Silicon Valley Code Camp

# Amazing New Features In JavaScript

Manoj Kumar

## Agenda

- Scope of Variables
- Parameters: default/rest/spread
- Destructuring
- Object Literals
- Arrow functions
- Iterators & Generators
- Promises & Proxies

# Scope of a Variable (ES5)

```
var x = 1   // global variable, a property in
   global obj
b = 2        // global variable, a property in
   global obj
function f() {
   a = b + 6  // “a” is a global variable, what
   about b?
   var b = 7; // b is local to this function
            // declaration is hoisted but not the
   assignment
}
a = 5
```

# Scope of a Variable (ES 6)

```
function f() {
    var a = 7;   // local to function f
    let c = 8;   // local to function f

    If ( a > 5) {
        var b = 9 + c;  // Visible anywhere in the
function
        let x = 1;   // local to this IF block
        const pi = 3.14; // constant and local to
IF block
                // redefining is not allowed
        const obj = { x:9, y:10 }; // obj.x is still
```

# const

```
const pi1;    // error: Missing initialization

const pi2 = 3.14159 ; // works!
pi2 = 3.14 ; // error: can't change the value

for( let i = 5;....) {
   const c = i * 9; // const can get a fresh
   value inside loop in different iteration
```

# Temporal Dead Zone

```
Let x = 6;
If ( true ) { // new block scope so TDZ starts
     console.log(x) ; // reference error (outer x
    not visible)
      x = 5;  // reference error
     let x;    // now x can be referenced
     console.log(x); // undefined

      x = 6;
      console.log(x); // 6
}
```

# Temporal => Time

```
If ( true ) { // new block scope so TDZ starts
     const   func1 = function () {
                                  Console.log(x);
                               };

    console.log(x) ; // still inside TDZ, so
    reference error

    let x = 8;
    func1(); // inside this call “let x” is effective
}
```

## Default Parameter Value

```
function f(x = 0, y = 0) {
}
f(); // 0,0
f(5); // 5,0
f(5, someVarUndefined); // 5,0
f(undefined, 6); // 0, 6
f(5,6); // 5,6
Function(x, y=x) { }    //OK
Function (x=y, y=5) { } // y is not available to
   use (TDZ)
```

# Rest of the Parameters

```
function func(x, y, ...others) {
     //others[0..n] contains rest of the
   parameters
}

func(5,6); // 5,6, others.length==0
func(5,6, 7); // 5,6, others.length==1,
   others[0] === 7
Func(5,6, 7,8,9); // 5,6,
     others.length==3,
     others[0..2] are 7,8,9
```

# Spread Operator

```
Math.max(2, 5, 6, 8);

Math.max( ...[2, 5, 6, 8]  )
    Same as
    Math.max.apply(null, [2,5,6,8])

[1, ...[2,3], 4]
=>
[1, 2, 3, 4]
```

# Destructuring

```
let [x, y] = [5, 8];      // x === 5, y ===8

var [a, , [b], c] = [5, null, [6]];
// a === 5 && b === 6 && c === undefined

[a, b] = [b, a]

var [a, b, c] = "xy";  // think “xy” as [“x”, “y”]
// a === "x" && b === "y" && c === undefined
```

# Destructuring

{ first: f, last: l } = { first: 'Manoj', last: 'Kumar' }

// f === 'Manoj' && l === 'Kumar'

let { x: x } = { x: 1, y: 2 }; // x = 1

let [ x, ...y ] = 'abc';  // x = 'a'; y = [ 'b', 'c' ]

let [ x, y ] = new Set( [ 'a', 'b' ] ); // x = 'a'; y = 'b';

[ ] = { }; // TypeError, empty objects are not iterable

[ ] = undefined; // TypeError, not iterable

[ ] = null; // TypeError, not iterable

let [x] = [ ]; // x = undefined

let {x:y} = { } ; // y = undefined because { }

# Object Literals

```
obj = {
  red  : 255, blue : 127, [ “green” ] : 255
};
red = 255;
blue = 127;
green = 255;
obj = {
  red  : red, blue : blue, green: green
};
obj = { red, blue, green  }          // ES 6
```

