CS 547

JavaScript

Week 13 Day 1

4/21/2015

Agenda

- Administration
  - Pillow on mac

Pillow on Mac

If you are using a mac to develop your code you will need to use pip3 to install pillow on a Mac with python 3.x

pip3 should be located in /usr/bin directory

JavaScript

_.JS

node

JavaScript

JavaScript: Origins

Origins: Java Scheme Self

Influenced by others including

C Java HyperText Python

Important to understand the Industry Context see the video at the end of the slides

JavaScript

JavaScript also called ECMAScript is a dynamic language originally know as LiveScript

Emerged from the browser wars of the mid 1990's . Sponsored by Netscape and Sun.

See the second Video for more context

Current version JS is ECAMAScript 5, (ES5) but ES6 is due out this summer.

JS: Origins

JavaScript was originally developed by Brendan Eich, while he was working for Netscape Communications Corporation in about a week and a half.

JS is classified as a prototype based scripting language with dynamic typing and first class functions.

JS is object oriented, but in a slightly different way than in Java.

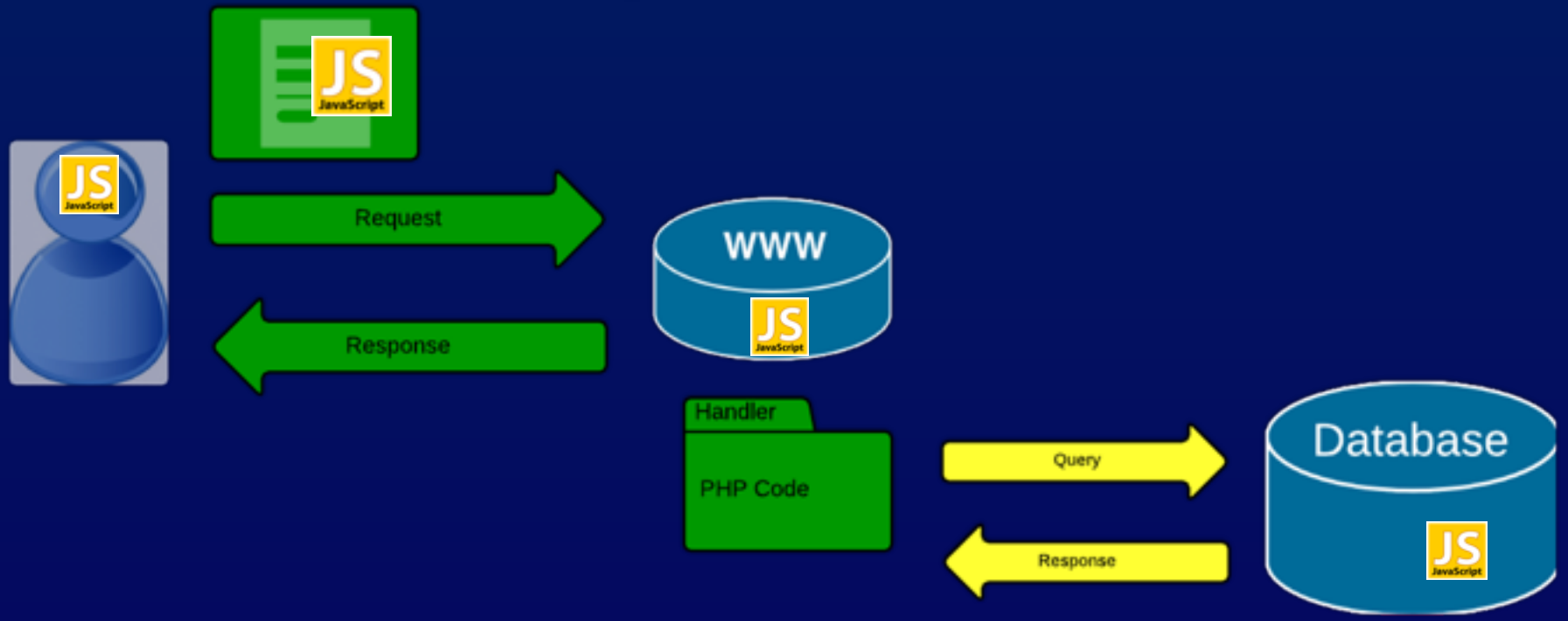
JS: Things to note

JavaScript is

- a full fledged language
- an interpreted language
- has no native input out functions – is meant to run in a hosted environment

JS: Things to note

runs in a hosted environment.. implies



JS Object Orientated Nature

In JS an **object** is a dynamic collection of **properties.**

Every property has a key string that is unique within that object,

* An object in JS is not an instance of a class. *

JS Properties

A property is a named collection of attributes.

- value: any JavaScript value
- writeable: Boolean
- enumerable: Boolean
- Configurable: Boolean
- get: function () {…return value}
- set: function (value) { … }

JS Properties

Two types of properties

- Data properties
- Accessor properties

JS Language Basics

- Syntax, Values, Operators, Expressions, Keywords, and Comments
- Literals, Variables, Operators, and Key Words
- Statements, Expressions and Semi colons
- Variables
  - Numbers, Strings, Arrays, Objects
- Objects and Functions

JS Syntax Values

Values: JS defines two types: Fixed and Variables

Fixed: (literals)

Numbers 101 0.0003 32Bit integers

Strings- “Jane Doe” ‘John Smith’

Expressions- 12 + 21 6 * 2

Variables:

JS Variables

Variables: used to hold stuff.

```
var x;

x = 6;
```

more in a bit

JS Operators

Two broad types, assignment and arithmetic operators

Equal sign (=) to assign values

X = 6

Arithmetic (+ - * / %) are used to compute values

(5 + 6) * 2

Decrement operators (++ --)

X++ --Y

JS Operators

Decrement Assignment operators (=+ =- =* =/)

x += 1;

