Intermediate JavaScript

Austin Sims

Introduction

- A look at some unique features of Javascript
- The good parts
- The awful parts

The Good Parts

Data Types

Boolean

String

Number

Object

Key-value store

```
var xby2 = {
    name: 'X by 2',
    location: 'Michigan',
    employees: 40
};
```

Data Types

Specialized objects

- Function
  - Can be passed around like any other value
- Array
  - Dynamically sized

Strings are weird

Cannot assign properties

```
> var s = "Hello, world";
> s.prop = 1234;
> s.prop;
undefined
```

Strings have some methods

```
> s.substring(7);
"world"

> s[0];
"H"
```

...very weird

Can extend prototype for new methods

```
String.prototype.reverse = function() {
    var rev = '';
    for (var i = this.length - 1; i >= 0; i--) {
        rev += this[i];
    }
    return rev;
}
```

```
> "asdf".reverse();
"fdsa"
```

Arrays are less weird... but still

Arrays are sparse

```
> var alphabet = ['a', 'b'];
> alphabet.length;
2

> alphabet[3] = 'd';
> alphabet;
["a", "b", undefined, "d"]

> alphabet.length;
4

> delete alphabet[1];
> alphabet;
["a", undefined, undefined, "d"]

> alphabet.length;
4
```

Values for Nothing

Two different values that mean "nothing"

undefined

Value for undelcared or unassigned names

```
> undeclaredName;
undefined

> var undeclaredName;
> undeclaredName
undefined

> var o = {};
> o.property;
undefined

> undeclaredName.property;
Uncaught ReferenceError: undeclaredName is not defined
```

Not-Quite Data Types

null

Intentionally not set

NaN

When number-returning functions go wrong

```
> parseInt("");
NaN
```

Truthy and Falsy

JavaScript will coerce any expression into either `true` or `false` whenever a boolean is expected, allowing for terser null checks. This C#:

```
private void ThingsMightBeNull(F arg) {
    if (arg.prop1 != null) {
        DoSomethingWith(arg.prop);
    }
}
```

looks like this in JS:

```
function thingsMightBeNull(arg) {
    if (arg.prop) {
        doSomethingWith(arg.prop);
    }
}
```

Checking for array contents

```
if (myArray.length) doSomethingWith(myArray);
```

… but be careful

The only falsy values:

- `false`
- `0`
- `""` (empty string)
- `null`
- `undefined`
- `NaN`

Surprisingly truthy:

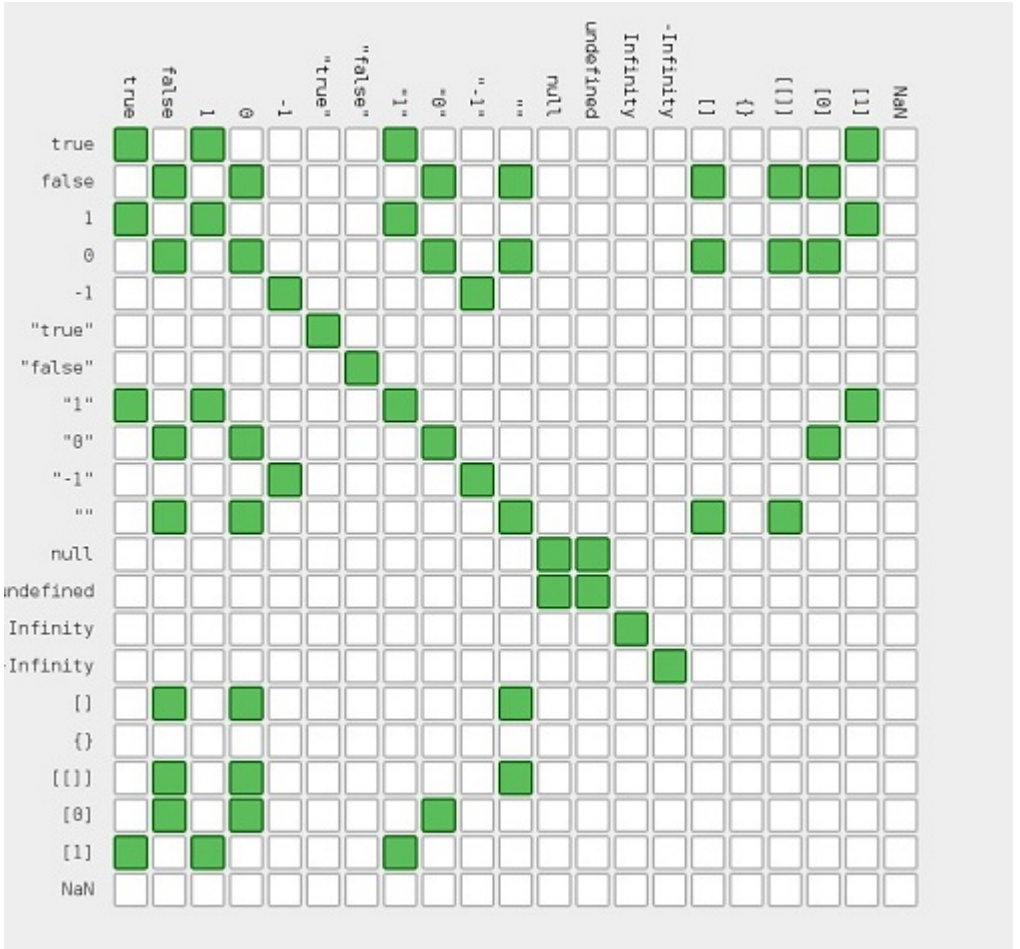
```
> Boolean([]) // empty array
true

> Boolean(-1)
true
```

