An Introduction to JavaScript

Godmar Back
JavaScript – The Basics

• Standardized as ECMAScript 262
• Combines elements of functional, object-based, and object-oriented languages
• Dynamically typed
• Typesafe & Garbage collected
• Interpreted
• Weak typing in that both implicit and explicit type coercions are allowed
• Uses static scoping with run-time dependent bindings
• C-like syntax
# Type System

**Types**
- Number Type
- String Type
- Boolean Type
- Null Type
- Undefined Type
- Object Type

**Values**
- IEEE FP Numbers
- Unicode Strings
- true, false
- null
- undefined
- objects, arrays, and functions
Operators

• Mostly like C or Java
• String concatenation via “+”
• Equality (==)
  – Performs conversions
  – Compares strings char-wise
  – undefined == null
  – Not transitive
• Identity (===, !==)
• Weird artifacts
  – ("0" == false && !"0" == false) is ?
  – && and || don’t always return Boolean, but type of last evaluated argument – type of (a && b) depends on value, not just type of a!
JavaScript Objects

• Objects are bundles of properties
• Properties can be of any type
  – A property that’s a function can be viewed as a method of the object
• Property can be added by simple assignment
  – a.x = 5
  – a.m = function () { .... }
  – a.o = { p1: “string” }
• Properties can be deleted via ‘delete’ operator
  – delete a.x
• Objects can be specified as literals { }
  – “JSON” – JavaScript object notation has become an interchange format
JavaScript Scoping

• Static scopes:
  – Properties of Global Object (default scope)
  – Function scopes (one per nested function) – form a scope chain for “var” declarations
• Does not use block {    } scoping
  – All “var” declared variables with function are visible on entry (multiple var are silently ignored)
  – Variables initialized to ‘undefined’
  – As are missing function arguments
• Object literals do not create a new scope
• Object properties are *not* on scope chain
  – E.g., ‘x’ does not resolve to ‘this.x’ within object method
JavaScript Functions

• First class objects
• Support closures
  – Free variables resolve based on the scope chain in effect when function was defined
  – Example:
    • // some context in which ‘d’ is defined
      var f = function (a, b) {
        var c = 1;
        d = a + b + c;
      }
      Here, ‘d’ is bound as per scope chain in ‘some context’
• Frequently used
What does this program output?

```javascript
function fiveguys() {
    var a = [];
    for (var i = 0; i < 5; i++) {
        a.push(function () {
            return i;
        });
    }
    return a;
}
```

```javascript
f = fiveguys();
for (var i = 0; i < f.length; i++)
    println(f[i]());
```

Leads to frequent errors when passing closures to handle future events, e.g. AJAX responses.
The ‘new’ operator

- JavaScript does not support class keyword, or concept
  - (though will be added in next revision of language)
- Instead, new is applied to a function
  - Creates empty object
  - Invokes the function
    - (“this” refers to the object during the call)
  - Returns a new object
  - Function object becomes the “.constructor” property
- Consequence
  - any runtime instance of a function can “double” as a constructor (and thus define a type in the conventional sense)
Built-in Objects

• Function (type: function)
  – new Function("x", "return x * x")(2) -> 4

• Array (type: function)
  – [ ] initializer convenience syntax
  – Arrays are sparse, length is (max {index} + 1)

• Number (type: function) – type coercion
• String (type: function) – type coercion
• Boolean (type: function) – type coercion
• Date
• RegExp
• Math
Prototypical Inheritance

- Let `function F() { }`
- Let `F.prototype = { <properties a, b, c> }`
- Then `o = new F()` means
  - reading `o.a` reads `F.prototype.a`
  - but writing `o.a` does not affect `F.prototype`
  - after write, subsequent reads will read per-object property
- Result: (somewhat) like dynamic classes: adding/removing properties to prototype object affects all “instances” created based on the prototype
- Recursively – forms prototype chain
  - Can be used to implement traditional inheritance
‘this’

- Binding depends on context
- At top-level, ‘this’ is the global object
- Inside functions, it depends on how the function is called:
  - If called via ‘new’ or using dot operator a.f(), ‘this’ is the current object
  - Else ‘this’ is the global object
- This (no pun intended) is confusing and extremely error prone
- Can use .bind() to create function with fixed ‘this’
What does this program output?

```javascript
// redundant, just for illustration
prop = undefined;
obj = {
    prop: "mine", // a "field"
    method: function () { // a "method"
        println("this.prop = " + this.prop);
        helper();
        // a nested function within a method
        function helper () {
            println("this.prop = " + this.prop);
        }
    }
};

obj.method();
m = obj.method;
m();
```
Real-life JavaScript

• JavaScript is embedded in environments
  – Most notably: in web pages
  – Global object here has additional properties
    • E.g., “window” (alias for global object)
    • “document”, “alert”, “setTimeout”, etc.
  – Allows interaction with the page, viewed as a hierarchical tree – the “DOM” referred to by “document”

• Lots of “ad-hoc” code, but most new code is structured

• 2 Trends for structuring
  – jQuery – style – not OO, but DOM-centered
  – OO-style JavaScript
    • Use prototypical facilities to superimpose classic OO concepts, such as packages, inheritance, and mix-ins
jQuery

$(document).ready(function() {
  $('a').click(function(event) {
    alert("Thanks for visiting!");
  });
});

• The entire library is contained in a single function called "\$"
  – returns a “selector” object that represents subset of elements in DOM and has chainable methods to operate on them (“for all”)
OO-style JavaScript

• Some codes use “manual” inheritance
  – Declare functions, name them, add prototype property, etc. – tedious, but amenable to static analysis because at least ‘function’ types are declared

• More common:
  – Classes are created on the fly using factory methods, e.g. libx.core.Class.create()
Sources of Errors

• Sheer confusion about scoping
  – Defaulting to global scope means “for (i = 0; i < 10; i++)” clobbers global i
  – ‘this’

• Namespace pollution (like globals in C)
  – “Helpful” code that changes prototype chains of all objects, e.g. “Object.prototype.usefulmethod = “

• Aliases (as in other OO languages)
  – Assigning a.x creates a local property, != b.x
  – Assigning a.x.y may be the same as b.x.y.

• Closures (see earlier example)
JavaScript Security

- JavaScript executes in Sandbox
  - No access to file system, etc.
- JavaScript has full access to the DOM of the current page
  - As well as to DOM of pages loaded from the same domain - can transform it in any way
- Cross-site scripting attack
  - Inject JavaScript into page by tricking server to serve it: via email, or post to form, etc.
- Implication for including third party code
  - Saying `<script src="http://somedomain.com" />` requires that you trust somedomain.com entirely – all or nothing
- No stack inspection
JavaScript & Concurrency

• JavaScript is single-threaded
  – Current event-handler runs to completion

• Consequence:
  – JavaScript must not run for “too long”
  – JavaScript code must not “block” – e.g., no synchronous network I/O

• Forces continuation-passing style
  – Great potential for concurrency bugs – execution order of network completion handlers is random
    • May even be synchronous if result is cached!
  – Plus, for big pages, execution of inlined JS is not uninterrupted and may interleave with event handlers
  – These errors are typically missed during testing
Further Pointers

ECMA-262:

Flanagan's JavaScript book, Chapters 1-4, available here – VT internal link:
http://proquest.safaribooksonline.com/?uiCode=vatech&xmlId=0596101996

Doug Crockford's pages make for easy and concise reading:
http://www.crockford.com/javascript/