JavaScript: Introduction, Types

Lecture 13
History

- Developed by Netscape
  - "LiveScript", then renamed "JavaScript"
  - *Nothing* to do with Java!
- Interpreted
- Browser-based, client-side execution
- Standardized by ECMA ("ECMAScript")
  - But no one calls it that!
  - MIME type text/javascript
- Becoming popular outside of browsers
  - E.g. Node.js
- *c.f.*, JScript (MS), Dart (Google)...
Client-Side Execution

GET /news/index.php HTTP/1.1
Host: www.osu.edu
User-Agent: Mozilla/5.0 (X11; Ubuntu;...

<!DOCTYPE html>
<html lang="en">
<head><title>My Page</title>
<meta charset="utf-8" />
...
Client-Side Execution

<!DOCTYPE html>
<html lang="en">
<head>
    <title>Something Short and Sweet</title>
    <meta charset="utf-8" />
</head>
<body>
    <p>Hello <a href="planet.html">World</a>!
        <br />
        <img src="globe.png" alt="a globe" />
    </p>
</body>
</html>
Client-Side Execution

<!DOCTYPE html>
<html lang="en">
  <head>
    <title>Something Short and Sweet</title>
    <meta charset="utf-8" />
    <script>
      window.alert("Annoying!");
    </script>
  </head>
  <body>
    <p>
      Hello <a href="planet.html">World</a>!
      <br />
      <img src="globe.png" alt="a globe"/>
    </p>
  </body>
</html>
Including Scripts

- **Head**: executed *before* body displays
  - Script (source) can be explicitly included
    ```javascript
    <script type="text/javascript">
    //default type in HTML 5!
    console.info("hi");
    ...
    </script>
    ```
  - Script can be linked in from external file
    ```html
    <script src="MyProgram.js"></script>
    ```
  - Recall: CSS
- **Inline**: executed as body is displayed
- **Browser blocks while downloading**
  - Common advice: put scripts at *end* of body
Demo

- Simple "hello world"
  - HTML file with javascript
  - Body is empty, script writes html output
  - Browser displays result

- Examining result
  - "view source": see javascript program
  - Firebug HTML tab: see rendered HTML

- Insert mistake
  - Replace writeln with println
  - Error console
Some Objects Provided Implicitly

- Some objects are created implicitly by the execution environment (browser)
- Document object
  - `writeln()` method puts output in body
- Window object
  - Refers to browser's display window
  - Alert method pops up a dialogue
    ```javascript
    window.alert("Say \"cheese\"!");
    ```
  - Prompt method pops up a dialogue
    ```javascript
    name = window.prompt("Enter name");
    ```
Demo with Popups

- Alert window
- Prompt window
- Console output (info, warn, error)
Familiar Minor Syntax

- Statement separator ;
  - Wrinkle: ;'s are optional!
    - Implicitly automatically inserted
    - But clearer and safer to include explicitly

- Statement blocks {...}

- Parentheses in expressions (...)

- Comments // and /*...*/
Familiar Operators

- Arithmetic (numbers are floats)
  - + - * / %
  - Wrinkles:
    - No diff in / between ints and floats!
    - % works on floats!

- Relational
  - < > <= >=
  - == !=
  - Wrinkle: === !==

- Logical
  - && || !
Familiar Statements

- **Assignment**
  - =
  - += -= *= /= %=
  - ++ -- (pre and post)

- **Conditionals**
  - if (...), if (...) ... else
  - switch (c)
    - case 'a': ... case 'b': ... default;

- **Iteration**
  - while (...), do...while(...)
  - for (...;...;...)
  - break, continue
Primitive vs Reference Types

- Distinction is similar to Java
- A variable is a "slot" in memory
- A variable can be primitive
  - The slot holds the value itself
  - Boolean, number, string, (null, undefined)
- A variable can be reference
  - The slot holds the pointer to the value
  - Arrays and objects (including functions!)
Primitive vs Reference Types

