JavaScript Programming Language:

The Language Everybody *Loves* to *Hate*

Roberto Chinnici
Senior Staff Engineer
Sun Microsystems, Inc.

TS-4986

Java

Sun microsystems

Forget about browser quirks for an hour.

Learn the modern, powerful programming language at the core of JavaScript™.

GOAL

Agenda

- Introduction
- Functional JavaScript programming language
- Object-oriented JavaScript technology
- Some cool stuff
- Conclusions

Introduction

Let's run an informal poll...

Google Results for "X suck(s)"

X	Hit count
JavaScript	622000
Ruby	496000
Taxes	485000
Python	389000
Death	372000
Java	205000

Snapshot taken on March 21, 2008

Common JavaScript Programming Language Complaints

- `24.88 + 4.35 ⇒ 29.229999999999997`
- `false 0 0.0 "0" "0.0" null undefined` are all false
- `0.0 + "0" ⇒ "00"`
- No `typeof(a) == "array"`
- No `class public private` keywords
- No `package` keyword either
- How does `this` work anyway?

The Reality

- A language frozen too soon
- With a puny standard library
- Lisp-1 in disguise
- C-like syntax with some innovations
- Prototype-based object system

Won't Talk About...

- Syntax
- `with` statement
- Coercions
- DOM
- Regular expressions
- Built-in library
- Specific Ajax toolkits (other than as examples)

Will Talk About...

- Core language
- Functions of all kinds
- Objects, properties, constructors, instances
- The case of the missing constructs
- Some cool ways to stretch the JavaScript programming language syntax

Functional JavaScript Programming Language

JavaScript Programming Language is a Lisp-1

- Functions are first-class objects
- Functions are closures
- Functions can be anonymous
- Functions can be higher-order

Functions Are First-Class

```
function triple(x) { return x * 3; }

var addTwo = function(y) { return y + 2; }

function compose(f, g) {
    return function(z) {
        return f(g(z));
    }
}
```

- `compose(addTwo, triple)(5)` ⇒ 17

Functions Are Closures

```
function createCounter() {
    var i = 0;
    return function() {
        return ++i;
    }
}

var counter = createCounter();
```

- counter() ⇒ 1
- counter() ⇒ 2
- counter() ⇒ 3

Only Functions Introduce a New Scope

```
function verbose() {
    var x = 1;

    function f() {
        var y = 2;
        return x + y + z;
    }

    var y = 3; // masked
    var z = 4;

    return f;
}

 (verbose())() ⇒ 7
```

Beware!
No Warning for Multiple Definitions of a Variable

```
function compute(x, y) {
    var t;

    // lots of convoluted code stores a value in t ...

    if (/* some condition */) {
        var t = // a complex expression ...
    }

    // algorithm continues but...
    // ... t has been overwritten!
}
```

Function Application

- Functions can be directly applied

```
function fn(x) { ... }
 fn(5)
```

- Works with variables too

```
var fn = function(x) { ... }
 fn(5)
```

- For advanced uses, function objects have two built-in methods

```
fn.call(    , arg1, arg2, ...)
fn.apply(     , args)
```

Function Application

```
function add(x, y) { return x + y; }
```

- `add.call(null, 3, 5)` ⇒ **8**
- `add.apply(null, [3, 5])` ⇒ **8**

Accessing the Arguments Passed to Function

- Predefined `arguments` variable
- Object with several properties:
  - `i` (an integer) — `i`-th argument
  - `length` — number of arguments
  - `callee` — the function being called

Generalized Sum

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

- `sum(3, 5, 7, 9)` ⇒ **24**
- `sum.call(null, 3, 5, 7, 9)` ⇒ **24**
- `sum.apply(null, [3, 5, 7, 9])` ⇒ **24**

Anonymous Factorial Function

```
var mysterious = function(n) {
    return n <= 1 ? 1 : n * arguments.callee(n - 1);
}

 mysterious(5) = 120;
```