# Computed Property

```
let prop = "red";
green = 255;

obj = {
  [ prop ]  : 255,                // [“red”] : 255 or red
   : 255
  [ "b" + "lue" ] : 127,
  green
};
```

<eot/>

# Arrow Functions

- New way of creating functions
- function square( x ) {   // ES 5 function
-       return x * x;
- }

```
x => x * x ;          // Arrow function
 (x, y) => x + y ;   ( ) => 5;
(x, y) =>  { f(x,y); return x + y; }
 x => { a:x+1, b:x+2}; // wrong!
 { means block
 x => ({ a:x+1, b:x+2 });
```

No line break before =>

# Arrow Functions vs Normal Functions

1. Following constructs are lexical
    - Arguments
    - this
    - super
    - new.target  (target object of new, null in normal functions)
2. Cannot be used as constructor

   `new ( () => { } )` throws an error

# Symbol

```
obj.red = 255;
obj[“red”] === 255;  // ES5
const my_prop = Symbol();   // ES6
obj[my_prop] = 127; // ES6
Obj = {
   [my_prop] : 127
};
const red = Symbol('my_red')
red.toString() === 'Symbol(my_red)'
Symbol()  !=  Symbol()
Symbol( 'red' )   !=  Symbol( 'red' )
```

## Symbol (Global Registry)

```
const globalRedProp = Symbol.for( 'red');
globalRedProp.toString() === 'Symbol(red)';
Symbol.for ( 'red' ) === globalRedProp
Symbol.keyFor( globalRedProp ) === 'red'
Symbol.for( 'red' ) ===  Symbol.for( 'red' )
Symbol( 'red' )    !=  Symbol( 'red' )
```

Built-in symbols:

- Symbol.iterator
- Symbol.match (=== String.prototype.match )

<eot/>

# Iterators

Iterable object makes its element accessible in for-of  loops and spread operators

```
for (let x of ['a', 'b', 'c']) {     // arrays are
  iterable
    console.log(x);
}
```

Iterable Objects:

- Arrays
- Strings
- Maps
- Sets
- DOM (not ready yet)

# Iterable

How to make any object iterable?

- Implement a method Symbol.iterator
- That returns Iterator object

```
IterableObject = {
    [Symbol.iterator] ( ) {
                // create an iterator object and
   return
         }
}
```

# Iterator

```
Iteartor = {
  next () {
     // keep returning IteratorResult in
  successive calls
  }
}
IteratorResult = {
  value :   // current value
  done:     // false, but at the end true
}
```

```
let iterable = {
    [Symbol.iterator]() {
        let step = 0;
        let iterator = {
            next() {
                if (step <= 2) step++;
                switch (step) {
                    case 1: return { value: 'hello', done:
    false };
                    case 2: return { value: 'world', done:
    false };
                    default: return { value: undefined,
    done: true };
                }
            }
        };
        return iterator;
    }
```

# Iterable

```
Let iterator = iterable [ Symbol.iterator ] ( ) ;
Iterator.next() === { value : 'hello', done:
   false }
Iterator.next() === { value : 'world', done:
   false }
Iterator.next() === { value : undefined, done:
   true }

While (true) {
   let result = iterator.next();
   if ( result.done ) break;
```

# Iterable in for-of

```
for (let x of iterable) {
    console.log(x);
}
for ( let [k,v] of map)
for ( let x of Array.from(arrayLike))
   //length, 0:, 1:..
for ( let [k,v] of array.entries())
```

# Generator

```
function * simpleGenFunc () {
     yield 'hello';
     yield 'world';
}
Iterator = simpleGenFunc ();
Iterator.next ( ) ; // { value: “hello”, done:
    false }
Iterator.next ( );  // { value: “world”, done:
    false }
Iterator.next ( );  // { value: undefined, done:
    true }
for ( x of simpleGenFunc () ) {
```

