CSCI-GA.3033.003
Scripting Languages

10/07/2013
Client-Side Scripting (JavaScript)
Outline

• JavaScript Basics
• Prototypes
About JavaScript

• Prototype-based object oriented language
  – There are no classes (unlike Java, C++, etc.)
  – Object inherits from prototype object
  – Similar to Self programming language

• Event-driven: callbacks for browser actions

• Mostly used for client-side scripting
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 = JavaScript 1.5

• Abandoned: ECMAScript v.4 = JavaScript 2.0
  – Would have had classes and gradual typing

• Current: ECMAScript v.5 = JavaScript 1.8.5

• Dialects: JScript (IE); ActionScript (Flash)
Lexical Peculiarities

- Embedded in HTML `<script>...</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]`
JavaScript

How to Write + Run Code

- Put in local file, then File→Open in web browser:

```html
<html><head>
  <meta http-equiv="Content-Script-Type" content="text/javascript" />
  <script>
    function greet() {
      var who = document.question.who.value;
      var answer = document.getElementById("answer");
      answer.innerHTML = "Hi, " + who + ".";
    }
  </script>
</head><body>
  <span class="result" id="answer"></span>
  <form name="question">
    Who shall be greeted:
    <input type="text" name="who" onchange="greet();">
    <input type="button" value="Greet" onclick="greet();">
  </form>
</body></html>
```
Call-backs

Concepts

User

Browser

JavaScript code

Server

click or type URL

HTTP request (GET or POST)

load from disk or run server-side script

update rendering

render HTML

change DOM tree

run event handler

click or type input

document.write()

run <script> tag

return

return

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

object
- array
- function
- RegExp
- ...

null
undefined
Variable Declarations

Implicit

\[ b = 5; \]

Reading an undeclared variable is a runtime error

Explicit

\[
\begin{align*}
\text{var x;} \\
\text{var i}=0, \text{msg="hi";} \\
\end{align*}
\]

Declaration without assignment is \textit{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>true</td>
<td>Identity</td>
<td>1</td>
<td>&quot;true&quot;</td>
<td></td>
</tr>
<tr>
<td>Number</td>
<td>0</td>
<td>false</td>
<td>&quot;0&quot;</td>
<td>Number object</td>
</tr>
<tr>
<td>NaN</td>
<td>false</td>
<td>Identity</td>
<td>&quot;NaN&quot;</td>
<td></td>
</tr>
<tr>
<td>Other</td>
<td>true</td>
<td></td>
<td>String</td>
<td></td>
</tr>
<tr>
<td>String</td>
<td>&quot;&quot;</td>
<td>false</td>
<td>Number or NaN</td>
<td>String object</td>
</tr>
<tr>
<td>Other</td>
<td>true</td>
<td></td>
<td>Identity</td>
<td></td>
</tr>
<tr>
<td>Object</td>
<td>true</td>
<td></td>
<td>toString()</td>
<td>Identity</td>
</tr>
</tbody>
</table>
## Operators

<table>
<thead>
<tr>
<th>Operator(s)</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>., [], (), new</td>
<td></td>
<td>Member/array/call access, creation</td>
</tr>
<tr>
<td>++, --, +, -</td>
<td>1</td>
<td>delete: remove property from object; typeof: return type; void: discard value</td>
</tr>
<tr>
<td>,</td>
<td>2</td>
<td>L Multiplicative</td>
</tr>
<tr>
<td>*, /, %</td>
<td></td>
<td>Multiplicative</td>
</tr>
<tr>
<td>+, -</td>
<td>2</td>
<td>L Additive; +: add numbers/concat strings</td>
</tr>
<tr>
<td>&lt;&lt;, &gt;&gt;, &gt;&gt;&gt;</td>
<td>2</td>
<td>L Bitwise shift</td>
</tr>
<tr>
<td>&lt;, &lt;=, &gt;, &gt;=, instanceof, in</td>
<td>2</td>
<td>L Comparison; in: check membership</td>
</tr>
<tr>
<td>==, !==, ===, !===</td>
<td>2</td>
<td>L Identity; ===, !===: equal value+type</td>
</tr>
<tr>
<td>&amp;</td>
<td>2</td>
<td>L Bitwise and</td>
</tr>
<tr>
<td>^</td>
<td>2</td>
<td>L Bitwise xor</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>&amp;&amp;</td>
<td>2</td>
<td>L Logical and</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>?:</td>
<td>3</td>
<td>R Conditional</td>
</tr>
<tr>
<td>=, +=, -=, ...</td>
<td>2</td>
<td>R Assignment</td>
</tr>
<tr>
<td>,</td>
<td>2</td>
<td>L Multiple evaluation</td>
</tr>
</tbody>
</table>

---

CS 5142 Cornell University
10/07/13
### Control Statements

<table>
<thead>
<tr>
<th>Category</th>
<th>Code Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Conditional</td>
<td><code>if (expr) ... [else ...]</code></td>
</tr>
<tr>
<td></td>
<td>`switch(expr) {case expr: ... ... default: ...}</td>
</tr>
<tr>
<td>Fixed-iter. loops</td>
<td><code>for (var in expr) ...</code></td>
</tr>
<tr>
<td>Indefinite loops</td>
<td><code>for (expr; expr; expr) ...</code></td>
</tr>
<tr>
<td></td>
<td><code>while (expr) ...</code></td>
</tr>
<tr>
<td></td>
<td><code>do ... while(expr);</code></td>
</tr>
<tr>
<td>Unstructured control</td>
<td><code>break [labelName];</code></td>
</tr>
<tr>
<td></td>
<td><code>continue [labelName];</code></td>
</tr>
<tr>
<td></td>
<td><code>return [expr];</code></td>
</tr>
<tr>
<td></td>
<td><code>throw expr;</code></td>
</tr>
<tr>
<td>Other</td>
<td><code>try {...} [catch(id){...}] [finally{...}]</code></td>
</tr>
<tr>
<td></td>
<td><code>with (expr) ...</code></td>
</tr>
</tbody>
</table>
Writing Subroutines

