JavaScript

Landscape, Margin all 0.1 except bottom 2.5, font, 14

[JavaScript online interpreter: 4](#_Toc315769364)

[ECMAScript 5 compatibility 4](#_Toc315769365)

[“use strict”: 4](#_Toc315769366)

[Property Attributes 4](#_Toc315769367)

[Object Attributes 4](#_Toc315769368)

[Constructors (Constructor Functions) 4](#_Toc315769369)

[var has 2 scopes ONLY: 4](#_Toc315769370)

[NaN – don’t use == or ===, just use isNaN: 4](#_Toc315769371)

[Strings 4](#_Toc315769372)

[Date (instance): 4](#_Toc315769373)

[Math (static): 4](#_Toc315769374)

[Reflection 4](#_Toc315769375)

[Prototype 4](#_Toc315769376)

[Simulate C# class’s instance properties/methods, class properties/methods 5](#_Toc315769377)

[Constructor object 5](#_Toc315769378)

[Prototype object 5](#_Toc315769379)

[Instance object 5](#_Toc315769380)

[Eample 5](#_Toc315769381)

[Defining Object: 7](#_Toc315769382)

[Object keys count 7](#_Toc315769383)

[Accessing Properties/functions: 7](#_Toc315769384)

[Clear Value 7](#_Toc315769385)

[Delete property/method to remove key: 7](#_Toc315769386)

[Global Object: 7](#_Toc315769387)

[Expando Object: 7](#_Toc315769388)

[Function Object: 7](#_Toc315769389)

[Augmenting Types 8](#_Toc315769390)

[Scope 8](#_Toc315769391)

[Module 8](#_Toc315769392)

[Array 8](#_Toc315769393)

[prototype: 8](#_Toc315769394)

[throw exception: 9](#_Toc315769395)

[try catch: 9](#_Toc315769396)

[If: 9](#_Toc315769397)

[switch: 9](#_Toc315769398)

[Loop: 9](#_Toc315769399)

[object.hasOwnProperty(variable) 9](#_Toc315769400)

[JSLint 9](#_Toc315769401)

[Pitfall 9](#_Toc315769402)

[document.write(“Hello World”): 9](#_Toc315769403)

[external javascript: 9](#_Toc315769404)

[Frame: 9](#_Toc315769405)

[Form: 9](#_Toc315769406)

[Browser: 10](#_Toc315769407)

[By Value vs By Reference: 10](#_Toc315769408)

[CSS 10](#_Toc315769409)

[Pattern 10](#_Toc315769410)

[Global 10](#_Toc315769411)

[for loop – good for Array 11](#_Toc315769412)

[for in loop (enumeration) – good for non-Array Object 11](#_Toc315769413)

[Not Augmenting Built-In (Object, Array, Function) Prototypes 11](#_Toc315769414)

[Use Strict Equality === or !== to Avoid Implicit Typecast 11](#_Toc315769415)

[Eval(someString) is Evil 11](#_Toc315769416)

[ParseInt(“someString”, base) MUST USE base 11](#_Toc315769417)

[Coding Convention 11](#_Toc315769418)

[Naming Convention 12](#_Toc315769419)

[Literals & Constructor 12](#_Toc315769420)

[Object Literal 12](#_Toc315769421)

[Custom Constructor 12](#_Toc315769422)

[Array 13](#_Toc315769423)

[JSON 13](#_Toc315769424)

[Primitive Types vs Primitive Wrappers 13](#_Toc315769425)

[Error Objects 13](#_Toc315769426)

[Function 14](#_Toc315769427)

[Hoisting 14](#_Toc315769428)

[1-time Immediate Function Initialization 14](#_Toc315769429)

[1-time Immediate Object Initialization 14](#_Toc315769430)

[Callback using object method 15](#_Toc315769431)

[Memoization: cache return result 15](#_Toc315769432)

[Curry 15](#_Toc315769433)

[Object 15](#_Toc315769434)

[Namespace 15](#_Toc315769435)

[Add References / Dependencies 16](#_Toc315769436)

[EXCELLENT – Expose Private to Public – w/o new 16](#_Toc315769437)

[EXCELLENT – Module – w/o new 17](#_Toc315769438)

[Sandbox 18](#_Toc315769439)

[EXCELLENT - Static/Base class Instance 18](#_Toc315769440)

[Constant 18](#_Toc315769441)

[Chaining, Fluent Interface 19](#_Toc315769442)

[method() Method 19](#_Toc315769443)

[Examples 19](#_Toc315769444)

[Instance members(both derived & base), Static members 19](#_Toc315769445)

[DOM Access 20](#_Toc315769446)

[Event 20](#_Toc315769447)

[Long Running Script 21](#_Toc315769448)

[Loading Strategies 21](#_Toc315769449)

[XHR 21](#_Toc315769450)

JavaScript online interpreter:

<http://math.chapman.edu/~jipsen/js/>

ECMAScript 5 compatibility

* <http://kangax.github.com/es5-compat-table/>

“use strict”:

to enforce a better syntax rules

Property Attributes

* ??? NEED DETAILS. writable/enumerable/configurable

Object Attributes

* ??? NEED DETAILS. prototype/class/extensible

Constructors (Constructor Functions)

* **Built-in**: Object(), Array(), Date(), RegExp(“”);

var has 2 scopes ONLY:

* **Global** and **Local**(within a **function**, **properties** of an object is **NOT** considered **var**!

NaN – don’t use == or ===, just use isNaN:

* NaN means undefined. NaN not equal to ANY value including itself, so to test a number use isNaN(number).
* Alternative:

var isNumber = **function** **isNumber**(value) { return typeof value === 'number' &&

**isFinite**(value);

}

Falsy & Truthy

* Falsy: false, null, undefined, empty string ‘’, number 0, NaN
* Truthy: All other values are truthy including: true, all object & the **string ‘false’!!!**.

Primitive vs Object Types

* Primitive Types: number, string, Boolean, null, undefined
* Object Types : All else. (inc. Number(), String(), Boolean())

Object object

* **Methods**: hasOwnProperty(propName);

Number object

* **Methods**: toExponential/toFixed/toPrecision/toString/

RegExp

* **Methods**: exec/test

String object

* **Methods**: charAt/charCodeAt/concat/indexOf/lastIndexOf/match/replace/search/splice/split/substring/toLowerCase/toUpperCase/fromCharCode
* A string literal can be wrapped in **single** **or** **double** **quotes**!
* **Concatenation**: **+**
* var text = “testing”; text.**length**; .**bold** .**fontsize** .**toLowerCase** .**replace** .**search** .**substring**

Date (instance):

* var mydate = **new Date()**; mydate.**getDate** .**getMonth** .**getYear**

Math (static):

* Math.**max**(x,y); Math.**round**(x); Math.**random**(x);