## Generator

```
let arr = [ ...simpleGenFunc ( ) ]; // [ 'hello',
    'world']

let [ x, y ] = simpleGenFunc ( );
x === 'hello'
y === 'world'
```

## Generator

Generators are

- Data Producers (as iterator)
- Data Consumer (Yield can get data from next())
- Communicating sequential tasks..

```
function * genFunc () {
    try {
        let hint = yield 'hello';
        // do something with hint
        yield 'world';
    } finally {
         // Do some cleanup here
    }
}
Iterator = genFunc();
Iterator.next();   // gives hello
Iterator.next("stop"); // gives 'world', hint ===
   "stop"
Iterator.next(); // cleanup code executes,
```

## Promise

```
setTimeout(
    function() { console.log("timeout!");},
    delay );

promise = timeout(delay) ;
promise.then(
    function(result) { console.log("Result: " +
   result);}
);
```

# Promise

```
function timeout(delay) {
    return new Promise(
        function(resolve, reject) {
            setTimeout(
                function() {
                    resolve();
                }, delay);
        });
    }
promise = timeout(delay) ;
promise.then( function(result) {....} );
```

# Promise

```
let promise = someAsyncOp() ;

promise.then ( function (result) {
        console.log(“Result: “ + result);
    }, function (err) {
        console.log(“Failed: “ + err);
    }
)
```

# Promise

```
promise.then ( null, function (err) {
        console.log(“Failed: “ + err);
   });

promise.catch(function (err) {
        console.log(“Failed: “ + err);
   } );
```

# Promise

- Life Cycle
  - Unsettled Promise
    - State: PENDING
  - Settled Promise
    - State: FULFILLED
    - State: REJECTED
  -
  - promise.then ( fulfilledCaseCallback, rejectedCaseCallBack);
  - promise.then(fulfilledCaseCallback);
  - promse.catch(rejectedCaseCallback);

## Promise
### Chaining

```
promise.then ( fulfillmentFunction1 )
            .then ( fulfillmentFunction2 );


promise2 = promise.then
  ( fulfillmentFunction1 )
promise2.then ( fulfillmentFunction2 );
```

## Promise

```
Promise.all([promise1, promise2])
    .then(function(results)  {
        // Both resolved
     })
    .catch(function(error) {
        // One rejected
     });
```

# Promise

```
Promise.race([promise1, promise2])
    .then(function(results)  {
        // one got resolved
     })
    .catch(function(error) {
        // First settlement was in rejection
     });
```

<eot/>

# Proxy

- Let proxy = new Proxy( target, handler );
- Handler: Object with one or more traps
- Traps:
  - Get
  - Set
  - Has
  - deleteProperty
  - defineProperty

## Proxy

More Traps:

- isExtensible
- preventExtensions
- getPrototypeOf
- setPrototypeOf
- ownKeys
- apply  (calling a function)
- Construct (using new)
-
-

```
let target = { name: "smartObject" };

let proxy = new Proxy(target, {
 set(trapTarget, key, value, receiver) {
    if (isNaN(value)) {
      throw new TypeError("Property must be a number.");
    }
    return Reflect.set(trapTarget, key, value, receiver);
   }
});

proxy.count = 1;  // numeric properties are okay
proxy.color = "red";  // throws exception
<eot/>
```

# References

- ecmascript 6 compatibility table
  - https://kangax.github.io/compat-table/es6/
- http://exploringjs.com/es6/index.html
  - Thanks to the author for tons of examples!
- leanpub.com/understandinges6
- JavaScript: The Good Parts
  - Douglas Crockford
- Effective JavaScript

# Thank You!

Please provide your feedback :)

Feedback from earlier sessions:

Speaker started with the concepts that were way too simple and then at the end it became way too complex.

