CSE 341
Lecture 24

JavaScript arrays and objects

slides created by Marty Stepp
http://www.cs.washington.edu/341/
Arrays

var name = []; // empty
var name = [expr, ..., expr]; // pre-filled

name[index]; // access value
name[index] = expr; // store value

var stooges = ["Larry", "Moe", "Shemp"];
stooges[2] = "Curly";

- the array is the only data structure included in JavaScript (other than objects)
Array features

- JS arrays can store elements of multiple types:
  ```javascript
  > var a = [42, true, "abc"];
  ```

- Arrays can be converted into strings (or call toString):
  ```javascript
  > print("hi " + a + " bye");
  hi 42, true, abc bye
  ```

- Caution: The typeof an array is object, not array:
  ```javascript
  > typeof(a)
  object
  ```
Array length

• use the `length` property to find the # of elements:
  ```javascript
  > a.length
  3
  ```

• you can set `length`;
  - if smaller, truncates the array to the new smaller size
  - if larger, all new elements will be `undefined`

  ```javascript
  > a.length = 2;
  > a
  42, true
  ```
Non-contiguous arrays

- there is no such thing as an array out-of-bounds error
  - get an element out of bounds → undefined
  - set an element out of bounds → length increases to fit
    - any elements in between old/new lengths are undefined

```javascript
> var a = [42, 10];
> a[10] = 5;
> a
42,10,,,,,,,,,5
> typeof(a[6])
undefined
> a.length
11
```
# Array instance methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>.concat(expr...)</code></td>
<td>returns new array with appended elements/arrays</td>
</tr>
<tr>
<td><code>.indexOf(expr)</code></td>
<td>index of first/last occurrence of <code>expr</code>; -1 if not found</td>
</tr>
<tr>
<td><code>.lastIndexOf(expr)</code></td>
<td></td>
</tr>
<tr>
<td><code>.join(separator)</code></td>
<td>glues elements together into a string</td>
</tr>
<tr>
<td><code>.pop()</code></td>
<td>remove and return last element</td>
</tr>
<tr>
<td><code>.push(expr...)</code></td>
<td>append value(s) to end of array</td>
</tr>
<tr>
<td><code>.reverse()</code></td>
<td>returns new array w/ elements in opposite order</td>
</tr>
<tr>
<td><code>.shift()</code></td>
<td>remove and return first element</td>
</tr>
<tr>
<td><code>.slice(start, end)</code></td>
<td>returns sub-array from start (incl.) to end (exclusive)</td>
</tr>
<tr>
<td><code>.sort()</code></td>
<td>sorts array in place, with optional compare function that takes 2 values, returns &lt;0, 0, &gt;0 (compareTo)</td>
</tr>
<tr>
<td><code>.sort(compareFn)</code></td>
<td></td>
</tr>
<tr>
<td><code>.splice(index, count, expr...)</code></td>
<td>Removes <code>count</code> elements from array starting at <code>index</code>, and inserts any given new elements there</td>
</tr>
<tr>
<td><code>.toString()</code></td>
<td>converts array to string such as &quot;42, 5, -1, 7&quot;</td>
</tr>
<tr>
<td><code>.unshift(expr...)</code></td>
<td>insert value(s) at front of array</td>
</tr>
</tbody>
</table>
Array methods example

```javascript
var a = ["Stef", "Jay"];  // Stef, Jay
a.push("Bob");            // Stef, Jay, Bob
a.unshift("Kelly");      // Kelly, Stef, Jay, Bob
a.pop();                  // Kelly, Stef, Jay
a.shift();                // Stef, Jay
a.sort();                 // Jay, Stef

• array serves as many data structures: list, queue, stack, ...

• methods: concat, join, pop, push, reverse, shift, slice, sort, splice, toString, unshift
  ▪ push and pop add / remove from back
  ▪ unshift and shift add / remove from front
    – shift and pop return the element that is removed
```
Split and join

```javascript
var s = "quick brown fox";
var a = s.split(" ");  // ["quick", "brown", "fox"]
a.reverse();          // ["fox", "brown", "quick"]
s = a.join("!");      // "fox!brown!quick"
```

- **split** breaks a string into an array using a delimiter
  - can also be used with regular expressions (seen later)
- **join** merges an array into a single string, placing a delimiter between them
"Multi-dimensional" arrays

- JS doesn't have true multi-dimensional arrays, but you can create an array of arrays:

```javascript
var matrix = [[10, 15, 20, 25],
               [30, 35, 40, 45],
               [50, 55, 60, 65]];

> matrix[2][1]
55

> matrix.length
3

> matrix[1].length
4
```
for (name in expr) { statements; } 

- JavaScript has a "for-each" loop, but it loops over each index, not each value, in the array.
  - in some impl.s, it also loops over the array's methods!
  - considered broken; discouraged from use in most cases

```javascript
var ducks = ["Huey", "Dewey", "Louie"];
for (x in a) { print(x); }
```

0 1 2
Array exercises

• Write a function `sum` that adds the values in an array.

• Write a function `longestWord` that takes a string and returns the word within that string with the most characters. If the string has no words, return "".

