CS472: WEB PROGRAMMING
MODULES AND OBJECTS IN JAVASCRIPT

Maharishi University of Management -Fairfield, Iowa © 2016

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 JavasScript are like associative arrays
- the keys can be any string
- you do not need quotes if the key is a valid JavasScript 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", //an object with a 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 al() { console.log(this); }
const bob = {
 log: function() {
   console.log(this);
 }
};

console.log(this); // this generally is window object
al(); //al() 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 multipleByTwo = multiply.bind(null, 2); // set a = 2
console.log(multipleByTwo(4)); // 8

const multipleByThree = multiply.bind(null, 3); // set a = 3
console.log(multipleByThree(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.

Main Point Preview
Inheritance

JavaScript supports prototype inheritance so that objects can inherit common functionality from a single ‘prototype’ object.

Science of Consciousness:

Pure consciousness is a level of awareness that is a common experience shared by everyone.

Prototype inheritance

- If use the IIFE syntax to make different instances of the module, then duplicate all the module code every time create a new module instance
- The makeCounter function is an object factory
  - – allows reuse of the module pattern functionality
- Problem: Why is an object factory inefficient when the methods become nontrivial?
- Solution: use Object.create() to inherit shared properties

Prototype Inheritance

- JavaScript supports prototype inheritance.
- Every JavaScript object has a (hidden) prototype property (__proto__)
  - __xyz__ underline notation meant to indicate this is not to be used by programmers, only the compiler
- objects get access to properties and methods of their prototype objects.
- `Object` is the end of the prototype chain.

```
const a = {};
// a.__proto__ is Object

const b = function() {};
// b.__proto__ is function
// b.__proto__.__proto__ is Object


const c = [];
// c.__proto__ is array
// c.__proto__.__proto__ is Object
```

Creating Objects via Object.create

- ES5 way to create objects is: **Object.create(object)**
- It sets **__proto__** property to the passed object for inheritance.

```
const person = {
 first: 'John',
 last: 'Doe',
 greet: function() { return 'Hi' + this.first; } //use
 this in methods to access properties
}

const p1 = Object.create(person);
console.log(p1.first); // John - Inheritance
console.log(p1.hasOwnProperty('first')); // false
p1.first = 'Tina';
console.log(p1.hasOwnProperty('first')); // true
console.log(p1); // {first: 'Tina'} - No last & greet()
p1.greet(); // Hi Tina
```

Object.create example of course object

```
// An Object
 const course = {
 coursename: 'Default',
 register: function() {
   return 'Register ' + this.coursename;
 }
}

const mwp = Object.create(course);
mwp.coursename = 'MWP';

console.log(mwp); // Object {coursename: "MWP"}
console.log(mwp.__proto__); // course Object
console.log(mwp.register()); // Register MWP
```

Function Constructors (“Classical”/class-like Inheritance)

- A constructor is a function that creates and *automatically* returns an object
  - Implicit (automatic) return of the constructed object
  - i.e., does not have an explicit *return* statement
- By convention function constructors start with a capital letter.
- To create new object from a function constructor *must* use the **new** keyword.

- new keyword does the following:
  - create an empty Object
  - set value of __proto__ property of new object to point to prototype object of the constructor function
    - every function object automatically has a *prototype* property
  - create **this variable** (implicitly) that points to the new object
  - Run constructor function
  - return **this** (implicitly)

- The ‘prototype’ property is used to add new functionality to any objects created from the constructor function

- new X() is Object.create(X.prototype) plus run constructor function

- “Bad” part of JavaScript: easy to forget the new
  - Function will run, but not behave as expected
  - ES6 ‘class’ syntax intended to remedy

Create objects via Function Constructors

```
function Person() {
 console.log(this);
 this.university = 'MUM';
 year = '2016';
}

const faculty1 = new Person(); // Person {university: "MUM"} - no
  year!

Person.prototype.greet = function() {
 return 'Hi ' + this.university;
}

const greeting = faculty1.greet();
console.log(greeting); // "Hi MUM"
```

Why this is awesome? Because we can create thousands of objects from the original function constructor with less memory space. And we can extend the functionality of all objects at runtime by adding methods and properties to the **prototype property.** ***(distinct from __proto__ which is used by the JSengine)***

Create course objects via function constructors

```
// By convention we use capital first letter for function constructor
function Course(coursename) {
 this.coursename = coursename;
 console.log('Function Constructor Invoked!');
  //implicit return of 'this' when called via 'new'
}

//add function register to prototype of all course objects (created from //Course
  constructor)
Course.prototype.register = function() {
 return 'Register ' + this.coursename;
}

const wap = new Course('WAP'); // Function Constructor Invoked!
console.log(wap); // Course {coursename: "WAP"}
console.log(wap.__proto__); // Course.prototype
console.log(wap instanceof Course); // true
console.log(Course.prototype.register); // function(){ ... }
console.log(wap.register()); // Register WAP
```

