COEN 168/268

Mobile Web Application Development

JavaScript and jQuery

Peter Bergström (pbergstrom@scu.edu)

Santa Clara University

JS

JavaScript!

Brings life to the web

JavaScript is the most widely deployed programming language

Everyone has it in their browsers (except if you're running Lynx or something)

Some history about JavaScript…

- Created by Brendan Eich at Netscape
- Was originally called Mocha, then LiveScript
- Name changed to JavaScript to jump on the Java marketing bandwagon of the 1990s
- It is not at all related to Java other than in name
- First shipped in September of 1996 in Netscape Navigator 2.0B3

For years JavaScript languished in the hell that is `<form>` validation

That's not longer the case...

JavaScript adds interactivity

Now, the basics

Imperative and structured

- C-style syntax for the most part
- However, scoping not block level, but rather function level
- Semi-colons are not required at the end of newlines

Dynamic

- Dynamically typed
  - `var` keyword is used to declare all types
- Object-based
  - All objects are hashes made up of `key` and `value` pairs
- Runtime evaluation is possible (but not recommend as `eval()` is evil)

Functional

- Functions are first-class citizens
- Functions are objects and can have properties and methods
- Closures are easy and in wide-spread use

Prototypical Inheritance

- JavaScript is not class-based
- You can simulate class-based inheritance if you want
- Functions are methods and the `this` property is bound to the object itself

Prototypical Inheritance Continued

- Functions are object constructors
  - `new` keyword can be used to create a new instance of a prototype that inherits properties and methods
  - `.prototype` property on a object determines what is carried over when `new` is called
  - This means that if you have an object declared on the `prototype`, it can be shared with all instances of the object (it's a bit weird)

JavaScript Syntax

I assume that you all know the basic C-style syntax so I won't go into all that

For most examples, we will be using the Chrome's Web Inspector

We will be using `console.log()` a lot

See:

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide

for more JS language examples like the next several slides

JavaScript Values

JavaScript recognizes the five following types of primitive values:

- **Numbers** like 42, 3.14159
- **Logical (Boolean)** like true / false
- **Strings** like "Howdy"
- **null** - a special keyword denoting a null value
- **undefined** - a top-level property whose value is undefined

Data type conversion

JS is dynamically typed so variables can be reused

```
var answer = 42;

// then later:
answer = "Thanks for all the fish...";

x = "The answer is " + 42 // "The answer is 42"
y = 42 + " is the answer" // "42 is the answer"

// but the + operator can cause strange things to happen:
"37" - 7 // 30
"37" + 7 // "377"
```

JavaScript Variables

```
var thisIsAString = "My voice is my passport.";

var thisIsANumber = 2;

var thisIsABoolean = false;

var thisIsAnObject = {
  key1: "value1",
  key2: "value2"
};

var thisIsAnArray = [1, 2, 3];
```

Variable identifier naming conventions

- Must start with letters (a-zA-Z), $, or _
- Subsequent characters can also be digits
- Should be `camelCase`
- Case sensitive!

All of these are valid, but pretty bad

```
var $;
var _;
var $parameter;
var __$myValue;
var this_is_a_long_variable_name;
```

Variables can declared all on one line (preferred)

```
var isVisible = false;
var idString = 'abc';

instead:

var isVisible = false,
    idString  = 'abc';
```

Also note that the alignment of the variable names and assignments.

Variable scope

- When you declare a variable outside of a function, it will be **global**
- This means that you can see it anywhere so be careful
- If it is declared inside of a function, it is in **function scope**

```
if (true) {
  var x = 5;
}
console.log(x); // 5 since there isn't block level scope
```

undefined and null

```
var isVisible;
console.log(isVisible); // undefined

var idString = null;
console.log(idString); // null

if (!isVisible) {
  // Will go in here
}

if (!idString) {
  // Will go in here
}
```

Converting strings to numbers

- Use `parseInt()` and `parseFloat()`
- Important to avoid strange behavior when:

```
"1.1" + "1.1"
// results in: "1.11.1"

parseFloat("1.1",10) + parseFloat("1.1", 10)
// results in 2.2
```

A Note about using == and !=

- Using == and != do work, but can yield strange results
- This is due to JavaScript's **type coercion** that gets around checking type

```
if("0" == 0) {
  // You go in here
}
```

If you care about type...

- Use === and !== to disable type coercion and enforce the type as well

```
if("0" === 0) {
  // You do not enter here
}
```

Strings

- Strings can be concatenated using +
- Can concatenate any type together with strings
- However, the results can get weird when mixing numbers and strings

Let's take a look...

```
var numBlue  = 2,
    numGreen = 5;

var colorString = "There are " + numBlue + numGreen + " objects.";
// Results in: "There are a total of 25 objects."

...

var colorString = "There are " + (numBlue + numGreen) + " objects.";
// Results in: "There are a total of 7 objects."
```

JavaScript Objects

Creating an `Object` is easy.

Let's create a `mustang` object

```
var mustang = {};
```

It has no properties and it is empty. However, there are `prototype` properties attached to it in the `__proto__` property.

