JS Study Guide

General Notes

Function Declaration

```
function diffOfSquares (a, b) {
         return a*a - b*b;
     }
```

Builds in memory immediately when the program loads

Function Expression

```
var diff = function diffOfSquares (a, b) {
         return a*a - b*b;
     };
```

The function keyword will now assign the following function to the variable.

Anonymous Function

```
var diff = function (a, b) {
         return a*a - b*b;
     };
```

Using FE's with Arrays and Map[]

```
var numbers = [12, 4, 3, 9, 8, 6, 10, 1];
```

```
var results = numbers.map(function (arrayCell) {
    return arrayCell *2;
});
```

Immediately Invoked Function Expression

```
( function ( ) {
        alert("Quick! You've got a Fast Pass to " + pass + "!");
} )();
```

Hoisting

1) First, memory is set aside for all necessary variables and declared functions. Variables are declared at the top of a scope and assigned a value of undefined. Functions are declared, then variables are assigned. 2) Function Expressions are never hoisted! They are treated as assignments.

Object Literal

```
var myBox = { height: 6, width: 8, length: 10, volume: 480,
        material: "cardboard", contents: booksArray
};
```

Property Access

```
myBox.width;

    myBox["volume"];
```

Brackets enable dynamic property access

```
for(var i = 1, i <= myBox["# of stops"]; i++){
          console.log( myBox["destination" + i] );
     }

     function addBook (box, name, writer){
         box["# of Books"]++;
         box["book" + box["# of Books"]] = {title: name, author: writer};
     }
```

ECMAScript 5 introduced a new method: Object.create(). Calling this method creates a new object. The prototype of this object is the first argument of the function: A second way to build objects USING Object.create() Using inheritance, we can create new Objects with our existing Objects as prototypes

```
var shoe = { sixe: 6, gender: "women", construction: "slipper"};

     var magicShoe = Object.create( shoe );
```

Examining the inheritance within our shoes We can use an inherited method to demonstrate our newly created prototype chain

```
Object.prototype.isPrototypeOf( shoe );
     shoe.isPrototypeOf( magicShoe );


     function Shoe (shoeSize, shoeColor, forGender, constructStyle) {
         this.size = shoeSize;
         this.color = shoeColor;
         this.gender = forGender;
         this.construction = constructStyle;

         this.putOn = function () { alert("Shoe's on!"); };
         this.takeOff = function () { alert("Uh, what's that smell?"); };
     }
```

```
var beachShoe = new Shoe
( 10, "blue", "women", "flipflop" );
```

Assigning a prototype to a constructor

By setting a constructor's prototype property, every new instance will refer to it for extra properties!

```
function Shoe (shoeSize, shoeColor, forGender, constructStyle) {
        this.size = shoeSize;
        this.color = shoeColor;
        this.gender = forGender;
        this.construction = constructStyle;
    }

    Shoe.prototype = {
        putOn: function () { alert ("Shoe's on!"); },
        takeOff: function () { alert ("Uh, what's that smell?"); }
    };

    Prototypes can also refer back to the instance!
    Shoe.prototype = {
        putOn: function () { alert ("Your " + this.construction + "'s" + "
    };
    beachShoe.
    takeOff
    ();
```

Finding an object's constructor and prototype

Some inherited properties provide ways to find an Object's nearest prototype ancestor

```
Object.constructor;
    Object.constructor.prototype;
```

Event Listeners

EventTarget.addEventListener (removeEventListener)

event.type

- DOMFocusIn
- DOMFocusOut
- DOMActivate

Mouse Events

- click
- mousedown
- mouseup
- mouseover
- mousemove
- mouseout

Mutation Event

- DOMSubtreeModified
- DOMNodeInserted
- DOMNodeRemoved
- DOMNodeRemovedFromDocument
- DOMNodeInsertedIntoDocument
- DOMAttrModified
- DOMCharacterDataModified

HTML Events

- load
- unload

- abort
- error
- select
- change
- submit
- reset
- focus
- blur
- resize
- scroll

More event types:

The registrable events depend on the EventTarget type , e.g.

TouchEvent includes:

- touchstart
- touchend
- touchmove
- touchcancel

Event Capturing and Event Bubbling

For nested objects (elem 1 with elem 2 nested inside) with the same type of event handler (onclick for example) Event capturing means the event on elem 1 takes place first Event bubbling means that the event on elem 2 takes place first

Events are handled in two phases: capturing and bubbling. During the capturing phase, events are dispatched to parent objects before they are dispatched to event targets that are lower in the object hierarchy. During the bubbling phase, events are dispatched to target elements first and then to parent elements. You can register event handlers for either event phase.

Full List of Events

Event > Event Reference [Event Reference - MDN]

JS Good Parts

Review Chapter 2 - Grammar for a collection of big picture railroad diagrams

Douglas Crockford uses this defined method constructor throughout the examples

```
Function.prototype.method = function (name, func) {
         this.prototype[name] = func;
         return this;
     };
```

Ch. 3 Objects

```
var empty_object = {};

     var stooge = {
         "first-name": "Jerome",
         "last-name": "Howard"
     };

     Nested -
     var flight = {
         airline: "Oceanic",
         number: 815,
         departure: {
             IATA: "SYD",
             time: "2004-09-22 14:55",
             city: "Sydney"
         },
         arrival: {
             IATA: "LAX",
             time: "2004-09-23 10:42",
             city: "Los Angeles"
         }
     };
```

Retrieval

```
stooge["first-name"]     // "Joe"
     flight.departure.IATA    // "SYD"
```

The undefined value is produced if an attempt is made to retrieve a nonexistent member

Attempting to retrieve values from undefined will throw a TypeError exception. This can be guarded against with the && operator

```
flight.equipment // undefined
     flight.equipment.model // throw "TypeError"
     flight.equipment && flight.equipment.model // undefined
```

Update

stooge[‘middle-name’] = ‘Lester’; stooge.nickname = ‘Curly’;

Reference

Objects are passed around by reference. They are never copied

Prototype

Every object is linked to a prototype object from which it can inherit properties. All objects created from object literals are linked to Object.prototype, an object that comes standard with JavaScript.

Reflection

Use the typeof operator to determinine the type of a property

```
typeof flight.number  // 'number'
```

Any property on the prototype chain can produce a value

```
typeof flight.toString    // 'function'
```

The other approach is to use the hasOwnProperty method, which returns true if the object has a particular property. The hasOwnProperty method does not look at the prototype chain

```
flight.hasOwnProperty('number')         // true
```

Enumeration

The for in statement can loop over all of the property names in an object. The enu- meration will include all of the properties filter out the values hasOwnProperty method and using typeof to exclude functions

```
var name;
     for (name in another_stooge) {
         if (typeof another_stooge[name] !== 'function') {
             document.writeln(name + ': ' + another_stooge[name]);
         }
     }
```

Delete

```
delete another_stooge.nickname;
```

It will remove a property from the object if it has one. It will not touch any of the objects in the proto- type linkage.

Global Abatement

Minimize the use of global variables by creating a single global variable for your application, and use it as a container for your application

```
var MYAPP = {};

     MYAPP.stooge = {
         "first-name": "Joe",
         "last-name": "Howard"
     };

     MYAPP.flight = {
         airline: "Oceanic",
         number: 815,
         departure: {
             IATA: "SYD",
             time: "2004-09-22 14:55",
             city: "Sydney"
         },
         arrival: {
             IATA: "LAX",
             time: "2004-09-23 10:42",
             city: "Los Angeles"
         }
     };
```

Ch. 4 - Functions

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

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

Since functions are objects, they can be used like any other value. Functions can be stored in variables, objects, and arrays. Functions can be passed as arguments to functions, and

functions can be returned from functions. Also, since functions are objects, functions can have methods.

