element node
  - **`.children`**
  - **`.firstElementChild, .lastElementChild`**
  - **`.childElementCount`**
  - **`.previousElementSibling`**
  - **`.nextElementSibling`**

Lab - Selection & Traversal Practice

- Start your local server, then visit:
  - http://localhost:3000/exercises/dom/
- Using the different selection methods, select:
  - The **<header>**
  - The **<nav>**
    - Then use traversal to select the **<li>**s within the **<nav>**
  - All **paragraphs**
    - Log out the "`innerHTML`" property of the **first** paragraph
  - Done? Experiment with traversal

Tip: Keep references to your selections with variables!


module

DOM MANIPULATION

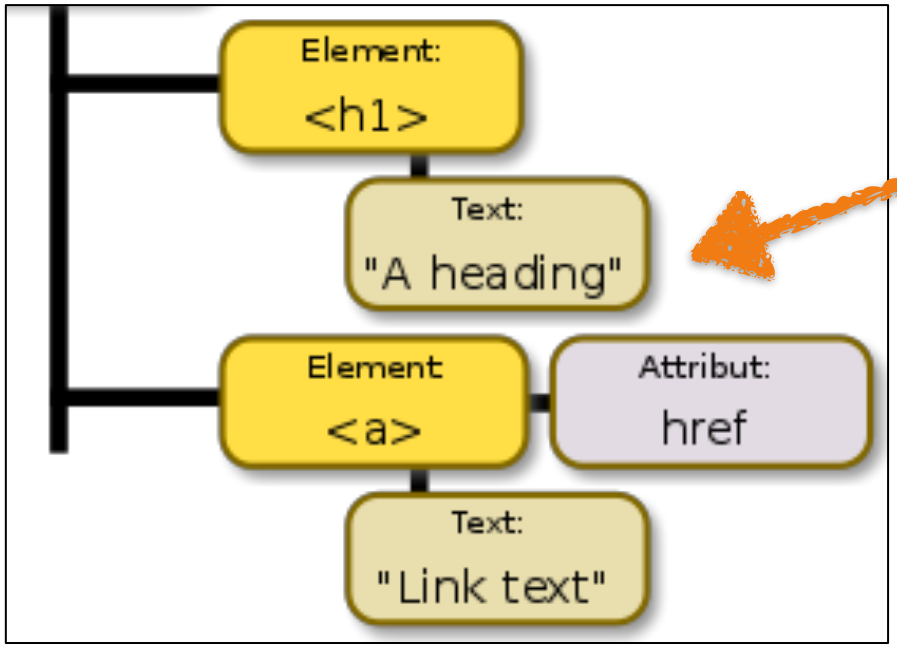
Adding content

1. Create the container node
   - Insert additional content node(s)
   - Insert text node(s) if working with text
2. Determine *which pre-existing* node you can use to insert the *new* node
3. Insert it into the DOM (append, prepend, insert, replace)

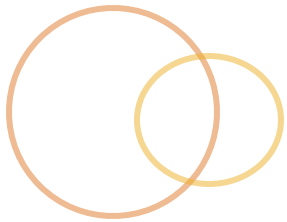
http://jsfiddle.net/mrmorris/ktwdye0w/

Creating new nodes

- **`document.createElement("div")`**
  - creates and returns a new node without inserting it into the DOM
- **`document.createTextNode("foo bar")`**
  - creates and returns a new text node with given content



Set element content



- el.**textContent**
  - text content of node and all children
- el.**innerHTML**
  - html content of node and all children
- el.**nodeValue**
  - text, comment, attribute node values
- el.**value**
  - form input values

Adding nodes to the tree

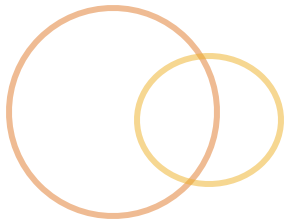
```
// given this set up
var parentEl = document.getElementById("users"),
    existingChild = parentEl.firstElementChild,
    newChild = document.createElement("li");


parentEl.appendChild(newChild);
// appends child to the end of
parentEl.childNodes


parentEl.insertBefore(newChild, existingChild);
// inserts newChild in parent.childNodes
// just before the existing child node
```

Moving and removing nodes

- Tree is “live”
  - **Selection then insertion** will **move** the element
  - **Removal** will **detach** it immediately

```
parentEl.replaceChild(newChild, existingChild);
// removes existingChild from parent.childNodes
// and inserts newChild in its place


parentEl.removeChild(existingChild);
// removes existingChild from parentEl.childNodes
```

Styling elements



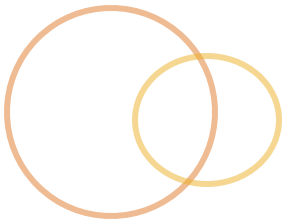
- Use element's "`style`" property
  - It's an object of style properties

```
el.style.color = "black";
el.style.marginLeft = "50px";
```

  - Some style names differ in JavaScript
  - Hyphens become camelCase
    - `background-color => backgroundColor`
  - Some names were keywords
    - float => cssFloat

http://jsfiddle.net/mrmorris/hJwCj/

ClassList API



- Ability to get, set and toggle classes on element(s)

```
el.classList.add("class");
el.classList.remove("class");
el.classList.toggle("class");
el.classList.contains("class");
```

DOM Performance

- DOM interaction comes with performance costs
  - *Searching*
  - *Accessing*
  - *A*nything that triggers a "*redraw*"
- How to address this:
  - Store a reference rather than re-selecting
  - Reduce the number of insertions; build up a set and do it in bulk