Object prototype properties and methods

For example, here are some methods

```
> mustang.toString();
  "[Object object]"

> mustang.constructor()
  Object {}

> mustang.constructor
  function Object() { [native code] }
```

Creating properties on `objects`

- Three ways:
  - dot notation
  - square brackets
  - literal notation

Dot notation

```
var mustang = {};

mustang.make = "Ford";
mustang.model = "Mustang";
mustang.year = 2015;
```

Square brackets

```
var mustang = {};

mustang["make"] = "Ford";
mustang["model"] = "Mustang";
mustang["year"] = 2015;
```

Literal Notation

```
var mustang = {
  make: "Ford",
  model: "Mustang",
  year: 2015
};
```

Myself, I prefer this way for complex objects.

You can also do a combination, if you need to.

```
// Define base properties
var mustang = {
  make: "Ford",
  model: "Mustang",
  year: 2015
};


// Define an additional property
mustang.color = "Red";
```

You can also add methods on objects:

```
// Define base properties
var mustang = {
  make: "Ford",
  model: "Mustang",
  year: 2015,

  description: function() {
    return this.year + " " + this.make + " " + this.model;
  }

};

> mustang.description();
  "2015 Ford Mustang"
```

Custom objects

- The previous example was just adding properties to a base `Object`
- However, most of the time, you will want to create custom `Objects` to represent your data
- This will help you scale up the complexity of your app

This won't scale well...

```
var mustang = {
  make: "Ford",
  model: "Mustang",
  year: 2015,

  description: function() {
    return this.year + " " + this.make + " " + this.model;
  }
};

var camaro = {
  make: "Chevrolet",
  model: "Camaro",
  year: 2015,

  description: function() {
    return this.year + " " + this.make + " " + this.model;
  }
};
```

Instead, create a `car` `Object` as the base

```
var car = {

  description: function() {
    return this.year + " " + this.make + " " + this.model;
  }

};
```

Then use the `Object.create()` method

```
var mustang = Object.create(car);
mustang.make = "Ford";
mustang.model = "Mustang";
mustang.year = 2015;

var camaro = Object.create(car);
camaro.make = "Chevrolet";
camaro.model = "Camaro";
camaro.year = 2015;
```

Why not just pass in the properties as values?

```
// Doesn't work
var mustang = Object.create(car, {
  make: "Ford",
  model: "Mustang",
  year: 2015
});


// Works
var mustang = Object.create(car, {
  make: {
    value: "Ford"
  },
  model: {
    value: "Mustang"
  },
  year: {
    value: 2015
  }
});
```

A side effect of JavaScripts prototypical nature

- If you define other objects on the `prototype` of an `Object`, any created instances will have that object referenced
- This means, that if you modify that object in one instance, it will be modified in all instances
- To fix this, create objects on initialization, not when the object is defined initially
- Primitives are OK

The side effect in action

```
var car = {
  colors: ['red', 'blue', 'yellow', 'green'],
  name: 'default car'
};

var ford = Object.create(car);
var bmw  = Object.create(car);

ford.colors.push('black');

> ford.colors
  "['red', 'blue', 'yellow', 'green', 'black']"

> bmw.colors
    "['red', 'blue', 'yellow', 'green', 'black']"
```

One way to fix this...

```
var car = {
  name: 'default car',
  init: function() {
    this.colors = ['red', 'blue', 'yellow', 'green'];
    return this;
  }
};

var ford = Object.create(car).init();
var bmw  = Object.create(car).init();

ford.colors.push('black');

> ford.colors
  "['red', 'blue', 'yellow', 'green', 'black']"

> bmw.colors
    "['red', 'blue', 'yellow', 'green']"
```

Demo

1 - JavaScript Variables

Code can be found at:

http://coen268.peterbergstrom.com/resources/demos/jslecturedemos.zip

JavaScript `function` declaration

- The name of the `function`.
- A list of arguments to the `function`, enclosed in parentheses and separated by commas.
- The JavaScript statements that define the function, enclosed in curly brackets, `{ }`.

```
function name(param, ...) {
 // Do stuff in here.
}
```

JavaScript `functions` can return values

```
function whoami() {
 return "Peter";
}
```

- Primitive parameters (such as a number) are passed to/from functions by value
- Non-primitive parameters (such as a object) are passed to/from functions by reference

Ways to declare `functions`

There are several ways to declare functions

- constructor
- declaration
- anonymous expression

JavaScript function constructor

```
var add = new Function("x", "y", "return x + y;");
```

JavaScript function declaration

```
function add(x, y) {
    return x + y;
}
```

JavaScript anonymous function expression

```
var add = function(x, y) {
    return x + y;
};
```

Nesting JavaScript functions

- One interesting thing about JavaScript is that you can nest functions
- This does enforce some encapsulation
- This is usually called a `closure`

What is a closure?