Operators with Equals Sign

- = (Assignment)
- == (Equality, type-coercing)
- === (Equality, strict)

Be careful with ==



Logical OR

Short-circuit OR like in other langs

Use to set defaults as well

```
function greet(location, name) {
    console.log('Welcome to '
                + location
                + ', '
                + (name || 'honored guest.')
                );
}
```

```
> greet('Lothlórien', 'Frodo');
Welcome to Lothlórien, Frodo

> greet('Rivendell')
Welcome to Rivendell, our honored guest.
```

Function return value

When a function has no `return` statement, it returns `undefined`.

```
function f() {
    var x = 2;
    var y = 3;
    // no return statement
}

> f();
undefined
```

JavaScript objects are flexible

Let's teach an array to sum itself:

```
var a = [1,2,3];
a.sum = function() {
    var sum = 0;
    this.forEach(function(n) {
        sum += n;
    });
    return sum;
};
```

```
> a[0];
1

> a.sum();
6

> a.push(6); a.sum();
12
```

Functions can have methods too

```
function f() {
    arguments.callee.g();
}
f.g = function() { console.log('Things are getting weird'); }
```

`f` calls `g`.

```
> f();
Things are getting weird
```

Something Useful

```
var genii = {
    name: 'GENII',
    bugs: 1000,
    linesOfCode: 200000,
    ratio: function() {
        console.log(this.name + ' has one bug per '
            + this.linesOfCode / this.bugs + ' lines of code.');
    }
}

var cpdm = {
    name: 'CPDM',
    bugs: 5,
    linesOfCode: 999999999,
    ratio: function() {
        console.log(this.name + ' has one bug per '
            + this.linesOfCode / this.bugs + ' lines of code.');
    }
}
```

```
> genii.ratio();
GENII has one bug per 200 lines of code.

> cpdm.ratio();
CPDM has one bug per 199999999.8 lines of code.
```

Most of that code is the same between GENII and CPDM… this leads us to OOP in JavaScript

Classes

- There is no class keyword
- Just write a constructor
- Methods are just function-valued properties

```
function Project(name, bugs, linesOfCode) {
    this.name = name;
    this.bugs = bugs;
    this.linesOfCode = linesOfCode;
    this.ratio = function() {
        console.log(this.name + ' has one bug per '
            + this.linesOfCode / this.bugs + ' lines of code.');
    };
}
```

Use the `new` keyword to instantiate:

```
> var abc = new Project('ABC Warehouse', 50, 30000);
> abc.ratio();
ABC Warehouse has one bug per 600 lines of code.
```

Be Careful...