DOM basics - Recap

- The **DOM** is a model of the web page document.
- Browsers offer a **JavaScript API** to interact with the DOM
  - You can access, manipulate, create any content
- ***jQuery*** is a lib that serves as an abstraction of the DOM
- Pay attention to DOM **performance** issues

Exercise: Find the flags

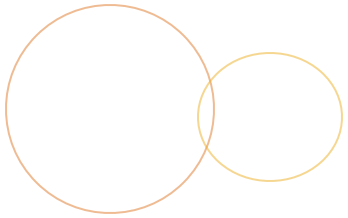
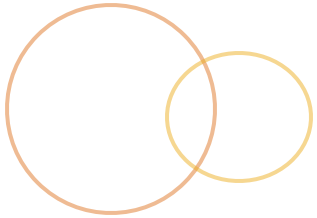
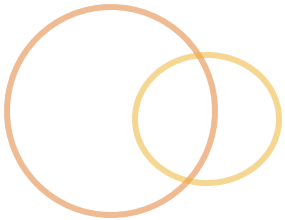
- Open the following file:

  `public/exercises/flags/flags.js`

- Complete the exercise
- Run the tests by visiting in your browser:

  `http://localhost:3000/exercises/flags/`

Solutions:
https://github.com/rm-training/web-dev-bc/tree/master/public/solutions/flags


module

EVENTS

JavaScript Programming Model

- JavaScript engine has an **single-threaded**, **event-driven**, **asynchronous** programming model
  - Single-threaded
    - One script runs top to bottom
    - Blocking!
  - Event-driven
    - Flow of the program is determined by events
    - Events happen and we can subscribe (listen) to them
  - Asynchronous
    - You can schedule future behavior
    - A block of code can run *later*
    - ~~Multiple operations can run at the same time~~

JS is ***still*** single-threaded

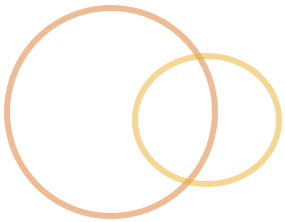
Event-driven

- As things happen
  - A user clicks a link
  - or a page completes loading
  - or a form is submitted
- Events are fired
  - click
  - or load
  - or submit
- Which triggers functionality
  - *On click change my color to blue*

So many events…

- **UI**
  - load, unload, error, resize, scroll
- **Keyboard**
  - keydown, keyup, keypress
- **Mouse**
  - click, dblclick, mousedown, mousemove mouseup mouseover, mouseout
- **Focus**
  - focus, blur
- **Form**
  - input, change, submit, reset, select, cut, copy, paste

Basic Event Handling



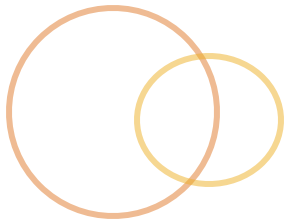
1. Select an element
   - The element that triggers the event
   - or element that event passes through
2. Determine which event you want to listen for
3. Define an *event handling function* to respond to the event when it occurs

Event Handling

- Use the **addEventListener** method to register a function to be called when an event is triggered
  - ie9+

```
var el = document.getElementById("main");

el.addEventListener("click", function(event) {
 console.log("Clicked!");
});
```

Handler options



- Inline

```
<p onclick="function(e){}"><p>
```

All handlers are passed an "event" object as the first argument

- Traditional DOM event handlers

```
el.onclick = function(e){}
```

- Event listeners (ie9+)

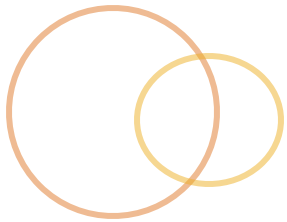
```
el.addEventListener(event, function [, flow]);
el.removeEventListener(event, function);
el.attachEvent(); // ie8- only
```