A closure is an expression (typically a function) that can have free variables together with an environment that binds those variables (that "closes" the expression).

```
function addSquares(a,b) {
    function square(x) {
        return x * x;
    }
    return square(a) + square(b);
}
a = addSquares(2,3); // returns 13
b = addSquares(3,4); // returns 25
c = addSquares(4,5); // returns 41
```

```
function outside(x) {
    function inside(y) {
        return x + y;
    }
    return inside;
}
// Think of it like: give me a function
// that adds 3 to whatever you give it
fn_inside = outside(3);
result = fn_inside(5); // returns 8

result1 = outside(3)(5); // returns 8
```

Nesting can actually continue forever...

```
function A(x) {
    function B(y) {
        function C(z) {
            alert(x + y + z);
        }
        C(3);
    }
    B(2);
}
A(1); // alerts 6 (1 + 2 + 3)
```

Demo

2 - JavaScript Functions

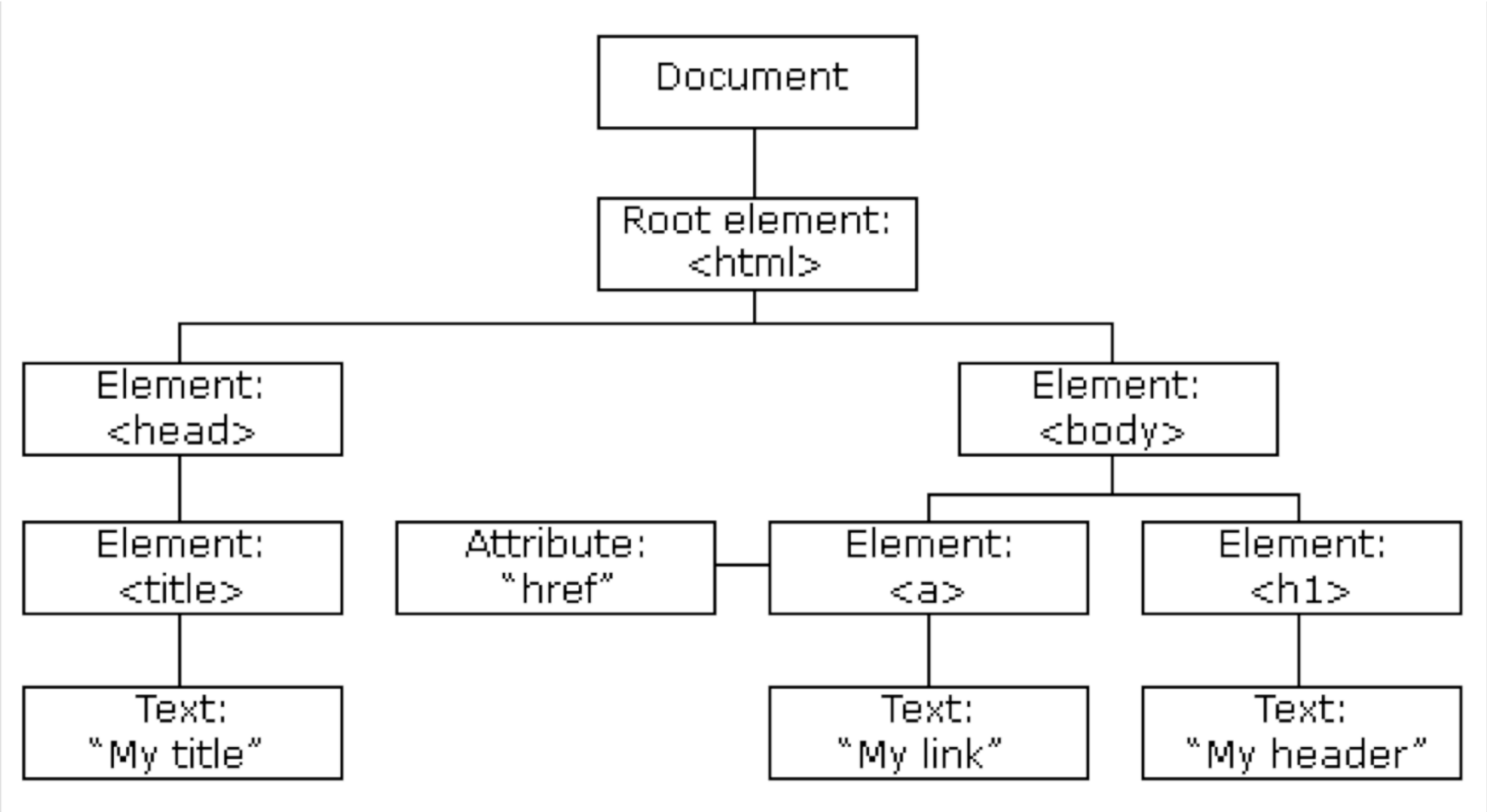
Code can be found at:

http://coen268.peterbergstrom.com/resources/demos/jslecturedemos.zip

Okay, that was a bit boring, let's bring things to life

Working with HTML and the DOM

- When a HTML document is loaded, the browser creates a **D**ocument **O**bject **M**odel representation of the page
- JavaScript's primary function is to interact with the HTML on the page
- It is a tree structure that you can navigate and modify in JavaScript
- In the context of DOM manipulation, HTML tags are called **DOM elements**



From: http://www.w3schools.com/js/js_htmldom.asp

What can you do with the DOM with JavaScript?