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

Function Literal

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

1) function 2) (optional) name 3) parameters 4) Statements with optional return

Declaration is function literal without the variable assignment, function expression requires variable assignment

Invocation

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

Functions also have this and arguments

Method Invocation

When a function is stored as a property of an object, we call it a method. When a method is invoked, this is bound to that object. If an invocation expression con- tains a refinement (that is, a . dot expression or [subscript] expression), it is invoked as a method

Function Invocation

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

When a function is invoked with this pattern, this is bound to the global object If a function is

invoked within a method, this has to be explicitly bound to a new variable, that

Augment myObject with a double method.

```
myObject.double = function () {
         var that = this; // Workaround.
         var helper = function () {
             that.value = add(that.value, that.value);
         };
         helper(); // Invoke helper as a function.
     };
```

Invoke double as a method.

```
myObject.double( );
     document.writeln(myObject.getValue()); // 6
```

Constructor Invocation

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

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

```
var Quo = function (string) {
         this.status = string;
     };

     Quo.prototype.get_status = function () {
         return this.status;
     };
```

Make an instance of Quo.

```
var myQuo = new Quo("confused");
     document.writeln(myQuo.get_status()); // confused
```

Use of this style of constructor functions is not recommended. We will see better alternatives in the next chapter.

Apply Invocation

The apply method lets us construct an array of arguments to use to invoke a func- tion. It also lets us choose the value of this. The apply method takes two parame- ters. The first is the value that should be bound to this. The second is an array of parameters.

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

Arguments

It gives the function access to all of the arguments that were sup- plied with the invocation, including excess arguments that were not assigned to parameters.

```
var sum = function () { var i, sum = 0;
          for (i = 0; i < arguments.length; i += 1) {
              sum += arguments[i];
          }
          return sum;
     };
```

Return

The return statement can be used to cause the function to return early. When return is

executed, the function returns immediately without executing the remaining statements. A function always returns a value. If the return value is not specified, then undefined is returned. If the function was invoked with the new prefix and the return value is not an object, then this (the new object) is returned instead.

Exceptions

```
var add = function (a, b) {
         if (typeof a !== 'number' || typeof b !== 'number') {
             throw {
                 name: 'TypeError',
                 message: 'add needs numbers'
             };
         }
         return a + b;
     }
```

The throw statement interrupts execution of the function. It should be given an exception object containing a name property that identifies the type of the exception, and a descriptive message property.

Make a try_it function that calls the new add function incorrectly.

```
var try_it = function () {
         try {
             add("seven");
         } catch (e) {
             document.writeln(e.name + ': ' + e.message);
         }
     }
     try_it( );
```

Augmenting Types

JavaScript allows the basic types of the language to be augmented. In Chapter 3, we saw that adding a method to Object.prototype makes that method available to all objects. This

also works for functions, arrays, strings, numbers, regular expressions, and booleans.

Recursion

Scope

JavaScript does have function scope. That means that the parameters and variables defined in a function are not visible outside of the function, and that a variable defined anywhere within a function is visible everywhere within the function.

In many modern languages, it is recommended that variables be declared as late as possible, at the first point of use. That turns out to be bad advice for JavaScript because it lacks block scope. So instead, it is best to declare all of the variables used in a function at the top of the function body.

Closure

```
var myObject = function () {
          var value = 0;
          return {
              increment: function (inc) {
                  value += typeof inc === 'number' ? inc : 1;
              },
              getValue: function () {
                  return value;
              }
          };
      }( );

      var quo = function (status) {
          return {
              get_status: function () {
                  return status;
              }
          };
      };
      // Make an instance of quo.
      var myQuo = quo("amazed");
      document.writeln(myQuo.get_status( ));
```

The get_status method still has privileged access to quo's status property even though quo has already returned. get_status does not have access to a copy of the parameter; it has access to the parameter itself. This is possible because the function has access to the context in which it was created. This is called closure.

Callbacks

Synchronous

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

Asynchronous

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

Module

From Adequately Good

```
(function () {
       // ... all vars and functions are in this scope only
       // still maintains access to all globals
     }());
```

Notice the () around the anonymous function. This is required by the language, since statements that begin with the token function are always considered to be function declarations. Including () creates a function expression instead.

Global import

```
(function ($, YAHOO) {
       // now have access to globals jQuery (as $) and YAHOO in this code
     }(jQuery, YAHOO));
```

By passing globals as parameters to our anonymous function, we import them into our code, which is both clearer and faster than implied globals.

Module Export

```
var MODULE = (function () {
         var my = {},
         privateVariable = 1;

         function privateMethod() {
             // ...
         }

         my.moduleProperty = 1;
         my.moduleMethod = function () {
             // ...
         };

         return my;
     }());
```

Augmentation

```
var MODULE = (function (my) {
         my.anotherMethod = function () {
             // added method...
         };

         return my;
     }(MODULE));
```

To augment modules. First, we import the module, then we add properties, then we export it.

Loose Augmentation - flexible multi-part modules that can load themselves in any order

```
var MODULE = (function (my) {
         // add capabilities...

         return my;
     }(MODULE || {}));
```

Tight Augmentation - implies a set loading order, but allows overrides.

```
var MODULE = (function (my) {
         var old_moduleMethod = my.moduleMethod;

         my.moduleMethod = function () {
             // method override, has access to old through old_moduleMethod
         };

         return my;
     }(MODULE));
```

Cross file private state

```
var MODULE = (function (my) {
         var _private = my._private = my._private || {},
         _seal = my._seal = my._seal || function () {
             delete my._private;
             delete my._seal;
             delete my._unseal;
         },
         _unseal = my._unseal = my._unseal || function () {
             my._private = _private;
```

```
            my._seal = _seal;
            my._unseal = _unseal;
        };

        // permanent access to _private, _seal, and _unseal

        return my;
    }(MODULE || {}));

    // Sub-modules

    MODULE.sub = (function () {
        var my = {};
        // ...

        return my;
    }());
```

Cascade

Methods that set or change the state of an object and return this

Curry

Currying applies a function with an explicit subset of arguments

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

Memoization

Functions use objects to remember the results of previous operations

```
var fibonacci = function () {
          var memo = [0, 1];
          var fib = function (n) {
              var result = memo[n];
              if (typeof result !== 'number') {
                  result = fib(n - 1) + fib(n - 2);
                  memo[n] = result;
              }
              return result;
          };
          return fib;
      }( );
```

Ch. 5 - Inheritance

Pseudoclassical

We can define a constructor and augment its prototype:

```
var Mammal = function (name) {
          this.name = name;
      };

      Mammal.prototype.get_name = function () {
          return this.name;
      };

      Mammal.prototype.says = function () {
          return this.saying || '';
      };
```

Generally there are better options for code reuse in JavaScript

Object Specifiers

Sometimes better to define a constructor that takes a single object instead of a number of

parameters

Prototypal

Prototypal inheritance is conceptually simpler than classical inheritance: a new object can inherit the properties of an old object.