JS Operators Comparison

Operator	Description	Comparing	Returns
==	equal to	x == 8	false
		x == 5	true
===	equal value and equal type	x === "5"	false
		x === 5	true
!=	not equal	x != 8	true
!==	not equal value or not equal type	x !== "5"	true
		x !== 5	false
>	greater than	x > 8	false
<	less than	x < 8	true
>=	greater than or equal to	x >= 8	false
<=	less than or equal to	x <= 8	*true*

JS Operators

Unary + and - . This operator precedes its operand and evaluates to its operand but attempts to converts it into a number, if it isn't already. Will return NaN (not a number) if it can not make the conversion.

```
+3     // 3
+"3"   // 3
+true  // 1
+false // 0
+null  // 0
```

JS Keywords

abstract	arguments	boolean	break	byte
case	catch	char	class*	const
continue	debugger	default	delete	do
double	else	enum*	eval	export*
extends*	false	final	finally	float
for	function	goto	if	implements
import*	in	instanceof	int	interface
let	long	native	new	null
package	private	protected	public	return
short	static	super*	switch	synchronized
this	throw	throws	transient	true
try	typeof	var	void	volatile
while	with	yield		

JS Statements

JS has a set of familiar statements:

- Expression
- Disruptive
- Conditionals- Try, If, and Switch
- Loops- While, For, and Do

JS Statements

Expression statements use the ***var*** keyword. Also be aware that by default JS puts everything into a **Global** namespace.

```
var   foo = “bar”;
```

Blocks of statements us curly brackets { } but the do not create a new scope

```
{
var foo = “bazz”; // this overwrites the above
}
```

JS Statements

Disruptive include

break – to break out of a loop or block

return – to exit a function and return control back to the calling function

throw – to raise an error

JS Semi Colons;

A note about semi colons (;). In JS semicolons are optional. The interpreter will fill them in for you. Because of this you should always use them to denote the end of a statement. Consider

```
function foo(x) {
    …
    return
    true
}
```

JS Semi Colons;

Did you mean????

```
function foo(x) {
    …
    return;
    true;
}
```

OR???

```
function foo(x) {
    …
    return
    true;
}
```

JS Comments

Javascript uses C or java style comments:

Single line comments start with // and go to the end of the line

```
var x = 12; // set x = 12
```

Block comments are denoted within /* … */ pairs

```
/*  This  is
      a block
      Comment  */
```

JS Statements Conditionals

JS has the regular if else and else if conditional statements

```
if ( something == true ) {
    // do some thing
}
else {
    // do something else
}
```

JS Statements Conditionals

Else If

```
if (time < 10) {
    greeting = "Good morning";
} else if (time < 20) {
    greeting = "Good day";
} else {
    greeting = "Good evening";
}
```

JS Statements Conditionals

JS Switch Statement

```
switch(expression) {
    case n:
        code block
        break;
    case n:
        code block
        break;
    default:
        default code block
}
```

```
switch (new Date().getDay()) {
    case 1:
    case 2:
    case 3:
    default:
        text = "Looking forward to the Weekend";
        break;
    case 4:
    case 5:
      text = "Soon it be Weekend";
       break;
    case 0:
    case 6:
      text = "Weekend!!!";
}
```

JS Loops

JS has four loop constructs similar to Java

- for - loops through a block of code a fixed number of times
- while - loops through a block of code while a condition is true
- for/in - loops through the properties of an object
- do/while - loops while a specified condition is true tests after block is executed

JS Variables

- ✓ Numbers
- ✓ Strings
- Arrays
- Objects

JS Variables Arrays

JS Arrays are declared with [ ] brackets

```
var cars = [ ‘Honda’, “Toyta” ];    // aka array literal
```

you can also use the new keyword to declare the array

```
var cars = new Array[ ‘Honda’, “Toyta” ];    // works but not prefered
```

JS Variables Arrays

JS Arrays can contain mixed types

```
var Stuff= [ ‘Honda’, Date.now(), 23];
```

and you can acess it by index

```
Stuff[0];  // Honda
Stuff[1];
Stuff [2] ; //23
```

JS Variables Arrays

JS arrays use array.length to get the length of the array

```
Stuff.length; // returns 3
```

JS Variables Arrays

JS arrays are not associative like in PHP, use objects for that instead

```
var person = [];
person["firstName"] = "John";
person["lastName"] = "Doe";
person["age"] = 46;
var x = person.length;         // person.length will return 0
var y = person[0];             // person[0] will return undefined
```

JS Objects

In JS everything is an object.

But objects are not like they are in Java.

JS Object Orientated Nature

In JS an **object** is a dynamic collection of **properties.**

Every property has a key string that is unique within that object,

* An object in JS is not an instance of a class. *

JS Objects

to create an object use curly brackets

```
var empty_object = { };

var rockStar= {
        firstName :  “Freddy”,
        lastName : “Mercury”
}
```

JS Objects

Retrieval: values can be retired in two ways: obj.[“string”] or obj.string

rockstar[“firstName”] // returns “Freddy”

rockstar.lastName; // returns “Mercury”

}

JS Objects

Objects can also contain functions. we will get into that next time.

JS Variables

- Numbers, Strings, Arrays, Objects

JS Variables

JS Resources

Videos:

Dougleas Crockford on Javascript https://www.youtube.com/watch?v=JxAXlJEmNMg

This video series provides context and history on the development of JS at 1:37:00 he describes the history computing and what led to the creation of JavaScript

And then there was JavaScript https://www.youtube.com/watch?v=RO1Wnu-xKoY

CS 547

JavaScript

Week 13 Day 2

4/23/2015

Agenda

- Administration
  - Pillow on mac
- JavaScript
  - objects
  - functions
  - IDEs / debuggers
- JSON

JS Language Basics

- ✓ Syntax, Values, Operators, Expressions, Keywords, and Comments
- ✓ Literals, Variables, Operators, and Key Words
- ✓ Statements, Expressions and Semi colons
- ✓ Variables
  - ✓ Numbers, Strings, Arrays, Objects
- Objects
- Functions

