Outline

- JavaScript Basics
- Prototypes
- Scopes
- At 7:30pm: Quiz 2

Related Languages

- JavaScript is not related to Java
  - Originally LiveScript, by Brendan Eich at Mozilla
  - Name change is Netscape/Sun marketing ploy
- 1999: ECMAScript v.3 official specification
  - JavaScript 1.5 (many browsers) = JScript 5.5 (IE)
- In progress: ECMAScript v.4 = JavaScript 2.0
- JavaScript is the only client-side scripting language that works in all browsers
  - VBScript only works in Internet Explorer

Lexical Peculiarities

- Embedded in HTML `<script>`...
- Case sensitive
- No sigils, no interpolation
- Semicolon optional at end of line, that can cause bug when prefix valid statement
- Single-line comments: `//...`, or at start `!/...`
- Multi-line comments: `/*...*/`
- Literals: `"s", 's', true, null, RegExp /.../, Object `{x:1, y:2}, Array [1,2,3]`
### Call-backs

<table>
<thead>
<tr>
<th>User</th>
<th>Browser</th>
<th>Server</th>
</tr>
</thead>
<tbody>
<tr>
<td>HTTP request (GET or POST)</td>
<td>HTTP response (HTML)</td>
<td>load from disk or run server-side script</td>
</tr>
<tr>
<td>render HTML</td>
<td>run &lt;script&gt; tag</td>
<td>return</td>
</tr>
<tr>
<td>click or type URL</td>
<td>document.write</td>
<td>change DOM tree</td>
</tr>
<tr>
<td>update rendering</td>
<td>return</td>
<td></td>
</tr>
</tbody>
</table>

**Call-back**

### Input and Output

**Input:**
- Call-backs triggered by events, when user interacts with HTML `<form>` etc.
- Reading information from DOM tree

**Output**
- `document.write("...")`
  - At HTML parse time: in-place insert
  - Later from event handler: overwrite document(!)
- DOM tree manipulation:
  - `document.getElementById("..."); innerHTML = "..."`
- Browser interaction, e.g., pop-up: `alert("...")`

### Types

- **primitive**
  - number
  - string
  - boolean
  - trivial
  - array
  - function
  - RegExp
- **null**
- **undefined**

### Variable Declarations

**Implicit**

- `b = 5;`
  - Reading an undeclared variable is a runtime error

**Explicit**

- `var x;`
- `var i=0, msg="hi";`
  - Declaration without assignment is undefined value

- **Scope of implicit declaration is global**
- **Scope of explicit declaration is local to function (there is no block scope)**

### Type Conversions

<table>
<thead>
<tr>
<th>Value</th>
<th>Boolean</th>
<th>Number</th>
<th>String</th>
<th>Object</th>
</tr>
</thead>
<tbody>
<tr>
<td>undefined</td>
<td>false</td>
<td>NaN</td>
<td>&quot;undefined&quot;</td>
<td>Error</td>
</tr>
<tr>
<td>null</td>
<td>false</td>
<td>0</td>
<td>&quot;null&quot;</td>
<td>Error</td>
</tr>
<tr>
<td>Boolean</td>
<td>false</td>
<td>identity</td>
<td>0</td>
<td>&quot;false&quot;</td>
</tr>
<tr>
<td>1</td>
<td>true</td>
<td>&quot;true&quot;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Number</td>
<td>0</td>
<td>false</td>
<td>identity</td>
<td>0</td>
</tr>
<tr>
<td>NaN</td>
<td>false</td>
<td>&quot;NaN&quot;</td>
<td>Number object</td>
<td></td>
</tr>
<tr>
<td>Other</td>
<td>true</td>
<td>String</td>
<td></td>
<td></td>
</tr>
<tr>
<td>String</td>
<td>false</td>
<td>Number of NaN</td>
<td>identity</td>
<td></td>
</tr>
<tr>
<td>Other</td>
<td>true</td>
<td>String object</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Object</td>
<td>true</td>
<td>identity</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>. (%)</td>
<td>Member, array, call, access, creation</td>
</tr>
<tr>
<td>==, !=</td>
<td>1</td>
</tr>
<tr>
<td>+</td>
<td>2</td>
</tr>
<tr>
<td>delete</td>
<td>3</td>
</tr>
<tr>
<td>&lt;, &gt;, &lt;=, &gt;=</td>
<td>4</td>
</tr>
<tr>
<td>instanceof</td>
<td>5</td>
</tr>
<tr>
<td>void</td>
<td>6</td>
</tr>
<tr>
<td>=</td>
<td>7</td>
</tr>
<tr>
<td>+, -</td>
<td>8</td>
</tr>
<tr>
<td>++, --</td>
<td>9</td>
</tr>
<tr>
<td>&amp;&amp;</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>? :</td>
<td>12</td>
</tr>
<tr>
<td>., .()</td>
<td>13</td>
</tr>
</tbody>
</table>