- a: 34.2
- b: "hi"
- c: 4
- d: width: 12
  height: 15
  color: "blue"
Primitives: Checking Equality

```javascript
var a = 5;
var b = 5;
var c = 7;

if (a == b)  //=>true, slots are equal
if (a == c)  //=>false

var x = "hello";
var y = "hello";

if (x == y)  //=>true! c.f. Java
```
Primitives: Assignment is Copy

```javascript
var a = 5;
var b = a; //copy contents of slot

b++;

if (a == 5) //=>true, a is unchanged
```
Assignment is Copy (of Slot)
Primitives: Argument Passing

```javascript
function inc (param) {
  param++;
}

var a = 5;
inc(a);  //copies contents of slot
if (a == 5)  //=>true
References: Equality/Assignment

```javascript
var a = {x:1, y:4}; //a new object
var b = {x:1, y:4}; //a new object

if (a == b) //=>false slots unequal

a = b; //copy contents of slot

if (a == b) //=>true
```
Assignment is Copy (of Slot)

\[
a = b;
\]

\[
a \neq b
\]

\[
a == b
\]
References: Argument Passing

```javascript
function inc (param) {
    param.x++;  
}

var a = {x: 1, y: 4};
inc(a);  //copy contents of slot
if (a.x == 2)  //=>true
```
References: Argument Passing