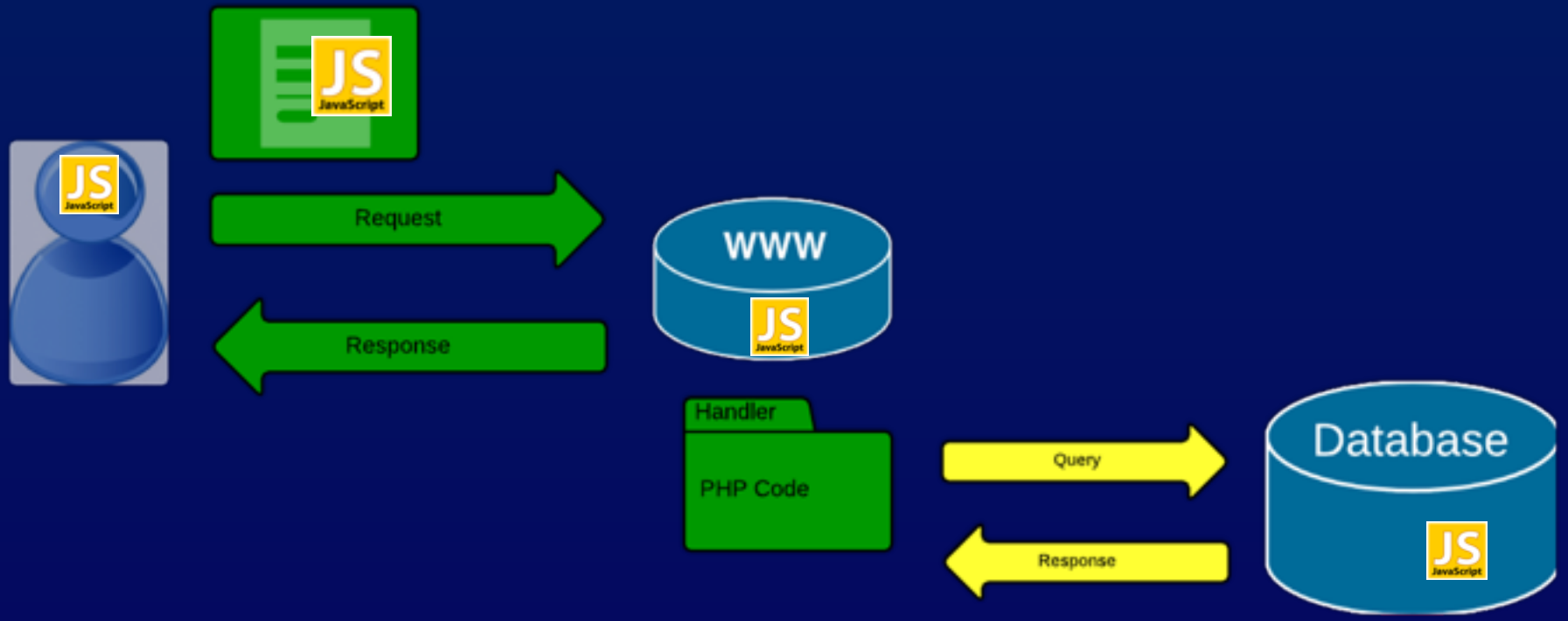
Pillow on Mac

JS IDEs

- Sublime Text 3 or 2 http://www.sublimetext.com/
- Brackets from brakets.io http://brackets.io/
- JetBrains WebStorm https://www.jetbrains.com/webstorm/
- Netbeans https://netbeans.org/features/html5/ (with appropriate plugins)

Reminder- JavaScript

Reminder- JS:


JS
JavaScript
Request
Response
WWW
Handler
PHP Code
Query
Response
Database

Reminder- JS Object Orientated Nature

In JS an **object** is a dynamic collection of **properties.**

Every property has a key string that is unique within that object,

* An object in JS is not an instance of a class. *

JS Objects

Simple types – immutable

numbers, strings, booleans (true and false), null, and undefined

Everything else is an object – mutable keyed collections

arrays, functions, regular expressions, and, objects are objects

But recall that JS objects are not like they are in Java.

JS Objects

An Object is a container of Properties

Properties have a Name and a Value

“first_name” : “Freddie”

A property name can be any string, including the empty string. A property value can be any JavaScript value except for undefined.

“” : “something” ;

“fname” : undefined;

JS Objects

Objects in JavaScript are class-free.

- No constraint on the names of new properties or on the values of properties.
- Objects are useful for collecting and organizing data.
- Objects can contain other objects.

JavaScript includes a ***prototype*** linkage feature that allows one object to inherit the properties of another.

JS Objects Creation

to create an object use curly brackets

```
var empty_object = { };    //object literal

var rockStar= {
        firstName :  “Freddy”,
        lastName : “Mercury”
}
```

JS Objects Retrieval

Retrieval: values can be retired in two ways: obj.[“string”] or obj.string

```
rockstar[“firstName”] // returns  “Freddy”

rockstar.firstName;  // returns  “Freddy”
}
```

JS Objects Update

A value in an object can be updated by assignment. If the property name already exists in the object, the property value is replaced

```
rockstar[“firstName”] = “Frederick”;
```

If the object does not already have that property name, the object is augmented:

```
rockstar.realName = “Farrokh Bulsara”;
```

JS Objects Reference

Objects are passed around by **reference**. They are never copied:

```
var a = {}, b = {}, c = {};
    // a, b, and c each refer to a
    // different empty object

a = b = c = {};
    // a, b, and c all refer to
    // the same empty object
```

JS Objects Prototype

Every object is linked to a prototype object from which it can inherit properties.

All objects created from object literals are linked to **Object.prototype**, an object that comes standard with JavaScript.