Higher-Order Functions (1)

```
function some(fn) {
    return function() {
        for (var i = 0; i < arguments.length; ++i) {
            if (fn(arguments[i])) return true;
        }
        return false;
    }
}

function even(n) { return n % 2 == 0; }

var isAtLeastOneEven = some(even);

 isAtLeastOneEven(3, 5, 7, 2) ⇒ true
```

Higher-Order Functions (2)

```
function all(fn) {
    return function() {
        for (var i = 0; i < arguments.length; ++i) {
            if (!fn(arguments[i])) return false;
        }
        return true;
    }
}

var areAllEven = all(even);

areAllEven(3, 5, 7, 2) ⇒ false
```

Higher-Order Functions (3)

```
function supertest(test, result) {
    return function(fn) {
        return function() {
            for (var i = 0; i < arguments.length; ++i) {
                if (test(fn(arguments[i]))) return result;
            }
            return !result;
        }
    }
}

var some = supertest(function(x) { return x; }, true);

var all = supertest(function(x) { return !x; }, false);
```

- `(some(even))(3, 5, 7, 2)` ⇒ **true**
- `(all(even))(3, 5, 7, 2)` ⇒ **false**

A Few More Things

- Need a basic library of higher-order functions (HOF)
  - e.g. Functional (http://osteele.com/sources/javascript/functional/)
- No tail recursion
  - Serious risk of blowing the stack
- Runtimes not particularly optimized for heavy use of HOF
  - But more sophisticated compilers on the way
- Mysterious first argument to `call`/`apply` explained later

Real-World Example (from Prototype)

```
// in this context, this is a function

function bind() {
    // argument validation code omitted

    var __method = this;
    var args = $A(arguments);  // turns it into an array
    var object = args.shift(); // pops first element

    return function() {
      return __method.apply(object,
                            args.concat($A(arguments)));
    }
}

var f2 = bind.apply(f1, [foo, bar])

f2(x, y, z) ⇒ f1.apply(foo, [bar, x, y, z])
```

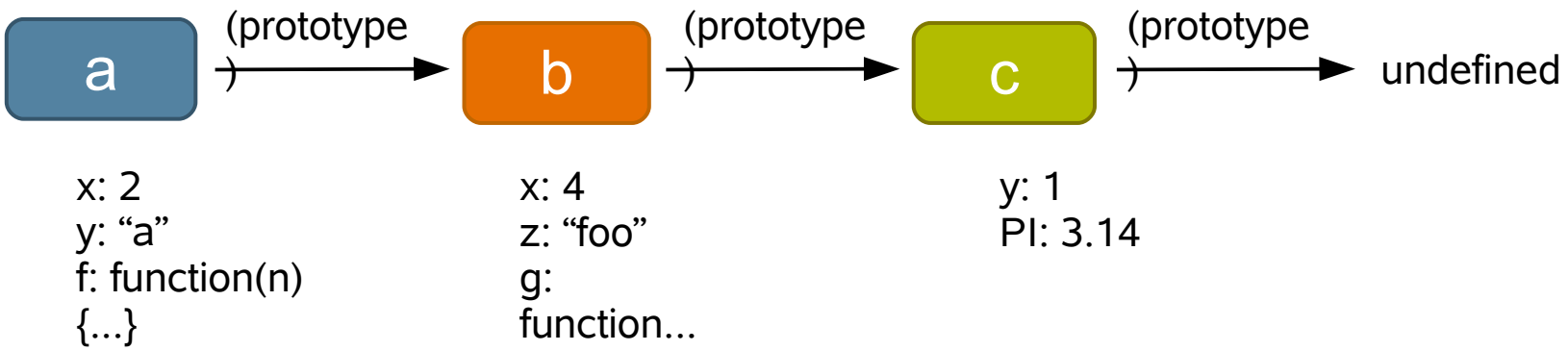
Object-oriented JavaScript Technology

Objects Without Classes

- JavaScript objects exist on their own account
- Objects have a bunch of named properties
- (Almost) every object has a prototype
- Prototypes form chains
- Property lookups follow prototype chain
- Object.prototype is the default “root”

```
Object.prototype.prototype === undefined
```

