Javascript Notes:

Classes

As a function:

Create a function, which acts as an object constructor, using ‘this’ within the function, which acts as a class. ‘this’ keyword ensures uniqueness of functions/properties within the object/class. Keeps them out of the ‘global’ namespace.



Functions can typically be added ‘after-the-fact’ via prototyping (adding functionality to the ‘prototype’ of the object/class), which makes the function itself sort of ‘static’ to the class – being shared across all instances of the object, and, ergo, more inexpensive.

function Apple (type) {

this.type = type;

this.color = "red";

}

Apple.prototype.getInfo = function() {

return this.color + ' ' + this.type + ' apple';

};

Singletons:

One offs.

var apple = new function() {

this.type = "macintosh";

this.color = "red";

this.getInfo = function () {

return this.color + ' ' + this.type + ' apple';

};

}

Prototype:

Functions, like almost all objects in JavaScript, contain a "prototype" object. When we call a JavaScript constructor to create an object, all the properties of the constructor's prototype are then made available to the new object. In this fashion, multiple Car objects can be created which access the same prototype.

Namespacing

Immediately-Invoked Function Expressions (aka Anonymous Self-Executing Functions):

(function(parameters)

Immediately loaded/executed into memory by surrounding parentheses. This one is anonymous, but, could be named.

{

//variables

//functions

})(parameters);

Empty object declaration if it doesn’t already exist.

var myNamespace = myNamespace || {};

myNamespace.Car = function(){};

Function added to namespace.

Full ‘module’ under namespace:

(function(namespaceParameter, otherParameters)

{

//variables

Namespace scoped in IIFE and passed

//functions

})(window.Namespace = window.Namespace || {});

Subclassing (from a base class):

Module Pattern

Object Literals:

In object literal notation, an object is described as a set of comma-separated name/value pairs enclosed in curly braces ({}). Names inside the object may be either strings or identifiers that are followed by a colon. There should be no comma used after the final name/value pair in the object as this may result in errors.

var Calculator = function () {

// private variables here

// private functions here

return {

// public members

// in the form of an object literal

function1: function () { },

function2: function () { }

};

};

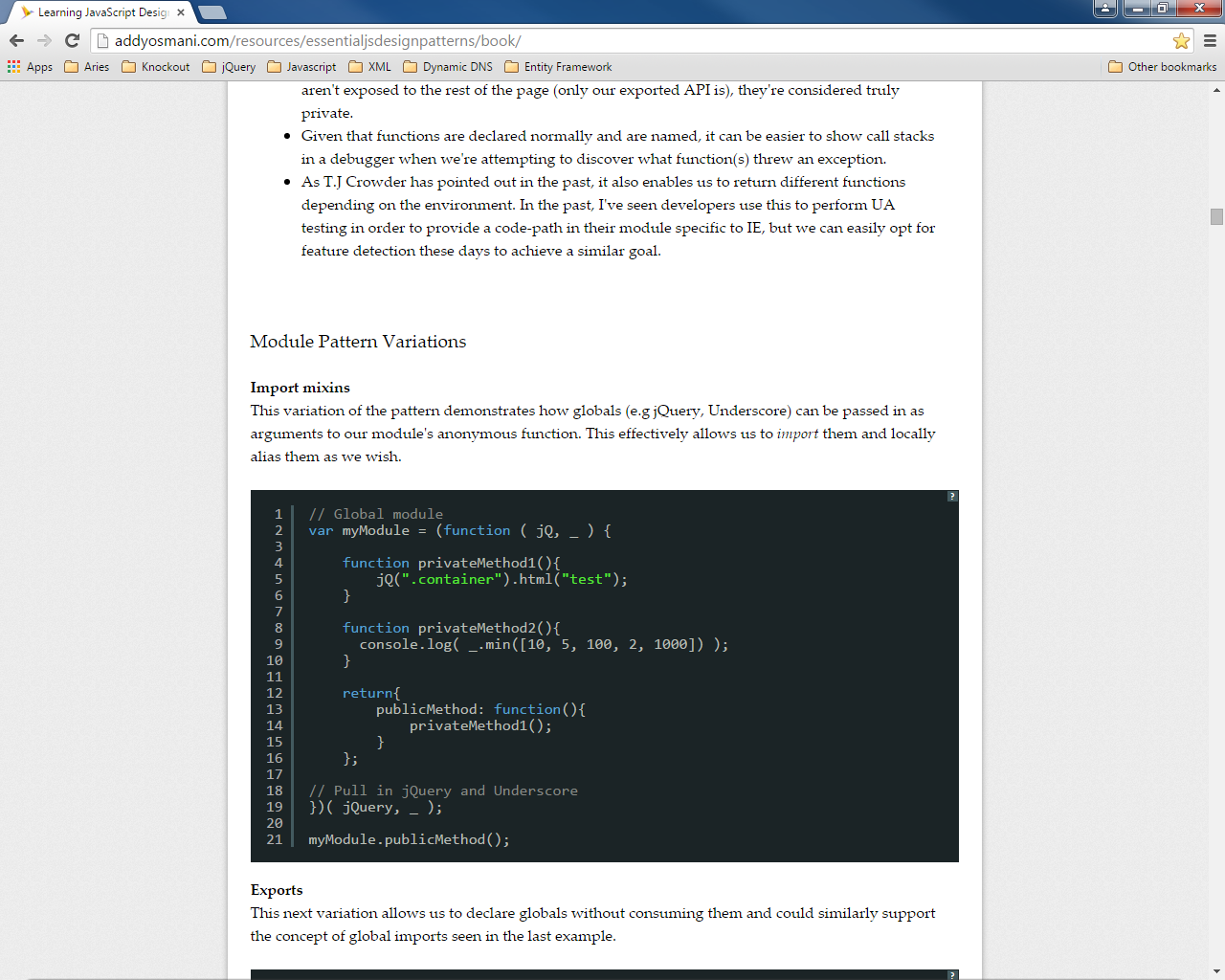
Within the Module pattern, variables or methods declared are only available inside the module itself thanks to closure. Variables or methods defined within the returning object however are available to everyone.