• Write a function `rotateRight` that accepts an array and an integer $n$ and "rotates" it by sliding each element to the right by 1 index, $n$ times.
  - `rotateRight([1, 2, 3, 4, 5], 2)`; changes the array to store [4, 5, 1, 2, 3]
Simulating other data structures

- JS has no other collections, but an array can be used as...
  - a **stack**: push, pop, length
  - a **queue**: push, shift, length
  - a **list**: push/pop/unshift/shift, slice/splice, indexOf...
**Array higher-order methods** *

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>.every(function)</code></td>
<td>accepts a function that returns a boolean value and calls it on each element until it returns false</td>
</tr>
<tr>
<td><code>.filter(function)</code></td>
<td>accepts a function that returns a boolean; calls it on each element, returning a new array of the elements for which the function returned true</td>
</tr>
<tr>
<td><code>.forEach(function)</code></td>
<td>applies a &quot;void&quot; function to each element</td>
</tr>
<tr>
<td><code>.map(function)</code></td>
<td>applies function to each element; returns new array</td>
</tr>
<tr>
<td><code>.reduce(function)</code></td>
<td>accepts a function that accepts pairs of values and combines them into a single value; calls it on each element starting from the front, using the given initialValue (or element [0] if not passed)</td>
</tr>
<tr>
<td><code>.reduce(function, initialValue)</code></td>
<td>reduceRight starts from the end of the array</td>
</tr>
<tr>
<td><code>.reduceRight(function)</code></td>
<td></td>
</tr>
<tr>
<td><code>.reduceRight(function, initialValue)</code></td>
<td></td>
</tr>
<tr>
<td><code>.some(function)</code></td>
<td>accepts a function that returns a boolean value and applies it to each element until it returns true</td>
</tr>
</tbody>
</table>

*most web browsers are missing some/all of these methods*
Objects

• **simple types**: numbers, strings, booleans, null, undefined
  ▪ object-like; have properties; but are *immutable*
  ▪ all other values in JavaScript are objects

• JavaScript objects are mutable key/value collections
  ▪ a container of properties, each with a name and value

• JavaScript **does not have the concept of classes** (!!)
  ▪ every object is "just an object"
  ▪ *(it is possible to relate one object to others; seen later)*
Creating an object

```javascript
{  name: expr,
    name: expr, ...
    name: expr }

• can enclose name in quotes if it conflicts with a keyword
```

```javascript
> var teacher = {  fullName: "Marty Stepp",
                    age: 31, height: 6.1, "class": "CSE 341"  };
> var emptyObj = {};
```

• an object variable stores a reference to the object:
  ```javascript
  > var refToTeacher = teacher;  // not a copy
  ```
Accessing object properties

\[ \text{object.propertyName} \]
\[ \text{object["propertyName"]} \]
\[ \text{object[expr]} \]

- use latter syntax if you don't know prop. name till runtime

\>` teacher.age
31
\>` teacher["fullName"]
\> Marty Stepp
\>` var x = "height";
\>` teacher[x]
\> 6.1
Modifying/removing properties

```javascript
object.propertyName = expr;
object["propertyName"] = expr;
delete object.propertyName;
delete object["propertyName"];```

- delete removes a property from the object

```javascript
> teacher.age = 29; // if only...
> teacher["height"] -= 0.2;
> delete teacher.age; // no one will know!
> typeof(teacher.age)
undefined```
More about properties

• property names can be anything but undefined:
  > var silly = {42: "hi", true: 3.14, "q": "Q"};

• you can add properties to an object after creating it:
  > silly.favoriteMovie = "Fight Club";
  > silly["anotherProp"] = 123;

• if you access a non-existent property, it is undefined:
  > silly.fooBar
  > typeof(silly.fooBar)
  undefined
Null/undefined objects

• trying to read properties of null/undefined is an error:
  
  ```javascript
  > var n = null;
  > var u;    // undefined
  > n.foo    // error
  > u.foo    // error
  ```

• You can guard against such errors with && and ||:
  
  ```javascript
  > teacher && teacher.name
  Marty Stepp
  > n && n.foo
  null
  > (n && n.foo) || 42    // 42 if n is falsey
  42
  ```
Object methods

- an object can contain **methods** (functions) as properties
  - method can use the **this** keyword to refer to the object

```javascript
function greet(you) {
    print("Hello " + you + ", I'm " + this.fullName);
}

> teacher.greet = greet;
> teacher.greet("students");
Hello students, I'm Marty Stepp
```
For-each loop on objects

```javascript
for (name in object) { statements; }
```

- "for-each" loops over each property's name in the object
  - it also loops over the object's methods!
  - usually not useful; discouraged. also order unpredictable

```javascript
> for (prop in teacher) {
  print(prop + "=" + teacher[prop]); }

fullName=Marty Stepp
age=31
height=6.1
class=CSE 341
greet=function greet(you) {
  print("Hello " + you + ", I'm " + this.fullName);
}
```
Objects as maps

- JS has no **map** collection, but an object can be used as one:
  - the "keys" are the object's properties (property names)

```javascript
> var phonebook = {};  
> phonebook["Marty"] = "685-2181";  
> phonebook["Stuart"] = "685-9138";  
> phonebook["Jenny"] = "867-5309";  
> phonebook["Stuart"]  
685-9138
```

```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
Arrays are (just) objects

• an array is (essentially) just an object with properties named 0, 1, 2, ..., and a length property
  ▪ arrays also contain methods like pop and slice

• it's hard to tell whether a given value even IS an array
  ▪ typeof({name: "Bob", age: 22}) → "object"
  ▪ typeof([1, 2, 3]) → "object"
• **duck typing**: Dynamic typing where an object's set of properties, rather than its class, determines its semantics.
  - "*If it walks like a duck, and quacks like a duck, ...*"

• JS code will "work" as long as a value is not used in a way that causes an error.

• Any JS parameter can be of any type, so a function that expects an array can be "tricked" by passing any object that "walks and quacks" like an array...
function sum(a) { // add up elements of an "array"
    var total = 0;
    for (var i = 0; i < a.length; i++) {
        total += a[i];
    }
    return total;
}

• anything with length and numeric props. up to that length works:

```javascript
> var a1 = [3, 4, 5];
> sum(a1)
12
> var o1 = {0:42, 9:77, 1:8, length:2}; // quack
> sum(o1)
50
```