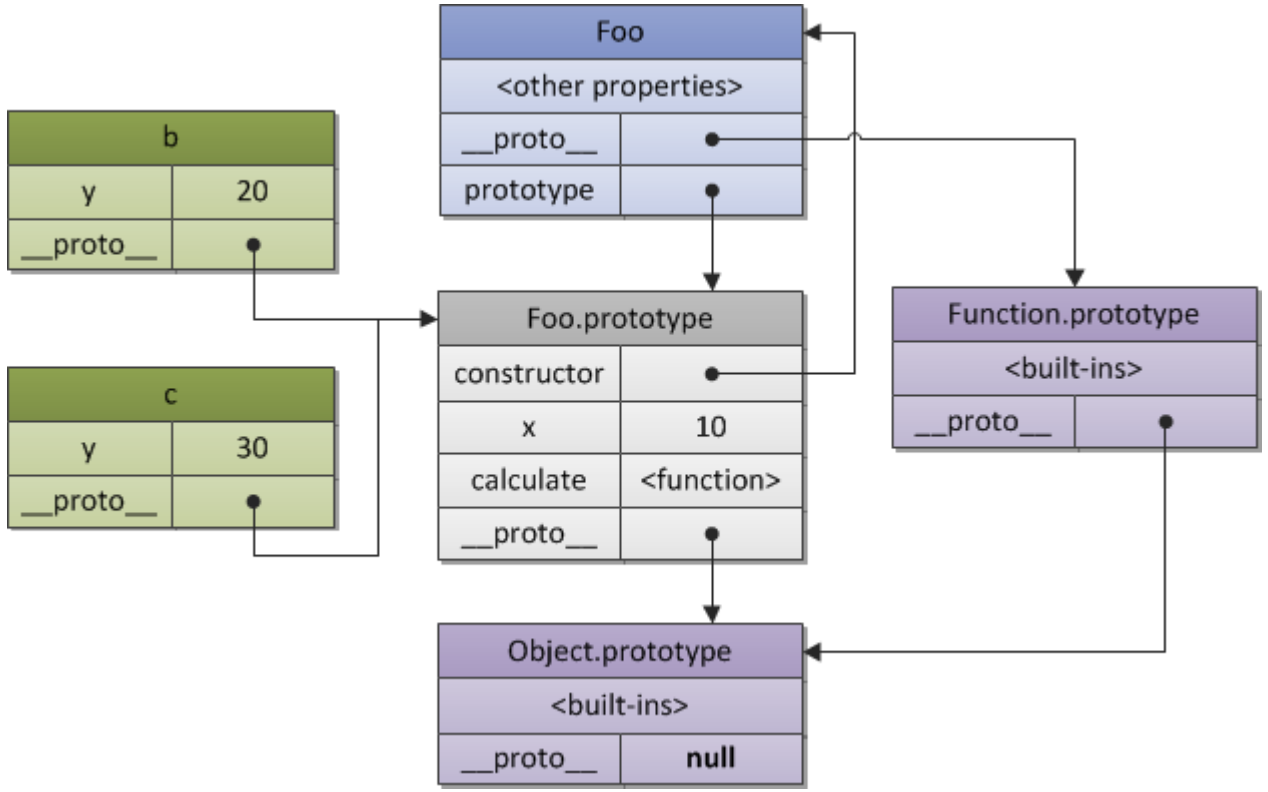
Function constructors prototype diagram

```
// a constructor function
function Foo(x) {
  this.y = x;
}


/*
__proto__ property of new objects point to prototype object of the constructor function so we may use it to define

shared/inherited properties or methods  "x“ and "calculate“
*/
Foo.prototype.x = 10;
Foo.prototype.calculate = function (z) {
  return this.x + this.y + z;
};


// now create our "b" and "c" objects using "pattern" Foo
var b = new  Foo(20);
var c = new  Foo(30);
);
```