Event handler context

- Functions are called in the context of the DOM element

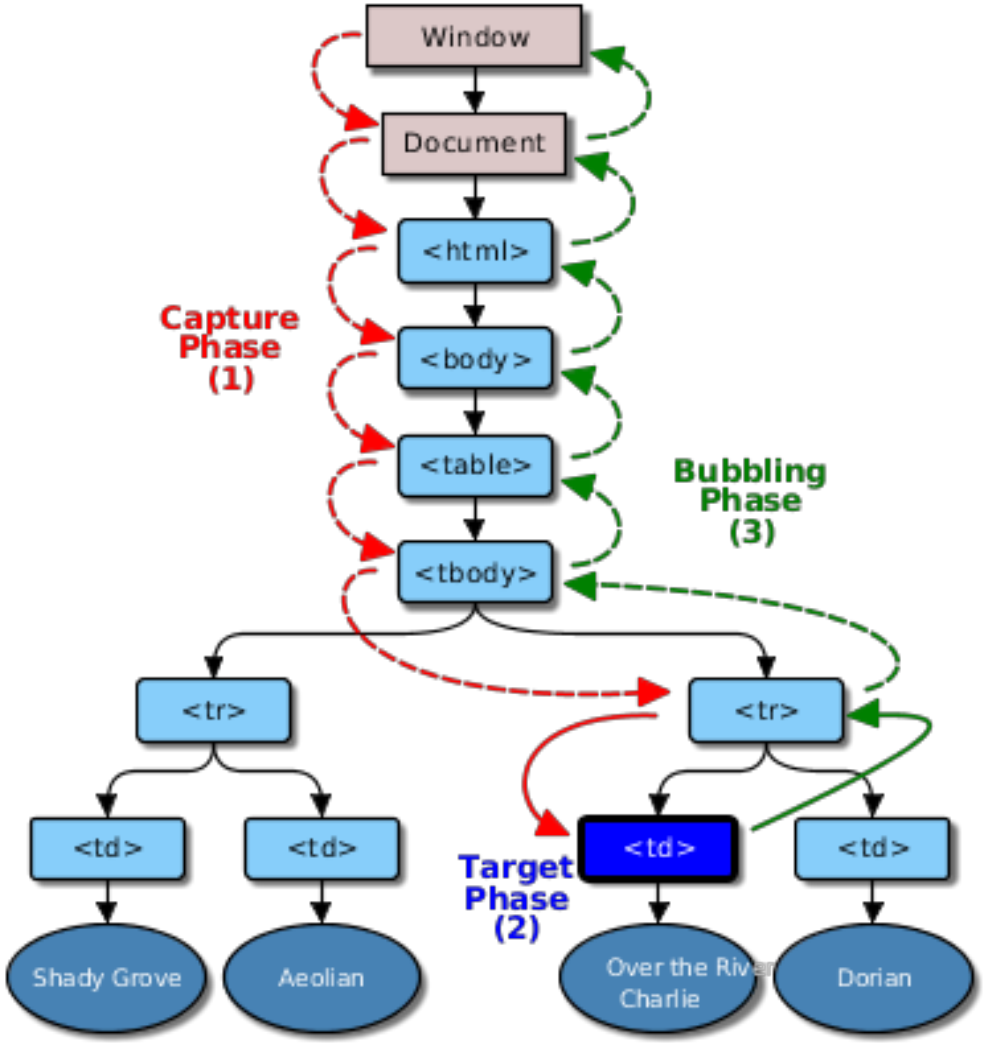
```
el.addEventListener("click", myHandler);


function myHandler(event) {
  this; // equivalent to el
  event.target; // what triggered the event
  event.currentTarget; // where handler is bound
}
```

Event Propagation

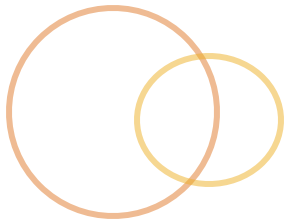


- An event triggered on an element is also triggered on all “ancestor” elements
- Two models
  - Trickling, aka Capturing (Netscape)
  - Bubbling (MS)

Event Propagation



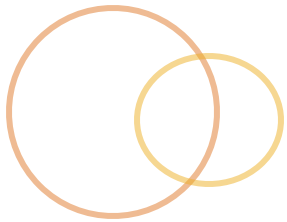
Controlling Events



- Event handlers can affect propagation

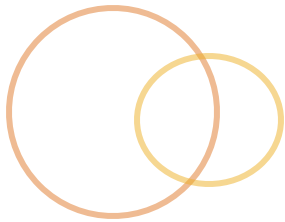
```
// no further propagation
event.stopPropagation();

// no browser default behavior
event.preventDefault();

// no further handlers
event.stopImmediatePropagation();
```

The event object



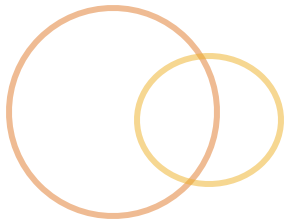
- Handlers are passed event object with lots of info about the event/user
  - Event.screenX
  - Event.screenY
  - Event.pageX
  - Event.pageY
  - Event.clientX
  - Event.clientY
- Key events include a "`keyCode`" property
- http://jsfiddle.net/mrmorris/8htsexcg/

Complete example

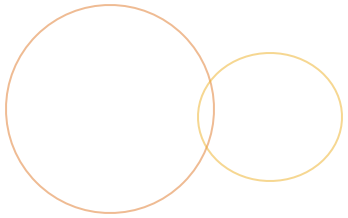
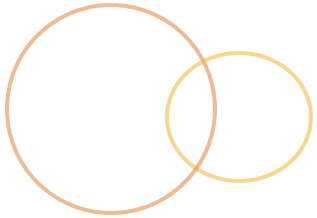
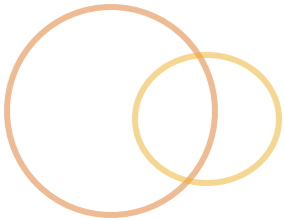


```
const el = document.getElementById("some-id");
el.addEventListener("click", function(event) {

        // “this” represents the element
        // handling the event
        this.style.color: "#ff9900";

        // “target” represents the element
        // that triggered
        event.target.style.color: "#ff9900";

        // you can stop default browser behavior
        event.preventDefault();

        // or you can stop the event from bubbling
        event.stopPropagation();

});
```

Debugging - Events



- View Event Listeners registered in the page
  - Event Listeners Panel
  - getEventListeners(document)
- Monitor events on an element
  - monitorEvents(node, eventType);
  - unmonitorEvents(node);