```javascript
function inc (param) {
    param = {x: 2, y: 7};
}

var a = {x: 1, y: 4};
inc(a);  //copies contents of slot
if (a.x == 2)  //=>false
```
Wrinkle: == vs ===

- Recall + operator in Java
  - Concatenation between strings
  - Addition between numbers
  - 3 + "4" also works! Results in "34"

- Similarly, == (!=) tries to make types match
  - 3 == "3" is true

- To prevent implicit type conversion, use === (!==)
  - 3 === "3" is false
Demo: Iteration

- Table generated by Javascript
  - Prompt for initial value
  - Calculate interest series
  - Print out a row of table for each year
Static vs Dynamic Types

- **Static:** known at compile time
  - E.g. C, C++, Java, Ada
    
    ```
    int x
    char[] a
    FluffyCloud t
    void* d
    ```

- **Dynamic:** known only at run time
  - E.g. Python, PHP, Ruby, JavaScript
    
    ```
    var x
    var a
    var t
    var d
    ```
Static Types

a
34.2
number

b
"hi"
string

c
num[]

Shape

4

0

-300

3.14

width: 12
height: 15
color: "blue"
Dynamic Types

- **a**: 34.2
- **b**: "hi"
- **c**: [4, 0, -300, 3.14]
- **d**: Object

- **d** has properties:
  - width: 12
  - height: 15
  - color: "blue"
Function Signatures

- **Statically typed**
  
  ```
  String parse(char[] s, int i) { ... return e; }
  out = parse(t, x);
  ```
  
  - Parameter types (i.e. s and i) are declared.
  - Return type (i.e. of parse) is declared.
  - The compiler checks conformance of
    - (Declared) types of arguments (t, x)
    - (Declared) type of return expression (e)
    - (Declared) type of expression using parse (out)

- **Dynamically typed**
  
  ```
  function parse(s, i) { ... }
  out = parse(t, x)
  ```
  
  - You are on your own!
Changing Types at Run-time

<table>
<thead>
<tr>
<th>Dynamic Types</th>
<th>Static Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>//a is undeclared</td>
<td>//a is undefined</td>
</tr>
<tr>
<td>var a;</td>
<td>String a;</td>
</tr>
<tr>
<td>//a is undefined</td>
<td>//a is null string</td>
</tr>
<tr>
<td>a = &quot;hi&quot;;</td>
<td>a = &quot;hi&quot;;</td>
</tr>
<tr>
<td>//load-time error</td>
<td>//compile-time err</td>
</tr>
<tr>
<td>a = &quot;hi&quot;;</td>
<td>a = &quot;hi&quot;;</td>
</tr>
<tr>
<td>a = 3;</td>
<td>int x = a;</td>
</tr>
<tr>
<td>//a is a number</td>
<td>//compile-time err</td>
</tr>
<tr>
<td>a.crazy();</td>
<td>x.length();</td>
</tr>
<tr>
<td>//run-time error</td>
<td>//compile-time err</td>
</tr>
</tbody>
</table>
Summary

- Executes at client-side, in browser
  - Interpreted (not compiled)
- Basic syntax: operators, statements
- Objects: document, window...
- Types
  - Primitives: boolean, number, string, null, undefined
  - References: arrays, objects (& functions)
- Working with primitives and references
  - Checking equality
  - Assignment
  - Parameter passing
- Dynamic types (vs static types)
JavaScript: Functions, Arrays
## Conversion of Primitive Values

<table>
<thead>
<tr>
<th></th>
<th>String</th>
<th>Number</th>
<th>Boolean</th>
</tr>
</thead>
<tbody>
<tr>
<td>undefined</td>
<td>undefined</td>
<td>&quot;undefined&quot;</td>
<td>NaN</td>
</tr>
<tr>
<td>null</td>
<td>null</td>
<td>&quot;null&quot;</td>
<td>0</td>
</tr>
<tr>
<td>booleans</td>
<td>true</td>
<td>&quot;true&quot;</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>false</td>
<td>&quot;false&quot;</td>
<td>0</td>
</tr>
<tr>
<td>strings</td>
<td>&quot;&quot;</td>
<td>0</td>
<td>false</td>
</tr>
<tr>
<td></td>
<td>&quot;1.2&quot;</td>
<td>1.2</td>
<td>true</td>
</tr>
<tr>
<td></td>
<td>&quot;0&quot;</td>
<td>0</td>
<td>true</td>
</tr>
<tr>
<td></td>
<td>&quot;one&quot;</td>
<td>NaN</td>
<td>true</td>
</tr>
</tbody>
</table>
## Conversion of Primitive Values

<table>
<thead>
<tr>
<th>numbers</th>
<th>string</th>
<th>number</th>
<th>boolean</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>&quot;0&quot;</td>
<td>0.0</td>
<td>false</td>
</tr>
<tr>
<td>-0</td>
<td>&quot;0&quot;</td>
<td>0.0</td>
<td>false</td>
</tr>
<tr>
<td>1</td>
<td>&quot;1&quot;</td>
<td>1.0</td>
<td>true</td>
</tr>
<tr>
<td>NaN</td>
<td>&quot;NaN&quot;</td>
<td>NaN</td>
<td>false</td>
</tr>
<tr>
<td>Infinity</td>
<td>&quot;Infinity&quot;</td>
<td>Infinity</td>
<td>true</td>
</tr>
<tr>
<td>-Infinity</td>
<td>&quot;-Infinity&quot;</td>
<td>-Infinity</td>
<td>true</td>
</tr>
<tr>
<td>6.022e23</td>
<td>&quot;6.022e+24&quot;</td>
<td>6.022e23</td>
<td>true</td>
</tr>
</tbody>
</table>
Easier: Column-Wise View

- How do things convert to Boolean?
  - Undefined and null are false
  - The empty string is false
  - 0 and NaN are false

- How do things convert to Numbers?
  - Undefined is NaN
  - Null is 0
  - Empty string is 0
  - Non-numeric strings are NaN
== Testing is... Different

- When types don't match, coerce:
  - Null & undefined equal only to each other
  - Strings & boolean converted to numbers
    - '1' == true
  - Objects converted via valueOf or toString
  - But note: NaN is not equal to NaN

- Some surprising consequences
  - false == 'false'    // false
  - false == '0'        // true

- See dorey.github.io/JavaScript-Equality-Table
Functions are People too

- Declaring an ordinary, named function
  ```javascript
  function foo(a, b) { ... }
  foo("hi", 3);
  ```

- Or functions can be anonymous
  ```javascript
  function(a, b) { ... }
  //how do we invoke such a thing?
  ```

- Functions are first-class citizens

- They can be assigned to variables!
  ```javascript
  var foo = function(a, b) {...};
  foo("hi", 3);
  ```
Functions Can Be Arguments

function apply(x, a) {
    return x(a);  //x is a function!
}

function square(i) {
    return i*i;
}

apply(square, 5)  //=>25
Functions Can Be Return Values