JS Objects Prototype

```
if (typeof Object.create !== 'function') {
        Object.create = function (o) {
                var F = function () {};
                F.prototype = o;
                return new F();
        };
}
var another_rockStar = Object.create(rockStar);
```

The prototype link has no effect on updating. The prototype link is used only in retrieval. JS will look up the chain till it finds Object.prototype.This is called delegation, it is dynamic and if we add a new property to the prototype

JS Objects Prototype

Delegation is dynamic and if we add a new property to the prototype that property will immediately be visible in all of the objects that are based on that prototype:

```
rockStar.profession = ‘singer';
another_rockStar.profession // ‘singer'
```

JS Objects Reflection

Reflection is a way of inspecting an object to determine what properties it has by attempting to retrieve the properties and examining the values obtained. Two methods to use reflection:

typeof

```
typeof rockStar.firstName ;  // string
```

hasOwnProperty

```
rockStar.hasOwnProperty("profession"); // true
```

JS Objects Enumeration

You can use for loops to loop through all the properties in an object. There are two forms

for .. in

– this will loop through all the properties including function and prototype properties. Does not guarantee order

for (start; stop; increment)

– this will loop through all the properties in order without the function and prototype properties.

JS Objects Enumeration

for .. in – this will loop through all the properties including function and prototype properties. Does not guarantee order

```
var name;
for (name in another_rockStar) {
 …
}
```

JS Objects Enumeration

for (start; stop; increment)– this will loop through all the properties in order without the function and prototype properties.

```
var i;
var properties = [ 'first_name', 'last_name', 'profession' ];
for (i = 0; i < properties.length; i += 1) {
    document.writeln(properties[i] + ': ' + another_rockStar[properties[i]]); }
```

JS Objects Delete

The delete operator can be used to remove a property from an object. It will remove a property from the object if it has one.
It will not touch any of the objects in the prototype linkage.

JS Global Abatement

JavaScript makes it easy to define global variables that can hold all of the assets of your application.

Unfortunately, global variables *weaken* the resiliency of programs and

should be avoided.

JS Global Abatement

One way to minimize the use of global variables is to create a single global variable for your application:

var MYAPP = {};
That variable then becomes the container for your application:

```
MYAPP.rockStar = {
    "first_name": “Taylor",
    "last_name": “Swift"
};
```

JS Global Abatement

By reducing your global footprint to a single name, you significantly reduce the chance of bad interactions with other applications, widgets, or libraries.
Your program also becomes easier to read because it is obvious that

MYAPP.rockStar refers to a top-level structure.

JS Objects

Objects can also contain functions… so let's talk about functions.

JS Functions

Functions are one of the best things about JavaScript.

Functions in JavaScript are objects.

Objects are collections of name/value pairs having a hidden link to a prototype object.

Objects produced from object literals are linked to Object.prototype.

JS Functions

Function objects are linked to Function.prototype (which is itself linked to Object.prototype).

Every function is also created with two additional hidden properties:

the function's context and

the code that implements the function's behavior.

Every function object is also created with a prototype property. Its value is an object with a constructor property whose value is the function.

This is distinct from the hidden link to Function.prototype.

JS Functions

Since functions are objects, they can be used like any other value.

Functions can be stored in variables, objects, and arrays.

Functions can be passed as arguments to functions, and functions can be returned from functions.

Also, since functions are objects, functions can have methods.

The thing that is special about **functions** is that they **can be invoked**.

Functions Literals

Function objects are created with function literals:

```
// Create a variable called add and store a function
// in it that adds two numbers.

var add = function (a, b) {
    return a + b;
};
```

Functions Literals

Four parts to a function:

```
var add = function (a, b) {
    return a + b;
};
```

1. Reserved word ‘function’
2. an optional function name – if there is no name it is called an ***anonymous*** function
3. set of optional parameters
4. the statements inside the { brackets}

Functions closure

```
var add = function (a, b) {
    return a + b;
};
```

A function literal can appear anywhere that an expression can appear.
Functions can be defined inside of other functions.
An inner function of course has access to its parameters and variables.
An inner function also enjoys access to the parameters and variables of the functions it is nested within. The function object created by a function literal contains a link to that outer context.
This is called **closure**. This is the source of enormous expressive power

Function Invocation

Invoking a function suspends the execution of the current function, passing control and parameters to the new function.

In addition to the declared parameters, every function receives **two** *additional* parameters:

- **this**
- **arguments.**

The **this** parameter is very important in object oriented programming, and its value is determined by the invocation pattern.

Function Invocation

There are **four** patterns of invocation in JavaScript:

- Method invocation
- Function invocation
- Constructor invocation
- Apply invocation

The patterns differ in how the bonus parameter **this** is initialized.

Function Invocation

The invocation operator is a pair of parentheses (b)that follow any expression that produces a function value.

The parentheses can contain zero or more expressions, separated by commas. Each expression produces one argument value. Each of the argument values will be assigned to the function's parameter names.

There is no runtime error when the number of arguments and the number of parameters do not match. If there are too many argument values, the extra argument values will be ignored. If there are too few argument values, the undefined value will be substituted for the missing values.

There is no type checking on the argument values: any type of value can be passed to any parameter.

Method Invocation Pattern

When a function is stored as a property of an object, we call it a method. When a method is invoked, this is bound to that object.

If an invocation expression contains a refinement (that is, a . dot expression or [subscript] expression), it is invoked as a method:

```
var myObject = {
    value: 0,
    increment: function (inc) {
        this.value += typeof inc === 'number' ? inc : 1;
    }
};
myObject.increment( );
document.writeln(myObject.value); // 1
myObject.increment(2);
document.writeln(myObject.value); // 3
```

Function Invocation Pattern

When a function is not the property of an object, then it is invoked as a function:

```
var sum = add(3, 4); // sum is 7
```

When a function is invoked with this pattern, this is bound to the global object. This was a mistake in the design of the language. Had the language been designed correctly, when the inner function is invoked, this would still be bound to the this variable of the outer function.

A consequence of this error is that a method cannot employ an inner function to help it do its work because the inner function does not share the method's access to the object as its this is bound to the wrong value.

Function Invocation Pattern

Fortunately, there is an easy workaround. If the method defines a variable and assigns it the value of this, the inner function will have access to this through that variable. By convention, the name of that variable is that:

```
// Augment myObject with a double method.
myObject.double = function ( ) {
      var that = this;    // Workaround.
      var helper = function ( ) {
            that.value = add(that.value, that.value);
      };
      helper( ); // Invoke helper as a function.
};
// Invoke double as a method.
myObject.double( );
document.writeln(myObject.getValue( )); // 6
```

Constructor Invocation Pattern

JavaScript is a prototypal inheritance language. That means that objects can inherit properties directly from other objects. The language is class-free.