- Change all the HTML elements and attributes in the page
- Add or remove existing HTML elements and attributes
- Change all the CSS styles in the page
- React to or create HTML events in the page

Finding DOM elements

```
document.getElementById('my-id');

document.getElementsByTagName('h2');

document.getElementsByClassName('my-classname');
```

Changing DOM elements

```
element.className = 'my-class';
element.id = 'my-id';
element.style.fontSize = '10px';

and more...
```

Adding and Removing DOM elements

```
document.createElement()
document.removeChild()
document.appendChild()
document.replaceChild()
```

Demo

3 - DOM traversal

Code can be found at:

http://coen268.peterbergstrom.com/resources/demos/jslecturedemos.zip

DOM events, bringing a page to life

- There are many mouse and touch events such as `click`, `scroll`, `mousedown`, `ontouchstart`, etc
  - These are the ones that bring your app to life
  - There are so many, so look them up
- There are events such as `load` that trigger when things load
  - A common one is `window.onload` to do things once the page is loaded

Several ways to add events

Inline in element

```
<div class="my-class" onclick="function() {alert(this.className;)}">
  This is my div
</div>
```

Hard to manage!

In JavaScript...

```
<div class="my-class">This is my div</div>

// assuming only 1 div
var myDiv = document.getElementsByTagName('div')[0];

myDiv.onclick = function() {
  alert(this.className);
};
```

Better, but there is one problem. You overwrite any other `click` handler

In JavaScript, using addEventListener():

```
<div class="my-class">This is my div</div>

// assuming only 1 div
var myDiv = document.getElementsByTagName('div')[0];

myDiv.addEventListener('click', function() {
  alert(this.className);
});
```

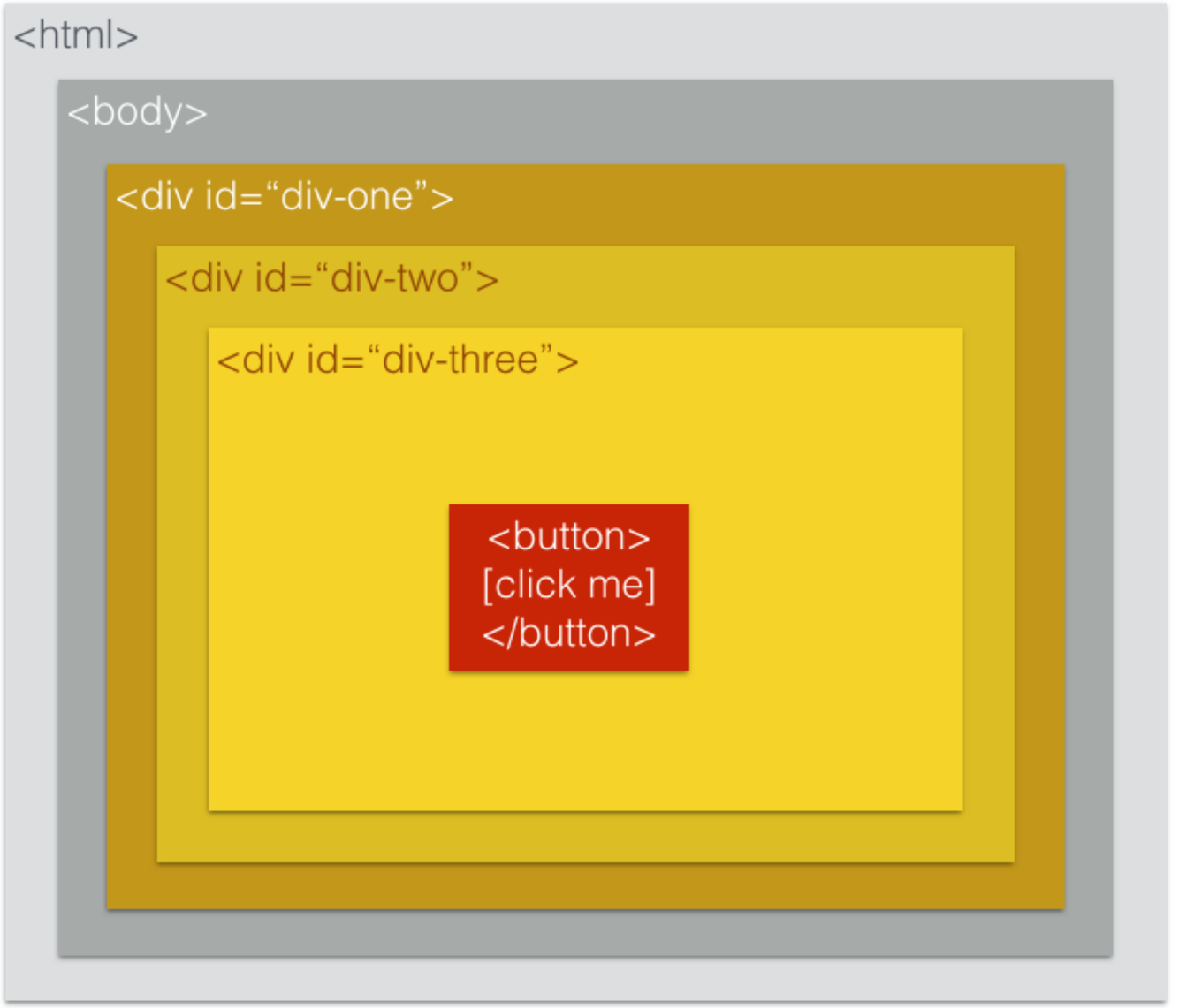
Now you can have multiple handlers going to the same event, if you want

JavaScript Event Capturing & Bubbling

- Events do not happen on their own as they are part of a hierarchy of elements
- As you know, HTML documents are all about nesting elements within each other
- Let's go through an example...