```javascript
function grantDegree() {
    function addTitle(name) {
        return "Dr. " + name;
    }
    return addTitle; // a function!
}

var phd = grantDegree();
phd("Turing"); // phd is a function!
```
function greaterThan(bound) {
    function compare (value) {
        return value > bound;
    }
    return compare; //1-arg function
}

var testPos = greaterThan(0);
testPos(4) //=>true
testPos(-3) //=>false
Closures + Anonymity

```javascript
function greaterThan(bound) {
    function compare (value) {
        return value > bound;
    }
    return compare; // 1-arg function
}

var testPos = greaterThan(0);
testPos(4) //=> true
testPos(-3) //=> false
```
function greaterThan(bound) {
    var compare = function(value) {
        return value > bound;
    }
    return compare; //1-arg function
}

var testPos = greaterThan(0);
testPos(4) //=>true
testPos(-3) //=>false
Closures + Anonymity

```javascript
function greaterThan(bound) {
    return function(value) {
        return value > bound;
    }
}

var testPos = greaterThan(0);
testPos(4)  //=>true
testPos(-3) //=>false
```
Arrays: Basics

- Numbered starting at 0
- Indexed with [ ]
- Property length is # of elements

```javascript
var sum = 0;
for (var i = 0; i < n.length; i++) {
    sum += n[i];
}
```
Array Instantiation/Initialization

- Instantiate with new
  ```javascript
  var n = new Array(3);
  ```
- Initially, each element is undefined
- Note: Elements can be a mix of types
  ```javascript
  n[0] = 10;
  n[1] = "hi";
  n[2] = new Array(100);
  ```
- Shorthand syntax: an array literal
  ```javascript
  var n = [10, 20, 30, 40];
  var m = ["hi", , "world", 3.14];
  [3, "hi", 17, [3, 4]].length == 4
  ```
Dynamic Size

- Arrays can grow
  ```javascript
  var n = ["tree", 6, -2];
  n.length == 3  //=>true
  n[8] = 17;
  n.length == 9  //=>true
  ```

- Arrays can shrink
  ```javascript
  n.length = 2;
  //n is now ["tree", 6 ]
  ```
Arrays are Dynamic

```javascript
var n = [];
```
Arrays are Dynamic

```javascript
var n = [];
```
Arrays are Dynamic

```javascript
var n[0] = 4;
```
Arrays are Dynamic
Arrays are Dynamic

```
var n[3] = 3.14;
```
Arrays are Dynamic

- undefined
- undefined
- 3.14

0
1
2
3
Arrays are Dynamic

```javascript
var n[1] = "hi";
```
Arrays are Dynamic
Summary

- Functions as first-class citizens
  - Can be passed as arguments
  - Can be returned as return values!
  - Closure: carry their context
- Arrays are fun too
  - Dynamically sized
  - Shorthand syntax
JavaScript: Array API
Accessors: Searching

- Find occurrence: `indexOf/lastIndexOf`
  - Returns -1 if not found
  - `indexOf(element[, startIndex])`
  - `lastIndexOf(element[, lastIndex])`
  - Optional parameter: start/end index
  - Uses strict equality (===)