Differential Inheritance

```
var myMammal = {
          name : 'Herb the Mammal',
          get_name : function () {
              return this.name;
          },
          says : function () {
              return this.saying || '';
          }
      };

      var myCat = Object.create(myMammal);

      myCat.purr = function (n) {
          var i, s = '';
          for (i = 0; i < n; i += 1) {
              if (s) {
                  s += '-';
              }
              s += 'r';
          }
          return s;
      };
```

Functional

Using the module pattern we can achieve private variables in inheritance

1) Create object (literal or call constructor or Object.create) 2) Optional private instances or methods 3) Augments object with methods that have privileged access to properties/methods in step 2 4) Returns object

```
var constructor = function (spec, my) {
         var that, other private instance variables;
         my = my || {};
         Add shared variables and functions to my
         that = a new object;
         Add privileged methods to that
         return that;
     };
```

The spec object contains all of the information that the constructor needs to make an instance.

```
var cat = function (spec) {
         spec.saying = spec.saying || 'meow';
         var that = mammal(spec);
         that.purr = function (n) {
             var i, s = '';
             for (i = 0; i < n; i += 1) {
                 if (s) {
                     s += '-';
                 }
                 s += 'r';
             }
             return s;
         };
         that.get_name = function () {
             return that.says() + ' ' + spec.name + ' ' + that.says();
         }
         return that;
     };
     var myCat = cat({name: 'Henrietta'});
```

Parts

Can compose objects from sets of parts

Ch. 6 Arrays

Array Literals

0 indexed like other languages

```
var empty = [];

     var numbers = ['zero', 'one', 'two', 'three', 'four','five', 'six', 's
```

Length

```
myArray.length;
     myArray[1000000] = true;
     myArray.length;
```

The length property is the largest integer property name in the array plus one. This is not necessarily the number of properties in the array The [] postfix subscript operator converts its expression to a string using the expres- sion's toString method if it has one. The length can be set explicitly. Making the length larger does not allocate more space for the array. Making the length smaller will cause all properties with a sub- script that is greater than or equal to the new length to be deleted

Delete

JavaScript's arrays are really objects,

```
delete numbers[2];
```

Leaves a hole in the array

better to use splice

Enumeration

Since JavaScript's arrays are really objects, the for in statement can be used to iter- ate over all of the properties of an array. To ensure ordering, use for

Confusion

The rule is simple: when the property names are small sequential integers, you should use an array. Otherwise, use an object. Have to explicitly define is_array function to test (see p 61) Having such a test, it is possible to write functions that do one thing when passed a single value and lots of things when passed an array of values.

Methods

Because an array is really an object, we can add methods directly to an individual array

Dimensions

To make a two-dimensional array or an array of arrays, you must build the arrays yourself

```
for (i = 0; i < n; i += 1) {
          my_array[i] = [];
     }
```

Note: Array.dim(n, []) will not work here. Each element would get a reference to the same array, which would be very bad.

The cells of an empty matrix will initially have the value undefined. If you want them to have a different initial value, you must explicitly set them.

Ch. 7 Regular Expressions

Many of JavaScript's features were borrowed from other languages. The syntax came from Java, functions came from Scheme, and prototypal inheritance came from Self. JavaScript's

Regular Expression feature was borrowed from Perl.

```
var url = "http://www.ora.com:80/goodparts?q#fragment";
     var result = parse_url.exec(url);
     var names = ['url', 'scheme', 'slash', 'host', 'port',
     'path', 'query', 'hash'];
     var blanks = '       ';
     var i;
     for (i = 0; i < names.length; i += 1) {
         document.writeln(names[i] + ':' +
         blanks.substring(names[i].length), result[i]);
     }
```

Construction

Regex literal or use new RegExp

- Choice - vertical bar | , attempts to match each of the sequences in order
- Sequence - factor followed by quantifier
- Factor - char, group, class, or escape sequence
- Escape - \ followed by a specifier
- Group -

```
  1. Capturing - wrapped in ()
       2. Non-capturing - (?:), simply matches does not capture the matched
       3. Positive Lookahead (?=), like noncapturing but rewinds after text
       4. Negative Lookahead - (?!) matches only if it failed to match
```

- Class - way to specify one of a set of characters
- Class Escape -
- Regexp Quantifier - Specifies the number of times a factor should match

Ch. 8 - Methods

Array

- Appends items to array - array.concat
- Turns all elements into string with separator - array.join
- array.pop
- array.push
- Modifies actual array by reversing elements - array.reverse
- Removes the first element from array and returns it - array.shift
- array.slice
- Takes a compare function and reorders - array.sort
- Removes items and replaces them - array.splice
- Pushes onto front - array.unshift

Function

- Invokes a function with a specified this and array of arguments (similar to call) - function.apply

Number

- Converts to string in exponential form - number.toExponential
- Converts to string in decimal form - number.toFixed
- Converts to string in decimal form - number.toPrecision
- Converts to string - number.toString

Object

- Returns true if the object has a property with the given name - object.hasOwnProperty

RegExp

- If regexp is matched within the string, it returns an array fith the matche substring at 0, the

text captured at group 1, etc. - rexgexp.exec

- Faster, returns true if matched, otherwise false - regexp.test

String

- string.charAt
- string.charCodeAt
- string.concat // Prefer + operator
- string.indexOf // searches for substring
- string.lastIndexOf // searches from end of string
- string.match // same as regexp.exec, with g flag returns array of all matches excluding capturing groups
- string.replace // Takes search value (either string or regexp) and a replace value
- string.search // like indexOf but takes regexp
- string.slice
- string.split
- string.substring
- string.toLocaleLowerCase
- string.toLocaleUpperCase
- string.toLowerCase
- string.toUpperCase
- string.fromCharCode

Falsy Values

- false
- null
- undefined
- ‘’
- 0
- NaN

The falsy values false, 0 (zero), and “” (empty string) are all equivalent and can be compared against each other The falsy values null and undefined are not equivalent to anything except themselves The value NaN is not equivalent to anything — including NaN typeof(NaN) returns “number”. Fortunately, the core JavaScript function isNaN() can be used to evaluate whether a value is NaN or not.

Things to Know -

1. Object and array literal syntax

2. Dot notation & subscript/bracket notation for methods

3. === and !==

4. Function (Lexical) Scope

5. Hoisting

6. Casting Values -

```
String(42);  // "42"
     Number("42");  // 42
     Boolean("true");  // true

     // cast string to number using the plus operator
     +"42";  // 42

     // cast number to string using an empty string and coercion
     "" + 42;  // "42"
```

7. Using number methods on number literals - (2).toFixed(); // “2”

8. Arguments object is not an array

9. Privatizing variables

```
var fn = (function() {
         var privatizedVariable = "private";
         return function() {
             return privatizedVariable;
         };
```

```
})();

fn();  // "private"
privatizedVariable;  // ReferenceError
```

10. Prototypal inheritance

11. call and apply

12. function expressions vs. function declarations vs. named function expressions

13. Name resolution order:

    1. Language-defined: All scopes are, by default, given the names this and arguments.

    2. Formal parameters: Functions can have named formal parameters, which are scoped to the body of that function.

    3. Function declarations: These are of the form function foo() {}.

    4. Variable declarations: These take the form var foo;.

The most important special case to keep in mind is name resolution order. Remember that there are four ways for names to enter a given scope. The order I listed them above is the order they are resolved in. In general, if a name has already been defined, it is never overridden by another property of the same name. This means that a function declaration takes priority over a variable declaration. This does not mean that an assignment to that name will not work, just that the declaration portion will be ignored. There are a few exceptions:

The built-in name arguments behaves oddly. It seems to be declared following the formal parameters, but before function declarations. This means that a formal parameter with the name arguments will take precedence over the built-in, even if it is undefined. This is a bad feature. Don’t use the name arguments as a formal parameter.

Trying to use the name this as an identifier anywhere will cause a SyntaxError. This is a good feature.

If multiple formal parameters have the same name, the one occurring latest in the list will take precedence, even if it is undefined.

Function Parameters

If a function is called with missing arguments (less than declared), the missing values are set to: undefined If a function is called with too many arguments (more than declared), these arguments cannot be referred, because they don't have a name. They can only be reached in the arguments object.