Property Search Algorithm



- `a.x` ⇒ **`2`**
- `a.z` ⇒ **`“foo”`**
- `a.PI` ⇒ **`3.14`**
- `b.y` ⇒ **`1`**

```
a.z = “bar”;
```

- `a.z` ⇒ **`“bar”`**
- `b.z` ⇒ **`“foo”`**

“Methods”

- Methods are function-valued properties
- Special notation to make calling them easier

```
 obj.m(x, y, z) ⇒ (obj[m]).apply(obj, [x, y, z])
```

- The first argument to `call/apply` is passed as `this`

```
var obj = {};
obj.m = function(x) { return x + this.y; }
obj.y = 3;
 obj.m(5) ⇒ 8
```

- `this` is a special variable, like `arguments`
- When writing HOFs that work with objects, a very common mistake is to forget to pass the correct value for `this`

How Not To Go Wrong with Objects

- Forget about constructors and classical inheritance
- Use Douglas Crockford's `object` function
  - `object(x)` ⇒ `new object whose prototype is x`
- By all accounts, it should have been a primitive, but isn't
- Build chains of objects, using supporting creation functions
- Don't use `new` statements until you understand all the details
  - And then don't use them

Using `object()`

```
var squareProto = {
    area: function() { return this.side * this.side; }
};

var square = object(squareProto);
square.side = 5;

 square.area() ⇒ 25

// now encapsulate construction process

function createSquare(side) {
  var square = object(squareProto);
  square.side = 5;
  return square;
}
```

Encapsulation using Closures

Similar to `private` modifier

```
function createBlackBox(n) {
  var hidden = n;
  var obj = {};
  obj.get = function() { return hidden; }
  obj.set = function(n) {
                if (n > hidden) { hidden = n;}
                return hidden;
            };
  return obj; // obj is leaked, hidden isn't
}

var bb = createBlackBox(42);

 bb.get() ⇒  42

// hidden cannot be modified, except by set method
 bb.set(21) ⇒  42
 bb.set(50) ⇒  50
```

Extending Objects

Mostly used with function-only objects (“traits”)

```
// from Prototype
Object.extend = function(destination, source) {
  for (var property in source)
    destination[property] = source[property];
  return destination;
};

// from Dojo
dojo._mixin = function(obj, props){
  var tobj = {};
  for(var x in props){
    if(tobj[x] === undefined || tobj[x] != props[x]){
      obj[x] = props[x];
    }
  }
  return obj;
}
```

How It Really Works

It's all backwards...

- Prototypes are set on constructor functions
- The prototype specified on a function is used as the prototype for all the object the function creates when called with `new`

```
function fn() { this.x = 4; }
fn.prototype = { y: 8 };
var obj = new fn();
```

- `obj.x ⇒ 4`
- `obj.y ⇒ 8`

How `object()` is Implemented

```
function object(o) {
    function F() {}
    F.prototype = o;
    return new F();
}
```

See http://javascript.crockford.com/prototypal.html

Typical Constructor/Prototype Use

From Mozilla Narcissus

```
function Tokenizer(s, f, l) {
    this.cursor = 0;
    this.source = String(s);
    this.tokens = [];
    // ...
}

Tokenizer.prototype = {
    get input() {
        return this.source.substring(this.cursor);
    },
    get done() {
        return this.peek() == END;
    },
    // ...
}

var t = new Tokenizer(s, f, l);
```

“But Library XYZ Supports Classes with Inheritance and `super` and...”