Let's use this HTML

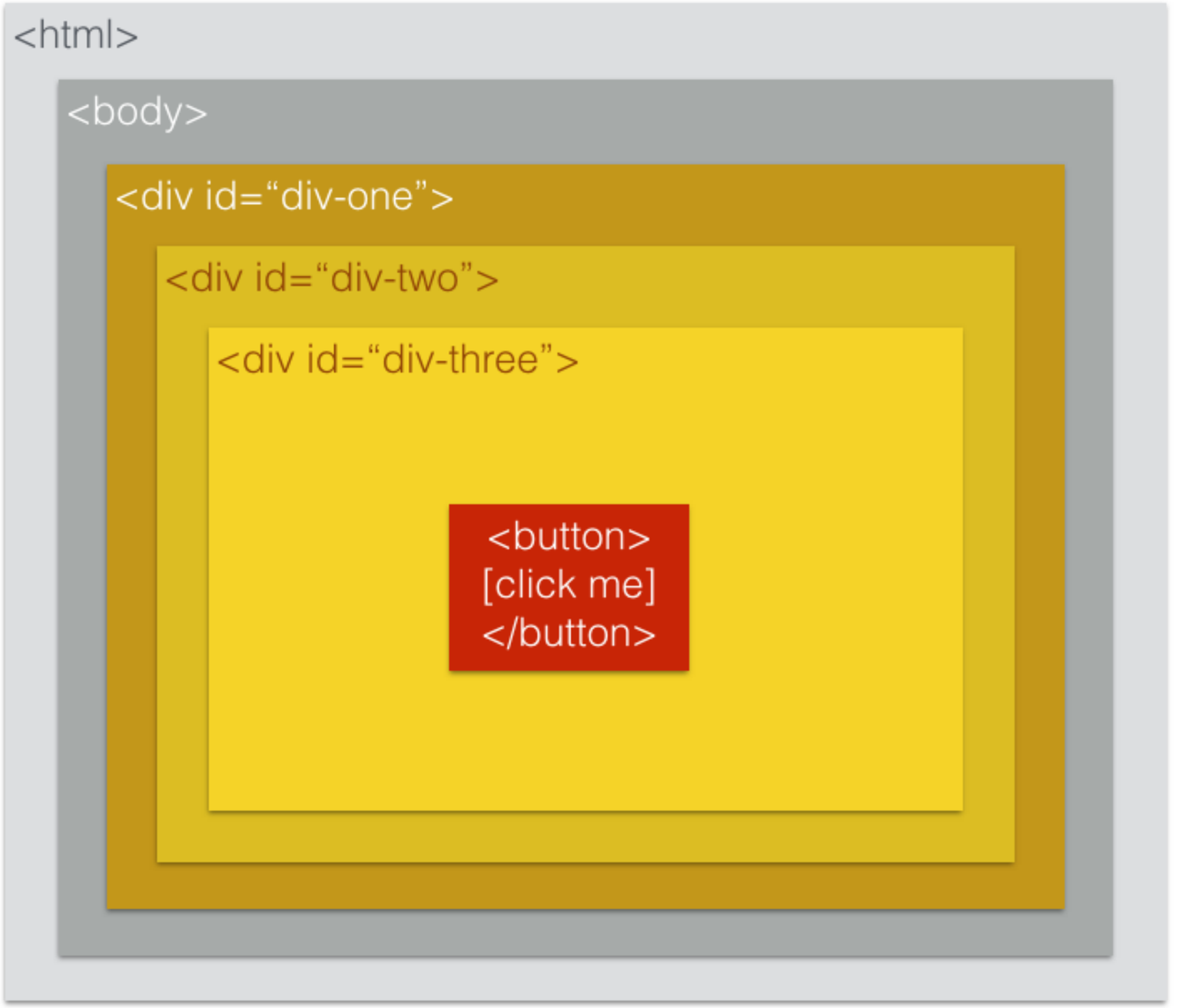
```
<div id="div-one">
    <div id="div-two">
        <div id="div-three">
            <button id="button">[click me]</button>
        </div>
    </div>
</div>
```