Double Exclamation Points - Defensive Programming "foo" = "foo" !"foo" = false !!"foo" = true

Truth Table -

```
''        ==   '0'           // false
     0         ==   ''            // true
     0         ==   '0'           // true
     false     ==   'false'       // false
     false     ==   '0'           // true
     false     ==   undefined     // false
     false     ==   null          // false
     null      ==   undefined     // true
     " \t\r\n" ==   0             // true

     NaN   ===  NaN     //false
     !!NaN === !!NaN    //true
```

Testing and Debugging

Debugging

Logging - Use console.log and others of the added console methods in chrome

```
function log() {
         try {
             console.log.apply(console, arguments);
         } catch(e) {
             try {
                 opera.postError.apply(opera, arguments);
             } catch(e){
                 alert(Array.prototype.join.call( arguments, " "));
             }
```

```
        }
    }
```

Logging method that works for outdated versions of Opera

- Breakpoints - Can set breakpoints within browser console
- Step over - Proceeds from breakpoint line to next line of code without executing any funciton/method calls in the breakpointed line
- Step into - Proceeds into breakpointed line and executes code
- Step out - Moves to return statement at the point where the function call one step higher then the function containing the breakpoint resumes control

Testing

- Repeatability - Can be run many times and produce exact same results
- Simplicity - Test just one thing
- Independence - Execute in isolation

Deconstructive - Whittle down code to bar minimum to reproduce the problem and then run tests Constructive - Start from a known, healthy functioning state for the code and rebuild until able to reproduce the bug in question

Test suites - custom built for frameworks and should display testing results

- The ability to simulate browser behavior (clicks, keypresses, and so on)
- Interactive control of tests (pausing and resuming tests)
- Handling asynchronous test timeouts
- The ability to filter which tests are to be executed

Assertion - core of testing, check if a premise (a function return, a variable value, no errors thrown), and log a message depending on the result

Effective JS

Ch.1

Know which JS

- Decide which versions of JavaScript your application supports.
- Be sure that any JavaScript features you use are supported by all environments where your application runs.
- Always test strict code in environments that perform the strict- mode checks.
- Beware of concatenating scripts that differ in their expectations about strict mode.

Floating Points

- JavaScript numbers are double-precision floating-point numbers.
- Integers in JavaScript are just a subset of doubles rather than a separate datatype.
- Bitwise operators treat numbers as if they were 32-bit signed integers.
- Be aware of limitations of precisions in floating-point arithmetic.

Implicit Coercions

- Type errors can be silently hidden by implicit coercions.
- The + operator is overloaded to do addition or string concatenation depending on its argument types.
- Objects are coerced to numbers via valueOf and to strings via toString.
- Objects with valueOf methods should implement a toString method that provides a string representation of the number produced by valueOf.
- Use typeof or comparison to undefined rather than truthiness to test for undefined values.

Primitives over Object Wrappers

- Object wrappers for primitive types do not have the same behavior as their primitive values when compared for equality.
- Getting and setting properties on primitives implicitly creates object wrappers.

Avoid Using == with Mixed Types

- The == operator applies a confusing set of implicit coercions when its arguments are of different types.
- Use === to make it clear to your readers that your comparison does not involve any implicit coercions.
- Use your own explicit coercions when comparing values of different types to make your program's behavior clearer.

Limits of Semicolon Insertion

- Semicolons are only ever inferred before a }, at the end of a line, or at the end of a program.
- Semicolons are only ever inferred when the next token cannot be parsed.
- Never omit a semicolon before a statement beginning with (, [, +, -, or /.
- When concatenating scripts, insert semicolons explicitly between scripts.
- Never put a newline before the argument to return, throw, break, continue, ++, or –.
- Semicolons are never inferred as separators in the head of a for loop or as empty statements.

Strings as sequence of 16 bit units

- JavaScript strings consist of 16-bit code units, not Unicode code points.
- Unicode code points 216 and above are represented in JavaScript by two code units, known as a surrogate pair.
- Surrogate pairs throw off string element counts, affecting length, charAt, charCodeAt, and regular expression patterns such as ".".
- Use third-party libraries for writing code point-aware string manipulation.
- Whenever you are using a library that works with strings, con- sult the documentation to see how it handles the full range of code points.

Ch. 2

Minimize Global Object Use

- Avoid declaring global variables.

- Declare variables as locally as possible.
- Avoid adding properties to the global object.
- Use the global object for platform feature detection.

Always decalre local variables

- Always declare new local variables with var.
- Consider using lint tools to help check for unbound variables.
- Avoid with - with calls a number of methods in sequence on an object
- Avoid using with statements.
- Use short variable names for repeated access to an object.
- Explicitly bind local variables to object properties instead of implicitly binding them with a with statement.

Get comfortable with closures

- Functions can refer to variables defined in outer scopes.
- Closures can outlive the function that creates them.
- Closures internally store references to their outer variables, and can both read and update their stored variables.

FUNCTIONS THAT KEEP TRACK OF VARIABLES FROM THEIR CONTAINING SCOPES ARE KNOWN AS CLOSURES

THE THIRD AND FINAL FACT TO LEARN ABOUT CLOSURES IS THAT THEY CAN UPDATE THE VALUES OF OUTER VARIABLES. CLOSURES ACTUALLY STORE REFERENCES TO THEIR OUTER VARIABLES, RATHER THAN COPYING THEIR VALUES

Understand variable hoisting

- Variable declarations within a block are implicitly hoisted to the top of their enclosing function.
- Redeclarations of a variable are treated as a single variable.
- Consider manually hoisting local variable declarations to avoid confusion.

Use immediately invoke function expressions to create local scopes

Remember: Closures store their outer variables by reference, not by value.

```
function wrapElements(a) {
         var result = [];
         for (var i = 0, n = a.length; i < n; i++) {
             (function() {
                 var j = i;
                 result[i] = function() { return a[j]; };        })();
             }
             return result;
         }

         function wrapElements(a) {
             var result = [];
             for (var i = 0, n = a.length; i < n; i++) { (function(j) {
                 result[i] = function() { return a[j]; }; })(i);
             }
             return result;
         }
```

◦ Understand the difference between binding and assignment. Binding - allocation of slots in memory for variables by a function when the function is called. Assignment - assigning a value to a variable.

◦ Closures capture their outer variables by reference, not by value.

◦ Use immediately invoked function expressions (IIFEs) to create local scopes.

◦ Be aware of the cases where wrapping a block in an IIFE can change its behavior. (The block cannot contain any break or continue statements that jump outside of the block, since it is ille- gal to break or continue outside of a function. Second, if the block refers to this or the special arguments variable, the IIFE changes their meaning.)

Beware of unportable scoping of block local function declarations

◦ Always keep function declarations at the outermost level of a program or a containing

function to avoid unportable behavior.

- Use var declarations with conditional assignment instead of condi- tional function declarations.

Avoid creating local variables with eval

- Calling eval interprets its argument as a JavaScript program, but that program runs in the local scope of the caller.
- Avoid creating variables with eval that pollute the caller's scope.
- If eval code might create global variables, wrap the call in a nested function to prevent scope pollution.

Prefer indirect eval to direct eval

- Wrap eval in a sequence expression with a useless literal to force the use of indirect eval.
- Prefer indirect eval to direct eval whenever possible.

```
var x = "global";
     function test() {
         var x = "local";
         return eval("x"); // direct eval
     }
     test(); // "local"

     var x = "global"; function test() {
         var x = "local";
         var f = eval;
         return f("x"); // indirect eval
     }
     test(); // "global"
```

Ch. 3 - Working with functions

Understand the difference between function, method and constructor calls

```
function hello(username) { return "hello, " + username;
     }
     hello("Keyser Söze"); // "hello, Keyser Söze"


     var obj = {
     hello: function() {
         return "hello, " + this.username; },
         username: "Hans Gruber"
     };
     obj.hello(); // "hello, Hans Gruber"


     function User(name, passwordHash) {
         this.name = name;
         this.passwordHash = passwordHash;
     }

     var u = new User("sfalken", "0ef33ae791068ec64b502d6cb0191387");
     u.name; // "sfalken"
```

- Method calls provide the object in which the method property is looked up as their receiver.
- Function calls provide the global object (or undefined for strict func- tions) as their receiver. Calling methods with function call syntax is rarely useful.
- Constructors are called with new and receive a fresh object as their receiver.
- Get comfortable using higher order functions

Higher-order functions are nothing more than functions that take other functions as arguments (callbacks) or return functions as their result.

