JavaScript: Sort of a Big Deal,

But Sort of Quirky...

David Johnston: dwtj@iastate.edu

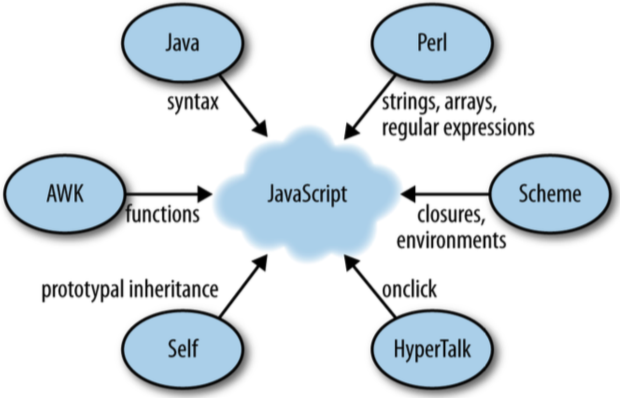
Iowa State University

March 20, 2017

"Lisp in C's Clothing" (Crockford, 2001)

- **Dynamically Typed:** no static type annotations or type checks.
- **C-Like Syntax:** curly-braces, `for`, semicolons, dot operator.
- **Object-Oriented:** OOP patterns, properties, inheritance (prototypal, not class-based).
- **Functional:** FP patterns, function literals, closures with lexical scoping.

“Programming languages that influenced JavaScript” (Rauschmayer, 2014)



Problem and Goal

- **Problem:** Though JS may syntactically look similar to other languages, many may find the language's semantics surprising.
- **Goal:** Present a few of these to promote awareness.

A Surprising Example

What is going on here?

```
function f() {
    x = new Object();
}
f()
window.x === x && x === this.x  // true?!
```

`x` is a variable, but somehow we can access it in 3 syntactically distinct ways. Here, we can see that `window`, the unique global object, is:

- An object by which *variables* are accessed.
- An object by which *properties* are accessed.
- An object produced by the `this` keyword.

Values and Types

Values in JavaScript

- **Dynamically Typed:** All type info is at runtime; variables have no static type annotations.
- **Weakly Typed:** Few type checks; lots of implicit type coercion.
- **Functions:** Functions are values that can be instantiated dynamically and treated as values.
- **Objects:** Objects have *properties* which store values; when a property lookup is called, it acts as a *method*.
- **Objects Are Just Maps:** Properties can be dynamically added, removed, and assigned values.
- **No UDTs:** No language support for static user-defined types (UDTs), but they can be emulated via constructors which manipulate object properties.

Some Familiar Literals

- String Literal: `"Foo"`
- Number Literal: `42`

Array Literals

```
var arr = ["Hello", "world", "!", 42];
```

Object Literals

```
var obj = {
    foo: "bar",
    n: 42
};
```

Function Literals

Aside from their names, these two functions are equivalent in every way. The former is syntactic sugar for the latter.

```
function f(x) {
    return x + 42;
};

var g = function(x) {
    return x + 42;
};
```

Nested Function Literals

```
var enclosed = 42;
function outer() {
    function inner() {
        return enclosed;
    }
    return inner;
}
enclosed === outer()();  // ==> true
```

We'll talk more about scopes later.

Functions and Arrays are actually objects.

ES6 defines seven datatypes:

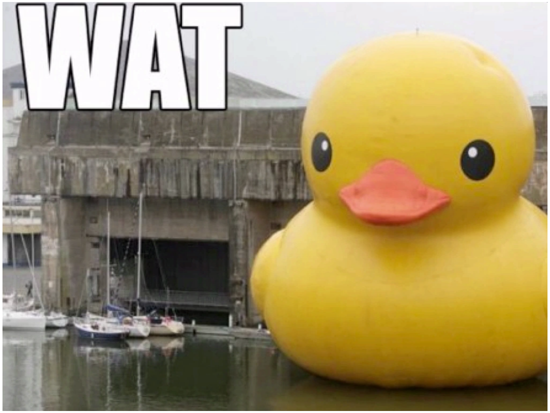
- Boolean
- Null
- Undefined
- Number
- String
- Symbol (new in ES6)
- Object

These types don't quite match typeof

ES6 defines the following behavior for the typeof operator:

- `typeof Undefined ==> "undefined"`
- `typeof Null ==> "object"`
- `typeof Boolean ==> "boolean"`
- `typeof Number ==> "number"`
- `typeof String ==> "string"`
- `typeof Symbol ==> "symbol"`
- `typeof Impl-dependent Function Object ==> "function"`
- `typeof Object ==> "object"`

Weak Typing in JavaScript



`https://www.destroyallsoftware.com/talks/wat`

What is `this`?