<html>
<body>

<button>
[click me]
</button>

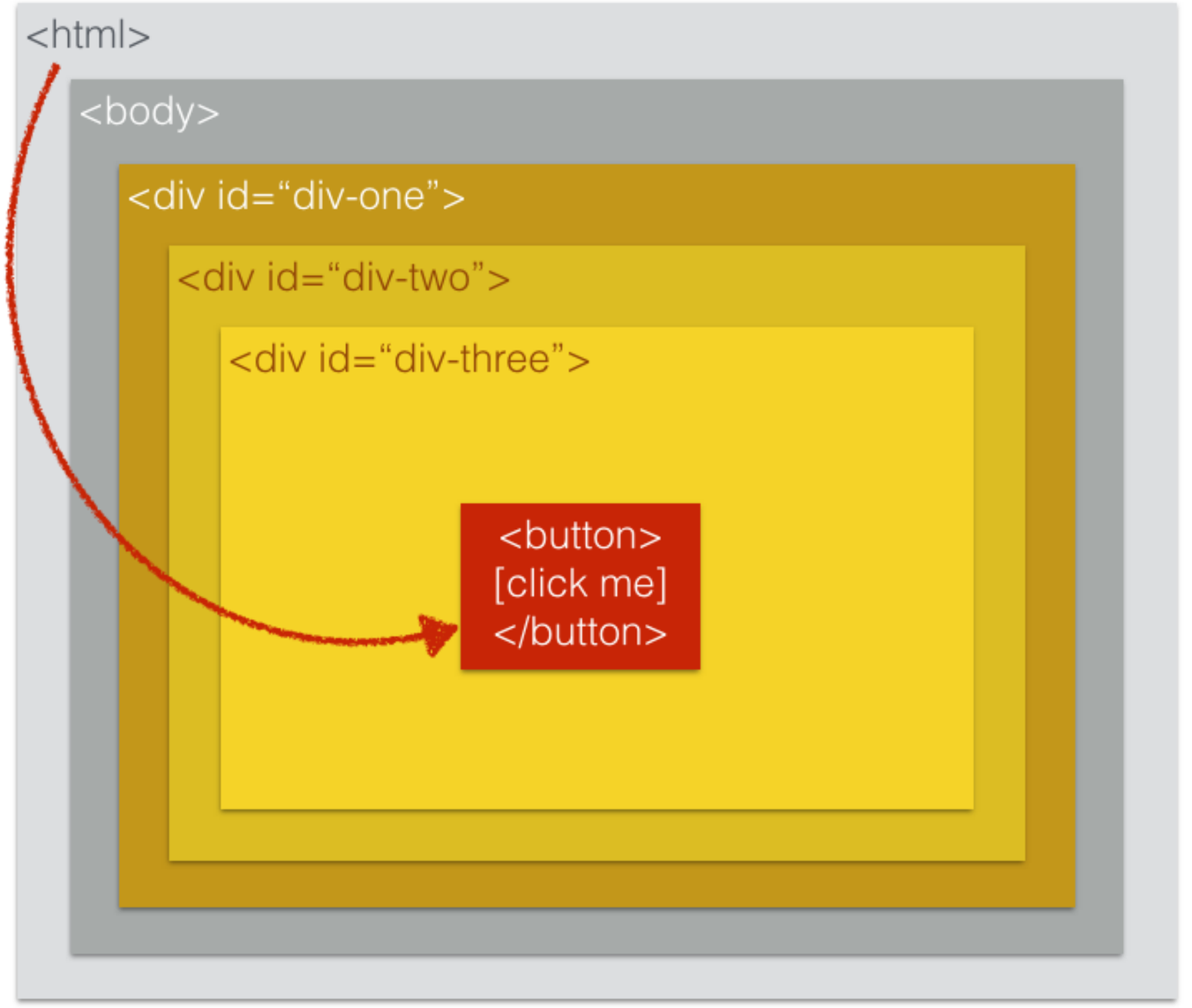
Two Phases: Capturing and Bubbling

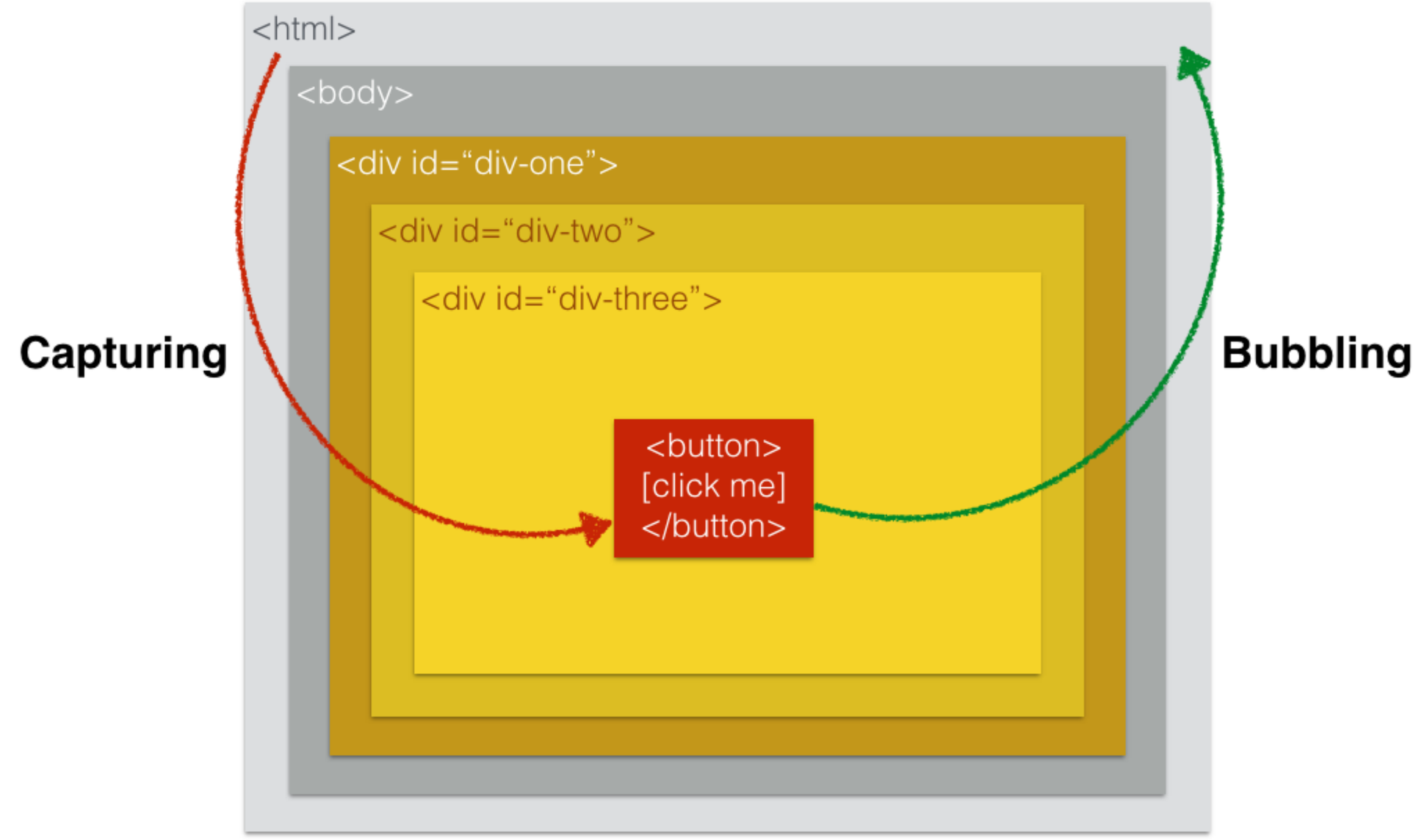
- Capturing goes down from the `root`, the `<html>` element down to the element that triggered the event
- Then, the event bubbles up from the element that triggered the event back up to the `root`