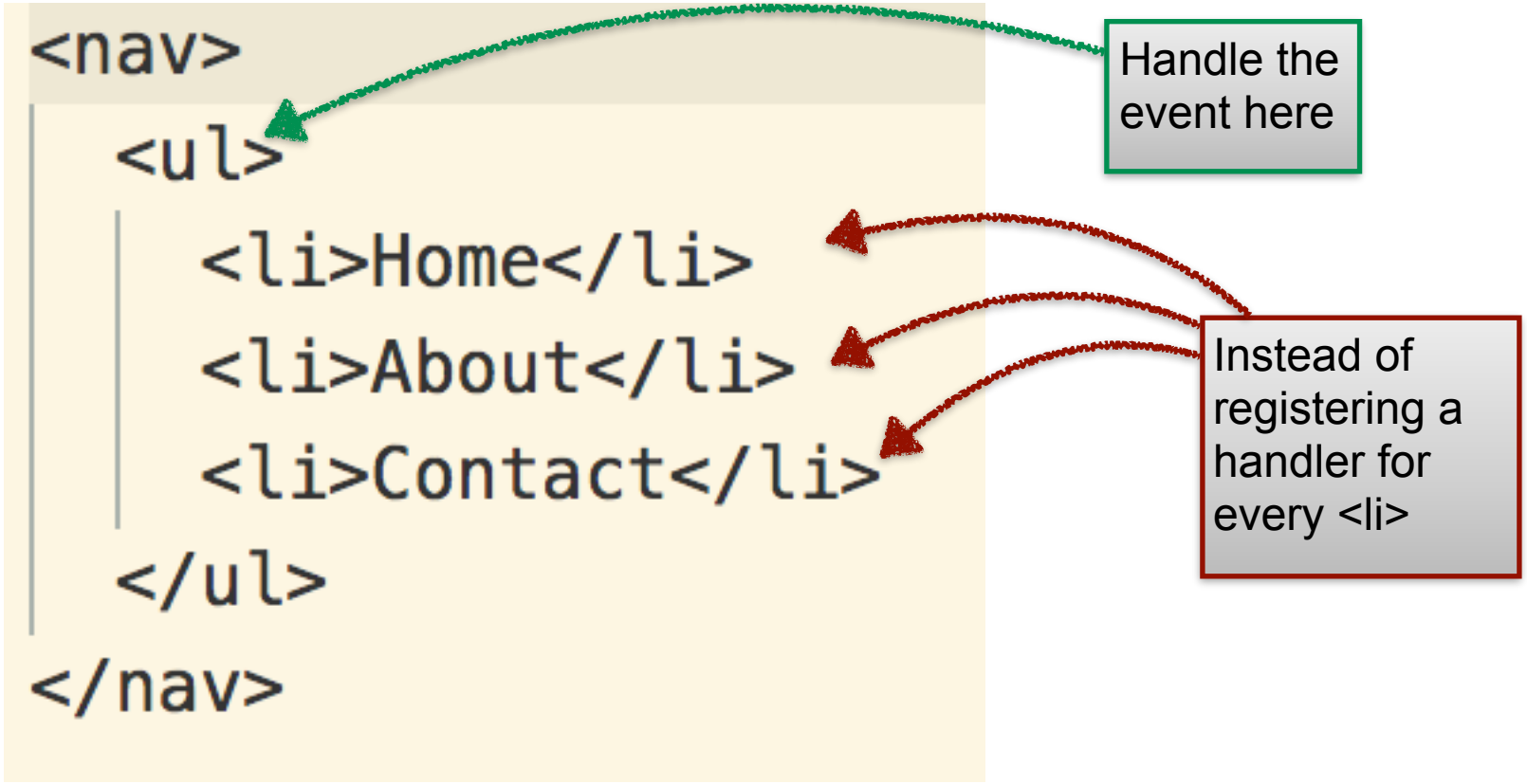





events

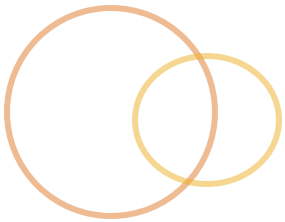
DELEGATION

Event Delegation

- When a parent element is responsible for handling an event that bubbles up from its children



Event Delegation

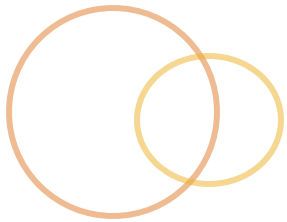


- Why delegate?
  - New child content can be added w/out a new handler
  - Fewer handlers registered, easier on memory
- Relies on some event object properties
  - `target`, which references the originating node of the event
  - `currentTarget` property refers to the element currently handling the event (where the handler is registered)

Example: Event Delegation

```
document
  .querySelector("ul")
  .addEventListener("click", myLiHandler);

function myLiHandler(event) {
  if (e.target && e.target.nodeName == "LI") {
    console.log(
      e.target.innerHTML, " was clicked!"
    );
  }
}
```

Exercise: Events



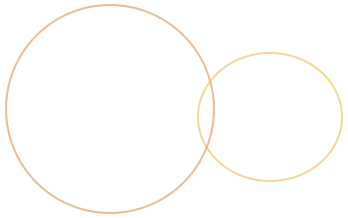
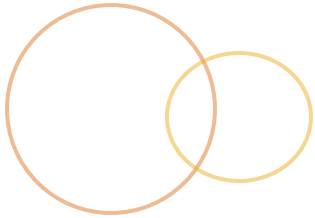
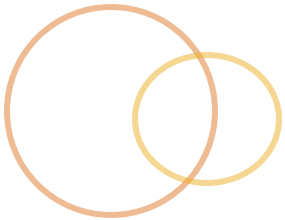
- Open the following file:

  `public/execises/events/events.js`

- Complete the exercise
- Run your tests by visiting in your browser:

  http://localhost:3000/exercises/events/

Solutions:
https://github.com/rm-training/web-dev-bc/tree/master/public/solutions/events


module

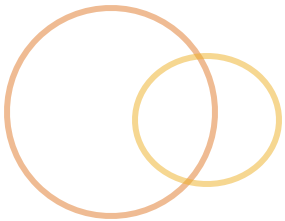
AJAX/XHR

AJAX/XHR

- Interface through which browsers can make HTTP Requests
- Handled by the `XMLHttpRequest` object
- Introduced by Microsoft in the 90s for ie, taken from there…

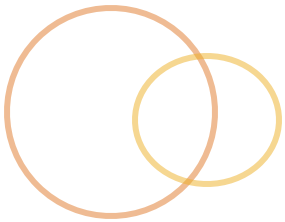
- ...There is a new `fetch()` API in ESNext
  - Not widely supported, lacks some features
  - Polyfill: https://github.com/github/fetch
  - https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API

AJAX/XHR



- Why use it?
  - Non-blocking
  - Dynamic page content/interaction
  - Supports many formats
- Limitations
  - Same-origin policy
  - History management

XHR– Step by step

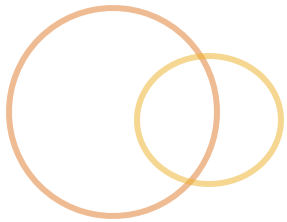


1. Browser makes a request to a server
2. And the script continues along it's merry way

…some time later…

3. the server responds in xml/json/html
4. Browser parses and processes response
5. Browser invokes our JavaScript callback

Making the request



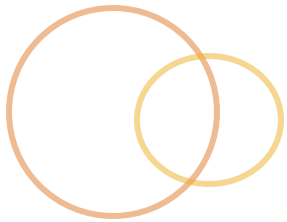
```
// create the request object
var req = new XMLHttpRequest();

// ...todo: attach listener (next slide)...

// initialize the request
req.open("GET", "url.json");

// set header (after open but before send)
// defaults to Accept */*
req.setRequestHeader("Accept","application/json");

// then send it!
req.send(null);
```

Handle the response

- “load” event will fire when response is received
- Request object will have `responseText` and `status`

```
req.addEventListener("load", function(e) {
      // HTTP status codes
      if (req.status == 200) {
            console.log(req.responseText);
      }
});
```

XHR Example

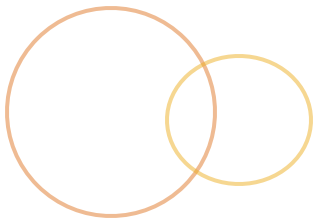
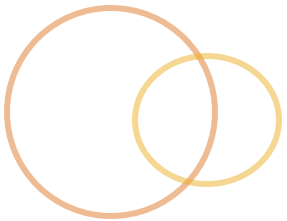


- Loading content from a weather API
  - http://jsfiddle.net/mrmorris/cfwa8v92/

Data formats

Format	Summary	PROS	CONS
HTML	Easiest for content in page	• Easy to parse • No need to process much	• Server must produce the HTML • Data portability is limited • Limited to same domain
XML	Looks similar to HTML, more strict	• Flexible and can handle complex structure • Processed using the DOM	• Very verbose, lots of data • Lots of code needed to process result • Same domain only
JSON	Similar object literal syntax	• concise! Small • Easy to use within JavaScript • Any domain, w/ JSONP or CORS	• Syntax is strict • Can contain malicious content since it can be parsed as JavaScript

JSON


- **J**ava**S**cript **O**bject **N**otation
- Most commonly used web data communication format
- Like an object literal, except:
  - Property names must be surrounded by double quotes
  - No function definitions, function calls or variables
- Methods
  - `JSON.stringify(object);`
  - `JSON.parse(string);`

JSON

```
{
  name: “Jason”,
  trophies: [
    “trophy1”,
    “trophy2”
  ],
  sayHi: function() {
    console.log(‘hi’);
  }
  age: user.age,
  car: {
    name: “toyota”,
    year: 1985
  }
}
```

to json

```
{
  “name”: “Jason”,
  “trophies”: [
      “trophy1”,
      “trophy2”
  ],
  “age”: 40,
  “car”: {
      “name”: “toyota”,
      “year”: 1985
  }
}
```

XHR with JSON

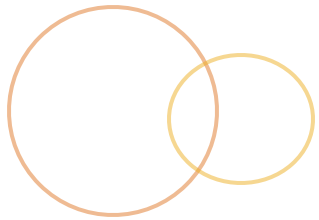
- It is sent and received as a string and will need to be de-serialized

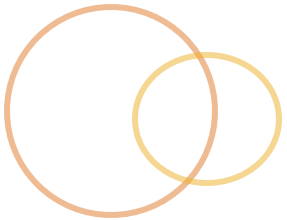
```
var data = JSON.parse(xhr.responseText);
var newContent = "";

for (var i=0; i< data.length; i++) {
  newContent += "<div class='event'>";
  newContent += "<img src='" + data[i].val+ "'/>";
}

document
 .getElementById('content')
 .innerHTML = newContent;
```

Cross-origin

- By default, ajax requests must be made on the same domain
- Alternatives to this are:
  - A proxy file on the server
  - JSON/p “Json with padding”
  - CORS (Cross-origin resource sharing), which involves new http headers between browser and server – ie10+
- For later: http://jsonplaceholder.typicode.com/

CORS





- Cross-Origin Resource Sharing
- A set of headers sent by the requesting client (XHR) and the responding server that can negotiate whom can request what from where
- Caveats
  - Supports **all** HTTP verbs
  - Usable with XMLHttpRequest
  - Simple in theory, complex in practice