What is `this`?

- w3schools: "In JavaScript, the thing called `this`, is the object that 'owns' the current code."
- `this` is a keyword which evaluates to the current *invocation context*, and how it is set depends upon how the current function was invoked.

`this` Is Set In One Of Four Ways

- **Method Invocation:** If the current fuction was invoked as a method, then `this` is the *receiver object*.
- **Constructor Invocation:** If the current function was invoked as a constructor (i.e. using the `new` keyword), then `this` is the new object being constructed.
- **Explicit Argument:** If the current function was called via its `call()` or `apply()` methods, then `this` will be the first argument passed in.
- **Otherwise:** If in the root of the script or in a normal function invocation, `this` is the global object.

Variables and Scopes

Why does this happen?

```
function outer() {
    function inner() {
        x = 5
    }
    inner();
}
outer();
console.log(x);  // ==> 5
```

C, C++, and Java Variables Are *Block Scoped*

```
public void f() {
    for (int i = 0; i<10; i++) {
        System.out.println(i);
    }
    // Cannot use `i` here.
}
```

JavaScript Variables are **Not** Block Scoped. They are function scoped.

```
function f() {
    for (var i = 0; i<10; i++) {
        console.log(i);
    }
    // Can use 'i' here.
    console.log(i)  // ==> 10
}
```

JavaScript Variable Declarations are *Hoisted*

```
function f() {
    // ‘i‘ is declared but uninitialized.
    console.log(i)  // ==> undefined
    var i = 42
    console.log(i)  // ==> 42
}
```

Variable Lookups in a Nested Function?

```
var x = new Object();
function outer() {
    function inner() {
        return x;
    }
    return inner;
}
x === outer()();  // ==> true
```

The x variable which is read is the “nearest” variable with this name in the scope chain. The search goes: inner, outer, then finally window (the global object).

Variable Assignments in a Nested Function?

```
function outer() {
    function inner() {
        x = 5
    }
    inner();
}
outer();
console.log(x);  // ==> 5
```

If no `x` variable is found in searching the scope chaing, then this implicitly declares a variable `x` as a property of the global object and initializes it.[1]

[1] As of ES5, one can prevent this foolishness by using strict mode.

How Is the Scope Chain Determined?

For some user-defined function `f`, the parent scope of `f` is the scope in which `f` was instantiated.

```
function wrap(wrapped) {
    function wrapper() {
        return wrapped;
    }
    return wrapper;
}
var a = {a: "a"};
var b = {b: "b"};
var a_wrapper = wrap(a);
var b_wrapper = wrap(b);
a_wrapper() === a;  // ==> true
b_wrapper() === b;  // ==> true
```

Our Surprising Example (Revisited)

This example illustrates how the global object is at the intersection of these three different language features: variable, properties, and invocation contexts.

```
function f() {
    x = new Object();
}
f()
window.x === x && x === this.x  // true!
```

We cannot normally use *property accesses* on our scopes: we don't have expressions which evaluate to a scope. But there's one exception: the global object.

Inheriting Properties

Objects Can Inherit Properties via Prototypal Inheritance

To the white board!

Three Ways to Designate an Object's Prototype

- **Explictly at Object Instantiation:** The `Object.create()` helper function creates a new object instance with a given prototype.
- **Constructor Functions:** Set a constructor function's `prototype` property, and this object will become the prototype of all object instanted from this constructor.
- **Set Manually**: `Object.setPrototypeOf()`

Prototype Chain Search

- **Reading From obj.p:** The prototype chain of `obj` is searched, looking for the first object which *defines* `p`. If `p` is never found, `undefined` is returned.
- **Assigning To obj.p:** The prototype chain of `obj` is searched, looking for the first object which *defines* `p`. If `p` is never found, `p` is defined directly on `obj`.

Scope Chains vs Prototype Chains

- **Scope Chain:**
  - Extended by function call.
  - Extends from function definition scope. (Lexical/static scoping)
  - Stored values accessed via variable accesses.
  - The chain's root is always the global object.
- **Prototype Chain:**
  - Extended by object creation.
  - Extends from a designated object.
  - Stored values accessed via property accesses.
  - The chain's root is always `null`.

Crockford, D. (2001). Javascript: The world's most misunderstood programming language.

Rauschmayer, A. (2014). *Speaking JavaScript: An In-Depth Guide for Programmers*. "O'Reilly Media, Inc.".