Always use `new` with constructors

Here's a mistake:

```
> var healthcare = Project('Healthcare.gov', 100000, 100000);
> healthcare.ratio();
Uncaught TypeError: Cannot read property 'ratio' of undefined
```

Where did the properties and method go??

```
> window.bugs
100000
```

this

How's `this` Bound?

Simple function call

Bound to global object

```
> function f() { return this; };
> f() === window;
true
```

How's `this` Bound?

Function called as an object method

Bound to object it belongs to

```
var a = [1,2,3];
a.sum = function() {
    var sum = 0;
    this.forEach(function(n) { sum += n; });
    return sum;
};
console.log(a.sum()); // prints 6
```

Same result:

```
var a = [1,2,3];
function sum(anArray) {
    var sum = 0;
    anArray.forEach(function(n) { sum += n; });
    return sum;
}
console.log(sum(anArray)); // prints 6
```

How's `this` Bound?

Function called as a constructor with `new`

The `new` keyword creates a special context for the function being called:

- An empty object is created and bound to `this`
- The function returns that object

How's `this` Bound?

Our Big Mistake Revisited

```
> var healthcare = Project('Healthcare.gov', 100000, 100000);
> heathcare
undefined

> window.bugs
100000
```

- Why is `healthcare` undefined?
- Why did 100000 get assigned to `window.bugs`?

How's `this` Bound?

Function `call` and `apply` methods

Specify what `this` is bound to for a function call

```
function add(c, d) {
    return   this.a
           + this.b
           + c
           + d;
}
var o = {a: 1, b:3};
```

```
> add.call(o, 5, 7) // 1 + 3 + 5 + 7...
16
```

`apply` is the same, but takes array for arguments

```
> add.apply(o, [5,7])
16
```

How's `this` Bound?

...when would you ever use that?

- Use `call` for class inheritance
- Call another ctor from within a subclass ctor, passing `this`

```
function Superclass(n) {
    this.single = n;
}
function Subclass(n) {
    Superclass.call(this, n);
    this.double = 2 * n;
}
```

```
> var inst = new Subclass(2);
> inst.single;
2

> inst.double;
4
```

How's `this` Bound?

The `bind` method

Creates a new function from an existing one with `this` and arguments specified

```
function f(addend1, addend2) {
    console.log((addend1 + addend2) / this.divisor);
}
var obj = {divisor: 2};
function g = f.bind(obj, 2,4);
```

```
> g();
3
```

Here's a practical example of `bind` in GENII:

```
// These calls to sendGetRequest are needed by both branches below
var sendHomeGetRequest = sendGetRequest.bind(this,
    window.FrameManager['MainSection'](tab.id).window, 'StartCustomerProfileGroup',
    'MainSection', 'MainSection', undefined, undefined, true );
var sendAccelPostingGetRequest = sendGetRequest.bind(this,
    window.FrameManager['MainSection'](tab.id).window, 'CloseCusHomeAndGoToCUPosting',
    'MainSection', 'MainSection', undefined, undefined, true);

// This branch is executed if any legacy screen is in the customer's tab
if (tab.displayLegacy) {
    // This will activate a GENII process and not return a value, so don't try
    // to assign it to anything.
    sendHomeGetRequest();

    // Specifying no controller location will cause this to navigate to
    // home--which is what we need
    tab.url = config.legacyUrlRoot;
    var navSubscription = observable(tab, 'mainFrameLoading').subscribe(function() {
        var controllerLocation = sendAccelPostingGetRequest();
        tab.url = config.legacyUrlRoot + controllerLocation;
    });
}

// This branch is executed if GENII Customer Home is in the customer's tab
// In this case, FMS home is in the background, navigation is not necessary
else {
    // This will return a controller location we need to append to the legacy
    // URL root and manually direct the iFrame to the result
    var controllerLocation = sendAccelPostingGetRequest();
    tab.url = config.legacyUrlRoot + controllerLocation;
}
```

The Wicked Sick Parts

Functional Programming

What?