## Slides from Older Presentations

- You may not find things in sequence in rest of the slides

# ES5 Equiv of Class

```
function Cube(size) {
    this.size = size;
}
Cube.prototype.rotate = function (direction) {
    // rotate the cube;
};
Let simpleCube = new Cube(3);

simpleCube.size === 3
simpleCube.rorate(“left”);
```

# ES 6 Class

```
class Cube {
   constructor (size) {
       this.size = size;
   }
   rotate(direction) {
       // rotate the cube
  }
}
let simpleCube = new Cube(3);
simpleCube.size === 3
simpleCube.rorate(“left”);
```

# Derived Class

```
class RubiksCube extends Cube {
   constructor (size, colors) {
       super(size);
       this.colors = colors;
   }
   solve() {
       // solve the cube
  }
}
```

# Derived Class

```
class RubiksCube extends Cube {
   constructor (size, colors) {
       super(size);
       this.colors = colors;
   }
   solve() {
       // solve the cube
  }
}
```

```
class BinaryTree {
    constructor(value, left=null, right=null) {
        this.value = value;
        this.left = left;
        this.right = right;
    }

    * [ Symbol.iterator ] ( ) {
        yield this.value;
        If ( this.left ) {
            yield* this.left;
        }
        If ( this.right ) {
            yield* this.right;
        }
    }
}
```

```
let tree = new BinaryTree(
                 'a',
                  new BinaryTree(
                       'b',
                       new BinaryTree('c'),
                       new BinaryTree('d')),
                  new BinaryTree('e'));

for (let x of tree) {
    console.log(x);
}
// Output:
// a
// b
// c
// d
// e
```

## Module

A JavaScript file is a module
One module is only one JavaScript file

Export entities in the module where declared
Import entities from a module in a module

## Module

```
//------ lib.js ------
export function square (x) {
    return x * x;
}
export function diag (x, y) {
    return sqrt(square(x) + square(y));
}
export const sqrt = Math.sqrt;
//------ main.js ------
import { square, diag } from 'lib';
console.log(square(11)); // 121
console.log(diag(4, 3)); // 5
```

# Module

```
import * as mylib from 'lib';
console.log ( mylib.square(11) ); // 121
console.log ( mylib.diag(4, 3) ); // 5
```

Imports are hoisted

Cyclic dependency is supported

Re-export some imported entities

- Export * from lib
- Export { square as num_square, diag } from lib

## Scope

- ES5
  - Function Scope
  - Global Scope
  - var keyword
- ES6
  - Block scope
  - let and const keywords

## Scope of a Variable (ES 6)

•

## New Features (1)

- Arrow Function
- Classes
- Enhanced object literals
- Template strings
- Destructuring
- Default, rest, spread
- Let, const
- Iterators, for..of
- Generators

## New Features (2)

Unicode
Modules
Module loaders
Map, set, weakmap, weakset
Proxies
Symbols
Subclassable built-ins
Promises

## New Features (3)

- Math, number, string, array, object APIs
- Binary and octal literals
- Reflect api
- Tail call optimization
-

# Silicon Valley Code Camp 2014

# Learn JavaScript by Modeling Rubik's Cube

Manoj Kumar

## Agenda

- Scripting Language
- Crash course in Rubik’s Cube
- Code Walk-through
- Modeling
- Finding moves

## Scripting Language

- Is a Programming Language
  - To manipulate
  - To customize
  - To automate
  - an existing system
- ECMAScript
  - Web Scripting Language
  - To work with web browser

# ECMA Script

- Object Based
  - Object: Collection of properties
    - Property
      - Type : Number, Boolean, String, Array, Function & other objects
      - Attributes
- Value, Writable, Configurable, Enumerable
- Functional
- Based on
  - Java, C
  - Self (Prototype)
  - Scheme (Functional)

## Types

- Primitive Value Types
  - Number
  - String
  - Boolean
  - Null
  - Undefined
- Object
- Array
- Function

## Number

- 64 bit floating point (sign bit, 11 exp, 52 frac)
- Represents integer and float
  - 1, 3.45, 5.345e-10, 0377, 0xFF,
- Infinity
  - >1.7976931348623157e+308
- NaN
  - NaN != NaN
- Representation for
  - MAX_VALUE, MIN_VALUE
  - NEGATIVE_INFINITY, POSITIVE_INFINITY
- +0 == -0  but 1/+0 != 1/-0