Here, basket[], doSomethingPrivate and doSomethingElsePrivate are local to the module.

They are accessible by addItem, getItemCount, doSomething, getTotal etc…

Importing Global Namespaces to Module Locally:

 Importing jQuery and Underscore (\_)

Revealing

var calculator = function (parameter) {

// private variables here

// private functions here

// private functions to point to

var internalFunction1 = function()

{

},

internalFunction2 = function()

{

};

Return object ‘reveals’ internal functionality through ObjectLiteral.

return {

// public members

// in the form of an object literal

function1: internalFunction1,

function2: internalFunction2

Self calling

};

}('parameter');

Prototype Pattern

Not only is the pattern an easy way to implement inheritance, but it can also come with a performance boost as well: when defining a function in an object, they're all created by reference (so all child objects point to the same function) instead of creating their own individual copies.

var Calculator = function(eq)

{

// state should go here

// this is 'NOT' shared amongst all

// just the instance

this.eqCtl = document.getElementById(eq);

};

‘this’ relates to the individual instance

Calculator.prototype = function () {

var add = function (x, y) {

var val = x + y;

this.eqCtl.innerHTML = val;

},

‘this’ relates to the individual instance instantiated in the constructor

subtract = function (x, y) {

var val = x - y;

this.eqCtl.innerHTML = val;

};

function YesNo(x, y) {

this.eqCtl.innerHTML = x;

}

};

Revealing

Calculator.prototype = function () {

var add = function (x, y) {

var val = x + y;

this.eqCtl.innerHTML = val;

},

subtract = function (x, y) {

var val = x - y;

this.eqCtl.innerHTML = val;

};

function YesNo(x, y) {

this.eqCtl.innerHTML = x;

}

Return object ‘reveals’ internal functionality.

return {

addFunction: add,

subtractFunction: subtract,

yesnoFunction: YesNo

};

Self calling

}();

Thissyness

The this keyword represents the latest caller in a call chain. It will, not necessarily be the original caller.

In the case of using the Prototype pattern, this will represent the object that is returned after calling the constructor:

Constructor creates instance object which is the ‘this’

var Calculator = function (eq) {

this.eqCtl = document.getElementById(eq);

};

Calculator.prototype = function ()

{

…

When a public facing function is called from the instance object, the internal function can use this to refer to the calling instance:

var calc = new Calculator('myElement');

calc.addFunction(1, 1);

Calculator.prototype = function () {

The instance calls addFunction which directly points to the internal add function. The this keyword can be used in the add function and it will represent the calling instance.

var add = function (x, y)

{

var val = 0;

if (itIsAnElement.call(this))

{

val = x + y;

}

else

{

val = 'No Value';

}

setVal.call(this, val);

},

…

return {

addFunction: add,

subtractFunction: subtract,

yesnoFunction: yesNo

};

}();

However, with subsequent internal calls, the this keyword will change to represent the instance of the caller:

var calc = new Calculator('myElement');

calc.addFunction(1, 1);

Calculator.prototype = function () {

var add = function (x, y)

{

var val = 0;

if (itIsAnElement())

{

val = x + y;

}

The calls to itIsAnElement and setVal will fail because, they are no longer being called by the instance – they are being called by a function within the prototype, which is defined as a shared/singleton. So the this keyword within both of these functions represents the prototype NOT the original calling instance.

else

{

val = 'No Value';

}

setVal(val);

},

…

function itIsAnElement()

{

return(this.elementControl.tagName ? true : false);

}

function setVal(val)

{

this.elementControl.innerHTML = val;

}

return {

addFunction: add,

subtractFunction: subtract,

yesnoFunction: yesNo

};

}();

To be able to reference the original calling instance, instead of having to pass the this keyword around to all of the necessary internal functions and those functions needing to have a parameter to hold the keyword, make internal function calls with the native .call(this,arguments) syntax. This allows the passing of the this keyword.

var calc = new Calculator('myElement');

calc.addFunction(1, 1);

Calculator.prototype = function () {

var add = function (x, y)

{

var val = 0;

if (itIsAnElement.call(this))

{

val = x + y;

}

else

By using the .call syntax, you can pass a reference to the original instance via the this keyword.

{

val = 'No Value';

}

setVal.call(this, val);

},

…

function itIsAnElement()

{

return(this.elementControl.tagName ? true : false);

}

function setVal(val)

{

this.elementControl.innerHTML = val;

}

return {

addFunction: add,

So here, the this keyword now represents the original calling instance and, therefore, can reference the elementControl variable.

subtractFunction: subtract,

yesnoFunction: yesNo

};

}();