- Higher-order functions are functions that take other functions as arguments or return functions as their result.
- Familiarize yourself with higher-order functions in existing libraries.
- Learn to detect common coding patterns that can be replaced by higher-order functions.

Use apply to call functions with different number of

arguments

a variadic or variable-arity function (the arity of a function is the number of argu- ments it expects): It can take any number of arguments.

- Use the apply method to call variadic functions with a computed array of arguments.
- Use the first argument of apply to provide a receiver for variadic methods.

Use arguments to create variadic functions

- Use the implicit arguments object to implement variable-arity functions.
- Consider providing additional fixed-arity versions of the variadic functions you provide so that your consumers don't need to use the apply method.

Never modify the arguments object

- Never modify the arguments object.
- Copytheargumentsobject to a real array using [].slice.call(arguments) before modifying it.

Use a variable to save a reference to arguments

- Be aware of the function nesting level when referring to arguments.
- Bind an explicitly scoped reference to arguments in order to refer to it from nested functions.

Use bind to extract methods with a fixed receiver

- Beware that extracting a method does not bind the method's receiver to its object.
- When passing an object's method to a higher-order function, use an anonymous function to call the method on the appropriate receiver.
- Use bind as a shorthand for creating a function bound to the appro- priate receiver.

Use bind to curry functions

- Use bind to curry a function, that is, to create a delegating function with a fixed subset of the required arguments.

- Pass null or undefined as the receiver argument to curry a function that ignores its receiver.

The technique of binding a function to a subset of its arguments is known as currying, named after the logician Haskell Curry, who popularized the technique in mathematics. Currying can be a suc- cinct way to implement function delegation with less boilerplate than explicit wrapper functions.

Prefer closures to strings for encapsulating code

- Never include local references in strings when sending them to APIs that execute them with eval.
- Prefer APIs that accept functions to call rather than strings to eval.

Avoid relying on the toString method of functions

- JavaScript engines are not required to produce accurate reflections of function source code via toString.
- Never rely on precise details of function source, since different engines may produce different results from toString.
- The results of toString do not expose the values of local variables stored in a closure.
- In general, avoid using toString on functions.

Avoid non-standard stack inspection properties

- Avoid the nonstandard arguments.caller and arguments.callee, because they are not reliably portable.
- Avoid the nonstandard caller property of functions, because it does not reliably contain complete information about the stack.

In some older host environments, every arguments object came with two additional properties: arguments.callee, which refers to the function that was called with arguments, and arguments.caller, which refers to the function that called it.

Ch. 4 - Objects and Prototypes

Understand the difference between prototype,

getPrototypeOf, and *proto*

- C.prototype is used to establish the prototype of objects created by new C().
- Object.getPrototypeOf(obj) is the standard ES5 mechanism for retrieving obj's prototype object.

- obj.**proto** is a nonstandard mechanism for retrieving obj's prototype object.

- C.prototype determines the prototype of objects created by new C().
- Object.getPrototypeOf(obj) is the standard ES5 function for retrieving the prototype of an object.
- obj.**proto** is a nonstandard mechanism for retrieving the proto- type of an object.
- A class is a design pattern consisting of a constructor function and an associated prototype.

Prefer Object.getPrototypeOf to **prot**

- Prefer the standards-compliant Object.getPrototypeOf to the non- standard **proto** property.
- Implement Object.getPrototypeOf in non-ES5 environments that support **proto**.

Make your constructors new agnostic

- If a caller forgets the new keyword, then the function's receiver becomes the global object
- If the User function is defined as ES5 strict code, then the receiver defaults to undefined
- Make a constructor agnostic to its caller's syntax by reinvoking itself with new or with Object.create.
- Document clearly when a function expects to be called with new.

Store Methods on Prototypes

- Storing methods on instance objects creates multiple copies of the functions, one per

instance object.

- Prefer storing methods on prototypes over storing them on instance objects.

Use Closures to store private data

- Closure variables are private, accessible only to local references.
- Use local variables as private data to enforce information hiding within methods.

Store instance state only on instance objects

- Mutable data can be problematic when shared, and prototypes are shared between all their instances.
- Store mutable per-instance state on instance objects.

Recognize the implicit binding of this

- The scope of this is always determined by its nearest enclosing function.
- Use a local variable, usually called self, me, or that, to make a this-binding available to inner functions.

Call superclass contructors from subclass constructors

```
SpaceShip.prototype = Object.create(Actor.prototype);
```

not

```
SpaceShip.prototype = new Actor();
```

- Call the superclass constructor explicitly from subclass construc- tors, passing this as the explicit receiver.
- Use Object.create to construct the subclass prototype object to avoid calling the superclass constructor.

Never reuse superclass property names

- Be aware of all property names used by your superclasses.
- Never reuse a superclass property name in a subclass.

Avoid Inheriting from standard classes

- Inheriting from standard classes tends to break due to special internal properties such as [[Class]].
- Prefer delegating to properties instead of inheriting from standard classes.

```
"Array"
     new Array(...), [...]
     "Boolean"
     new Boolean(...)
     "Date"
     new Date(...)
     "Error"
     new Error(...), new EvalError(...), new RangeError(...), new Referen
     new TypeError(...), new URIError(...)
     "Function"
     new Function(...), function(...) {...}
     "JSON"
     JSON
     "Math"
     Math
     "Number"
     new Number(...)
     "Object"
     new Object(...), {...}, new MyClass(...)
     "RegExp"
     new RegExp(...), /.../
     "String"
     new String(...)
```

Treat prototypes as an implementation detail

- Objects are interfaces; prototypes are implementations.

- Avoid inspecting the prototype structure of objects you don’t control.
- Avoid inspecting properties that implement the internals of objects you don’t control.

Avoid reckless monkey patching

Since prototypes are shared as objects, anyone can add, remove, or modify their properties. This con- troversial practice is commonly known as monkey-patching.

- Avoid reckless monkey-patching.
- Document any monkey-patching performed by a library.
- Consider making monkey-patching optional by performing the mod- ifications in an exported function.
- Use monkey-patching to provide polyfills for missing standard APIs.

Ch. 5 - Arrays and Dictionaries

- Build lightweight dictionaries from direct instances of object

```
var dict = {};
     dict.alice = 34;
     dict.bob = 24;
     dict.chris = 62;
```

- Use object literals to construct lightweight dictionaries.

- Lightweight dictionaries should be direct descendants of Object.prototype to protect against prototype pollution in for…in loops.
- use null prototypes to prevent prototype pollution
- In ES5, use Object.create(null) to create prototype-free empty objects that are less susceptible to pollution.
- In older environments, consider using { **proto**: null }.
- But beware that **proto** is neither standard nor entirely portable and may be removed in future JavaScript environments.

- Never use the name “**proto**” as a dictionary key since some environments treat this property specially.
- Use Has own property to protect aginst prototype pollution

Every object descended from Object inherits the hasOwnProperty method. This method can be used to determine whether an object has the specified property as a direct property of that object; unlike the in operator, this method does not check down the object’s prototype chain.

```
o = new Object();
     o.prop = 'exists';

     function changeO() {
         o.newprop = o.prop;
         delete o.prop;
     }

     o.hasOwnProperty('prop');   // returns true
     changeO();
     o.hasOwnProperty('prop');
```

- Use hasOwnProperty to protect against prototype pollution.
- Use lexical scope and call to protect against overriding of the hasOwnProperty method.
- Consider implementing dictionary operations in a class that encapsulates the boilerplate hasOwnProperty tests.
- Use a dictionary class to protect against the use of “**proto**” as a key.