This is a radical departure from the current fashion. Most languages today are classical. Prototypal inheritance is powerfully expressive, but is not widely understood.

Constructor Invocation Pattern

If a function is invoked with the new prefix, then a new object will be created with a hidden link to the value of the function's prototype member, and this will be bound to that new object.

The new prefix also changes the behavior of the return statement.

Constructor Invocation Pattern

```
// Create a constructor function called Quo.
// It makes an object with a status property.

var Quo = function (string) {
      this.status = string;
};
// Give all instances of Quo a public
// called get_status.

Quo.prototype.get_status = function ( ) {
      return this.status;
};
// Make an instance of Quo.
var myQuo = new Quo("confused");
document.writeln(myQuo.get_status( ));   // writes confused
```

Apply Invocation Pattern

Because JavaScript is a functional object-oriented language, functions can have methods.

The apply method lets us construct an array of arguments to use to invoke a function. It also lets us choose the value of this.

The apply method takes two parameters. The first is the value that should be bound to this. The second is an array of parameters.

Apply Invocation Pattern

```
// Make an array of 2 numbers and add them.
var array = [3, 4];
var sum = add.apply(null, array); // sum is 7
// Make an object with a status member.
var statusObject = {
      status: 'A-OK'
};
// statusObject does not inherit from Quo.prototype,
// but we can invoke the get_status method on
// statusObject even though statusObject does not have
// a get_status method.
var status = Quo.prototype.get_status.apply(statusObject);
 // status is 'A-OK'
```

Apply Invocation Pattern

```
// Make an array of 2 numbers and add them.
var array = [3, 4];
var sum = add.apply(null, array); // sum is 7
// Make an object with a status member.
var statusObject = {
      status: 'A-OK'
};
// statusObject does not inherit from Quo.prototype,
// but we can invoke the get_status method on
// statusObject even though statusObject does not have
// a get_status method.
var status = Quo.prototype.get_status.apply(statusObject);
 // status is 'A-OK'
```

Function Arguments

A bonus parameter that is available to functions when they are invoked is the arguments array. It gives the function access to all of the arguments that were supplied with the invocation, including excess arguments that were not assigned to parameters.

This makes it possible to write functions that take an unspecified number of parameters.

Function Arguments

```
// Make a function that adds a lot of stuff.
var sum = function ( ) {
    var i, sum = 0;
    for (i = 0; i < arguments.length; i += 1) {
        sum += arguments[i];
    }
    return sum;
};
document.writeln(sum(4, 8, 15, 16, 23, 42)); // 108
```

Return

When a function is invoked, it begins execution with the first statement, and ends
when it hits the } that closes the function body. That causes the function to return
control to the part of the program that invoked the function.
The return statement can be used to cause the function to return early. When return is executed, the function returns immediately without executing the remaining statements.
A function always returns a value. If the return value is not specified, then undefined is returned.
If the function was invoked with the new prefix and the return value is not an object, then this (the new object) is returned instead.

Recursion

A recursive function is a function that calls itself, either directly or indirectly. Recursive functions can be very effective in manipulating tree structures such as the browser's Document Object Model (DOM). Each recursive call is given a smaller piece of the tree to work on:

Scope

Scope in a programming language controls the visibility and lifetimes of variables and parameters.
Most languages with C syntax have block scope. All variables defined in a block (a list of statements wrapped with curly braces) are not visible from outside of the block. The variables defined in a block can be released when execution of the block is finished. This is a good thing.
Unfortunately, JavaScript **does not have block scope** even though its block syntax suggests that it does.
This confusion can be a source of errors.

Scope

JavaScript does have function scope. That means that the parameters and variables defined in a function are not visible outside of the function, and that a variable defined anywhere within a function is visible everywhere within the function.
In many modern languages, it is recommended that variables be declared as late as possible, at the first point of use. That turns out to be bad advice for JavaScript because it lacks block scope.
So instead, it is best to declare all of the variables used in a function at the top of the function body.

Closure

The good news about scope is that inner functions get access to the parameters and variables of the functions they are defined within (with the exception of this and arguments). This is a very good thing.
A more interesting case is when the inner function has a longer lifetime than its outer function.
More on this in a bit

Callbacks

Callback are a powerful feature of JavaScript. Functions can make it easier to deal with discontinuous events. For example, suppose there is a sequence that begins with a user interaction, making a request of the server, and finally displaying the server’s response.

Callbacks

The naïve way to write that would be:

```
request = prepare_the_request( );
response = send_request_synchronously(request);
display(response);
```

The problem with this approach is that a synchronous request over the network will leave the client in a frozen state. If either the network or the server is slow, the degradation in responsiveness will be unacceptable.

Callbacks

A better approach is to make an asynchronous request, providing a callback function that will be invoked when the server's response is received. An asynchronous function returns immediately, so the client isn't blocked:

```
request = prepare_the_request( );
send_request_asynchronously(request, function (response) {
        display(response);
    });
```

We pass a function parameter (display) to the send_request_asynchronously function that will be called when the response is available.

Modules

In JS we can use functions and closure to make modules.
A module is a function or object that presents an interface but that hides its state and implementation.
By using functions to produce modules, we can almost completely eliminate our use of global variables, thereby mitigating one of JavaScript's worst features.
Many of the popular 'libraries' are indeed modules.

Cascade Functions

In JS some methods do not have a return value. For example, it is typical for methods that set or change the state of an object to return nothing.
If we have those methods return this instead of undefined, we can enable cascades.
In a cascade, we can call many methods on the same object in sequence in a single statement.