Prototype

* It is an object. The “derived” object will **inherit the properties** of the prototype object.
* Prototype Chain
  + Used **ONLY** in **Retrieval**.
  + Date/Array/Function.prototype = Object.prototype; same as = {};
    - Function is a bit different: Function.prototype = { constructor.this };
  + Object.prototype is itself an object BUT it is the last object in the prototype chain.
  + Object.prototype.prototype = null.
    - // **Object.prototype** has **NO prototype**!
    - ALL built-in, user-defined object **HAS prototype**.
* To find out if p is the prototype of o, use **p.isPrototypeOf(o)**.
* To find out if o is an instance of MyObject(), -> **o instanceOf MyObject** // var o = new MyObject();
  + instanceOf doesn’t check whether o was initialized by the MyObject ctor. It checks whether it inherits from MyObject.prototype (i.e. **o.prototype === MyObject.prototype**)
* An incomplete **example** of creating a new object that inherits from a prototype, support **BOTH ECMAScript 3 & 5**

// inherit() returns a newly created object that inherits properties from the

// prototype object p. It uses the ECMAScript 5 function Object.create() if

// it is defined, and otherwise falls back to an older technique.

function inherit(p) {

if (p == null) throw TypeError(); // p must be a non-null object

if (Object.create) // If **ECMAScript5** **Object**.**create**() is defined...

return **Object.create(p)**; // then just use it.

var t = typeof p; // Otherwise do some more type checking

if (t !== "object" && t !== "function") throw TypeError();

function **f() {};** // Define a dummy constructor function.

**f.prototype = p;**  // Set its prototype property to p **BEFORE f** is **instantiated**.

return **new f();** // Use **new** f() to create an "heir" of p.

}

* **Property Inheritance occurs ONLY for query NOT for setting!!! You can still do a Set, but the Set affected the Derived object, NOT the inherited object. Allows us to selectively override inherited properties:**

var **unitcircle** = { r:1 }; // An object to inherit from

var c = inherit(**unitcircle**); // c inherits the property r

c.x = 1; c.y = 1; // c defines two properties of its own

c.r; // => 1 the unitcirle value is inherited

c.r = 2; // c overrides its inherited property, i.e. the **derived** **object** **c**.r is modified but **NOT** the **prototype**.r, the **inherited r** is **HIDDEN.** Note: **w/o setting c.r, c.r will return unitcircle.r**.

unitcircle.r; // => 1: the prototype object is not affected

c.r; // => 2: the original object c.r is modified.

// Notice that BOTH get & set of r proprerty of the c object is on the c object itself, NOT the inherited unitcircle object!

* **constructor function** – replaces the need for the above impl.
  + Use Upper Camel case
  + var MyObject = function() { … }
  + **MyObject**.**prototype** = { **someSharedMethod**: function() { **this**.someShared/Non-SharedProperty: … } , someSharedProperty: someValue }
  + Whenever you use new to create an object:
  + var myObject = **new** MyObject();
  + js engine
    - auto create an **empty** **object**: **var this = {};**
      * if support **ECMAScript5**, then **Object**.**create**(prototypeObject, optionalProperties) will be used.
    - **Preserve Inheritance**: auto **copy** **ctor** **function MyObject() ‘s prototype** : **this.prototype = MyObject.prototype;**
      * this gives you the **inherited instance methods** **from** the **function** **object**
    - **auto copy** **ctor** **function MyObject()** to **this**.**prototype**.**constructor**
    - **call** the **ctor** **function** for final **initialization**
  + p.s. the above MyObject.prototype is **NOT** **setting** the **constructor** **property**, to remedy this problem:
  + Choice 1:

MyObject.prototype = {

**constructor: MyObject, // Explicitly set the constructor back-reference**

someSharedMethod: function() { **this**.someShared/Non-SharedProperty: … },

someShareProperty: someValue

};

* + Choice 2:

// **Extend** the predefined MyObject.prototype object so **we don't overwrite the original MyObject.prototype!**

// the automatically created Range.prototype.constructor property.

MyObject.**prototype.someSharedMethod** = function() { **this**.someShared/Non-SharedProperty: … };

MyObject.**prototype.someShareProperty** = someValue;

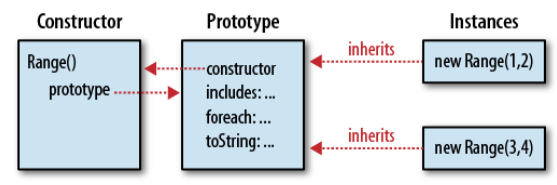
* **constructor** **property -** 
  + var F = function() {}; // This is a function object.

**F.prototype.constructor === F** // => true

* + It also means an object ‘o’ typically inherits(when using **new**) a constructor property defined in the prototype(i.e. **F.prototype.constructor**) which points to the constructor function(**F**) and becomes **o.constructor**:
    - var o = **new** F(); // Create an object o of class F
    - **o.constructor === F** // => true: the constructor property
  + The following illustrates this relationship between the constructor function, its prototype

object, the back reference from the prototype to the constructor, and the instances

created with the constructor.



* myObj **instanceOf** MyObject // return **true** if myObj **inherits** from **MyObject**.**prototype**. It does **NOT** actually checking whether it is init by the **MyObject** **ctor**!

Simulate C# class’s instance properties/methods, class properties/methods

Constructor object

* ctor function itself is an object. Properties you add to this ctor object serve as **class properties/methods**.

Prototype object

* properties & methods of this object are **inherited & shared** by all instances of the class & behave like **instance properties/methods**.

Instance object

* Each instance of the class is an object. Properties defined on an instance behave like **instance properties/methods** BUT are **NOT shared** by any other instance.
  + e.g. **x, y, z, foo are all Non-Shared instance properties/method**

function MyObject(x, y) { this.**x** = x; this.**y**=y; }

var o = new MyObject(1,2);

o.**z** = 3;

o.**foo** = function() { … }

Eample

* Implementation

/\*

\* This constructor function defines the **instance fields r and i** on every

\* instance it creates. These fields hold the real and imaginary parts of

\* the complex number: they are the state of the object.

\*/

**function** Complex(real, imaginary) {

if (isNaN(real) || isNaN(imaginary)) // Ensure that both args are numbers.

throw new TypeError(); // Throw an error if they are not.

**this.r** = real; // The real part of the complex number.

**this.i** = imaginary; // The imaginary part of the number.

}

/\*

\* The **instance methods** of a class are defined as function-valued properties

\* of the **prototype** object. The methods defined here are inherited by all

\* instances and provide the shared behavior of the class. Note that JavaScript

\* instance methods must use the this keyword to access the instance fields.

\*/

// Add a complex number to this one and return the sum in a new object.

Complex.**prototype**.add = function(that) {

return new Complex(this.r + that.r, this.i + that.i);

};

// Multiply this complex number by another and return the product.

Complex.**prototype**.mul = function(that) {

return new Complex(this.r \* that.r - this.i \* that.i,

this.r \* that.i + this.i \* that.r);

};

// Return the real magnitude of a complex number. This is defined

// as its distance from the origin (0,0) of the complex plane.

Complex.**prototype**.mag = function() {

return Math.sqrt(this.r\*this.r + this.i\*this.i);

};

// Return a complex number that is the negative of this one.

Complex.**prototype**.neg = function() { return new Complex(-this.r, -this.i); };

// Convert a Complex object to a string in a useful way.