## String

- Within double/single quotes
  - “Hello world”
  - ‘\u0041 world’
- Sequence of 16 bit unicode chars
- Supports + operator
- Used for character type too
-
-

## Boolean

- Only two values
  - true
  - false
- 6 more falsy values
  - 0, -0, “”, NaN, null, undefined
- Rest all values are true
  - Including ‘false’ :)
-

## Undefined and Null

- Undefined Type
  - Only one value: undefined
  -
- Null Type
  - Only one value: null
-

# Binary Operators

- Binary + (Addition or concatenation)
  - 1 + 2 = 3,
  - ‘1’ + ‘2’ = ‘1’ + 2 = **1 + ‘2’ = ‘12’**
- -, * , /, %
  - **2 * ‘3’ = 6**
- >=, <=, >, <
- ==, !=
- **=== !==**
- Logical &&, ||

# Prefix Operators

- + to number
  - 1 + +'2' // 3
- - negate
- ! logical not
- Typeof
  - typeof 1 // ‘number’
  - typeof ‘a’ // ‘string’
-
-

## Bit Operators

| | |
|---|---|
| & | and |
| \| | or |
| ^ | xor |
| ~ | not |
| >> | signed right shift |
| >>> | unsigned right shift |
| << | left shift |

## And more

- = assignment
- +=, -=, *=, /= %=
  - X op= y ==> x = x op y
- ++ increment
  - X++ ==> x = x+1
- -- decrement
  - X-- ==> x = x-1

## A Simple Object

```
point = { x : 100, y : 200 };
point.x   // 100
point['x']  // 100
point.y = 300;

ap = { "x-coord" : 100, "y-coord" : 200 };
ap.x-coord    // Error, - means subtraction
ap["x-coord"] // 100
ap["X-coord"]  // undefined, (note the upper
case X)
```

## Arrays

**var x = [5, 3];**

**x.length** => 2

**x.push(7)** ===> add a new element

**x[20] = 9** ===> 3 to 19 elements are empty

**delete x[1]**  == remove one element

**x.splice**(beginIndex, noOfElemToRemove)

typeof x ==> ‘object’  .. not a good design

x.constructor == Array

concat, join, pop, reverse, slice, shift, sort

## Functions

```
function add(x, y) {
    return x+y
}
sum = add(4,5)
myAdd = add
sum1 = myAdd(4, 5)
```

## Function Var Assignment

```
myAdd = function add(x, y) {
    return x+y
}
sum1 = myAdd(4, 5)
sum2 = add(4, 5)
ReferenceError: add is not defined
```

## Function Var Assignment

```
myAdd = function (x, y) {
    return x+y
}
sum1 = myAdd(4, 5)
```

# Anonymous Function

```
sum = function (x, y) {
                return x+y
            } (4,5)
sum = (function (x, y) {
                return x+y
            }) (4,5)
```

## Arguments

```
function add(  ) {
    var sum = 0
    for( var i = 0; i < arguments.length; i++) {
        sum += arguments[i]
    }
    return sum
}
add(4, 5)  => 9
add(4,5,3) => 12
add() => 0
```

## Functional Programming

- Function as a first class object
  - Assign to variables
  - Pass to other functions
- Avoid State change, mutability, side effects
- Prefer recursion over loop
- Higher Order Functions
  - ForEach (function, collection)
  - Map (function, collection)
  - Filter (function, collection)
  - Reduce (function, accumulator, collections)
  - Curry (function)

## Code Walkthrough

Model Rubik's Cube
with
Arrays and Functions

## Scope of a Variable

```
function f() {
    a = 6  // “a” is a global variable
}

a = 5

f()
// a is 6 now
```

## Scope of a Variable

```
function f() {
    var a = 6   // “a” is a local variable
    alert("After assignment : a = " + a)
}
a = 5
alert("Before Calling function: a = " + a)
f()
alert("After Calling function: a = " + a)
```