- **Declaration**: `function id(arg*) {...}
  - 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](arg*) {...}
  - Expression creates anonymous function at runtime
  - `id` optional; only visible internally for recursion
Anonymous Functions

• Literal: \texttt{function \ [id] (arg*) \{ ... \}}

• Example script:

\begin{verbatim}
<html><head>
<meta http-equiv="Content-Script-Type" content="text/javascript"/>
<script>
  function callFn(fn, x) { fn(x); }
  function printIt(it) { document.write("printIt " + it + "\n<br/>"); }
  //pass reference to named subroutine; prints "printIt Hello"
  callFn(printIt, "Hello");
  //pass reference to anonymous subroutine; prints "lambda Hi"
  callFn(function(it) { document.write("lambda " + it + "\n<br/>"); }, "Hi");
</script>
</head><body>
</body></html>
\end{verbatim}
Outline

• JavaScript Basics
• Prototypes
# Using Objects

JavaScript

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

a2.color = "red";

document.write(
    a1.prepare("slice") + "<br/>\n");
document.write(
    a2.prepare("squeeze") + "<br/>\n");
```

<table>
<thead>
<tr>
<th>Constructor calls</th>
<th>Property access</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>new Apple(150, &quot;green&quot;)</code></td>
<td><code>a2.color = &quot;red&quot;</code></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Method calls</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>a1.prepare(&quot;slice&quot;) + &quot;&lt;br/&gt;\n&quot;</code></td>
</tr>
</tbody>
</table>
Prototypes and Constructors

Concepts

- **Function**: Callable object
- **Constructor**: Function that initializes new object
- **Prototype**: Object stored as `prototype` property of constructor and new objects
- **Method**: Function stored as property of object
- **Property**: Name + value
Defining Classes

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

Conventions:

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

Apple.prototype.pluck = function() {
    return this.color + " apple";
}

Assign object properties (fields) in constructor
Assign prototype properties (methods) before any calls to constructor
### Operators on Objects

| new $C(...) $ | - Create new empty object $o$
|              | - Set $o.prototype = C.prototype$
|              | - Set $o.constructor = C.prototype.constructor$
|              | - Call $C(...)$, pass $o$ as value for `this`
|              | - Return result of $C(...) $ or $o$ if none |
| $o.p$         | - 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 |
| $o instanceof C$ | - If $o.constructor$ is $C$, return `true`
|                | - Otherwise, look in $o.prototype$ |
Prototype Inheritance

**Fruit** : Function

- `pluck = function{...}`
- `prepare = function{...}`

**Apple** : Function

- `pluck = function{...}`

**a1** : Apple
- `weight = 150`
- `color = “green”`

**a2** : Apple
- `weight = 150`
- `color = “red”`
Inheritance in JavaScript

```javascript
function Fruit(weight) {
    this.weight = weight;
}
Fruit.prototype.pluck = function() {
    return "fruit(" + this.weight + "g);"
}
Fruit.prototype.prepare = function(how) {
    return how + "d " + this.pluck();
}

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

- Constructor assigns fields
- Top-level assigns methods
- Inherit from Fruit.prototype
- Remove spurious property
- Enable instanceof checks
## Receiver Object

<table>
<thead>
<tr>
<th>y = o.f(x)</th>
<th>During f, this == o</th>
</tr>
</thead>
<tbody>
<tr>
<td>y = f.call(o, x)</td>
<td></td>
</tr>
<tr>
<td>y = f.apply(o, [x])</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>y = new f(x)</th>
<th>During f, this == new object</th>
</tr>
</thead>
<tbody>
<tr>
<td>Top-level code (outside any function)</td>
<td>this == global object</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>y = f(x)</th>
<th>During f, this == global object</th>
</tr>
</thead>
</table>
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)
# More Operators on Objects

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>o[&quot;p&quot;]</code></td>
<td>• If object <code>o</code> has property <code>p</code>, return it</td>
</tr>
<tr>
<td></td>
<td>• Otherwise, look in <code>o.prototype</code></td>
</tr>
<tr>
<td><code>o[&quot;p&quot;] = expr</code></td>
<td>• If object <code>o</code> has property <code>p</code>, assign it</td>
</tr>
<tr>
<td></td>
<td>• Otherwise, create it and assign it</td>
</tr>
<tr>
<td>&quot;p&quot; in o</td>
<td>• If object <code>o</code> has property <code>p</code>, return <code>true</code></td>
</tr>
<tr>
<td></td>
<td>• Otherwise, look in <code>o.prototype</code></td>
</tr>
<tr>
<td><code>typeof o</code></td>
<td>• String describing JavaScript type, e.g., <code>typeof new Fruit == &quot;object&quot;</code></td>
</tr>
<tr>
<td><code>delete o.p</code></td>
<td>• If object <code>o</code> has property <code>p</code>, set its value to <code>undefined</code></td>
</tr>
<tr>
<td></td>
<td>• <code>for(x in o)</code> loop omits <code>undefined p</code></td>
</tr>
</tbody>
</table>
Last Slide

- Nothing to announce
- Today’s lecture
  - Client-side scripting
  - JavaScript
  - Prototypes
  - Closures
- Next lecture
  - JavaScript
  - Closures