Complex.**prototype**.toString = function() {

return "{" + this.r + "," + this.i + "}";

};

// Test whether this Complex object has the same value as another.

Complex.**prototype**.equals = function(that) {

return that != null && // must be defined and non-null

that.constructor === Complex && // and an instance of Complex

this.r === that.r && this.i === that.i; // and have the same values.

};

/\*

\* **Class** **fields** (such as constants) and **class** **methods** are defined as

\* **properties of the constructor**. Note that class methods do not

\* generally use the this keyword: they operate only on their arguments.

\*/

// Here are some class fields that hold useful predefined complex numbers.

// Their names are uppercase to indicate that they are constants.

// (In ECMAScript 5, we could actually make these properties read-only.)

**Complex.ZERO** = new Complex(0,0);

**Complex.ONE** = new Complex(1,0);

**Complex.I** = new Complex(0,1);

// This **class method** parses a string in the format returned by the toString

// instance method and returns a Complex object or throws a TypeError.

**Complex.parse** = **function**(s) {

try { // Assume that the parsing will succeed

var m = Complex.\_format.exec(s); // Regular expression magic

return new Complex(parseFloat(m[1]), parseFloat(m[2]));

} catch (x) { // And throw an exception if it fails

throw new TypeError("Can't parse '" + s + "' as a complex number.");

}

};

// A **"private" class field** used in Complex.parse() above.

// The underscore in its name indicates that it is intended for internal

// use and should not be considered part of the public API of this class.

**Complex.\_format** = /^\{([^,]+),([^}]+)\}$/;

* Usage:

var c = new Complex(2,3); // Create a new object with the constructor

var d = new Complex(c.i,c.r); // Use instance properties of c

c.add(d).toString(); // => "{5,5}": use instance methods

// A more complex expression that uses a class method and field

Complex.parse(c.toString()). // Convert c to a string and back again,

add(c.neg()). // add its negative to it,

equals(Complex.ZERO) // and it will always equal zero

Defining Object:

* Object inherit from Object.prototype
* <http://mckoss.com/jscript/object.htm>
* <http://bonsaiden.github.com/JavaScript-Garden/#object>
* **Object** **Literals (**A pair of curly braces with comma-delimited ‘,’ named properties**)**:
  + Create an **Empty** **Object** that inherit from Object.prototype: var myEmptyObject = **{};**
  + Create an **object** inline initialization: var myObject = {x**:** 1**,** y**:**2, add**:** function() { … }};
* **Object.create()**
  + ECMAScript5
  + **Object**.**create**(prototypeObject, optionalProperties)
* **DO NOT USE new** to create an object, but if you insist this is how:
  + **function MyObj**(x, y) { … } OR var **MyObj** = **function**(x, y) { … }
  + **var myObj = new MyObj(1, 2);**

Object keys count

* **keys(**theObject**).length**

Accessing Properties/functions:

* foo**.name**; OR foo**[“name”]**;
* returns **undefined** for non-existent member
* **for**( prop **in** obj) **obj[prop]** … // which will **include** **proptotype** **chain**
* **for**(i=0; i < someNum; i++) obj[i] … //will **NOT** **include** prototype chain. someNum = key(obj).length
* **Provide Default** value **||** instead of undefined: e.g. var f = foo[“name”] || “none”;
* **TypeError Exception:** if try to retrieve value from an undefined.
* **To avoid TypeError Exception**, make sure object exists before reference its properties:
  + e.g. if(**book && book.subtitle**) len = book.subtitle.length;

Clear Value

* myObject.x = undefined/null; will only **“clear”** the **value not remove the key**!

Delete property/method to remove key:

* **delete** myObject.x;

Global Object:

* **Best** **Practice**: Minimize to a Single Global Variable
* var myGlobalObject = “something”; // if **var** is **NOT** **inside** any **function**, it is **Global**
* **this** refers to **Global** **unless** use inside a **method**
* Add a property to the Enviroment’s global object e.g. In browser, you use **windows.foo** = “something”; **foo** is **Global**
* **If you forget** to use the **var myGlobalObject**, **still** a **global** object: myGlobalObject = “something”; // called implied Global
* Usage: var x = myGlobalObject YES; var y = **this**.myGlobalObject; var z = **this**[“myGlobalObject”];

Expando Object:

* Anything **non-primitive** types **IS** an **object** and **primitive** type can **become** an **object** with a **wrapper**.
* e.g. Turn **primitive** **type** **to** **Object** using **Wrapper**: var myObject = **new** Number(123); // 123 is primitive number type
* And you can **expand** the object by adding properties, arrays, functions.

Function Object:

* **Methods**: **apply**
* **function** Foo(a, b, c, …) { return … }
* Function Foo is an object -> Foo.prototype -> Object.prototype
* Function object has 2 **hidden** properties: context & code
* **Default** Foo.**prototype** property: its value is an **object** with a .**constructor** **property** whose value is the current **function**.
* **Closure** in **Nested** functions: **inner** function can access to the **params** and **variables** of the **outer** functions it is nested within **EXCEPT** “**this**“ & “**arguments**”, “**this**” will refer to global **unless** it is a **method** instead of a function, arguments will refer to the current function.
* # of arguments vs # of params:
  + Too many args: ignored
  + Too few args: the missing args’s values are undefined.
* **Method**: function stored as a **property** of an **object** is called method. E.g.
  + function Foo() { this.boo = function() { … } };
  + var o = new Foo(); o.boo(); // **boo is a method**
* **this – how to use**:
  + **Method** Invoke Pattern: a **method** use **“this”** to access the containing object.
  + **Function** Invoke Pattern: not a method, so “***this***” **ALWAYS** bound to the **Global** object, even **nested** (used **inside** a **method**) (**BAD** Design)! e.g. **inside** a **method** you have a function: **var** **foo** = **function**() { ***this***.someProp … } // ***this*** bounds to **Global**.
    - Work around: **inside the method,** **var that = this**; **var foo** = function(){ **that**.someProp … } //avoid ‘this’ & use closure
  + **Constructor** Invoke Pattern (**BAD** Design)
    - var **MyCtor** = function () { … }; var myBoo = new **MyCtor**(); // note: capitalize
    - **BAD** **things** will happen if use **ctor** **function** **w/o using new**. e.g. use ‘this’ inside the function have different meanings
  + **Apply** **Invoke** Pattern
    - someMethod.**apply**(theValueThatShoudBeBoundTo**this**, **arrayOfParams**);
    - Above **invoke someMethod**() and **replace ‘this’** with ValueThatShoudBeBoundTo**this & replace arguments** w/ **arrayOfParams**
* **Arguments**:
  + Always bounds to the immediate function.
  + Access to **ALL** of the **arguments** that were **not** **assigned** to **parameters**.
  + for(i = 0; i < **arguments.length**; i++) { … **arguments[i]**; }
* **return**: a function **ALWAYS** **return** a value
  + normal invoke: if return **NOT specified**, then **undefined** is returned.
  + invoked using **new**: if return **NOT specified or NOT an Object**, then **this (i.e. the new Object)** is returned.