# Scope of a Variable

```
function f() {
   a = 6
   ....
   var a = 7    // makes “a” a local
   variable!
    // declaration is hoisted but not the
   initializer!
}
a = 5

f()
```

# Scope of a Variable (ES 6)

```
function f() {
   var a = 7;   // local to function f

   If ( a > 5) {
      var b = 9;  // Visible anywhere in the
   function
      let c = 1;   // local to this IF block
      const pi = 3.14; // constant and local to
   IF block
   }
}
```

# loop variable with var

```
// Objective
// funcs [ 0 ] = function( ) { return 0; } ;
// funcs [ 1 ] = function( ) { return 1; } ;
// funcs [ 2 ] = function( ) { return 2; } ;
let funcs = [  ];
for (var i=0; i < 3; i++) {
    funcs.push( function() { return i;} );
}
funcs[0]() ; // 3
funcs[1]() ; // 3
funcs[2]() ; // 3
```

## loop variable with var

```
Funcs = [  ];
functionCreator(n) {
        return function() { return n;}
      }
}
for (var i=0; i < 3; i++) {
   funcs.push( functionCreator(i));
}
funcs[0]() ; // 0
funcs[1]() ; // 1
funcs[2]() ; // 2
```

## loop variable with var

```
for (var i=0; i < 3; i++) {
    funcs.push(
       function(n) {
         return function() { return n;}
       }(i)
    );
}
funcs[0]() ; // 0
funcs[1]() ; // 1
funcs[2]() ; // 2
```

# loop variable with let

```
let funcs = [  ];
for (let i=0; i < 3; i++) {
    // new binding of “ i ” is created in every
   iteration
    funcs.push( function() { return i;} );
}
funcs[0]() ; // 0
funcs[1]() ; // 1
funcs[2]() ; // 2
```

<eot/>

# Semicolon Insertion

**You can only leave out ;**

- Before }
  ```
  A = 6  }
  ```
- After new line(s)
  ```
  A = 6
  }
  ```
- End of the program

Cannot leave ; within ‘for’ header

- **for (var i=0; i < 7** .. NO ; inserted here
  **i++) {**

# Semicolon Insertion

Inserted only if next token cannot be parsed

A = 6 (; is inserted automatically)

X = 5

What if next line seems to be continuation?

A = b (; is NOT inserted automatically)

(add(3,4),......)

- So problem starting chars are

  ( [ + - /

- Statements before such chars must have ;

## Building Custom Object

```
frontColor = { }   // create empty
  object
frontColor.red = 255
frontColor.blue = 0
frontColor.green = 128

redComponent = frontColor.red
```

## Object using constructor

```
function Color ( r, g, b ) {
    this.red = r
    this.green = g;
    this.blue = b
}

red = new Color(255, 0, 0)
```

# Object Using Constructor

```
function Color ( r, g, b ) {
    this.red = r
    this.green = g;
    this.blue = b
}
myColor = {  }
myColor.red    // undefined
Color.apply(  myColor, [255, 65,
  127]  )
Color.call(  myColor, 255, 65, 127 )
myColor.red   // 255
```

## Bad Usage of Constructor

```
function Color ( r, g, b ) {
    this.red = r
    this.green = g
    this.blue = b
}
Color(255, 127, 65)
this.red // 255.. but what is “this”
  here?
```

## Immutable Object Using Constructor

```
function Color ( r, g, b ) {
        this.getRed = function( ) { return r };
        this.getGreen = function() { return g };
        this.getBlue = function() { return b };
}
red = new Color(255, 0, 0)
red.getRed( ) // 255
?? red.r = 128 // creates a new property r
red.getRed()  // 255!
```

## Closure

- Closure is an object having
  - Function
  - Environment when function was created
    - Local Variables of outer function
    - Local functions declared in outer function
    - Parameters of outer function
    - this and arguments of outer function are not available but can be saved in local variables of outer function and then used in inner function