XHR Recap

- A means for the browser to make additional requests without reloading the page
- Enables very fast and dynamic web pages
- Best with small, light transactions
- JSON is the data format of choice
- Requests across domains are possible but require jumping through some extra hoops (and your server must support it)

Exercise: Making Ajax Requests

- Open the following file:

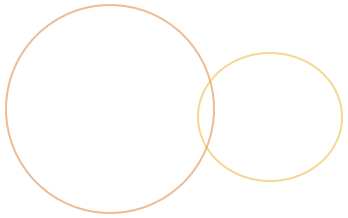
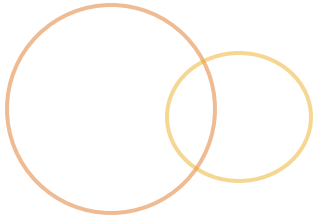
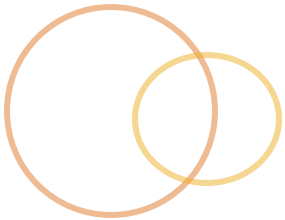
  `public/exercises/ajax/ajax.js`

- Complete the exercise
- Run it by visiting in your browser:

  http://localhost:3000/exercises/ajax/

- We'll use a public API that supports CORS
  - https://jsonplaceholder.typicode.com/posts
  - ...Or we could use a local, fake API...

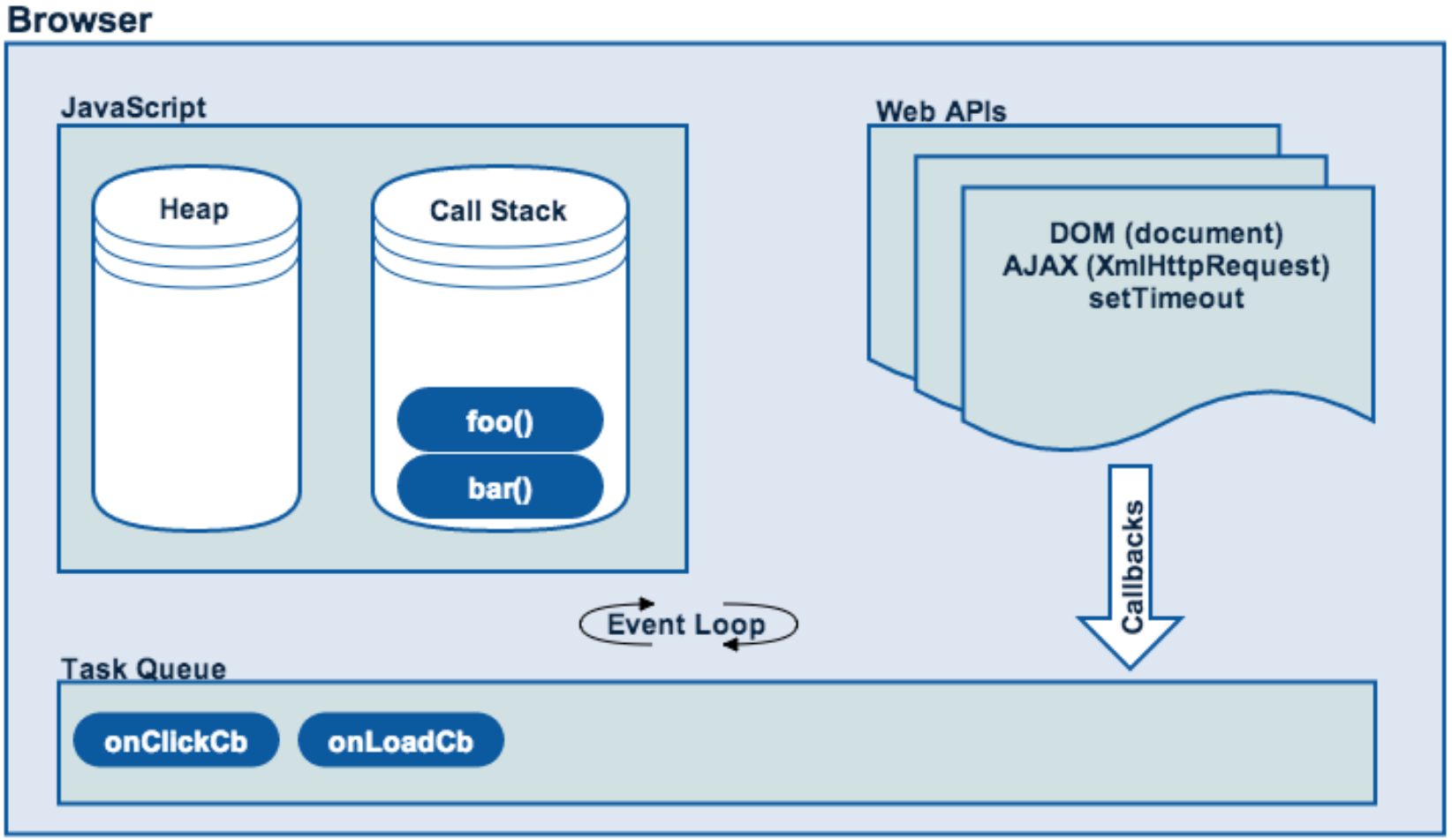
Solutions:
https://github.com/rm-training/web-dev-bc/tree/master/public/solutions/ajax