Augmenting Types via prototype

* Extend Existing Object via prototype inheritance
* Use the following to **augment ANY function object** including **Number(), String()**:
  + Function.prototype.**method** = function (name, func) {

If(!this.prototype[name]) { **this**.**prototype[name]** = **func**; return this; } }; // inject into **this’s** **prototype**

* Usage e.g. :
  + **Number**.**method**(‘integer’, function() { return Math[this < 0 ? ‘ceiling’ : ‘floor’](this); });
  + **String**.method(‘trim’, function() { return this.replace(/^\s+|\s+$/g, '');

Reflection on Type property that is data. NOT function

* Use **typeof** myObj.dataProperty & myObj.**hasOwnProperty**(propName)

Scope

* **Function** Scope: **YES**
* **Block** Scope: **NO**. i.e. variable defined inside a block { var boo = …; } IS NOT LOCAL to the Block! **BAD**!

Closure

Module

* **Elminate the use of global variables, avoid any need to use ‘this’ or ‘that’. Promotes encapsulation via private variables.**
* The general pattern of a module is a function that defines private variables and functions; creates privileged functions which, through closure, will have access to the private variables and functions; and that returns the privileged functions or stores them in an accessible place.

Cascade / Chaining / Fluent methods

* **Some methods do not require a return value**, we can have those methods **return ‘this’ instead of undefined**, we can enable cascade

Array Object

* **Methods:** concat/join/pop/push/reverse/shift/slice/sort/splice/unshift
* Array is just an object. Array subscripts are auto turned into strings that are used to make properties of the Array Object. The 1st value will get the property name ‘0’, the 2nd one becomes property name ‘1’ …
* Inherits from **Array**.**prototype** which comes with a few **useful methods & 1 property** for array e.g. length, push()/unshift()/…
* Allow **ANY** **mixture** of **types**!
* 0-based
* **Array Literal**: var **empty** = []; var numbers = [1, 2, 3, 4];
* numbers.**length** // 4.
  + Object Type doesn’t have the length property, only the Array Type
  + Note: length equals to the **largest integer property name + 1**. Not necessary the # of properties in the array

var myArray = [];

myArray.**length** // **0**

**myArray[1000000]** = true;

myArray.**length** // **1000001**

* + Also it **won’t** **count** **non-integer** properties
  + Setting the length smaller than the actual # of properties can ‘delete’ all the upper range properties.
* **Access**: numbers[0];
* .**push**(aValue); // add to end, i.e. **append**
* .**pop**(); // remove the last item
* .**unshift**(aValue); // add to **1st item**, i.e. **prepend**
* .**shift**(); // remove the **1st item**
* **delete** myArray[2]: will **leave** **an** undefined **hole** in the array
* .**splice**(**index**, **numOfElementToDelete**); // delete element from array **w/o leaving a hole**. **Cons**: **slow** bcos reinsert elements.
* **Testing** if an object **is** an **array**:
  + **Simple**:

var is\_array = function (value) {

return value &&

typeof value === 'object' &&

value.constructor === Array;

};

* + **Complete:**

var is\_array = function (value) {

return value &&

typeof value === 'object' &&

typeof value.length === 'number' &&

typeof value.splice === 'function' &&

!(value.propertyIsEnumerable('length'));

};

* **Enumeration**
  + **for in**: **indetermine** **order** of properties
  + **for**: give u the **proper** **order**

throw exception:

* throw should be given an exception **Object** containing a **name** and **message** **property**.
* e.g. **throw** **{** **name**: “**TypeError**”, **message**: “some message here” **}**; // throw an Object Literal

try catch:

* **try** { … } **catch**(e) { if(e.**name == “TypeError”)** e.**message**; … }

if:

* if (x==0) { y = 1 }; if (…){ … } else{ … } // similar to C++
* y= (x==0) ? 1 : 2;
* **Avoid using ==, using === instead.**

switch:

* **switch** (expression) { **case** 2: … **break**; case 5: … break; **default**: … } // similar to C++

Loop:

* **for**(var i=0; i < Count; i++) { … }
* **for**(prop **in** obj) { obj[prop]; // **returns the value of the property prop in the object obj** Different from For Each…Next. This will include the prototype. If want to skip prototype use obj.hasOwnProperty(prop)
* **while**(x == 0){ … } / **do**{ … }**while**(x==0)
* **break**; **continue**; to control the loop

object.hasOwnProperty(propName)

* Test whether the property name is truly a member of the object or was found on the prototype chain.
* for(myProp in obj) { if(obj.**hasOwnProperty**(myProp)) … }

Bad Part

Global Variable – 3 ways

* 1. Use var outside ANY function
* 2. Use the environment global object, e.g. window object in browser
* 3. Implied global: foo=value

Scope

JSLint

* **Global Variables Declaration**: /\*global myGlobalVar1, myGlobalVar2, … \*/

Awful JS

* **Auto insert semi-colon** causing problem. Therefore always explicit add semi-colon.
* **NaN**: **typeof** cannot test NaN bcos it returns ‘number’ (**BAD**), **NaN === NaN** // returns false (**BAD**). Use **isNaN(myValue)** to test NaN.
* **Testing objectness & null: typeof cannot differentiate null and object.** i.e. typeof null returns ‘object’. To truly test an object myValue use: **if(myValue && typeof myValue == ‘object’)** // **then it is an object** or an **array**, bcos if **myValue** is **null**, **myValue is falsy**!
* **Test objectness & Array**: **typeof cannot differentiate array and object**. A better test:
  + **if(myValue && typeof myValue == ‘object’ && myValue.constructor == Array)**
* parseInt stops when it sees a non digit, so **parseInt(“15”) & parseInt(“15 cm”)** gives the **same** **result**

Bad JS

* == / != ONLY works when comparing same data type. Use === / !== instead
* **Never use with**. Confusion & Performance Issues.
* Don’t use eval: performance issue, security issue, hard to read
* Don’t use Block-less statement: easy to confuse ppl
* **Don’t use** Typed Wrappers e.g. new Boolean(false), new Number, new String
* **Never use new Object or new Array**. **Use** **Literal** **{}** and **[]** instead. // disaster if you forgot to use new to create and object via function
* **Never use void.**

document.write(“Hello World”):

output some text

external javascript:

external file with .js extension. cannot contain <script> tag. Import this script using <script src=”xxx.js”> … <script>

Frame:

breakout frame: if(window.top != window.self) window.top.location = “theHTML.htm”

update 2 frames: parent.upperframe.location.href=”a.htm” // or parent.lowerframe

Form:

Validation: x=document.myFormName; text=x.myEditBox.value; <form name=”myFormName” action=”xyz.htm” onsubmit=”return thevalidatefuncntion”>

Get Value: document.formname.fieldname.value

Set focus: document.forms[0].fieldname.focus(); // .forms[0] can also be formname

Text selected: document.formname.fieldname.select();

Dropdown selected Text: text = document.formname.dropdownname.options[document.formname.dropdownname.selectedIndex].text <form> <select name=”dropdownname” onchange=”myfunc”>