<html>
<body>

<button>
[click me]
</button>

Capturing




Capturing
<html>
<body>

<button>
[click me]
</button>
Bubbling

When to trigger the event, capturing or bubbling?

When calling `addEventListener()`, you can specify if you want to trigger on capture or on bubbling

```
// triggers on capture
addEventListener('click', myFunction, true);

// triggers on bubble
addEventListener('click', myFunction, false);

// default, triggers on capture
addEventListener('click', myFunction);
```

Why does it matter?

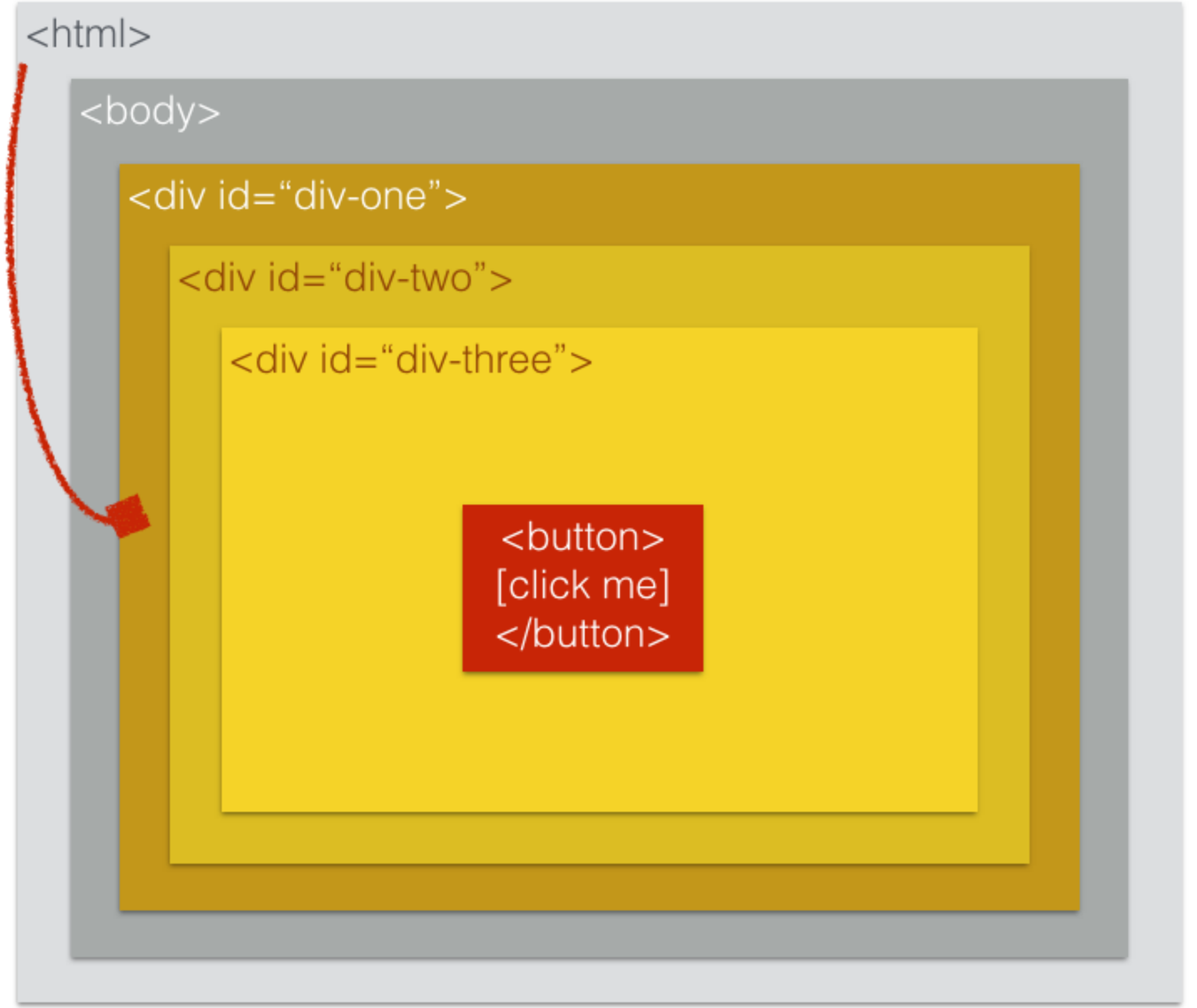
- Usually it doesn't
- However, there can be times when you want to alter behavior
- If you trigger on capture, instead of bubble, you can deal with the order of
  events in one way, and vice versa