Cascade Functions

Cascades allow us to write in a style like this:

```
getElement('myBoxDiv').move(350, 150).width(100).height(100).color('red').
    border('10px outset').padding('4px').appendText("Please stand by").
    on('mousedown', function (m) {
        this.startDrag(m, this.getNinth(m));
        }).
    on('mousemove', 'drag').
    on('mouseup', 'stopDrag').
    later(2000, function ( ) {
        this.color('yellow').
        setHTML("What hath God wraught?").
        slide(400, 40, 200, 200);
    }).tip('This box is resizeable');
```

Curry Functions

Functions are values, and we can manipulate function values in interesting ways. Currying allows us to produce a new function by combining a function and an argument:

```
var add1 = add.curry(1);
document.writeln(add1(6)); // 7
```

Curry Functions

JavaScript does not have a curry method, but we can fix that by augmenting Function.prototype:

```
Function.method('curry', function ( ) {
    var slice = Array.prototype.slice,
        args = slice.apply(arguments),
        that = this;
    return function ( ) {
        return that.apply(null, args.concat(slice.apply(arguments)));
    };
});
```

Memoization Functions

Functions can use objects to remember the results of previous operations, making it possible to avoid unnecessary work.
This optimization is called ***memoization***. JavaScript's objects and arrays are very convenient for this.

Memoization Functions

The memoizer function will take an initial memo array and the fundamental function. It returns a shell function that manages the memo store and that calls thebfundamental function as needed. We pass the shell function and the function's parameters to the fundamental function:

```
var memoizer = function (memo, fundamental) {
        var shell = function (n) {
            var result = memo[n];
            if (typeof result !== 'number') {
                result = fundamental(shell, n);
                memo[n] = result;
            }
            return result;
        };
        return shell;
};
```

Memoization Functions

By devising functions that produce other functions, we can significantly reduce the amount of work we have to do. For example, to produce a memoizing factorial function,we only need to supply the basic factorial formula:

```
var factorial = memoizer([1, 1], function (shell, n) {
    return n * shell(n - 1);
});
```

Function Summary

Functions in JavaScript are objects.

Functions can be stored in variables, objects, and arrays.

Functions can be passed as arguments to functions, and functions can be returned from functions.

Also, since functions are objects, functions can have methods.

Function Summary

There are **four** patterns of invocation in JavaScript:

- Method invocation
- Function invocation
- Constructor invocation
- Apply invocation

Functions can be Cascaded, use Closures, be turned ins Asynchronous Callback, and be Curried.

JSON

JSON JavaScript Object Notation is a very popular data interchange format.

It was developed by Douglas Crockford.

JSON is text-based, lightweight, and a human-readable format for data exchange between clients and servers.

JSON is derived from JavaScript and bears a close resemblance to JavaScript objects, but it is not dependent on JavaScript.

JSON is language-independent, and support for the JSON data format is available in all the popular languages, some of which are C#, PHP, Java, C++, Python, and Ruby.

JSON is a format and not a language!

JSON vs XML

students.xml

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!-- This is an example students feed in XML -->
<students>
    <student>
        <studentid>101</studentid>
        <firstname>John</firstname>
        <lastname>Doe</lastname>
        <classes>
            <class>Business Research</class>
            <class>Economics</class>
            <class>Finance</class>
        </classes>
    </student>
    <student>
        <studentid>102</studentid>
        <firstname>Jane</firstname>
        <lastname>Dane</lastname>
        <classes>
            <class>Marketing</class>
            <class>Economics</class>
            <class>Finance</class>
        </classes>
    </student>
</students>
```

students.json