Browser:

Detection: navigation.name, .appName, .appVersion, .platform, document.referrer

Client’s monitor: window.screen.width, .height, .availWidth, .avail.Height, .colorDepth

Popup Box: window.alert, whichButton=confirm(), textTyped=prompt()

New Window: window.open(url)

Redirect: window.location = “new url”

Reload: window.location.refresh

Status Bar: window.status = “text for status bar”

Print: window.print();

By Value vs By Reference:

Numbers, Booleans: Copy/Pass/Compare by Value

Objects, Arrays, Functions: Copy/Pass/Compare by Reference

Strings: Copy/Pass by Reference, Compare by Reference(both are objects), Compare by Value(if one or both is a literal or string value)

CSS

Cascading Order: Inline -> Internal Style Sheet(inside <head>) -> External Style Sheet -> Browser

Syntax: selector {property : value} // “” for multiple words, ; to separate multiple attribute

e.g.: body {color:black}, p {font-family:”sans serif”}, p {text-align:center; color:red}

Grouping: Can group selectors

e.g.: h1, h2, h3 { color:red }

class attribute:

e.g. CSS: p.right{text-align:right} p.left{text-align:left}

e.g. HTML: <p class=”right”>xyz</p> <p class=”left”>xyz</p>

e.g. CSS: .right {text-align:center}

e.g. HTML: <h1 class=”right”>xyz</h> <p class=”right”>xyz</p>

id attribute:

e.g. CSS: #**intro** { … } // match ANY elements with the <xyz **id**=”**intro**”>

e.g. CSS: **p**#**intro** { … } // match ONLY **p** element <**pid**=”**intro**”>

comment: /\* … \*/

External style sheet:

HTML: <head> <**link** rel=”stylesheet” type=”text/css” href=”xyz.css”/></head>

CSS: NO HTML tag in this file

Internal style sheet: <head> <**style** type=”text/css”> … </style> … </head>

Inline styles: <xyz **style**=”text-align:right” >

<**div**>: define division. Browser will put line-break before and after the tag

<**span**>: define a section. Browser will **NOT** put line-break before and after the tag

Pattern

Global

* **Global variable:**

var a = 0; // a is global when outside a function

b = 0; // b is ALSO global, var is optional if you want global

* **Single line Multi-variable initialization**

function foo() {

var a = b = 0; // a is local BUT **b** is **global** }

* **Single var Pattern**
* Top of the function, single var, seprate variable with comma, optional initialization with proper datatype so ppl know what type the variable is intended to use.
* function foo() {

var a = 1,

b = 1,

myObject = { … ),

}

* **Avoid Hoisting Bug:**

myname = "global"; // global variable

function func() {

**alert(myname); // "undefined"**, hoisting makes the local **var** “**DECLARTION**” goto TOP of the function, **NOT** the “**Initilization**”

var myname = "local123"; // the **initialization** value **local123 WON’T** get **hosited**

alert(myname); // "local123"

}

func();

for loop – good for Array

* function looper() {

var i = 0, // single var pattern

max,

myarray = [];

// cache the length, so myarray.length will called only ONCE

**for** (i = 0, max = myarray.**length**; i < max; i++) {

// do something with myarray[i]

}

}

for in loop (enumeration) – good for non-Array Object

* Use **for** loop on **Array**
* Use **for in** loop on non-array **Object**
* Use .**hasOwnProperty** to filter out inherited proptotype members

**for** (var i **in** man) {

if (man.**hasOwnProperty**(i)) { // filter

console.log(i, ":", man[i]);

}

}

Not Augmenting Built-In (Object, Array, Function) Prototypes

* bcos it will globally affect ALL derived types and could have unintended side-effects.
* Code do not use .hasOwnProperty will include your built-in prototypes
* May not be that bad.

Use Strict Equality === or !== to Avoid Implicit Typecast

* **=== & !==** compare against **same type same value**.
* JavaScript did a lot of implicit typecast. To protect yourself use **===** instead of **==** to test variable against value.

var zero = 0; // zero is a **number** 0, not a **boolean** false

if (**zero === false**) {

// **not executing** because zero is 0, not false

}

// antipattern

if (**zero == false**) {

// this block is executed...

}

Eval(someString) is Evil

* Dynamic Code Composition by string: **Security** implication similar to SQL Injection. E.g. Ajax Injection.
* **Same as - setInterval**(), **setTimeout**(), **Function**() has to be careful!. They all use similar mechanism.

setTimeout(**"myFunc()"**, 1000); // BAD

setTimeout(**"myFunc(1, 2, 3)"**, 1000); // BAD

setTimeout(**myFunc**, 1000); // GOOD

setTimeout(**function () {** // GOOD

myFunc(1, 2, 3);

}, 1000);

ParseInt(“someString”, base) MUST USE base

* W/O using base, date won’t work!
* var month = "06",

year = "09";

month = parseInt(month, **10**);

year = parseInt(year, **10**); // w/o the base, year will be 0 bcos “09” will be treated as Octal(base 8) by default if a string starts with zero 0.

* Alternative:
  + +”09” // gives you numeric 9 bcos of the unary operator
  + Number(“09”) // will return numeric 9

Coding Convention

* Always use curly braces even in the case of single statement like if & for.
* Opening Brace Location
  + **Pitfall** caused by auto semicolon insertion!
  + function func() {

**return // if the { is on the next line, a semicolon will auto insert after return, and that is BAD!!!**

**{**

name: "Batman"

};

}

* + equivalent to this:
  + function func() {

**return undefined;** // a semi-colon is auto inserted right after return.

// unreachable code follows...

{

name: "Batman"

};

}

* + **ALWAYS Position the Brace on the same line** e.g.
  + function func() {

**return { // here on the same line**

name: "Batman"

};

}

Naming Convention

* **Constructor** **function**: **Upper** **camel** case
* **Regular** **function**: **lower** **camel** case
* **Regular** **Variable**: **all** **lower** case delimted by underscore \_
* **Private** **members**: **underscore prefix** e.g. **\_**myPrivateProperty
* **Constant**: **ALL Upper** case.

Literals & Constructor

Object Literal

* **ALWAYS USE literal { … }**,
* **DON’T USE built-in Object()** constructor **new Object(); , other ctors like Number(), Date(), String(), Array() are fine.**

Custom Constructor

* Usage, calling **Constructor Function** using **new**: e.g. var person = **new Person(‘vincent’);**
* Here’s how the Person constructor function could be defined.

**var Person** = **function (name) {**

this.name = name;

this.say = function () {

return "I am " + this.name; // this is still refers to the containing object, bcos this function is still a method.