```javascript
var i = n.indexOf(elt);
while (i !== -1) {
  report(i);
  i = n.indexOf(elt, i + 1);
}
```
Accessors: Extracting

- None of the following change the array
  - Return a new array/string with result
- Concatenate: `concat`
  ```javascript
  concatenate(a1, a2, ..., aN)
  var d = n.concatenate(n);
  ```
- Extract a sub-section: `slice`
  ```javascript
  slice(startIndex, endIndex)
  k = n.slice(1,3) //k is n[1], n[2]
  ```
- Combine into string: `join`
  ```javascript
  join(separator)
  s = n.join(" "); //default is ","
  ```
Mutators: Growing/Shrinking

- Add/remove from end: `push/pop`
  ```javascript
  var n = [10, 20];
  newLength = n.push(30, 40); //=>4
  lastValue = n.pop(); //=>40
  ```
- Add/remove from beginning: `unshift/shift`
  ```javascript
  var n = [10, 20];
  newLength = n.unshift(30, 40); //=>4
  firstValue = n.shift(); //=>30
  ```
- Push/shift gives FIFO queue
Push Example

function findAll(n, elt) {
    var indices = [];
    var i = n.indexOf(elt);
    while (i != -1) {
        indices.push(i);
        i = n.indexOf(elt, i + 1);
    }
    return indices;
}
Mutators: Extract

- Extract sub-array: `splice`
  
  `splice(index, howMany[, e1, e2, ..., eN])`
  
  - Modifies array (c.f. `slice`, an accessor)
  - Returns array of removed elements

```javascript
var magic = [34, -17, 6, 4];
var removed = magic.splice(2, 0, 13);
//removed is []
//magic is [34, -17, 13, 6, 4]

removed = magic.splice(3, 1, "hi", "yo");
//removed is [6]
//magic is [34, -17, 13, "hi", "yo", 4]
```
Mutators: Rearrange

- **Transpose all elements:** `reverse`
  ```javascript
  var n = [5, 300, 90];
  n.reverse(); // n is [90, 300, 5]
  ```

- **Order all elements:** `sort`
  ```javascript
  var f = ["blue", "beluga", "killer"];
  f.sort(); // f is ["beluga", "blue", "killer"]
  n.sort(); // n is [300, 5, 90]
  ```
Mutators: Rearrange

- Transpose all elements: `reverse`
  ```javascript
  var n = [5, 300, 90];
  n.reverse(); //n is [90, 300, 5]
  ```

- Order all elements: `sort`
  ```javascript
  var f = ["blue", "beluga","killer"];
  f.sort(); //f is ["beluga", "blue", "killer"]
  n.sort(); //n is [300, 5, 90]
  ```

- Problem: Default ordering is based on string representation (lexicographic)
- Solution: Use a function that compares
Sorting with Comparator

- A comparator \((a, b)\) returns a number
  - \(< 0\) iff \(a\) is smaller than \(b\)
  - \(== 0\) iff \(a\) is same size as \(b\)
  - \(> 0\) iff \(a\) is greater than \(b\)

- Examples

  ```javascript
  function lenOrder(a, b) {
    return a.length - b.length;
  }
  
  function compareNumbers(a, b) {
    return a - b;
  }
  ```
Sorting with Comparator

- Optional argument to sort
  
  ```javascript
  sort([compareFunction])
  ```

- Example
  
  ```javascript
  names.sort(lenOrder);
  n.sort(compareNumbers);
  ```

