

CS472: WEB PROGRAMMING

MODULES AND OBJECTS IN JAVASCRIPT

Some contents of this document are Copyright 2012 Marty Stepp, Jessica Miller, Victoria Kirst and Roy McElmurry IV. All rights reserved. Any redistribution, reproduction, transmission, or storage of part or all of the contents in any form is prohibited without the author's expressed written permission. Slides have been modified for Maharishi University of Management Computer Science course CS472 in accordance with instructors agreement with authors.

Maharishi University of Management -Fairfield, Iowa © 2016



All rights reserved. No part of this slide presentation may be reproduced or utilized in any form or by any means, electronic or mechanical, including photocopying or recording, or by any information storage and retrieval system, without permission in writing from Maharishi University of Management.

Main Point Preview

JavaScript Objects

Objects are a widely used encapsulation mechanism in JavaScript. They are easily created with object literals. They can dynamically add new properties; behave like associative arrays; must use 'this' to refer to properties;

Science of Consciousness:

Objects make code efficient and effective by encapsulating related pieces of functionality. With repeated experience of transcending unbounded awareness develops to encapsulate individual awareness. Individual awareness becomes more efficient and effective if it is supported by unbounded awareness.

How about classes and objects?

- small programs are easily written without objects
- larger programs become cluttered with disorganized functions
- objects group related data and behavior
 - helps manage size and complexity, promotes code reuse
- You have already used many types of JavaScript objects
 - Strings, arrays, HTML / XML DOM nodes
 - global DOM objects
 - The jQuery object (following lessons)

Creating objects via object literal

```
const name = {  
  'fieldName': value,  
  ...  
  'fieldName': value  
};  
const pt = {  
  'x': 4,  
  'y': 3  
};  
alert(pt.x + ", " + pt.y);
```

- in JavaScript, you can create a new object without creating a class
- the above is like a Point object; it has fields named x and y
- the object does not belong to any class; it is the only one of its kind, a singleton
 - `typeof(pt) === "object"`

JavaScript objects

- objects in JavaScript are like associative arrays
- the keys can be any string
- you do not need quotes if the key is a valid JavaScript identifier
- values can be anything, including functions
- you can add keys dynamically using associative array or the . syntax
- object properties that have functions as their value are called 'methods'

```
const x = {  
  'a': 97,  
  'b': 98,  
  'c': 99,  
  'd': 199,  
  'mult': function(x,y) {  
    return x * y;  
  }  
};
```

Common examples of using object literals

```
$.ajax("http://example.com/app.php",  
{  
  'method': "post", //object with field named method  
  'timeout': 2000 //and a field name timeout  
});
```

```
$("<div>", {  
  'css': {'color': red }, //a css field  
  'id': 'myid', //an id field  
  'click': myClickHandler //and a method called click  
});
```

- the parameters in {} passed to jQuery functions and methods are object literals
- object literals are the basis of JSON

Main Point

JavaScript Objects

Objects are a widely used encapsulation mechanism in JavaScript. They are easily created with object literals. They can dynamically add new properties; behave like associative arrays; must use 'this' to refer to properties;

Science of Consciousness:

Objects make code efficient and effective by encapsulating related pieces of functionality. With repeated experience of transcending unbounded awareness develops to encapsulate individual awareness. Individual awareness becomes more efficient and effective if it is supported by unbounded awareness.

Main Point Preview

the keyword 'this'

In JavaScript, like Java, the keyword 'this' refers to the containing object. However, in JavaScript the same 'this' can refer to many different types of objects depending on the context.

Science of Consciousness: The keyword 'this' is an important form of self-referral and understanding this self-referral is critical to writing successful JavaScript. Experiencing and understanding self-referral consciousness is critical to living a successful life.



Objects that have behavior

```
const name = {  
  ...  
  methodName: function(parameters) { statements; }  
};  
  
const pt = {  
  x: 4,  
  y: 3,  
  distanceFromOrigin: function() {  
    return Math.sqrt(this.x * this.x + this.y * this.y);  
  }  
};  
alert(pt.distanceFromOrigin()); // 5
```

- like in Java, objects' methods run "inside" that object
 - inside an object's method, the object refers to itself as `this`
 - unlike Java, `this` keyword is mandatory inside JS objects

this

- In Java, every method has an implicit variable 'this' which is a reference to the object that contains the method
 - Java, in contrast to JavaScript, has no functions, only methods
 - So, in Java, it is always obvious what 'this' is referring to
- In JavaScript, 'this', usually follows the same principle
 - Refers to a property on the containing object
 - If in a method, refers to a property of the object, just like Java
 - If in a function, then the containing object is 'window'
 - Not in "use strict" mode → undefined
 - Methods and functions can be passed to other objects!!
 - 'this' is then a portable reference to an arbitrary object