};

};

**SAME AS this non-anonymous version:**

**function Person(name) {** … }

* It’s as if something like this happens behind the scenes:

var Person = function (name) {

**// create a new object**

**// using the object literal**

**// var this = {}; it happens implicitly, equivalent to var this = Object.create(Person.prototype);**

**// and the default Person.prototype is just the built-in “Object”**

**this**.name = name;

**this**.say = function () {

return "I am " + this.name;

};

**// return this; by default it implicitly return “this”, unless you override by EXPLICITLY “return someObject;”**

};

* + **Explicity return:**

var Objectmaker = function () {

// this `name` property will be ignored

// because the constructor

// decides to return another object instead

this.name = "This is it";

// creating and returning a new object

**var that = {};**

**that.name = "And that's that";**

**return that;**

};

// test

var o = **new** Objectmaker();

console.log(o.name); // o.name **returns** **"And that's that" NOT** “This is it”.

* **Pitfall**
  + if you **forgot** using **new**, you are just calling a **Regular Function** rather than **Construtor Function**.
    - For Regular Function, the **“this”** inside the function will point to the **Global** object, therefore this.name will **create** a **name** property in the **Gobal** object **instead** **of** the **Person** object.
    - p.s. in strict mode, “this” will NOT point to the Global object.
    - **Fixes**:
      * Naming Convention: **Capitalize** the object name
      * **Cache this: var that = {}; that.name; that.say() = function() { …} return that;**
      * **This fix has its own problem, the prototype chain is broken!**
      * **Alternative:**

function Person() {

**if (!(this instanceof Person)) { // check whether this is an instance or your ctor**

**return new Person(); // if not, invoke itself property with new and return.**

}

this.name = "This is it";

}

* + Inefficient: Everytime a **new Person();** is called, a new **function(name)** is created and a new **name** & **say()** are created. The better way is to put the **say()** method into the prototype: **Person.prototype.say = function() {** … }

Array

* **ALWAYS** Create using **Literal [ ]**:

**BAD:** var a = **new Array("itsy", "bitsy", "spider");**

**GOOD:** var a = **["itsy", "bitsy", "spider"];**

* + **BAD: Array(3)** **set** the **length** of an array **NOT** create **an** **element** **of** **value** **3** in the array, some ppl found it **confusing**
* **Test Arrayness:** 
  + **typeof** a // "**object**", because arrays are objects, **correct BUT NOT USEFUL**!
  + Therefore 2 ways of checking:
    - 1. myArray**.constructor === Array**,
    - 2. **Array.isArray(**myArray);

JSON

* JavaScript Object Notation:
  + {**“**name**”**: “kayla”, “age”: 12, “someArray” : [1,2,3] }
  + **Properties** **ONLY**, **NO functions**
  + **Property Name** need to wrap in **quotes “”**
* **Serialize**:
  + **Native: JSON.stringify**(someObjectOrArray);
* **Deserialize**:
  + **Native:** **JSON.Parse(**someJSONString); // if will throw an exception if the text contains anything dangerous.
  + **jQuery:** **$.parseJSON(**someJSONString);
  + **BAD: eval()**;

Primitive Types vs Primitive Wrappers

* **5 primitive types**: number, string, boolean, null, undefined. **3 immutable**: number, string, boolean
* **3 primitive wrappers** ctors: new Number(), new String(), new Boolean().
* **Always use Primitive Types instead of Wrapper ctor**. bcos, even Primitive Type support augmenting. The only exception is Date

Error Objects

* **Avoid** throw 3 **built-in** Error(), SyntaxError(), TypeError(). All with properties: name, message. Use by throw new SyntaxError().
* **ALWAYS** throw **Custom Object** instead

try {

// something bad happened, **throw a Custom error *object literal***

**throw** **{**

***name: "MyErrorType", // custom error type***

***message: "oops",***

***extra: "This was rather embarrassing",***

***remedy: genericErrorHandler // who should handle it***

**}*;***

} catch (e) {

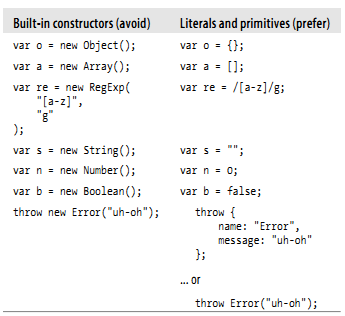
// inform the user

alert(e.message); // "oops"

// gracefully handle the error

e.remedy(); // calls genericErrorHandler()

}

* 

Function

Hoisting

* **Inside a function,** 
  + a local function **declaration** get **HOISTED**,
  + a local function’s **Initialized** **expression NOT HOISTED.**

// global functions

function foo() {

alert('global foo');

}

function bar() {

alert('global bar');

}

function hoistMe() {

console.log(typeof foo); // "function"

console.log(typeof bar); // "undefined"

foo(); // "local foo"

bar(); // bar is defined BUT bar is not a function, throws TypeError

// function **declaration** get **hoisted**:

**function foo() {**

***alert('local foo');***

**}**

// function bar is **Initialized** with the **expression** which is **NOT hoisted**:

// BUT var **'bar' Declaration** **gets hoisted !!!**

**var bar = function () {**

alert('local bar')s;

**};**

}

hoistMe();

1-time Immediate Function Initialization

* **e.g. (function () {** // an **anonymous** 1-time initialization function.

**var** x = 1; **// a temporary local scope var to help other initialization code.**

alert('watch out!');

}**() ); // execute here!**

* Same as Self-Invoke, Self-Excute Pattern. i.e. the function **executes as soon as it is defined**!
* Best use: Provide **1-time initialization** using **anonymous** function with **temporary local** **scoped vars** inside to help initialization.
* 1. using function **expression**.
* 2. Add parenthese **()** at the end e.g. (function() { … } **()** );
* 3. Wrap the entire function with parentheses **( function() {…}() );**
  + **Except** when **func** is **assigned** to a **var**, then **NO** NEED for **parentheses** & **var** is the **return** **value(NOT** the **function)**:
  + e.g. **var test = function() { … }();**

1-time Immediate Object Initialization

* **Same** reason and benefit **as** the **Immediate Function Initialization**. But using Object Literal to provide **a more manageable and structured** **initialization** using a **OOP** approach
* **({**

// here you can define setting values

// a.k.a. configuration constants

maxwidth: 600,

maxheight: 400,

// you can also define utility methods

**gimmeMax**: function () {

return this.maxwidth + "x" + this.maxheight;

},

// initialize

**init**: **function** () {

console.log(this.**gimmeMax**());

// more init tasks...

**return this**; // optionally return this if you want to access properties defined in the object.

}

**}).init()**; // **call init() to kick-off initialization**

* Both patterns behave the same
  + ({...} **)**.init();
  + ({...}.init() **)**;

Callback using object method

* Read more …
* **Pitfall**:

// **Red** is the **callback**, **Purple** is the **callback’s param**

var myapp = {};

myapp.color = "green";

**myapp**.**paint** = function (**node**) {

**node**.style.color = **this.color; // this.color === myapp.color most of the time, BUT** **!==** **WHEN** **method** use as **callback**

};

var findNodes = function (**callback**) {

// some impl. to look up some Node …

if (typeof callback === "function") {

**callback**(**found**); // pass the found Node to the callback as argument.

}

};

findNodes(**myapp**.**paint**); // WONT WORK, bcos myapp.paint’s **this.color** is **undefined**

* **Solution**: pass the callback as well as the object this callback belongs to

findNodes(**myapp, myapp.paint**);

var findNodes = function (**callback\_obj**, callback) {

// some impl. to look up some Node …

if (typeof callback === "function") {

**callback.call(callback\_obj, found);**

}

};

* + Optional, you can pass the method name as a string!