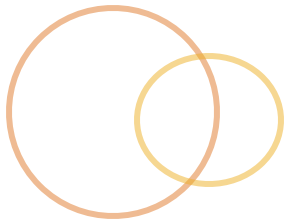

module

ASYNCHRONOUS PROGRAMMING

Single-threaded JavaScript



Being Asynchronous



- Because JavaScript cannot do more than one thing at a time…
  - Callbacks
  - Promises
  - [ES6] `async` and `await`
  - Observables

Callback Pattern

- A function passed to another function as a parameter
  - …so that it can be invoked later by the calling function.
- Aren't asynchronous on their own
  - …but we tend to use them for such things
  - ex: event handling, ajax handling, file operations, etc

```
function callLater(fn) {
  // do some async work
  return fn();
}

callLater(function() {
  console.log("I'm done!");
});
```

Callback Context

- **`this`** inside a callback may change, be careful

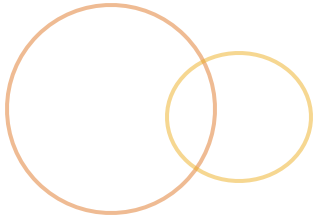
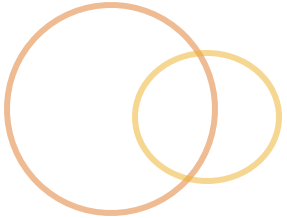
```
setTimeout(function() {
    console.log("I was called later");
}, 1000);

$("a").on("click", function() {
    console.log(this); // ?
});
```

The Downside to Callbacks

- Can become deeply nested and not easy to reason
- There is no guarantee that the callback will be invoked

```
// callback hell
async1(function(err, result1) {
    async2(function(err, result2) {
        async3(function(err, result3) {
            async4(function(err, result4) {
                /*…*/
            });
        });
    });
});
```

Promises


- A **Promise** represents a proxy for a value
- They represent the *promise of future value*
- They still use callbacks under the hood

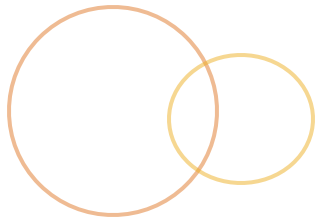
Event handler/callback

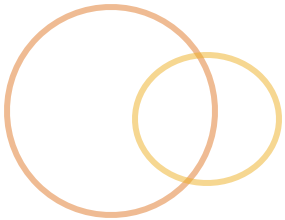
```
xhr.addEventListener('load', function(data) {
  // do something with the data
});
```

Promise callback

```
const prom = getData();
prom.then(function(data) {
  // do something with the data
});
```

Promises





- **Benefits:**
  - Guarantees that callbacks are invoked
  - Composable (can be chained)
  - Immutable (one-way latch)
  - You can continue to use them after resolved
  - They are objects you can pass around
- **Bummers:**
  - ES6+
  - No `.finally()`

Making Promises

- Construct a Promise to represent a future value
- Constructor expects a single argument:
  - A function with **`fulfill`** and **`reject`** functions

```
var promise1 = new Promise(function(fulfill, reject) {
      // likely will use an async operation here
      setTimeout(function(err, data) {
            if (err) {
                  reject(err);
            } else {
                  fulfill(data);
            }
      }, 1000);
});
```

As the ***maker*** of this promise, YOU define what success (fulfill) and failure (reject) are

Using Promises

- When you have a promise, you can attach functionality that will run either when the promise **fails** or **succeeds**
- Attach handlers using **`then`** method
  - When promise is resolved it's "then" is called

```
const onFulfilled = function(data) {
  console.log("We got data!", data);
};
const onRejected = function(err) {
  console.log("Error happened", err);
};

promise.then(onFulfilled, onRejected);
```

Promises Terminology

- Specification: https://promisesaplus.com
  - **pending** – the action is not fulfilled or rejected
  - **fulfilled** – the action succeeded
  - **rejected** – the action failed
  - **settled** – the action is fulfilled or rejected

```
var p = new Promise(
  function(resolve, reject){
  ...
  if(something)
     resolve({});
  else{
     reject(new Error());
  }
})
```

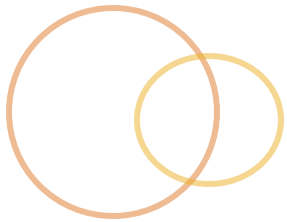
```
p.then(
  function(data){
    ...
  },
  function(err){
    ...
  }
);
```

Promise Errors

- Use the reject/error handler argument in `then()`
- ES6 Promises also support a `.catch()` callback, which will do the same thing.

```
prom.then(null, function(error){
   console.log("Something went wrong", error);
});


prom.catch(function(err) {
    console.log(err);
});
```

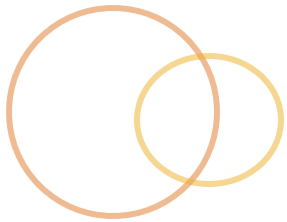
Chaining Promises



- **`.then()`** always wraps any return value as a ***new* Promise**
  - We can chain then() to create asynchronous sequences
- You can also ***specify a new promise*** to return

```
// when promise 1 completes...
promise1.then(function(data){
    console.log(data); // 5
    return data + 2; // returns a new promise
}).then(function(data) {
    // after promise 2 ^
    console.log(data); // 7!
}).catch(function(err) {
    // if anything goes wrong
    console.log(err);
});
```

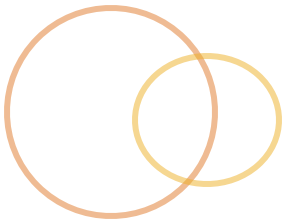
Fixing callback hell



- Remember this? Let's see what that would look like if we wrapped each async operation in a promise