We can abstract out this pattern into a Dict constructor that encapsulates all of the techniques for writing robust dictionaries in a single datatype definition:

```
function Dict(elements) {
     // allow an optional initial table
     this.elements = elements || {};
     }
     // simple Object

     Dict.prototype.has = function(key) {
     // own property only
```

```
return {}.hasOwnProperty.call(this.elements, key);
};

Dict.prototype.get = function(key) { // own property only
return this.has(key)
? this.elements[key] : undefined;
};

Dict.prototype.set = function(key, val) {
this.elements[key] = val;
};

Dict.prototype.remove = function(key) { delete this.elements[key];
};
```

Prefer arrays to dictionaries for ordered collections

- Avoid relying on the order in which for…in loops enumerate object properties.
- If you aggregate data in a dictionary, make sure the aggregate operations are order-insensitive.
- Use arrays instead of dictionary objects for ordered collections.

Never add enumerable properties to Object.prototype

- Avoid adding properties to Object.prototype.
- Consider writing a function instead of an Object.prototype method.
- If you do add properties to Object.prototype, use ES5's Object.defineProperty to define them as nonenumerable properties.

Avoid modifying an object during enumeration

✦ Make sure not to modify an object while enumerating its properties with a for…in loop. ✦ Use a while loop or classic for loop instead of a for…in loop when iterating over an object whose contents might change during the loop. ✦ For predictable enumeration over a changing data structure, con- sider using a sequential data structure such as an array instead of a dictionary object.

Prefer for…in loops for array interation

- Always use a for loop rather than a for…in loop for iterating over the indexed properties of an array.
- Consider storing the length property of an array in a local variable before a loop to avoid recomputing the property lookup.

Prefer iteration methods to loops

- Use iteration methods such as Array.prototype.forEach and Array.prototype.map in place of for loops to make code more read- able and avoid duplicating loop control logic.
- Use custom iteration functions to abstract common loop patterns that are not provided by the standard library.
- Traditional loops can still be appropriate in cases where early exit is necessary; alternatively, the some and every methods can be used for early exit.

Reuse generic array methods on array-like objects

- Reuse generic Array methods on array-like objects by extracting method objects and using their call method.
- Any object can be used with generic Array methods if it has indexed properties and an appropriate length property.

Prefer array literals to the array constructor

If you call the Array constructor with a single numeric argument, it does something completely different: It attempts to create an array with no elements but whose length property is the given argument. This means that [“hello”] and new Array(“hello”) behave the same, but [17] and new Array(17) do completely different things.

- The Array constructor behaves differently if its first argument is a number.
- Use array literals instead of the Array constructor.

Ch. 6 Library and API Design

Maintain consistent conventions

- Use consistent conventions for variable names and function signatures.
- Don’t deviate from conventions your users are likely to encounter in other parts of their development platform.

Treat undefined as no value

- Avoid using undefined to represent anything other than the absence of a specific value.
- Use descriptive string values or objects with named boolean proper- ties, rather than undefined or null, to represent application-specific flags.
- Test for undefined instead of checking arguments.length to provide parameter default values.
- Never use truthiness tests for parameter default values that should allow 0, NaN, or the empty string as valid arguments.

Accept options objects for keyword arguments

- Use options objects to make APIs more readable and memorable.
- The arguments provided by an options object should all be treated as optional.
- Use an extend utility function to abstract out the logic of extracting values from options objects.

Avoid unnecessary state

- APIs are sometimes classified as either stateful or stateless. A stateless API provides functions or methods whose behavior depends only on their inputs, not on the changing state of the program.
- Prefer stateless APIs where possible.
- When providing stateful APIs, document the relevant state that each operation depends on.

Use structural typing for flexible interfaces

This sentence contains a *bold phrase* within it. This sentence contains an *underlined phrase* within it. This sentence contains an /italicized phrase/ within it.

- Use structural typing (also known as duck typing) for flexible object interfaces.

- Avoid inheritance when structural interfaces are more flexible and lightweight.
- Use mock objects, that is, alternative implementations of interfaces that provide repeatable behavior, for unit testing.

Distinguish between array and array-like

- Never overload structural types with other overlapping types.
- When overloading a structural type with other types, test for the other types first.
- Accept true arrays instead of array-like objects when overloading with other object types.
- Document whether your API accepts true arrays or array-like values.
- Use ES5's Array.isArray to test for true arrays.

Avoid excessive coercion

- Avoid mixing coercions with overloading.
- Consider defensively guarding against unexpected inputs.

Support method chaining

- Use method chaining to combine stateless operations.
- Support method chaining by designing stateless methods that produce new objects.
- Support method chaining in stateful methods by returning this.

Ch. 7 Concurrency

Dont block the event queue on I/O

Rather than blocking on the network, this API initiates the download process and then immediately returns after storing the callback in an internal registry. At some point later, when the download has com- pleted, the system calls the registered callback, passing it the text of the downloaded file as its argument.

JavaScript is sometimes described as providing a run-to-completion guarantee: Any user code that is cur- rently running in a shared context, such as a single web page in a browser, or a single running instance of a web server, is allowed to finish executing before the next event handler is invoked.

In effect, the system maintains an internal queue of events as they occur, and invokes any registered callbacks one at a time.

The single most important rule of concurrent JavaScript is never to use any blocking I/O APIs in the middle of an application's event queue. In the browser,

- Asynchronous APIs take callbacks to defer processing of expensive operations and avoid blocking the main application.
- JavaScript accepts events concurrently but processes event han- dlers sequentially using an event queue.
- Never use blocking I/O in an application's event queue.

Use nested or named callbacks for asynchronous sequencing

- Use nested or named callbacks to perform several asynchronous operations in sequence.
- Try to strike a balance between excessive nesting of callbacks and awkward naming of non-nested callbacks.
- Avoid sequencing operations that can be performed concurrently.

Be aware of dropped errors

- Avoid copying and pasting error-handling code by writing shared error-handling functions.
- Make sure to handle all error conditions explicitly to avoid dropped errors.

Use recursion for asynchronous loops

- Loops cannot be asynchronous.
- Use recursive functions to perform iterations in separate turns of the event loop.
- Recursion performed in separate turns of the event loop does not overflow the call stack.

When a program is in the middle of too many function calls, it can run out of stack space, resulting in a thrown exception. This condition is known as stack overflow.

Dont block the event queue on computation

- Avoid expensive algorithms in the main event queue.
- On platforms that support it, the Worker API can be used for running long computations in a separate event queue.
- When the Worker API is not available or is too costly, consider breaking up computations across multiple turns of the event loop.

Use a counter to perform concurrent operations

- Events in a JavaScript application occur nondeterministically, that is, in unpredictable order.
- Use a counter to avoid data races in concurrent operations.

Never call asynchronous callbacks synchronously

- Never call an asynchronous callback synchronously, even if the data is immediately available.
- Calling an asynchronous callback synchronously disrupts the expected sequence of operations and can lead to unexpected interleaving of code.
- Calling an asynchronous callback synchronously can lead to stack overflows or mishandled exceptions.
- Use an asynchronous API such as setTimeout to schedule an asynchronous callback to run in another turn.

Use promises for cleaner asynchronous logic

One way to think about a promise is as an object that represents an eventual value—it wraps a concurrent operation that may not have completed yet, but will eventually produce a result value. The then method allows us to take one promise object that represents one type of eventual value and generate a new promise object that represents another type of eventual value—whatever we return from the callback.

- Promises represent eventual values, that is, concurrent computations that eventually produce a result.
- Use promises to compose different concurrent operations.
- Use promise APIs to avoid data races.
- Use select (also known as choose) for situations where an inten- tional race condition is

required.

JS Design Patterns

JavaScript Design Patterns

Constructor Pattern

1. Literal

2. Object.create(Object.prototype)

3. new

Property Assignment

1. .