//Ref: http://dmitrysoshnikov.com/ecmascript/javascript-the-core/

Built-in Function Constructors

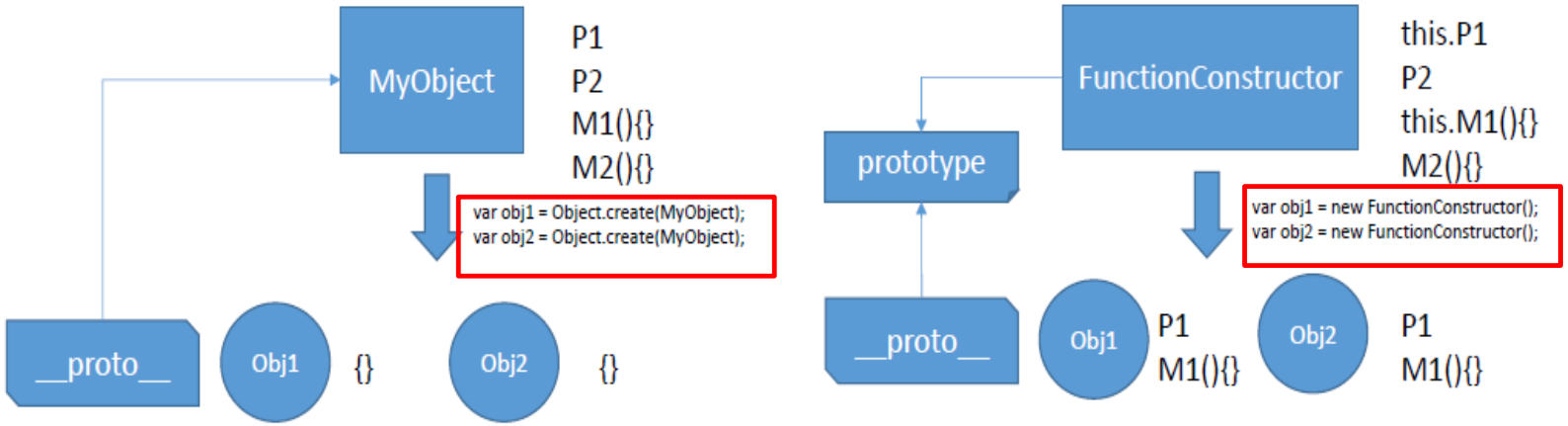
```
const a = new Number(12);
const b = new String("Hello");
const c = new Date(2016, 03, 01);

/* Number.prototype, String.prototype, Date.prototype
 are objects with helper methods
 available because objects were created using new()
 keyword */

a.toString(); // "12"
b.italics(); // "<i>Hello</i>"
c.getMonth(); // 3
```

Creating Objects Comparison

- Obj1 and Obj2 __proto__ properties point to
  - MyObject for Object.create
  - FunctionConstructor.prototype for FunctionConstructor
- Extensions are made by adding new properties to
  - the prototype object with Object.create
  - FunctionConstructor.prototype with function constructors.
- new X() is Object.create(X.prototype) plus run constructor function



Review – How to create Objects in JS

- Object.create();
  - The prototype chain (__proto__) will refer to object passed to create(someObj)
  - Have to explicitly add any nonprototype (“own”) fields to each new object
    - Could be done in a factory function
- Function Constructors: new FunctionConstructor();
  - properties and methods in the constructor with `this` become “own” fields on each new object
    - Do not have to explicitly add own fields to each new instance outside constructor
  - The prototype chain(__proto__) will refer to the function constructor’s `prototype` property
- to extend functionality must have
  - if use Object.create then need the prototype object
  - If use function constructor then need the function constructor
    - Use its prototype property, which will reference the prototype
  - “Bad” part of JavaScript: easy to forget the new
    - Function will run, but not behave as expected
    - ES6 ‘class’ syntax intended to remedy

Inheritance hierarchy via prototype object vs function constructor

```
const Mammal = {
    name: "unknown",
    saySomething: function () {
        console.log(my name is ' + this.name)
    },
    doSomething() {
        console.log(this.name + ' is walking
about.');
    }
}
//create subclass—prototype object that has
//Mammal prototype object as its __proto__
const Dog = Object.create(Mammal);
Dog.saySomething = function () {
    console.log(Woof, my name is ' +
this.name);
}
//create an object
const snoopy = Object.create(Dog);
snoopy.name = Snoopy';
snoopy.saySomething();
snoopy.doSomething();
```

```
const Mammal = function (name) {
    this.name = name;
}
Mammal.prototype.saySomething = function () {
    console.log(' my name is ' + this.name);
}
Mammal.prototype.doSomething = function() {
    console.log(this.name + ' is walking about.');
}

//create subclass—constructor function whose prototype
//object has its __proto__ point to prototype object of the
//Mammal constructor function
const Dog = function (name) {
 Mammal.call(this, name);
}
Dog.prototype = Object.create(Mammal.prototype);
Dog.prototype.saySomething = function () {
    console.log('Woof, my name is ' + this.name);
}
//create an object
const snoopy = new Dog(Snoopy ');
snoopy.saySomething();
snoopy.doSomething();
```

Inheritance hierarchy via ES6 class syntax

```
class Mammal {
    constructor(name) {  this.name = name;   }
    saySomething() {  console.log('Hi, my name is' + this.name);   }
    doSomething() {console.log(this.name + ' is walking about.');   }
}

//create a sub class
class Dog extends Mammal {
    constructor(name) {    super(name);    }
    saySomething() { console.log('Woof, my name is ' +this.name);  }
}

//create an object of type Dog
const lassie = new Dog('Lassie');
lassie.saySomething();
lassie.doSomething();
```

```
//class syntax
class MyClass {
constructor() { ... }
  method1() { ... }
  method2() { ... }
}
//example
class User {
  constructor(name) {   this.name = name;  }
  sayHi() { alert(this.name);  }
}
// Usage:
let user = new User("John");
user.sayHi();
```

- What class User {…} does:
  - Create a constructor function, User, from constructor method
  - Stores class methods, e.g, sayHi, in User.prototype.

Main Point
Inheritance

JavaScript supports prototype inheritance so that objects can inherit common functionality from a single ‘prototype’ object.

Science of Consciousness:

Pure consciousness is a level of awareness that is a common experience shared by everyone.

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/