  ```javascript
  n.sort(function(a, b) {
    return a - b;
  });
  ```
function isBig(elt, index, array) {
    return (elt >= 10);
}

- Universal quantification: `every`
  
  `[5, 8, 130, 44].every(isBig); //false`
  
  `[54, 18, 130, 44].every(isBig); //true`

- Existential quantification: `some`
  
  `[5, 8, 130, 44].some(isBig); //true`
  
  `[5, 8, 1, 4].some(isBig); //false`

- Neither modifies original array
Iteration: Filter

- Pare down an array based on a condition: `filter`
  ```
  filter(predicate)
  ```
  ```
  predicate(element, index, array)
  ```

- Returns a new array, with elements that satisfied the predicate
  - Does not modify the original array

- Example
  ```
  t = [12, 5, 8, 13, 44].filter(isBig);
  ```
Iteration: Map

- Transform an array into a new array, element by element: map
  - E.g. an array of strings into an array of their lengths
  - ["hi", "there", "world"] → [2, 5, 5]
  - `map(callback)`
    - `callback(element, index, array)`

- Example
  ```javascript
  len = names.map(function(elt, i, a) {
    return elt.length();
  });
  ```
Iteration: For Each

- Similar to map, but preferred for side-effects and changing an array in place
  ```javascript
  forEach(callback)
  callback(element, index, array)
  ```

- Example
  ```javascript
  function logArrayElts(elt, i, array) {
    console.log("[" + i + "] = ", elt);
  }
  [2, 5, 9].forEach(logArrayElts);
  ```
Iteration: Reduce

- Applies a binary operator between all the elements of the array
  - E.g., to sum the elements of an array
  - \([15, 10, 8] \rightarrow 0 + 15 + 10 + 8 \rightarrow 33\)
  - `reduce(callback[, initialValue])`
    - `callback(previous, elt, index, array)`

- Examples
  - `function sum(a, b) { return a + b; }`
  - `function acc(a, b) { return a + 2 * b; }`
  - `\([2, 3, 7, 1].reduce(sum) \//=>13\)`
  - `\([2, 3, 7, 1].reduce(sum, 0) \//=>13\)`
  - `\([2, 3, "7", 1].reduce(sum) \//=>"571"\)`
  - `\([2, 3, 7, 1].reduce(acc) \//=>24\)`
  - `\([2, 3, 7, 1].reduce(acc, 0) \//=>26\)`
Iteration: Reduce

- Examples with anonymous functions
  
  \[2, 3\].reduce(function(a, b) {
    return a + b;
  });  //\#=> 5
  
  \[[0, 1], [2, 3], [4, 5]\].reduce(function(a, b) {
    return a.concat(b);
  });  //\#=> [0, 1, 2, 3, 4, 5]
Your Turn

Given: roster of students (an array)

Write a JavaScript program that outputs an html list of students (name and midterm score) whose gpa is > 3.0, such that the list is sorted by midterm score

1. Xi Chen (85)
2. Mary Smith (80)
3. Alessandro Reis (74)
Example Input

```javascript
var roster = [
    { name: "Mary Smith",
      gpa: 3.7,
      midterm: 80 },
    { name: "Xi Chen",
      gpa: 3.5,
      midterm: 85 },
    { name: "Alessandro Reis",
      gpa: 3.2,
      midterm: 74 },
    { name: "Erin Senda",
      gpa: 3.0,
      midterm: 68 } ];
```
One Solution

document.writeln("<ol><li>");
document.writeln(  
    roster.filter(function (e, i, a) {
        return e.gpa > 3.0;
    }).sort(function (a, b) {
        return b.midterm - a.midterm;
    }).map(function (e, i, a) {
        return e.name + " (" + e.midterm + ")";
    }).join("</li><li>")
);
document.writeln("</li></ol>");
Resources

- MDN (Mozilla Developer Network)
  - developer.mozilla.org/docs/JavaScript
- jsfiddle.net
  - HTML, CSS, Javascript → result
- "Eloquent Javascript", by Haverbeke
  - http://eloquentjavascript.net/
- "JavaScript: The Good Parts", by Crockford
Summary

- Array accessors and mutators
  - Accessors: indexOf, slice
  - Mutators for extraction: push/pop, unshift/shift, splice
  - Mutators for rearranging: reverse, sort

- Array iteration
  - Quantification: every, some, filter
  - Map (foreach for side-effects & mutating)
  - Reduce