## Property Attributes

- Value (Named Data Property)
  - Default value
- Get and Set (Named Accessor Property)
  - Getter and Setter function
  - Either Value or Get/Set can be used, but not both
- Writable
  - False => Read Only Property
- Enumerable
  - False => Obj.keys or for (key in Obj) will not show
- Configurable
  - False => delete obj.prop, or redefine will not work

# Defining Property

```
function Color(r, g, b) {
  Object.defineProperties( this,
     {
        red : {
           value: r,
           writable : false,
           enumerable : true,
           configurable: false
        }, ...
     }  ); }
```

# Freezing an Object

```
Rubik = {};
Rubik.Slope = {};
Rubik.Slope.HORIZONTAL = "Horizontal";
Rubik.Slope.VERTICAL   = "Vertical";
Rubik.Slope.SLANTED = "Slanted";

// Make Rubik.Slope immutable
Object.freeze(Rubik.Slope);
Object.defineProperty( Rubik, "Slope",
{ writable : false, enumerable : true,
  configurable : false }
);
```

# Sealing an Object

```
Object.seal(Rubik.Slope);
```

No new properties can be added.

Writable properties can be re-written.

Configurable properties can be re configured.

```
Object.isSealed(Rubik.Slope)  // true
```

## ES 6 (Harmony) Features

- Block Scoping: let and const
- Destructuring
    - [a, b] = [b, a]
- Default Parameter
- Rest and spread parameters
    - function(p1, ...restAllParams)
    - calling(p1, ...restAllParamsInArray)
- Module Rubik { export function ..}
- Import MyRubik as Rubik
- Class, extends and super
- For-in/for-of, Iterators, Generators

# Code Walkthrough

Model Rubik's Cube
with
Objects

## **Next step..**

- DOM
- JQuery
- Java Script Design Patterns
- Coding Style/Documentation
- Books to read:
  - **JavaScript – The Good Parts**
  - **Effective JavaScript**

## Q & A

- Source Code
  - GitHub
- This presentation is available on SVCC
- 
- 
- kr_manoj@yahoo.com

## Many Variables in one declaration

```
function X () {
    var a = 5,
          b = 6
    var c = 7, d=8
    alert ( "a=" + a + ", b=" + b + ",
  c=" + c)
}
X()
//alert ( "a=" + a + ", b=" + b + ",
  c=" + c)
```

# Spot the mistake!

```
function X () {
     var a = 5
          b = 6
     var c = 7
     alert ( "a=" + a + ", b=" + this.b + ",
   c=" + c)
}
X()
//alert ( "a=" + a + ", b=" + b + ", c=" +
   c)
alert ( "b=" + window.b)
```

# Spot the mistake!

```
function X () {
    var a = 5,
        b = 6
    var c = 7
    alert ( "a=" + a + ", b=" + this.b + ",
  c=" + c)
}
X()
//alert ( "a=" + a + ", b=" + b + ", c=" +
  c)
alert ( "b=" + window.b)
```

# Constants in JavaScript

```
 "use strict";
Object.defineProperty(this,  "PI",  {
      value : 3.14,
      writable : false,
      enumerable : true,
      configurable : false
    });
PI = 7  // TypeError: "PI" is read-only
```

## Constants in JavaScript

```
"use strict";
var MyConst = { }
MyConst.PI = 3.14
Object.freeze(MyConst)

MyConst.PI = 8 //TypeError: "PI" is read-only
```

## Rubik's Cube

- Cube
  - Mini Cube/Cubelet/Atom
    - Corner (8)
    - Edge (12)
    - Center (6)
  - Sides/Layer
    - Front/Back/Left/Right/Up/Down

## Naming Atoms

- Corner: RFU
  - Right, Front, Up corner
  - RFU, FRU, URF … refers to same corner
- Edge : RF
  - Middle cubelet of the edge shared by Right and Front layers
- Center: R
  - Center of the right layer