- They all have subtle differences
  - Single or multiple inheritance
  - Order of initializers
  - Handling of `super`
  - Modifying built-in prototypes
- Bugs have been discovered after years of use
- Know what's inside the box
- If you cannot explain how it works, don't use it

Putting Objects and Functions Together

Currying function from Functional

```
Function.prototype.curry = function() {
  var fn = this;
  var args = Array.slice(arguments, 0);
  return function() {
      return fn.apply(this,
                  args.concat(Array.slice(arguments, 0)));
  };
}


var f1 = function(x, y) { return x + y; }
var f2 = f1.curry(5);
```

- f2(10) ⇒ **f1(5, 10)** ⇒ **15**

Bind Revisited (from Prototype)

```
Object.extend(Function.prototype, {
  bind: function() {
    if (arguments.length < 2 &&
        Object.isUndefined(arguments[0]))
        return this;
    var __method = this;
    var args = $A(arguments);
    var object = args.shift();
    return function() {
        return __method.apply(object,
                              args.concat($A(arguments)));
    }
  }
});

var f2 = f1.bind(foo, bar)
```

- `f2(x, y, z)` ⇒ **`f1.apply(foo, [bar, x, y, z])`**

Some cool stuff

Objects-as-packages Idiom

From jMaki

```
function Jmaki() {
    var _jmaki = this;
    this.version = '1.1beta1';
    this.genId = function() { ... }
    // ...
}

if (typeof jmaki == 'undefined') {
    var jmaki = new Jmaki();
    jmaki.widgets = {};
    // ...
}

// all the jMaki functions are under jmaki, e.g.
//   var id = jmaki.genId();
```

Getters and Setters in JavaScript 1.5 Technology

- Define functions to be invoked when a property is accessed
- Transparent to the client

```
var squareProto = {
  side: 0,
  get area() { return this.side * this.side; }
};

var mySquare = object(squareProto);
mySquare.side = 5;

 mySquare.area ⇒ 25
```

Objects and Chaining in jQuery

- `$` as the universal entry point (i.e. the `jQuery` function)
- First select some objects
- Then manipulate them by chaining operations

```
$("div.contents p.marked").hide("slow");;

$("p").click(function() {
                $(this).css("background-color", "red");
             });

 $("a")
   .filter(".one").click(function(){...})
   .end()
   .filter(".two").click(function(){...})
   .end();
```

Protoscript

- DSL for animations in JavaScript Technology
- Program-as-data using object literals

```
$proto('#avatar', {
  Click: {
    delay: 2,
    onClick: {
      Fade: {
        opacity: {to: 0},
        onComplete: {Close : {} }
      }
    }
  }
});
```

JavaScript as a Capability Language (1)
Google Caja

- Suppose you are function `f` in object `o` called with argument `x`

```
o.f(x)
```

- What can you access?
  - `o` using `this`
  - x, your argument(s)
  - yourself, using `arguments.callee`
  - any variable in scope (if you are a closure)
  - any globals (= variables in the top-level scope)
  - any properties reachable from those objects, recursively
- Otherwise, you are boxed in

JavaScript as a Capability Language (2)
Google Caja

- The object references you have determine what you can do
- Follows from the small language core
- Some technical complications due to language semantics, browser environment
  - e.g. by default `this` is bound to the top-level window
- Caja rewrites JavaScript programming language into a subset of JavaScript programming languages that makes the constraints enforceable
- Goal is to run third-party code in a logical sandbox without requiring the use of iframes

Conclusions

Love JavaScript Programming Language Now?

- Naturally, a functional language (Lisp-1)
- Prototype-based object system with `object()`
- Don't blindly duplicate Java language patterns
- Learn from the masters

Resources

- Anything by Douglas Crockford, John Resig, etc.
  - http://javascript.crockford.com/
  - http://ejohn.org/
- Source code for many popular toolkits:
  - Prototype, jQuery, Ext JS, Dojo, jMaki, Functional, etc.
- Google Caja
  - http://code.google.com/p/google-caja/

THANK YOU

Roberto Chinnici
roberto.chinnici@sun.com

TS-4986