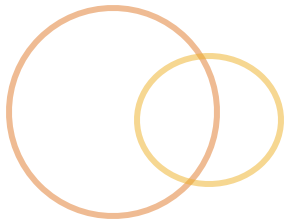
```
async1(function(err, result1) {
     async2(function(err, result2) {
          async3(function(err, result3) {
          });
     });
});
```

Promised Land



```
prom1 // when prom1 resolves
  .then(function() {
    // ...
    return prom2;
  })
  .then(function() {
    // ...
    return prom3;
  })
  .catch(function(err) {
     // deal with thrown error
  });
```

Promise breaking



- What is wrong with the below promise sequence?

```
fetchResult(query)
    .then(function(result) {
        // this is an async operation
        $.ajax(result.id);
    })
    .then(function(newData) {
        console.log(newData);
    });
    .catch(function(error) {
        console.error(error);
    });
```

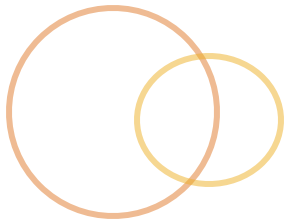
Promise breaking

- What is wrong with the below promise sequence?

```
fetchResult(query)
    .then(function(result) {
        // this is an async operation
        return $.ajax(result.id);
    })
    .then(function(newData) {
        console.log(newData);
    });
    .catch(function(error) {
        console.error(error);
    });
```

This is asynchronous so we **should** pass the new promise back (for the next)

Composing Promises



- `Promise.all([…])`
  - Returns a promise that resolves when all promises passed in are resolved or at the first rejection
  - Fulfilled value is an array of all returned promise values
- `Promise.race([…])`
  - Returns a promise that resolves when any one promise is fulfilled or rejected

Composing Promises Example

```
var p1 = Promise.resolve(3);
var p2 = 1337;
var p3 = new Promise(function(resolve, reject) {
    setTimeout(resolve, 1000);
});

Promise.all([p1,p2,p3]).then(function(data) {
    console.log(values); // ?
});

Promise.any([p1,p2,p3]).then(function(data) {
    console.log(data); // ?
});
```

Async and await [ES6]

- Two new keywords allow us to write asynchronous code that looks and feels synchronous
- **`async function`**
  - Defines an asynchronous Function that can **yield flow of control** back to the caller
  - The function immediately returns a Promise that will be resolved when the function returns a value or rejected when it has an error
    - The function is resolved with any return value
    - Errors with any error thrown
- **`await`**
  - Informs code *within* an async function to yield/wait for an *internal* Promise to resolve before proceeding

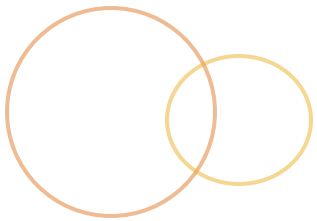
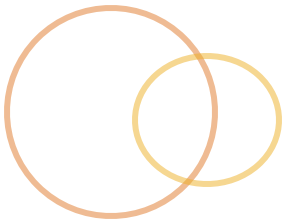
From this…

```
function getAndRenderArtists() {
  let artists;
  Ajax.get("/api/artists/1")
   .then(function(data){
      artists = data;
      return Ajax.get("albums");
    })
    .then(function(data){
      artists.albums = data;
      View.set("artist", artist);
    })
    .catch(function(err){});
}
```

This code is getting two dependent pieces of data.

But only sets the final View data once both are available.

… to this


```
async function getAndRenderArtists() {
  var artist = await Ajax.get("/api/artists/1");
  artist.albums = await Ajax.get("/api/artists/1/albums");
  View.set("artist", artist);
}

var rendered = getAndRenderArtists();
rendered.then(function(response) {
  console.log("Page is loaded");
});
```

Exercise: Ajax Promise

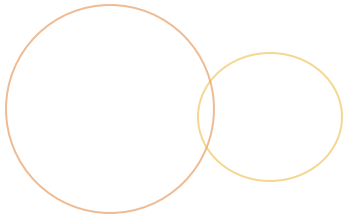
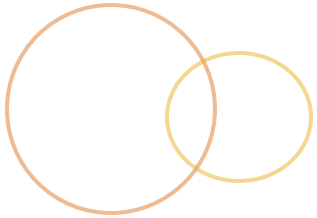
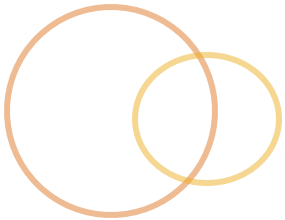
- Open the following file:

  `public/exercises/promises/promises.js`

- Complete the exercise
- Run the tests by visiting in your browser:

  http://localhost:3000/exercises/promises

Solutions:
https://github.com/rm-training/web-dev-bc/blob/master/public/solutions/promises/promises.js


WRAPPING UP

That's a wrap

- Any questions?
- Best practice reminders
  - Semantic
  - Don't re-select
  - Don't select more often than you need to
  - Be non-blocking
  - Keep scope clean
- Staying sharp
  - Code Kata

Final Lab: Todos

- Open the following file:

  `public/exercises/todos/script.js`

- Complete the exercise
  - You can use vanilla JS or jQuery
  - You can use your Ajax lib or fetch() or XHR
  - We can use handlebars for templating
- Run the app by visiting in your browser:

  http://localhost:3000/exercises/todos

Solutions:
https://github.com/rm-training/web-dev-bc/blob/master/public/solutions/todos/script.js