© Martin Hirzel
### Control Statements

<table>
<thead>
<tr>
<th>Type</th>
<th>Code</th>
</tr>
</thead>
</table>
| Conditional          | if (expr) {...} [else {...}]
|                      | switch(expr) {case expr: ... default:...}                            |
| Fixed-iteration      | for (expr; expr; expr) ...
| loops                | for (var in expr) ...
| Indefinite           | while (expr) ...
| loops                | do ... while (expr) ;                                               |
| Unstructured control | break [labelName];
|                      | continue [labelName];
|                      | return [expr];
|                      | throw expr;                                                        |
| Other                | try {...} [catch(id){...}] [finally{...}]                            |
|                      | with (expr) ...                                                     |

### Writing Subroutines

- **Declaration**: `function id(args){...}
  - Declaration creates named function at compile time
  - id mandatory; visible externally in enclosing scope

- **Arguments**: `arg := id`
  - User may pass more or fewer than declared
  - Missing arguments have value `undefined`
  - All (declared and surplus) arguments are in arguments array-like object

- **Literal**: `function [id](args){...}
  - Expression creates anonymous function at runtime
  - id optional; only visible internally for recursion

### Outline

- JavaScript Basics
- Prototypes
- Scopes
- At 7:30pm: Quiz 2

### Using Objects

```javascript
a1 = new Apple(150, "green");
a2 = new Apple(150, "green");
a2.color = "red";
```

- **Constructor calls**
- **Property calls**
- **Method calls**

### Defining Classes

```javascript
function Apple(weight, color) {
    this.weight = weight;
    this.color = color;
}
```

- **All functions f are objects with a prototype property**
  - f.prototype initially points to an almost empty object
  - f.prototype.constructor points back to f

### Prototypes and Constructors

- **Constructor = function that initializes new object**
- **Prototype = object stored as .prototype property of constructor and new objects**
- **Method = function stored as property of object**

Conventions:
- Assign object properties (fields) in constructor
- Assign prototype properties (methods) before any calls to constructor
JavaScript

Operators on Objects

- Create new empty object $o$
- Set $o.prototype = C.prototype$
- Set $o.constructor = C.prototype.constructor$
- Call $C(\ldots)$, pass $o$ as value for this
- Return result of $C(\ldots)$ or $o$ if none

```
new C(\ldots)
```

- If object $o$ has property $p$, return it
- Otherwise, look in $o.prototype$

```
$\ o.p$
```

- If object $o$ has property $p$, assign it
- Otherwise, create it and assign it

```
$\ o.p = expr$
```

- If $o.constructor$ is $C$, return true
- Otherwise, look in $o.prototype$

```
$\ o.prototype\ instanceof C$
```

Concepts

Prototype Inheritance

```
Fruit : Function
  constructor = Fruit
  prototype = new Fruit

Apple : Function
  constructor = Apple
  prototype = new Fruit
```

Soap-box

Evaluating Prototypes

Strengths
- Orthogonality
  - No class/object duality
- Flexibility
  - Can emulate classes
  - But don’t have to
  - E.g., can borrow (copy) method instead of inheriting

Weaknesses
- Lack of familiarity
- Lack of static guarantees
  - Less error checking
  - Harder to optimize
  (but, Self pioneered many optimizations later used for Java)

JavaScript

Inheritance in JavaScript

```
function Fruit(weight) {
  this.weight = weight;
}
Fruit.prototype.pluck = function() {
  return "fruit(" + this.weight + ")";
}
Fruit.prototype.prepare = function(how) {
  return how + " fruit(");
}
Apple.prototype.pluck = function() {
  this.weight = weight;
}
function Apple(weight, color) {
  this.color = color;
  this.weight = weight;
}
Apple.prototype.constructor = Apple;
delete Apple.prototype.weight;
Apple.prototype = new Fruit;
```

More Operators on Objects

```
o["p"]
```

- If object $o$ has property $p$, return it
- Otherwise, look in $o.prototype$

```
o["p"] = expr
```

- If object $o$ has property $p$, assign it
- Otherwise, create it and assign it

```
"p" in o
```

- If object $o$ has property $p$, return true
- Otherwise, look in $o.prototype$

```
typeof o
```

- String describing JavaScript type, e.g.,
  - typeof new Fruit = "object"

```
delete o.p
```