2. [“”]

3. defineProperty

4. defineProperties

Inheritance -

```
var truck = Object.create(auto);
```

Constructor -

```
function *Object*(params) {
       this.params = params;

       this.method = function () {};
     }

     Constructor with Prototype -
     *Object*.prototype.method = function () {};
```

Module Pattern -

Options

1. Object literal notation
2. Standard module pattern
3. AMD modules
4. CommonJS modules
5. ECMAScript Harmony Modules

Object Literals

Creates module pattern by encapsulating and organizing data

Module Pattern -

(See notes in above) Encapsulates privacy, state and organization using closures Similar to IIFE except object is returned instead of function

```
(IIFE's -
     (function(){ /* code */ }()); // Crockford recommends this one

     (function(){ /* code */ })(); // But this one works just as well

     with a function as the return value
     )

     var module = (function () {
         var x;
         return {
             method: function (foo) {
                 x = foo;
                 return x;
             }
         }
     })();
```

Simple Module Template

```
var myNamespace = (function () {

          var myPrivateVar, myPrivateMethod;

          // A private counter variable
          myPrivateVar = 0;

          // A private function which logs any arguments
          myPrivateMethod = function( foo ) {
              console.log( foo );
          };

          return {

              // A public variable
              myPublicVar: "foo",

              // A public function utilizing privates
              myPublicFunction: function( bar ) {

                  // Increment our private counter
                  myPrivateVar++;

                  // Call our private method using bar
                  myPrivateMethod( bar );

              }
          };

      })();
```

Imports - pass variables (global or outside scope) into function as arguments and parameters

Exports - create object variable and explicitly set properties/methods and return object

Frameworks with support for modules - Dojo (with AMD compatible version), ExtJS, YUI, jQuery

Advantages of vanilla modules

cleaner (control over variables in global namespace), private data (code can touch private data but outside world cannot) Disadvantages - If changes to visibility are made, each reference to the data must be changed, cannot access private members in methods that are added later, no automated unit tests for private members

Revealing Module Pattern

```
var myRevealingModule = (function () {
          var privateVar = "Ben Cherry",
          publicVar  = "Hey there!";
          function privateFunction() {
              console.log( "Name:" + privateVar );
          }
          function publicSetName( strName ) {
              privateVar = strName;
          }
          function publicGetName() {
              privateFunction();
          }
          // Reveal public pointers to
          // private functions and properties
          return {
              setName: publicSetName,
              greeting: publicVar,
              getName: publicGetName
          };
          })();

          var myRevealingModule = (function () {
              var privateCounter = 0;
              function privateFunction() {
                  privateCounter++;
              }
              function publicFunction() {
                  publicIncrement();
              }
              function publicIncrement() {
                  privateFunction();
```

```
    }
    function publicGetCount(){
        return privateCounter;
    }
    // Reveal public pointers to
    // private functions and properties
    return {
        start: publicFunction,
        increment: publicIncrement,
        count: publicGetCount
    };
    })();
```

Makes clear what can be accessed, but may be more fragile

Singleton Pattern

Class that is only allowed to instantiate one single object, only returns a reference to the original afterwards.

- There must be exactly one instance of a class, and it must be accessible to clients from a well-known access point.
- When the sole instance should be extensible by subclassing, and clients should be able to use an extended instance without modifying their code.
- Singleton defers execution by isolating the unpredictability of dynamic initialization, returning control to the programmer
- Singleton and static objects/classes are distinct
- The singleton is useful when exactly one object is needed to coordinate others across a system.
- When needed in JavaScript it usually means that the code base needs to be refactored or the larger design needs to be reconsidered

```
var mySingleton = (function () {

        // Instance stores a reference to the Singleton
        var instance;
```

```
function init() {

    // Singleton

    // Private methods and variables
    function privateMethod(){
        console.log( "I am private" );
    }

    var privateVariable = "Im also private";

    var privateRandomNumber = Math.random();

    return {

        // Public methods and variables
        publicMethod: function () {
            console.log( "The public can see me!" );
            },

            publicProperty: "I am also public",

            getRandomNumber: function() {
                return privateRandomNumber;
            }

        };

    };

    return {

        // Get the Singleton instance if one exists
        // or create one if it doesn't
        getInstance: function () {

            if ( !instance ) {
                instance = init();
            }

            return instance;
        }

    };
```

```
        })();
```

Observer Pattern

- Subject (type of object) notifies a list of observers of any changes of state
- Observer attaches to subject, registering (updating) state change upon receiving a notify message from the subject, detaching when necessary

```
function ObserverList(){
        this.observerList = [];
    }

    ObserverList.prototype.add = function( obj ){
        return this.observerList.push( obj );
    };

    ObserverList.prototype.count = function(){
        return this.observerList.length;
    };

    ObserverList.prototype.get = function( index ){
        if( index > -1 && index < this.observerList.length ){
            return this.observerList[ index ];
        }
    };

    ObserverList.prototype.indexOf = function( obj, startIndex ){
        var i = startIndex;

        while( i < this.observerList.length ){
            if( this.observerList[i] === obj ){
                return i;
            }
            i++;
        }

        return -1;
    };

    ObserverList.prototype.removeAt = function( index ){
```

```
            this.observerList.splice( index, 1 );
        };

        function Subject(){
            this.observers = new ObserverList();
        }


        Subject.prototype.addObserver = function( observer ){
            this.observers.add( observer );
        };

        Subject.prototype.removeObserver = function( observer ){
            this.observers.removeAt( this.observers.indexOf( observer, 0 ) )
        };

        Subject.prototype.notify = function( context ){
            var observerCount = this.observers.count();
            for(var i=0; i < observerCount; i++){
                this.observers.get(i).update( context );
            }
        };
```

Observer vs. Publish/Subscribe

- Pub/sub implements a topic/event channel (event aggregator) to sit between subject (publisher) and object (subscriber or observer)
- publish(),subscribe() and unsubscribe() follow a similar line of thought to event listeners
- Advantages - Observer good to maintain consistency between related objects, both are good for creating decoupled systems
- Disadvantages - with pub/sub it creates a possibility of issues with the subscriber that the publisher does not see due to decoupling, subscribers do not see each other

Publish

```
// jQuery: $(obj).trigger("channel", [arg1, arg2, arg3]);
     $( el ).trigger( "/login", [{username:"test", userData:"test"}] );

     // Dojo: dojo.publish("channel", [arg1, arg2, arg3] );
```

```
    dojo.publish( "/login", [{username:"test", userData:"test"}] );

    // YUI: el.publish("channel", [arg1, arg2, arg3]);
    el.publish( "/login", {username:"test", userData:"test"} );
```

Vanilla JS implementations - Amplify JS, Radio.js, PubSubJS, or Pure JS PubSub

```
var pubsub = {};

    (function(myObject) {

        // Storage for topics that can be broadcast
        // or listened to
        var topics = {};

        // An topic identifier
        var subUid = -1;

        // Publish or broadcast events of interest
        // with a specific topic name and arguments
        // such as the data to pass along
        myObject.publish = function( topic, args ) {

            if ( !topics[topic] ) {
                return false;
            }

            var subscribers = topics[topic],
            len = subscribers ? subscribers.length : 0;

            while (len--) {
                subscribers[len].func( topic, args );
            }

            return this;
        };

        // Subscribe to events of interest
        // with a specific topic name and a
        // callback function, to be executed
```

```
        // when the topic/event is observed
        myObject.subscribe = function( topic, func ) {

            if (!topics[topic]) {
                topics[topic] = [];
            }

            var token = ( ++subUid ).toString();
            topics[topic].push({
                token: token,
                func: func
                });
                return token;
            };

            // Unsubscribe from a specific
            // topic, based on a tokenized reference
            // to the subscription
            myObject.unsubscribe = function( token ) {
                for ( var m in topics ) {
                    if ( topics[m] ) {
                        for ( var i = 0, j = topics[m].length; i < j; i++
                            if ( topics[m][i].token === token ) {
                                topics[m].splice( i, 1 );
                                return token;
                            }
                        }
                    }
                }
                return this;
            };
        }( pubsub ));
```