'this' inside vs outside object

```
function foo() { console.log(this); }  
const bob = {  
  log: function() {  
    console.log(this);  
  }  
};
```

console.log(this); // this generally is window object
foo(); //foo() is called by global window object
bob.log(); //log() is called by the object, bob

this inside a function (-> window)



```
bob.f() { console.log(this); } // an object—"bob";  
f() { console.log(this); } //window
```

```
function sam() {  
  this.newvar = "hello";  
}  
console.log(newvar); // Uncaught ReferenceError: newvar is not defined  
sam(); // this = window  
console.log(newvar); //hello
```

this inside event handler

- When using `this` inside an event handler, it will always refer to the invoker. (event.target)
 - A very useful feature of 'this' for JavaScript and DOM manipulation
 - Portable context
 - Rule: 'this' refers to the object that called the function

```
const changeMyColorButton1 = document.getElementById("btn1");  
const changeMyColorButton2 = document.getElementById("btn2");
```

```
changeMyColorButton1.onclick = changeMyColor;  
changeMyColorButton2.onclick = changeMyColor;
```

```
function changeMyColor() {  
  this.style.backgroundColor = "red";  
}
```

Self Pattern – problem with inner functions



```
1  var a = {  
2    name: "",  
3    log: function() {  
4      this.name = "Hello";  
5      console.log(this.name); //Hello  
6      var setFrench = function(newname) {  
7        this.name = newname;  
8      };  
9      setFrench("Bonjour");  
10     console.log(this.name); //Hello  
11   }  
12 };  
13  
14 a.log();|
```

- Line 10: why not Bonjour?
- JavaScript functions treat 'this' as 'window' (even inner functions)
 - "bad part of JS"
- Line 9: calling setFrench from window!
- Line 7: creating variable window.name = newname;



Self Pattern – Legacy Solution

```
var a = {  
  name: "",  
  log: function() {  
    var self = this;  
    self.name = "Hello";  
    console.log(self.name); //Hello  
    var setFrench = function(newname) {  
      self.name = newname;  
    };  
    setFrench("Bonjour");  
    console.log(self.name); //Bonjour  
  }  
};
```

```
a.log();
```

- Self Pattern: Inside objects, always create a “self” variable and assign “this” to it. Use “Self” anywhere else
- JavaScript functions (versus methods) always use ‘window’ as ‘this’, even inner functions in methods



this inside arrow function (ES6)

- Also solves the Self Pattern problem
- 'this' will refer to surrounding lexical scope inside arrow function

```
const a = {  
  name: "",  
  log: function() {  
    this.name = "Hello";  
    console.log(this.name); //Hello  
    const setFrench = (newname => this.name = newname);  
    setFrench("Bonjour");  
    console.log(this.name); //Bonjour  
  }  
};  
  
a.log();
```

.call() .apply() .bind()

- There are many helper methods on the Function object in JavaScript
 - .bind() when you want that function to be called back later with a certain context, useful in events. (ES5)
 - .call() or .apply() when you want to invoke the function immediately, and modify the context.
 - <http://stackoverflow.com/questions/15455009/javascript-call-apply-vs-bind>

```
var func2 = func.bind(anObject , arg1, arg2, ...) // creates a copy of  
func using anObject as 'this' and its first 2 arguments bound to arg1  
and arg2 values
```

```
func.call(anObject, arg1, arg2...);
```

```
func.apply(anObject, [arg1, arg2...]);
```



.bind() values without invoking function (ES5)

```
//Recall closure example for event handling from previous lesson
//needed to pass a parameter without executing the function
// returned an inner function with closure over the bound parameter
// can have same effect by binding a null context value with the required parameter
```

```
<a href="#" id="size-12">Size 12</a>
<a href="#" id="size-16">Size 16</a>
<a href="#" id="size-18">Size 18</a>
```

```
function makeSizer(size) {
  return function() {
    document.body.style.fontSize = size + "px";
  };
}
```

```
function makeSizerSimple(size){ //can use this version with .bind()
  document.body.style.fontSize = size + "px";
}
document.getElementById("size-12").onclick = makeSizer(12);
document.getElementById("size-16").onclick = makeSizer(16);
document.getElementById("size-18").onclick = makeSizerSimple.bind(null,18); //null ok if not using 'this'
```



‘Borrow’ a method that uses ‘this’

```
const me = {
  first: 'Tina',
  last: 'Xing',
  getFullName: function() {
    return this.first + ' ' + this.last;
  }
}

const log = function(height, weight) { // 'this' refers to the invoker
  console.log(this.getFullName() + height + ' ' + weight);
}

const logMe = log.bind(me);
logMe('180cm', '70kg'); // Tina Xing 180cm 70kg

log.call(me, '180cm', '70kg'); // Tina Xing 180cm 70kg
log.apply(me, ['180cm', '70kg']); // Tina Xing 180cm 70kg
```