## Moves

- R => right layer 90 deg **clockwise** **looking from right**
- R’ => right layer 90 deg **anticlockwise** looking from right
- R2 => right layer 180 deg
- RRRR, RR’, R2R2 =>No change
- (XY)’ = Y’X’

# Effect of a Move

- Rotating front layer clockwise

  ( F) ==>

  [ fru -> fdr -> fld -> ful -> fru ]

  [ fr -> fd -> fl -> fu -> fr ]

- FRU ->FDR

–Corner FRU has moved to FDR

–Right side color of FRU has gone to Down side of FDR

## Useful Moves

- Moves that produce the minimal disturbance
- One cycle of 3 corners (Placement)
- Rotating corners (Orientation)
- One cycle of 3 edges (Placement)
- In-place flipping edges (Orientation)

## Basic Corners Moves

- One cycle of 3 corners
  - (R'D'LD RD'L'D) => [ fru -> drf -> ful -> fru ]
  - (RF'L'F R'F'LF) => [ fru -> lfu -> drf -> fru ]
- Rotate corner at its own place

(R'D'LD RD'L'D RF'L'FR'F'LF) ==>

[ dfr -> rdf ]

[ flu -> luf ]

# Basic Edges Moves

- One cycle of 3 edges

  (V'F2VF2)      ==> [ fu -> bu -> fd -> fu ]

  (V'F'VFFV'F'V) ==> [ fr -> fl -> df -> fr ]

- Flipping edges in its own positions

  (RFBU2F2U'FUFU2B'R'F') ==>

  [ fr -> rf ]

  [ fu -> uf ]

## Cube Representation

- Cube
- Atom
  - Corner
  - Edge
  - Center
- Side
- Move
- MoveSequence
- MoveFinder

## Built-in Objects

- Object
- Function
- Array
- String
- Boolean
- Number
- Math, Date, RegExp, JSON, Error objects

## Call and Apply

- add(4, 5, 2 ,3)
- add.call(null, 4, 5, 2, 3)
- add.apply(null, [4, 5, 2, 3])
- add.apply(undefined, [4, 5, 2, 3])

## Variables

- No need to declare a variable

```
sum = 5
```

- Local Variables

```
var sum = 0;
```

- Declaring many variables in one declaration

```
var sum = 0, average = 0, stddev = 0;
```

- Always use semicolon OR know the rules precisely

## Object

- Collection of properties
- Property (optional)
  - primitive value
  - function
  - other object
- Prototype (optional)
  - To share property from others

## Literal Object

```
frontColor = {
    red : 255
    blue : 0
    green : 128
}
redComponent = frontColor.red
greenComponent = frontColor [ “green”
   ]
```

# Immutable Object

```
function makeColor ( r, g, b ) {
    return {
        getRed : function( ) { return r },
        getGreen : function() { return g },
        getBlue : function() { return b }
    }
}
color1 = makeColor(255, 0, 0)
color1.getRed( ) // 255
color1.getGreen() // 0
color1.blue = 128  // red has no property called blue!
    Error!
```

## Arrow Functions

```
( ( ) => 5 ) ( ) === 5;
var b = x => x + " Arrow";
b("Hello") === "Hello Arrow"

var d = { x : "AB",
            y : function() { return z => this.x +
   z; }
          }.y();
d( "CD" ) === "ABCD"

var e = { x : "XX", y : d };
e.y("XY") === "ABXY";
```

# Promise

```
let promise = new Promise(
                                function(resolve,
   reject) {
                              console.log(“1");
                              resolve();});
promise.then(function() {
                             console.log(“3");});
console.log(“2”);
```

You actually don't understand a concept.

You just get used to it!

And you brain makes you believe you got it!

Very important for our technical/mental health.

## References

- ECMAScript Support Matrix
  - http://pointedears.de/scripts/test/es-matrix/
- http://www.slideshare.net/Solution4Future/javascript-17363650
- https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference
-

- https://code.google.com/p/traceur-compiler/wiki/
-