Mediator Pattern

- Similar to the topic/event channel in pub/sub but different due to semantics
- Mediator coordinates interactions (logic & behavior) between multiple objects
- Event aggregator always deals with events, whereas mediator does not have to deal with events
- Event aggregator - unknown number of sources to unknown number of handlers, all logic

and workflow of the aggregator is handled inside of the objects that both trigger and handle the events

- The mediator handles the business logic and workflow itself, and knows when an object should have a method called or state changed
- Event aggregator is “fire and forget”

When to Use

- Event Aggregator
- Large number of objects listening for one event that could detoriate performance
- Indirect relationships that are dependent on eachother, ie view objects that have dependent activity but are not directly linked

Mediator

- When two or more objects have an indeirect working relationship

Event Aggregator and Mediator together

- Mediator handles many objects that related to eachother but unrelated to source whereas many of the changes that are unrelated are better served with an event aggregator

Advantages & Disadvatnages

- Mediator - reduces communication channels needed in a system to one, may cause performance inssues. Decoupled systems have the benefit of avoiding single changes (object throwing error) causing a domino effect through the whole system

The Prototype Pattern

- Can just be simplified in JavaScript to prototypal inheritance, objects receiving properties and methods from other objects.
- Objects are created based on a template of an existing object through cloning

Benefits

- Natively implemented in JavaScript
- Increases performance because methods reference the original instead of creating their

own copy

- ECMAScript 5 defines real prototypal inheritance through Object.create
- Prototypal inheritance without object.create

```
var vehiclePrototype = {

          init: function ( carModel ) {
              this.model = carModel;
          },

          getModel: function () {
              console.log( "The model of this vehicle is.." + this.model);
          }
      };


      function vehicle( model ) {

          function F() {};
          F.prototype = vehiclePrototype;

          var f = new F();

          f.init( model );
          return f;

      }

      var car = vehicle( "Ford Escort" );
      car.getModel();
```

The Command Pattern

- Encapsulates method invocation, requests or operations into a single object, allowing the actions to be decoupled from the object, and alllowing us to swap out concrete classes
- Abstract class - defines an interface but doesn't necessarily provide the implementation
- Concrete class - expands on an abstract classes interface to provide the implementation

```
CommandObject.execute("method", "receiving object", "data"); // for insta
```

The Facade Pattern

- An object that provides a convenient higher leve interface to abstract out some of the underlying complexity
- Often seen in libraries like jQuery
- Example - creating methods or functions that check for cross-browser features and performs an action based on the presence of certain features

```
jQuery's $(document).ready()

     bindReady: function() {
         ...
         if ( document.addEventListener ) {
             // Use the handy event callback
             document.addEventListener( "DOMContentLoaded", DOMContentLoa

             // A fallback to window.onload, that will always work
             window.addEventListener( "load", jQuery.ready, false );

             // If IE event model is used
             } else if ( document.attachEvent ) {

                 document.attachEvent( "onreadystatechange", DOMContentLo

                 // A fallback to window.onload, that will always work
                 window.attachEvent( "onload", jQuery.ready );
```

Can combine with module pattern to present a simplified interface to a module

```
var module = (function() {

         var _private = {
             i:5,
             get : function() {
```

```
            console.log( "current value:" + this.i);
            },
            set : function( val ) {
                this.i = val;
                },
                run : function() {
                    console.log( "running" );
                    },
                    jump: function(){
                        console.log( "jumping" );
                    }
                };

                return {

                    facade : function( args ) {
                        _private.set(args.val);
                        _private.get();
                        if ( args.run ) {
                            _private.run();
                        }
                    }
                };
            }());


// Outputs: "current value: 10" and "running"
module.facade( {run: true, val:10} );
```

Facades general have few disadvantages, but may hurt performance due to abstraction. getElementById is much faster than $().

The Factory Pattern

◦ Similar to other creational objects, but instead of calling new explicitly to create an object, we call the Factory object directly, asking for it to create some sort of sub-object, which it instantiates and returns

// Types.js - Constructors used behind the scenes

// A constructor for defining new cars function Car(options) {

```
    // some defaults
         this.doors = options.doors || 4;
         this.state = options.state || "brand new";
         this.color = options.color || "silver";
```

}

// A constructor for defining new trucks function Truck(options){

```
    this.state = options.state || "used";
         this.wheelSize = options.wheelSize || "large";
         this.color = options.color || "blue";
```

}

// FactoryExample.js

// Define a skeleton vehicle factory function VehicleFactory() {}

// Define the prototypes and utilities for this factory

// Our default vehicleClass is Car VehicleFactory.prototype.vehicleClass = Car;

// Our Factory method for creating new Vehicle instances
VehicleFactory.prototype.createVehicle = function (options) {

```
    switch(options.vehicleType){
             case "car":
             this.vehicleClass = Car;
             break;
             case "truck":
             this.vehicleClass = Truck;
             break;
             //defaults to VehicleFactory.prototype.vehicleClass (Car)
         }
```

```
        return new this.vehicleClass( options );
```

};

// Create an instance of our factory that makes cars var carFactory = new VehicleFactory(); var car = carFactory.createVehicle({ vehicleType: “car”, color: “yellow”, doors: 6 });

```
// Test to confirm our car was created using the vehicleClass/prototype C

     // Outputs: true
     console.log( car instanceof Car );

     // Outputs: Car object of color "yellow", doors: 6 in a "brand new"
     console.log( car );
```

Use the factory pattern:

- Setup involves complexity
- Generate different instances depending on the environment
- Working with many objects that share the same or similar properties
- When composing instances that only need to satisfy an API contract

Don’t use when:

- Generally stick to explicit constructors to avoid overhead. Can also introduce problems with unit testing

Abstract Factories

- Similar to factory but allows for an interface to define types of sub-objects whereas factory only implements classes that fulfill the sub-object contract

The Mixin Pattern

- Mixins are classes which offer functionality for sub-classes to access function re-use

- Base object provides form to sub-object, and so on

```
var Person =  function( firstName , lastName ){

        this.firstName = firstName;
        this.lastName =  lastName;
        this.gender = "male";

      };

      // a new instance of Person can then easily be created as follows:
      var clark = new Person( "Clark" , "Kent" );

      // Define a subclass constructor for for "Superhero":
      var Superhero = function( firstName, lastName , powers ){

          // Invoke the superclass constructor on the new object
          // then use .call() to invoke the constructor as a method of
          // the object to be initialized.

          Person.call( this, firstName, lastName );

          // Finally, store their powers, a new array of traits not found
          this.powers = powers;
      };

      Superhero.prototype = Object.create( Person.prototype );
      var superman = new Superhero( "Clark" ,"Kent" , ["flight","heat-visi
      console.log( superman );

      // Outputs Person attributes as well as powers
```

- In JavaScript, mixins are equivalent to extension

- Can use Underscore.js helper _extend() method

Advantages and Disadvantages

- Decrease functional repetition and increase function re-use, keep shared functionality in a mixin

- Extending prototypes can lead to prototype pollution and questionable code origins

Decorator Pattern

- Extend a class to extend object functionality

```
var truck = new Vehicle( "truck" );

      // New functionality we're decorating vehicle with
      truck.setModel = function( modelName ){
          this.model = modelName;
      };

      truck.setColor = function( color ){
          this.color = color;
      };
```

Another example is to override functions of the super class

Interfaces

- An interface is a way of defining the methods an object should have but not necessarily defining how they are implemented