- If object $o$ has property $p$, set its value to undefined
- For $(i in o)$ loop omits undefined $p$

Arrays

- Creation: $a=[4,5,6]$; $b=new Array(size)$; $c=new Array(7,8,9)$
- Indexing: e.g., $a[2] = a[-1] = a["k"] = 99$
  - Both arrays and non-array objects can be indexed
  by both numbers and strings
- Arrays are special:
  - length property is 1 + last used integer index
  - Write to non-existent index inserts
- Setting element to undefined: $delete a[i]$
- Resizing array: $a.length = newLength$
Native ECMAScript Objects

- Array: Literal notation: `[]`
  - length property behaves specially
- RegExp: Literal notation: `/FLAGS`
- Function: Callable, can serve as constructor
- String: Wrappers for auto-boxing
- Boolean: (Implicit conversion upon `r, p` access)
- Number: Auto-unboxing for primitive operators

- Also: Global, Object, Math, Date, Error
- See ECMA-262 v.3 specification for properties & methods

Outline

- JavaScript Basics
- Prototypes
- Scopes
- At 7:30pm: Quiz 2

Scope Chain

```
var a = 'ga';
function x(b) {
  c = 'gc';
  var d = 'xd';
  var e = 'ye';
  y('ya', 'yd');
  x('xb');
}
```

Variables as Properties

- Globals = properties of "global object"
  - Client-side JavaScript: window is global object; separate windows for each frame
  - Top-level functions are also properties of global object, e.g., `x.Apple`
- Locals = properties of "call object"
  - Call objects are chained in scope chain
  - Call object has properties `arguments`, and `arguments.callee` is current function

Receiver Object

```
y = o.f(x)
y = f.call(o, x)
y = f.apply(o, [x])
y = new f(x)
```

Abbreviated Member Access

```
with(o) stmt
```

© Martin Hirzel
**JavaScript**

### Static Scoping

**Bound in closest nesting scope in program text**

```
x = 'a';
function g() {
    var x = 'd';
    h();
}
function h() {
    document.write(x); // print 'a'
}  // global
```

---

### Closures

**Closure = Function + Environment**

```
function f(y) {
    function g() {
        document.write(y);
    }
    return g;
}
```

---

### Structure of a JavaScript Application

<table>
<thead>
<tr>
<th>Parent</th>
<th>Global object</th>
<th>Included file</th>
<th>Object</th>
<th>Call object</th>
<th>Function</th>
<th>Statements</th>
</tr>
</thead>
<tbody>
<tr>
<td>frames()</td>
<td>Own window object, all top-level variables and functions are properties</td>
<td>Included with <code>&lt;script&gt;</code> tag, no separate scope (part of global object)</td>
<td>Contains properties (including functions); can emulate package by convention</td>
<td>Created at call time, may survive when nested function gets returned as closure</td>
<td>Callable object, may serve as constructor</td>
<td>In function or directly at top-level</td>
</tr>
</tbody>
</table>

---

### JavaScript Packages

- **HTML tag `<script src="mod.js">`** and **JavaScript coding conventions**
  - No separate JavaScript language feature!
  - Convention: module should never define more than a single global name
  - Usually: object, which provides properties
  - Convention: nested objects named by reverse domain, e.g., `edu.nyu.cs.Counter`
  
### Scopes and Visibility

```javascript
var edu = // put this code in file "edu/nyu/cs/Counter.js"
if (!edu) edu = {}; // only one global symbol, don't pollute namespace
if (!edu.nyu) edu.nyu = {}; // if (!edu.nyu.cs) edu.nyu.cs = {};
if (edu.nyu.cs.Counter) throw new Error("Counter already exists");
function() {
    var private_counter = 0; // after return only visible from closures
    edu.nyu.cs.Counter.read = function() { return private_counter; };
}
```

---

© Martin Hirzel
JavaScript Documentation

- Tutorial: [http://www.w3schools.com](http://www.w3schools.com)
- Browser support summary: [http://www.webdevout.net/browser-support](http://www.webdevout.net/browser-support)
- W3C DOM: [http://www.w3.org/DOM/DOMTR](http://www.w3.org/DOM/DOMTR)
- JavaScript Archive Network: [http://www.openjsan.org](http://www.openjsan.org)

Last Slide

- Nothing to announce
- Today’s lecture
  - Client-side scripting
  - JavaScript
  - Prototypes
  - Closures
- Next lecture
  - Web applications
  - Databases
  - Session state
  - Form validation
  - AJAX

Outline

- JavaScript Basics
- Prototypes
- Scopes
- At 7:30pm: Quiz 2