Memoization: cache return result

* **Function is an Object**, therefore we can **add properties**. **Memoization** is **adding** a **property** to **cache** the **return** **value**.
* var myFunc = function (param) {

if (!myFunc.**cache**[param]) { // if it is a new input value, cache the result

var result = {};

// ... some expensive operation that the result needs to be cached ...

myFunc.**cache**[param] = result;

}

return myFunc.**cache**[param];

};

// cache storage

myFunc.**cache** = {};

Curry

* TBD

Object

Namespace

* var myNamespace = { namespaces: ‘cool’, prop1=’’ …};
* Usage: myNamespace.prop1;
* **Namespace** Creation **Helper** which will check for defined or not, to make sure existed namespace won’t got erased by new namespace.

var MYAPP = MYAPP || {}; // **||** specifies the **default** value, here defaults to an empty Object literal

MYAPP.**namespace** = **function** (ns\_string) {

var parts = ns\_string.split('.'),

parent = MYAPP,

i;

// strip redundant leading global

if (parts[0] === "MYAPP") {

parts = parts.slice(1);

}

for (i = 0; i < parts.length; i += 1) {

**// create a property if it doesn't exist**

if (**typeof** parent[parts[i]] === "**undefined**") {

parent[parts[i]] = {};

}

**parent = parent[parts[i]];**

}

return **parent**;

};

* This implementation enables all of these 3 different uses:

1. var **module2** = MYAPP.namespace('MYAPP.modules.module2'); **// cache to local var, FAST, preferred way**

**module2** === **MYAPP.modules.module2**; // true, the exact same object!

1. **MYAPP.namespace('modules.module2');** // using **namespace()** helper **method (**You can **skip initial** `MYAPP`**)**, **SLOW**
2. **MYAPP.modules.module2**.property; // **DIRECT** **Reference**, **SLOW**

Add References / Dependencies

* **Cache** a local variable at **top of your function or module**

var myFunction = function () {

**var event = YAHOO.util.Event**, // **cache** **dependencies**

dom = YAHOO.util.Dom; …

};

EXCELLENT – Expose Private to Public – w/o new

* Using **ctor function**. Make sure the property is local to that function. E.g.

function **Gadget**() {

// **private** member

**var** **name** = 'iPod';

// **public** function

**this**.**getName** = function () {

return **name**;

};

}

var toy = **new** **Gadget**();

// `name` is undefined, it's **private to the outside**

console.log(**toy.name**); // **undefined**

// **public** method has access to `name`, **public** to other members **within** the **same** **object**.

console.log(toy.getName()); // "iPod"

* **EXCELLENT:** Using
  + 1. **immediate function** to
  + 2. **return** **object literal**
  + 3. **expose** **private** members using **closure**

**var** **myobj**; // this will be the object, or namespace e.g. var myobj1 = { myobj2: {} };

(function () { // **immediate** **function**

// **private** members

**var** **name** = "my, oh my";

// implement the public part

// note -- no `var`, assign to myobj which can be a Global or an object i

**myobj** = {

// **public**/privileged method

getName: function () {

return **name**; // access **name** using **closure**

}

};

}**())**;

**myobj**.getName(); // access the public method "my, oh my"

* **Revelation** Pattern
  + Having private methods but also expose them as public methods. In the case of public methods got overwritten, the private methods won’t be affected.
  + **var** **myarray**; // note: myarray is an object with method to test array

(**function** () { // **immediate** **function**

**var** **astr** = "[object Array]", // **private variables**

**toString** = Object.prototype.toString;

// **alternative syntax**: continue with **var** **isArray =** function(a) { …

**function** isArray(a) { // **private** **function**

return toString.call(a) === astr;

}

**function** indexOf(haystack, needle) { // **private** **function**

var i = 0,

max = haystack.length;

for (; i < max; i += 1) {

if (haystack[i] === needle) {

return i;

}

}

return −1;

}

**myarray** = {

**isArray: isArray, // expose private function to public via closure**

**indexOf: indexOf, // expose private function to public via closure**

**inArray: indexOf // expose private functionto public via closure**

};

}**())**;

* + Usage:
    - myarray.isArray([1,2]); // true
    - myarray.isArray({0: 1}); // false
    - myarray.indexOf(["a", "b", "z"], "z"); // 2
    - myarray.inArray(["a", "b", "z"], "z"); // 2
  + let’s say the public myarray.indexOf got overwritten/redefined, the msarray.inArray will still work bcos the private indexOf is still intact.
    - myarray.indexOf = null;
    - myarray.inArray(["a", "b", "z"], "z"); // 2

EXCELLENT – Module – w/o new

* Module Pattern: A combination of: Namespaces, Immediate functions, Private & privileged members, Declaring dependencies.
* MYAPP.**namespace**('MYAPP.utilities.array'); **// create namespaced object**

**// immediate function**

MYAPP.utilities.array = (function () { // using immediate function to create an Array Utility Object

// **dependencies, single line mult-variable initialization**

**var** uobj = MYAPP.utilities.object,

ulang = MYAPP.utilities.lang,

**// private properties**

array\_string = "[object Array]",

ops = Object.prototype.toString;

**// private methods**

// ...

**// end var**

// optionally one-time init procedures

// ...

**// public API by returning Object Literal**

**return** {

**// public methods**

**inArray**: function (needle, haystack) {

for (var i = 0, max = haystack.length; i < max; i += 1) {

if (haystack[i] === needle) {

return true;

}

}

},

**isArray**: function (a) {

return ops.call(a) === **array\_string**; // reference **private** **property** via **closure**

}

// ... more methods and properties

}; // **end** **return**

**}()); // immediate function**

* **Revelation** pattern used in Module

MYAPP.**namespace**('MYAPP.utilities.array'); **// create namespaced object**

MYAPP.utilities.**array** = (function () {

// **private** **properties**

var array\_string = "[object Array]",

ops = Object.prototype.toString,

// **private** **methods**

**inArray** = function (haystack, needle) {

for (var i = 0, max = haystack.length; i < max; i += 1) {

if (haystack[i] === needle) {

return i;

}

}

return −1;

},

**isArray** = function (a) {

return ops.call(a) === array\_string;

};

// end var

**// revealing public API**

**return {**

**isArray: isArray, // private function to public** via **closure**

**indexOf: inArray // private function to public** via **closure**

**};**

}**())**; // immediate function

* **Global** and **Root** **App** Reference

MYAPP.utilities.module = (function (app, global) {

// references to the **global object**

// and to the **global app namespace object**

// are now localized

}(**MYAPP**, **this**));

* Constructor Function: TBD

Sandbox

EXCELLENT - Static/Base class Instance

* **Public Static VS Instance method**

// **constructor**

var **Gadget** = function () {}; // **Gadget is a function & function is also an Object**, therefore we can a**dd properties/methods to Gadget**

// a **static** **method** of the **Gadget function object**

**Gadget.isShiny** = function () {

**this**.prop1 = “guess where I go”; // **this** is bound to the **ctor** **function** **object Gadget**

return "you bet";

};

// a normal **instance** **method** added to the prototype, **prototype** **is** **Object** **bcos** using **new**

**Gadget.prototype.setPrice** = function (price) {

this.price = price; // create a **new** **property** at **derived level**: **this.price. Value** of price will be **different** on **different** object **instances!**

};

Calling:

// calling a **static** method of the **function object**.