Stopping event propagation

- There are times when you want to stop the capturing or bubbling traversal
- If each nested `<div>` had a click handler, it would trigger all of them in order unless you call `e.stopPropagation()` at some point:

```
divOne.addEventListener('click',  // divOne points to id="div-one"
    function myFunction(e) {
      e.stopPropagation(); // stop it cold and do your thing.
    });
```

Capturing



Stopping event default actions

- Another call that you can use is `e.preventDefault()`
- HTML elements have default actions like, clicking an `<input>` element will focus on it.
- With `e.preventDefault()` you can turn off those things and control it completely
- This is used quite often when you get into writing custom controls

CalculatorDemo

Adding JavaScript interactivity

Code can be found at:

https://github.com/pbergstr/calculator-demo/tree/v8-add-javascript

https://github.com/pbergstr/calculator-demo/tree/v9-organize-javascript

jQuery is DOM manipulation framework

- Takes care of browser incompatibilities
- Makes it easier to look up DOM elements
- Makes it easier to add and respond to events
- Contains a lot of nice utility methods

How jQuery works

```
<!doctype html>
<html>
<head>
    <meta charset="utf-8" />
    <title>Demo</title>
</head>
<body>
    <a href="http://jquery.com/">jQuery</a>
    <script src="jquery.js"></script>
    <script>

    // Your code goes here.

    </script>
</body>
</html>
```

Launching code on Document Ready

```
// Many developers do this:
window.onload = function() {
    alert("welcome");
}

// But blocks on image loads so this is better:
$( document ).ready(function() {
    // Your code here.
});
```

Adding Events on Document Ready

Since the DOM is loaded, but images might not be, you can add events:

```
$(document).ready(function() {

    $("a").click(function(event) {

        alert("Thanks for visiting!");

    });

});
```

A complete example

```
<!doctype html>
<html>
<head>
    <meta charset="utf-8" />
    <title>Demo</title>
</head>
<body>
    <a href="http://jquery.com/">jQuery</a>
    <script src="jquery.js"></script>
    <script>

    $(document).ready(function() {
        $("a").click(function(event) {
            alert( "The link will no longer take you to jquery.com" );
            event.preventDefault();
        });
    });

    </script>
</body>
</html>
```

With jQuery it is easy to add and remove class names

```
<style>
a.test {
    font-weight: bold;
}
</style>

$("a").addClass("test"); // Now bold

$("a").removeClass("test"); // Not bold
```

jQuery makes it easy to show and hide elements

```
$("a").hide(); // Now it is set to display: none

$("a").show(); // Now it is no longer display: none
```

You can even animate it

```
$("a").click(function(event) {

    event.preventDefault();

    $(this).hide("slow");

});
```

CalculatorDemo

Converting to jQuery real quick

Code can be found at:

https://github.com/pbergstr/calculator-demo/tree/v10-add-jquery

Demo

4 - jQuery Events

Code can be found at:

http://coen268.peterbergstrom.com/resources/demos/jslecturedemos.zip

HTML 5 and JavaScript

- HTML5 has a lot of features that use JavaScript
- Local Storage, Location, Accelerometer, and more…

Local Storage

- Very simple to use
- Supported in modern browsers
- Typically 5MB per site
- Key value store
- Stores only strings natively

Local Storage example

```
var data = {
  firstName: "Peter",
  lastName: "Bergström"
}

// Convert to JSON so we can store it
localStorage.setItem('name', JSON.stringify(data));

var data = JSON.parse(localStorage.getItem('name'));
```

Location

- Based on browser permission, you can get lat and long
- Works for mobile devices and desktops
- Great for apps

Location example

```
function getLocation() {
  if (navigator.geolocation) {
    var options = {
      enableHighAccuracy: true,
      timeout: 5000,
      maximumAge: 0
    };
    navigator.geolocation.getCurrentPosition(showLocation, showError, options);
  } else {
    showError();
  }
};
```

Demo

5 - Local Storage and Location

Code can be found at:

http://coen268.peterbergstrom.com/resources/demos/jslecturedemos.zip

Accelerometer

- Captures the motion of the device in physical space
- Enables a lot of interactivity
- Can be used in games or apps

Demo: http://www.html5rocks.com/en/tutorials/device/orientation/devicemotionsample.html

COEN 168/268

Mobile Web Application Development

JavaScript and jQuery

Peter Bergström (pbergstrom@scu.edu)

Santa Clara University