- Writing code *imperatively* instead of *declaratively*
- Tell computer *what* you want instead of *how* to get it
- SQL is imperative

Consider this SQL:

```
select p.title, p.date, a.email
from posts p join authors a on p.authorId = a.authorId
where p.category = 'Programming'
order by p.date desc
```

Expressed imperatively:

```
var results = [];
for (var i=0; i<posts.length; i++) {
    if (posts[i].category == 'Programming') {
        result.title = posts[i].title;
        result.date = posts[i].date;
        var author = null;
        for (var j=0; j<authors.length; j++) {
            if (authors[j].authorId == post.authorId) {
                result.email = authors[j].email;
            }
        }
        results.push(result);
    }
}
sort(results);
```

Yuck

Here's the SQL again:

```
select p.title, p.date, a.email
from posts p join authors a on p.authorId = a.authorId
where p.category = 'Programming'
order by p.date desc
```

Expressed imperatively in native JS

```
var results = posts
    .filter(function(p) {
        return p.category == 'Programming';
    })
    .map(function(p) {
        return {
            title: p.title,
            date: p.date,
            email: authors.filter(function(a) {
                return a.authorId == p.authorId;
            })[0].email
        };
    })
    .sort(function(a,b) { return b.date - a.date; });
```

Functional Programming

Pass built-in functions to FP methods to simplify your code.

This sums numbers from all HTML form inputs and get the sum.

```
var inputs = $('input[type=text]');
_(inputs)
    .invoke('val')
    .map(parseInt)
    .reduce(function(memo, next) { return memo + next; } , 0)
    .value()
```

IIFE

Immediately Invoked Function Expression

- Remember, functions are values
- Can invoke right away instead of storing in variable

```
// Assign a color to a given id from a cyclical palette such that, once assigned,
// an ID would always return the same color
var getColor = (function() {
    var colors = ['#fefefe', '#0fabcc', '#aaabba', '0123abc', '#0123ff'];
    var memo = {};
    var i = 0;
    return function(id) {
        if (memo[id]) return memo[id];
        else return (memo[id] = colors[(i++) % colors.length]);
    }
})();
```

But why don't `colors`, `memo`, and `i` go away after the anonymous function is done?

Closures

- Functions retain references the variables in the scope where they were defined

```
function makeFontResizer(size) {
    return function() {
        document.body.style.fontSize = size + 'px';
    };
}

document.getElementById('resize-text-12pt-button').onclick = makeSizer(12);
document.getElementById('resize-text-16pt-button').onclick = makeSizer(16);
document.getElementById('resize-text-20pt-button').onclick = makeSizer(20);
```

Unexpected Side Effects of Closures

References can persist for longer than you think

```
<button>Button 1</button>
<button>Button 2</button>
<button>Button 3</button>
<script>
    var buttons = document.getElementsByTagName('button');
    var i = 0;
    Array.prototype.forEach.call(buttons, function(button) {
        button.addEventListener('click', function() {
            alert('You clicked button number ' + ++i);
        });
    });
</script>
```

Expected: Click on a button, get popup with its number

Actual: Each click of any button increments the number in the message by 1

Unexpected Side Effects of Closures

Fix it with `bind`

```
<button>Here's a button!</button>
<button>Here's another!</button>
<button>There's so many to choose from!</button>
<script src="https://code.jquery.com/jquery-2.1.3.min.js"></script>
<script>
    var buttons = document.getElementsByTagName('button');
    var i = 0;
    Array.prototype.forEach.call(buttons, function(button) {
        button.addEventListener(
            'click',
            alert.bind(this, "You clicked button number " + ++i)
        );
    });
</script>
```

The Awful Parts

Global Variables

Not using 'var' makes a global variable

```
myVar = 1234;
```

Variable Hoisting

In most languages, using curly braces makes a new scope. You can do this in C#:

```
private void MyMethod(bool condition) {
    if (condition) {
        int x = 2;
    } else {
        int x = 3;
    }
}
```

Variable Hoisting

- In JS, only a function declaration makes a new scope
- Can still use IIFE for that
- Ugly and awkward

```
function myMethod() {
    if (condition) {
        (function() {
            var x = 2;
        })();
    } else {
        (function() {
            var x = 3;
        })();
    }
}
```

Semicolon Insertion

- Semicolons technically required
- Interpreter will guess if they're missing

```
function formatName(first, middle, last) {
    return
        last
        + ', '
        + first
        + ' '
        + middle
        + '.';
}
```

```
> formatName('Zaphod', 'M', 'Beeblebrox');
undefined
```

Questions?

slideshow created with remark