**Gadget**.**isShiny**(); // "you bet"

// creating an **instance** and calling a method

var **iphone** = new **Gadget**();

**iphone**.**setPrice**(500);

* Won’t work: calling instance statically. Calling static with an instance

typeof Gadget.setPrice; // "undefined", calling instance statically

typeof iphone.isShiny; // "undefined", calling static with an instance

* There are also pattern that support calling both types interchangeably
* Private Static:
  + …

Constant

* No such thing as const.
* Just use a **naming** **convention** like using **ALL** **CAPS**. E.g. Math.PI = 3.1415926535, and add them as **static** **properties** to the **ctor** **functions**.

// **constructor**

var **Widget** = function () {

// implementation...

};

// constants

Widget.**MAX\_HEIGHT** = 320;

Widget.**MAX\_WIDTH** = 480;

Chaining, Fluent Interface

* Create methods that return this, the instance of the current object.
* Pros & Cons: concise code that read like a sentence, splitting a large do all function into smaller specialized functions. Harder to debug since chain into 1 single line.

method() Method

* Turn JavaScript to a class-like language
* NOT a recommended, but could see in some libraries
* Create **method()** so methods defined in ctor function will not duplicate in ALL INSTANCES of objects.
* Implementation of **method() at proptotype:**

if (typeof Function.prototype.method !== "function") {

**Function.prototype.method** = **function** (name, implementation) {

**this.prototype[name]** = implementation;

return this;

};

}

* Define a class

var **Person** = **function** (name) {

this.name = name;

}.

**method**('getName', function () {

return this.name;

}).

**method**('setName', function (name) {

this.name = name;

return this;

});

* Usage

var a = **new** **Person**(**'Adam'**);

a.getName(); // **'Adam'**

a.setName('Eve').getName(); // 'Eve'

Examples

Instance members(both derived & base), Static members

* var MyObject = function(x, y) {

**// Derived class instance method sumOfXY, instance properties x & y**

this.x = x; // if you dont use new, this.x will go to Global!!!, now it just goto containing object

this.y = y;

// this method will showup on ALL new object instance, consider move it to base class prototype

this.sumOfXY = function() {

return this.x + this.y; // this is a method, therefore this points to the containing object

}

}

**// Static method myStaticFoo, and Static property z**

MyObject.myStaticFoo = function(z) {

this.z = z; // this points to the function object

return this.z + 1

}

// this is a good place to implement methods that shared across multiple instance.

**// Base class instance method baseFoo & instance property a & b**

MyObject.prototype = {

// baseFoo will be inherited into derived class

baseFoo: function(a) {

this.a = a; // this.a will be inherited into derived class

return this.a + this.sumOfXY(); // p.s. using **PROPERTIES from BOTH BASE & DERIVED**

},

b: "default hello: " // b will also be inherited into derived class

}

// Create 2 instances and show the instance properties ARE NOT SHARED

// Also show Static properties ARE SHARED

var myObject = new MyObject(100, 200);

// create a 2nd instance to show that the instance property x, y, a, b are not shared across instances

var myObject2 = new MyObject(1000, 2000);

console.log(myObject.x); // 100 – INSTANCE Property from derived

console.log(myObject.y); // 200 – INSTANCE Property from derived

console.log(myObject.sumOfXY()); // 300

console.log(MyObject.myStaticFoo(400)); // 401- STATIC Method using STATIC Property

console.log(MyObject.z); // 400 - STATIC Property

console.log(myObject.baseFoo(500)); // 800 – INSTANCE Method from base using INSTANCE Properties from base **AND** derived

console.log(myObject.a); // 500 – INSTANCE Property from base

myObject.a += 12; //set

console.log(myObject.a); // get 512

myObject.b += " world"; // append to INSTANCE Property from base

console.log(myObject.b); // hello world

console.log(myObject2.x); // 1000

console.log(myObject2.y); // 2000

console.log(myObject2.sumOfXY()); // 3000

console.log(MyObject.myStaticFoo(4000)); // 4001

console.log(MyObject.z); // 4000

console.log(myObject2.baseFoo(5000)); // 8000

console.log(myObject2.a); // 5000

myObject2.a += 12; //set

console.log(myObject2.a); // get 5012

console.log(myObject2.b); // getting the default: hello. NOT Affected by myObject

console.log(myObject.x); // 100

console.log(myObject.y); // 200

console.log(myObject.sumOfXY()); // 300

console.log(MyObject.myStaticFoo(400)); // 401

console.log(MyObject.z); // 400

console.log(myObject.baseFoo(500)); // 800

console.log(myObject.a); // 500

myObject.a += 12; //set

console.log(myObject.a); // get 512

myObject.b += " world";

console.log(myObject.b); // hello world

//MyObject

DOM Access

* Avoid DOM Access in loops
* Assign DOM reference to local var and work with that local var
* Use selector API
* Minimize adding to “Live” document tree.
  + Create & manipulate document fragment first, and then add it to “Live” document tree.

var frag = document.**createDocumentFragment**();

1or more: frag.appendChild(…);

**document.body.appendChild(frag);**

* + **Update**: **Clone** & manipulate document fragment first, and then add it to “Live” document tree.

var oldnode = document.getElementById(…);

var clone =oldnode.**cloneNode**(true);

// … manipulate the clone

oldnode.**parentNode**.**replaceChild**(clone, oldnode);

Event

* Cross-browser Event Handling

var b = document.getElementById('clickme');

if (document.addEventListener) { // W3C

b.**addEventListener**('click', myHandler, false);

} else if (document.attachEvent) { // **Before** **IE9**

b.**attachEvent**('onclick', myHandler);

} else { // last resort

b.onclick = myHandler;

}

* **Event Delegate**
  + Take advantage of event **bubbling** to allow **1 event handler** to **handle** **multiple** **events** nested inside a single div element.
  + Filter by e.**srcElement** & e.**target**

Long Running Script

* setTimeout()
* Web Worker – background thread
  + Create a separate background.js file for the background processing
  + Run it:

var ww = new Worker(“background.js”);

ww.onmessage = function(event) {

event.data // this returns the message, which can be ANY data.

* + background.js

postMessage('Starting');

while (end) {

postMessage(‘some progress status’);

}

}

postMessage('Finished');

Loading Strategies

* Place <script> at the end of the page, right before </body>. Bcos <script> element BLOCKS other downloads.

XHR

* var xhr = new XMLHttpRequest();
* xhr.onreadystatechange = handleResponse;
* xhr.open("GET", "page.html", true);
* xhr.send();