Function (method) Borrowing with 'apply'

```
var me = {  
  first: 'Tina',  
  last: 'Xing',  
  getFullName: function() {  
    return this.first + ' ' + this.last;  
  }  
};
```

```
var you = {  
  first: 'Rujuan',  
  last: 'Xing'  
};
```

```
console.log(me.getFullName.apply(you)); // Rujuan Xing  
//would it work with call?  How about bind?
```



Function Currying with 'bind'

- Recall: `bind` takes a context (`this`) and set of parameters, and returns a function that has that context and parameters set.
- This can be used in a functional programming technique called 'currying' which creates a new function from an old one by fixing the values of one or more parameters.

```
function multiply(a, b) {  
  return a * b; //no usage of 'this'  
}
```

```
const multiplyByTwo = multiply.bind(null, 2); // set a = 2  
console.log(multiplyByTwo(4)); // 8
```

```
const multiplyByThree = multiply.bind(null, 3); // set a = 3  
console.log(multiplyByThree(4)); // 12
```

Main Point Preview

the keyword 'this'

In JavaScript, like Java, the keyword 'this' refers to the containing object. However, in JavaScript the same 'this' can refer to many different types of objects depending on the context.

Science of Consciousness: The keyword 'this' is an important form of self-referral and understanding this self-referral is critical to writing successful JavaScript. Experiencing and understanding self-referral consciousness is critical to living a successful life.

Main Point Preview

Revealing Module Pattern

The revealing module pattern is widely used to provide a public API to an underlying implementation of private methods and properties.

Science of Consciousness: The Transcendental Meditation program is a sort of API to access the support of all the laws of nature through the experience of pure consciousness, the source of all the laws of nature.

Encapsulation and namespace protection with closures

- Languages such as Java provide private methods
 - can only be called by other methods in the same class.
- JavaScript does not provide this, but possible to emulate with closures.
- also provide powerful way of managing global namespace,
- Here's how to define public functions that access private functions and variables, using closures
 - **module pattern:**
- “Every real JavaScript programmer should know this if he or she wants to become great” Joe Zim

Module pattern - IIFE

```
(function(params) {  
    statements;  
}) (params);
```



```
(function(params) {  
    statements;  
} (params));
```

- declares and immediately calls an anonymous function
 - parens around function are a special syntax that means this is a function expression that will be immediately invoked
 - – “immediately invoked function expression (IIFE)”
 - used to create a new scope and closure around it
 - can help to avoid declaring global variables/functions
 - used by JavaScript libraries to keep global namespace clean



Module Pattern Example

```
// old: 3 globals
var count = 0;
function incr(n) {
  count += n;
}
function reset() {
  count = 0;
}
```

```
incr(4);
incr(2);
console.log("count: " + count);
```

```
// new: 0 globals
(function() {
  var count = 0;
  function incr(n) {
    count += n;
  }
  function reset() {
    count = 0;
  }

  incr(4);
  incr(2);
  console.log("count: " + count);
})();
```

- Declare-and-call protects your code and avoid globals
 - Avoids common problem with namespace/name collisions



IIFE and ES6

```
(function() {  
  var count = 0;  
  function incr(n) {  
    count += n;  
  }  
  function reset() {  
    count = 0;  
  }  
  
  incr(4);  
  incr(2);  
  console.log("count: " +  
    count);  
})();
```

```
{  
  let count = 0;  
  const incr = function(n) {  
    count += n;  
  }  
  const reset = function(n) {  
    count = 0;  
  }  
  incr(4);  
  incr(2);  
  console.log(count);  
};  
//block scope can replace IIFE for  
//namespace encapsulation
```

RECALL: Common closure bug with fix



```
var funcs = [];  
for (var i = 0; i < 5; i++) {  
  funcs[i] = function() {  
    return i;  
  };  
}  
console.log(funcs[0]());  
console.log(funcs[1]());  
console.log(funcs[2]());  
console.log(funcs[3]());  
console.log(funcs[4]());
```

- Closures that bind a loop variable often have this bug.
- Why do all of the functions return 5?

```
var helper = function(n) {  
  return function() {  
    return n;  
  }  
}  
var funcs = [];  
for (var i = 0; i < 5; i++) {  
  funcs[i] = helper(i);  
};  
console.log(funcs[0]());  
console.log(funcs[1]());  
console.log(funcs[2]());  
console.log(funcs[3]());  
console.log(funcs[4]());
```