```json
/* This is an example students feed in JSON */
{
    "students": {
        "0": {
            "studentid": "101",
            "firstname": "John",
            "lastname": "Doe",
            "classes": [
                "Business Research",
                "Economics",
                "Finance"
            ]
        },
        "1": {
            "studentid": "102",
            "firstname": "Jane",
            "lastname": "Dane",
            "class": [
                "Marketing",
                "Economics",
                "Finance"
            ]
        }
    }
}
```

JSON

JSON is used by web applications to transfer data.

Prior to JSON, many application used XLM to exchange data.

Over the weekend read up on JSON at json.org

End of Slides

For next time

- introduction to JS libraries
  - jQuery
- Assignment 3

CS 547

JavaScript Frameworks

jQuery

Week 14 Day 2

4/30/2015

Agenda

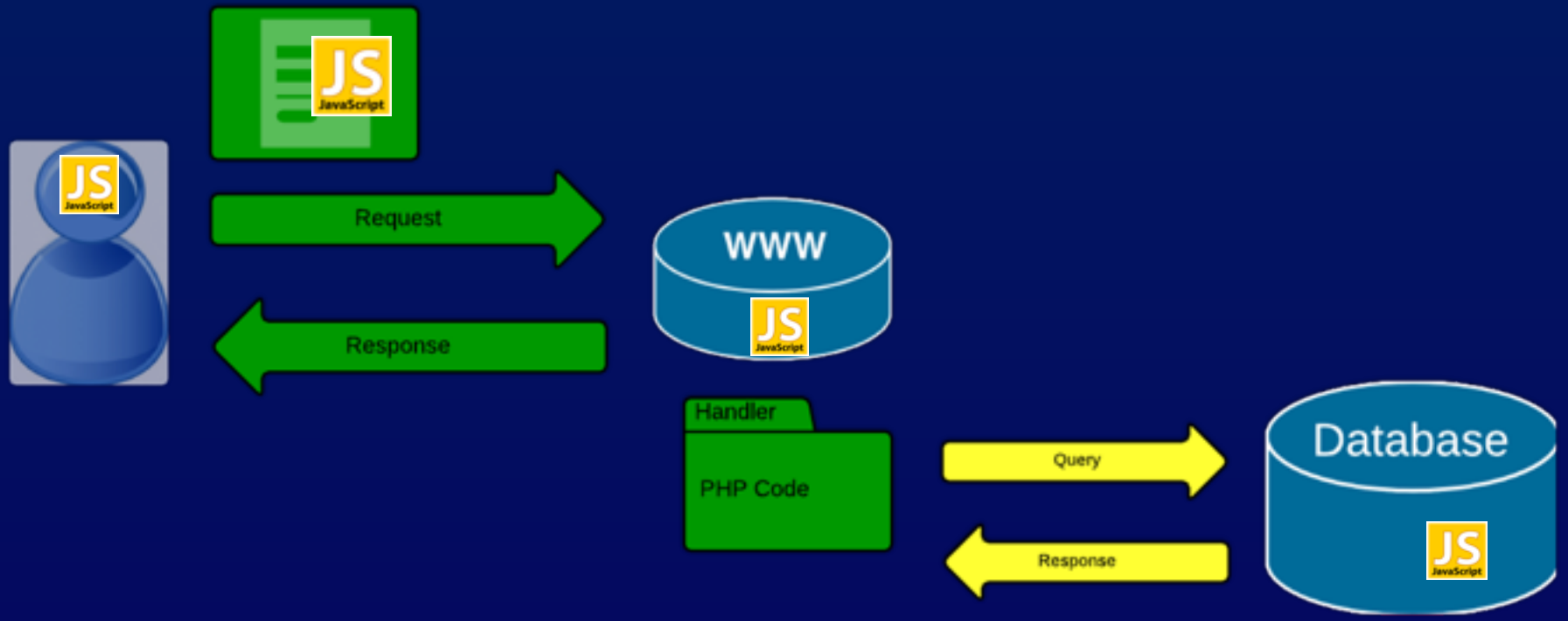
- jQuery

Pillow on Mac

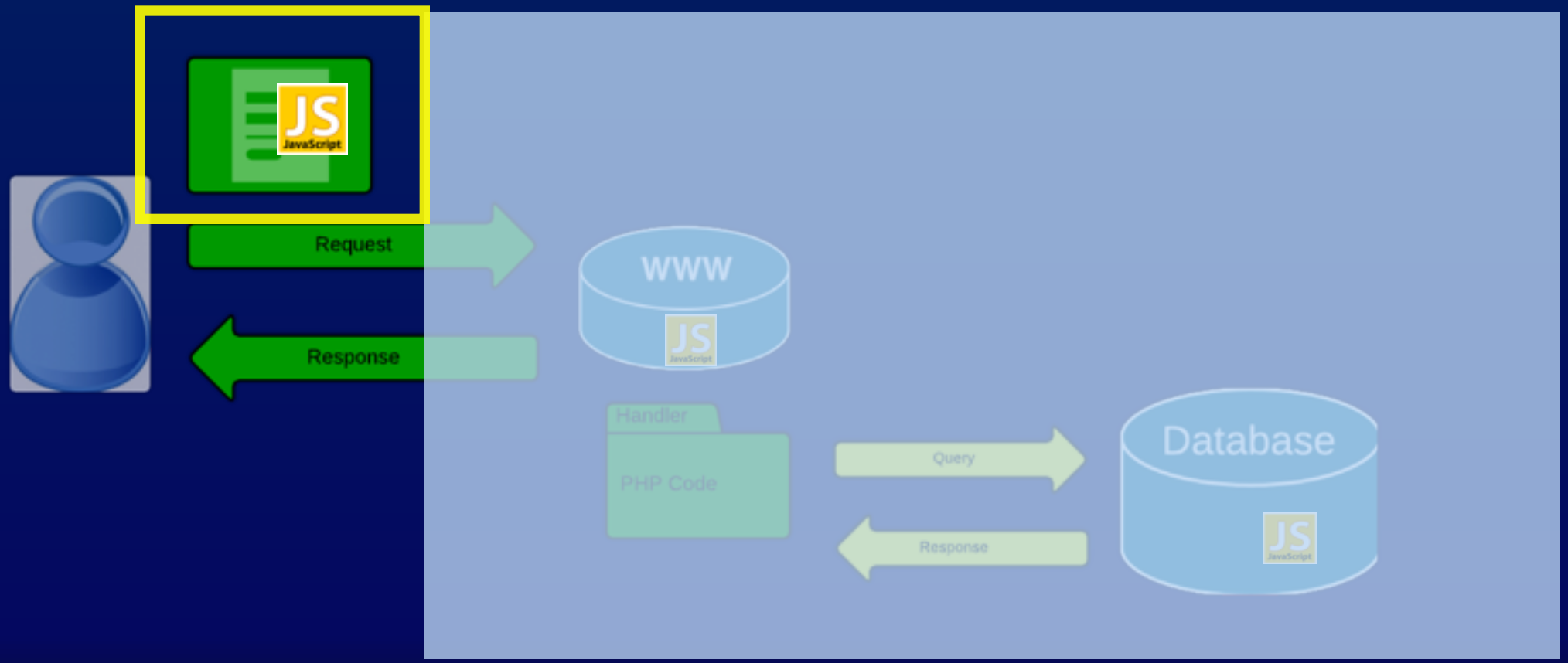
More students are reporting issues on mac

- use homebrew to install python 3
- use pip3 to install Pillow

JS Frameworks:



JS Frameworks



Frameworks

Frameworks are libraries written for the purpose of making some task easier.

One can manipulate the DOM directly, but it is easier to use a framework such as jQuery.

Using jQuery

Since jQuery is a library, you can include it in your web pages:

Locally:

```
<script src = “jquery.js”></script>
```

via CDN:

```
<script src = “//code.jquery.com/jquery-2.1.3.min.js”></script>
```

Using jQuery

There are two usage styles for jQuery:

Command:

- via the $ function, (factory method) chainable and return a jQuery object

Utility:

- via the $.-prefixed function. These do not act on the jQuery object directly

Using jQuery

```
<script>
    $(document).ready(function(){

   // jQuery methods go here...
</script>
```

Using jQuery

```
<script>
    $(document).ready(function(){

   // jQuery methods go here...
</script>
```

jQuery: Element Selector

```
<div id ="myDiv">
    <p> First Paragraph </p>
    <input id="mybutton" type="button" onclick="myClick();" value="Click here">
    <p> Second Paragraph </p>
    <button>Click me</button>