Recall: Fix Closure bug with IIFE (“Module Pattern”)

```
var funcs = [];  
for(var i = 0; i < 5; i++){  
  funcs[i] = function() {  
    return i;  
  };  
}  
console.log(funcs[0]());  
console.log(funcs[1]());  
console.log(funcs[2]());  
console.log(funcs[3]());  
console.log(funcs[4]());
```



```
var funcs = [];  
for (var i = 0; i < 5; i++) {  
  funcs[i]=(function(n) {  
    return function()  
    {  
      return n;  
    }  
  }(i));  
};  
  
console.log(funcs[0]());  
console.log(funcs[1]());  
console.log(funcs[2]());  
console.log(funcs[3]());  
console.log(funcs[4]());
```

Revealing Module Pattern

```
/* widely used in single page web apps */  
const Module = (function() {  
  const privateFunction = function() {  
    // private  
  };  
  const someFunction = function() {  
    // public  
  };  
  const anotherFunction = function() {  
    // public  
  };  
  return {  
    someMethod: someFunction,  
    anotherMethod: anotherFunction  
  };  
})();
```


Accessing Private Methods

```
const myModule = (function() {  
  const privateFunction = function(message) {  
    console.log(message);  
  };  
  const publicFunction = function(text) {  
    privateFunction (text);  
  };  
  return {  
    publicMethod: publicFunction  
  };  
})();
```

```
// Example of passing data into a private method  
// Private method will console.log() 'Hello!'  
myModule.publicMethod('Hello!');
```

Access Private Variables

```
const Module = (function() {  
  const privateArray = [];  
  const publicFunction = function(something) {  
    privateArray.push(something);  
  };  
  return {  
    publicMethod: publicFunction  
  };  
})();
```

Extending Modules

```
/* very easy due to dynamic nature of JavaScript—can dynamically add properties to
   objects */
const Module = (function() {
  const privateFunction = function() {
    // private
  };
  const someFunction = function() {
    // public
  };
  const anotherFunction = function() {
    // public
  };
  return { someMethod: someFunction, anotherMethod: anotherFunction };
})();

Module.extension = function() {
  // another method! (Q: public or private?)
};
```



Example (revealing module pattern)

```
const counter = (function() {  
  let privateCounter = 0; //private data  
  function changeBy(val) { //private inner function  
    privateCounter += val;  
  }  
  return {  
    increment: function() { // three public functions are closures that share the same environment.  
      changeBy(1);  
    },  
    decrement: function() {  
      changeBy(-1);  
    },  
    value: function() {  
      return privateCounter;  
    }  
  }  
})();  
  
alert(counter.value()); /* Alerts 0 */  
counter.increment();  
counter.increment();  
alert(counter.value()); /* Alerts 2 */  
counter.decrement();  
alert(counter.value()); /* Alerts 1 */
```



Module factory example

```
const makeCounter = function() {  
  let privateCounter = 0;  
  function changeBy(val) {  
    privateCounter += val;  
  }  
  return {  
    increment: function() {  
      changeBy(1);  
    },  
    decrement: function() {  
      changeBy(-1);  
    },  
    value: function() {  
      return privateCounter;  
    }  
  }  
};
```

- We could store this function in a separate variable and use it to create several counters.

```
const counter1 = makeCounter();  
const counter2 = makeCounter();  
alert(counter1.value()); /* Alerts 0  
*/  
counter1.increment();  
counter1.increment();  
alert(counter1.value()); /* Alerts 2  
*/  
counter1.decrement();  
alert(counter1.value()); /* Alerts 1  
*/  
alert(counter2.value()); /* Alerts 0  
*/
```

Main Point

Revealing Module Pattern

The revealing module pattern is widely used to provide a public API to an underlying implementation of private methods and properties.

Science of Consciousness: The Transcendental Meditation program is a sort of API to access the support of all the laws of nature through the experience of pure consciousness, the source of all the laws of nature.

CONNECTING THE PARTS OF KNOWLEDGE WITH THE WHOLENESS OF KNOWLEDGE

Life Is Structured in Layers

1. JavaScript is a functional OO language with objects but no classes.
 2. Closures and objects are fundamental to JavaScript best coding practices, particularly for promoting encapsulation, layering, and abstractions in code.
-
3. **Transcendental consciousness** is the experience of the most fundamental layer of all existence, pure consciousness, the experience of one's own Self.
 4. **Impulses within the transcendental field:** The many layers of abstraction required for sophisticated JavaScript implementations will be most successful if they arise from a solid basis of thought that is supported by all the laws of nature.
 5. **Wholeness moving within itself:** In unity consciousness, one appreciates that all complex systems are ultimately compositions of pure consciousness, one's own Self.



References

<http://www.cs.washington.edu/education/courses/cse341/10au/lectures/slides/27-scope-closures.pdf>

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

<http://www.jblearning.com/catalog/9780763780609/>