</div>
<script>
    $(document).ready(function(){
        $("button").click(function(){
            $("p").hide();     });
        $("#mybutton").click(function(){
            $("p").show();     });
    });


</script>
```

jQuery: #id Selector

```
<div id ="myDiv">
    <p> First Paragraph </p>
    <input id="mybutton" type="button" onclick="myClick();" value="Click here">
    <p> Second Paragraph </p>
    <button>Click me</button>

</div>
<script>
    $(document).ready(function(){
        $("button").click(function(){
            $("p").hide();     });
        $("#mybutton").click(function(){
            $("p").show();     });
    });


</script>
```

jQuery: class Selector

```
<div id ="myDiv">
    <p  class="myClass"> First Paragraph </p>
    <input id="mybutton" type="button" onclick="myClick();" value="Click here">
    <p> Second Paragraph </p>
    <button>Click me</button><input id="mySpecialbutton" type="button"  value="Click to hide myClass">


…
        $("button").click(function(){
            $("p").hide();     });
        $("#mybutton").click(function(){
            $("p").show();     })
        $("#mySpecialbutton").click(function(){
            $(".myClass").hide();     });
    });


</script>
```

jQuery Selectors

Syntax	Description
$("*")	Selects all elements
$(this)	Selects the current HTML element
$("p.intro")	Selects all <p> elements with class="intro"
$("p:first")	Selects the first <p> element
$("ul li:first")	Selects the first element of the first
$("ul li:first-child")	Selects the first element of every
$("[href]")	Selects all elements with an href attribute
$("a[target='_blank']")	Selects all <a> elements with a target attribute value equal to "_blank"
$("a[target!='_blank']")	Selects all <a> elements with a target attribute value NOT equal to "_blank"
$(":button")	Selects all <button> elements and <input> elements of type="button"
$("tr:even")	Selects all even <tr> elements
$("tr:odd")	Selects all odd <tr> elements

jQuery Events

Mouse Events	Keyboard Events	Form Events	Document/Window Events
click	keypress	submit	load
dblclick	keydown	change	resize
mouseenter	keyup	focus	scroll
mouseleave		blur	unload

jQuery Events Syntax

In jQuery, most DOM events have an equivalent jQuery method.

1. assign a click event

```
$("p").click();
```

2. define what should happen when the event fires. You must pass a function to the event:

```
$("p").click( function(){
    // action goes here!!
});
```

jQuery Effects

jQuery offers convenience methods for most native browser events.

These methods — including .click(), .focus(), .blur(), .change(), etc. — are shorthand for jQuery's .on() method.

Note the .on() can only create event listeners on elements that exist *at the time you set up the listeners*. Similar elements created after the event listeners are established will not automatically pick up event behaviors you've set up previously.

jQuery Effects

In jQuery, effects are actions that your webpage can do or respond to .

- Can be based on
  - user actions – text entered in a field
  - other events – mouse or clicks
  - browser itself – page load or unload

jQuery: class Selector

```
<div id ="myDiv">
    <input id="myFadebutton" type="button"  value="Click to fade this div myClass">
    <div id="div1" style="width:80px;height:80px;display:none;background-color:red;"></div><br>
…
        $("button").click(function(){
            $("p").hide();     });
        $("#mybutton").click(function(){
            $("p").show();     })
        $("#mySpecialbutton").click(function(){
            $(".myClass").hide();     });
        $("#myFadebutton").click(function(){
            $("#div1").fadeIn(3000);     });
    });


</script>
```

jQuery Effects

Method	Description
animate()	Runs a custom animation on the selected elements
clearQueue()	Removes all remaining queued functions from the selected elements
delay()	Sets a delay for all queued functions on the selected elements
dequeue()	Removes the next function from the queue, and then executes the function
fadeIn()	Fades in the selected elements
fadeOut()	Fades out the selected elements
fadeTo()	Fades in/out the selected elements to a given opacity
fadeToggle()	Toggles between the fadeIn() and fadeOut() methods

jQuery Effects

Method	Description
finish()	Stops, removes and completes all queued animations for the selected elements
hide()	Hides the selected elements
queue()	Shows the queued functions on the selected elements
show()	Shows the selected elements
slideDown()	Slides-down (shows) the selected elements
slideToggle()	Toggles between the slideUp() and slideDown() methods
slideUp()	Slides-up (hides) the selected elements
stop()	Stops the currently running animation for the selected elements
toggle()	Toggles between the hide() and show() methods

jQuery HTML

jQuery is used to manipulate and interact with the DOM or the html page. Three important methods are:

- text() - Sets or returns the text content of selected elements
- html() - Sets or returns the content of selected elements (including HTML markup)
- val() - Sets or returns the value of form fields

jQuery HTML

Get Context:

```
$("#test").text());     // gets the text of the test element

$("#test").html());   // gets the html code from the test element

$("#test").val());     // returns the value of an input element
```

jQuery HTML

the same three methods from before are used to set content:

- text() - Sets or returns the text content of selected elements
- html() - Sets or returns the content of selected elements (including HTML markup)
- val() - Sets or returns the value of form fields

jQuery HTML

Set Context:

```
$("#test").text(“some text”));     // sets test element to some text

$("#test").html(“some html”));   // sets test element to some html

$("#test").val(“a value”));     // sets the value of an input element
```

jQuery Add to HTML

There are four jQuery methods that are used to add new content:

- append() - Inserts content at the end of the selected elements
- prepend() - Inserts content at the beginning of the selected elements
- after() - Inserts content after the selected elements
- before() - Inserts content before the selected elements

jQuery New HTML

There are three ways of creating new HTML elements:

1. using html – var txt1 = " Text. ";

2. using jQuery – var txt2 = $(" ").text("Text.");

3. using DOM – txt3 = document.createElement("p");

jQuery Removing HTML

There are two jQuery methods to remove elements

- remove() - Removes the selected element (and its child elements)
- empty() - Removes the child elements from the selected element

jQuery Validating forms

You can use several plugins to validate form data with jQuery

Search the jQuery plugin site http://plugins.jquery.com/ for validators.