CS 454 Node.js & AngularJS

Albert F. Cervantes
Cydney Auman
California State University, Los Angeles

Iterating Over Arrays (cont'd)

- You may encounter the following syntax in the wild. Avoid using when iterating over arrays:

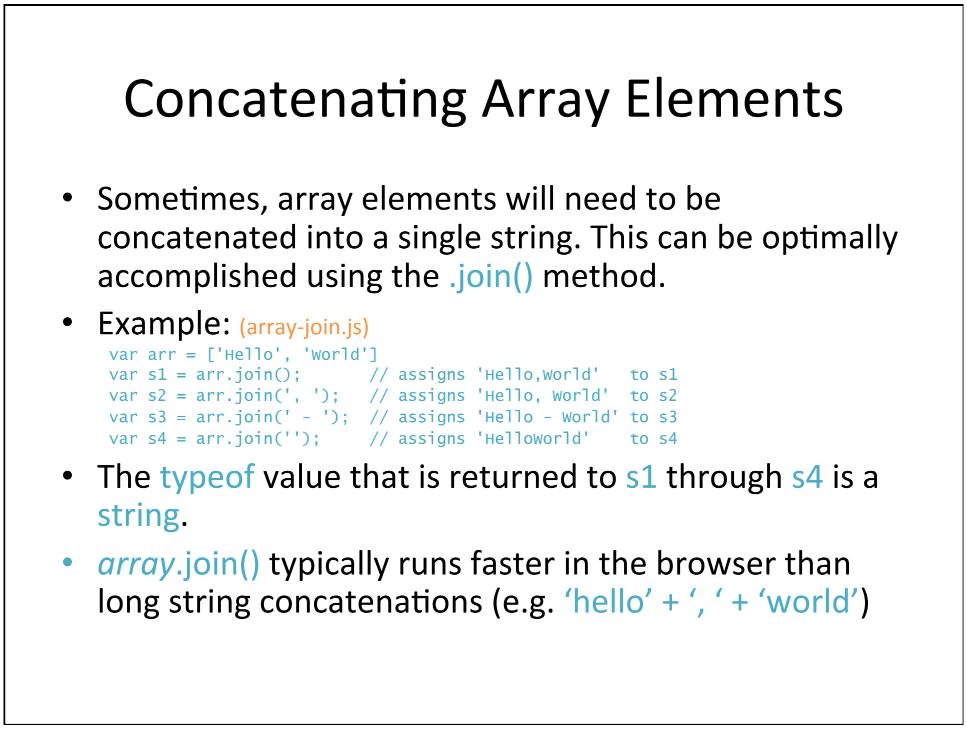
```
for (var el in arr)
```

- This approach will yield unexpected results in certain scenarios

```
var tmp = [1,2,3,4,5]

Array.prototype.foo = function() {
    return 'bar'
}

for (val in tmp)
    console.log(val)
```

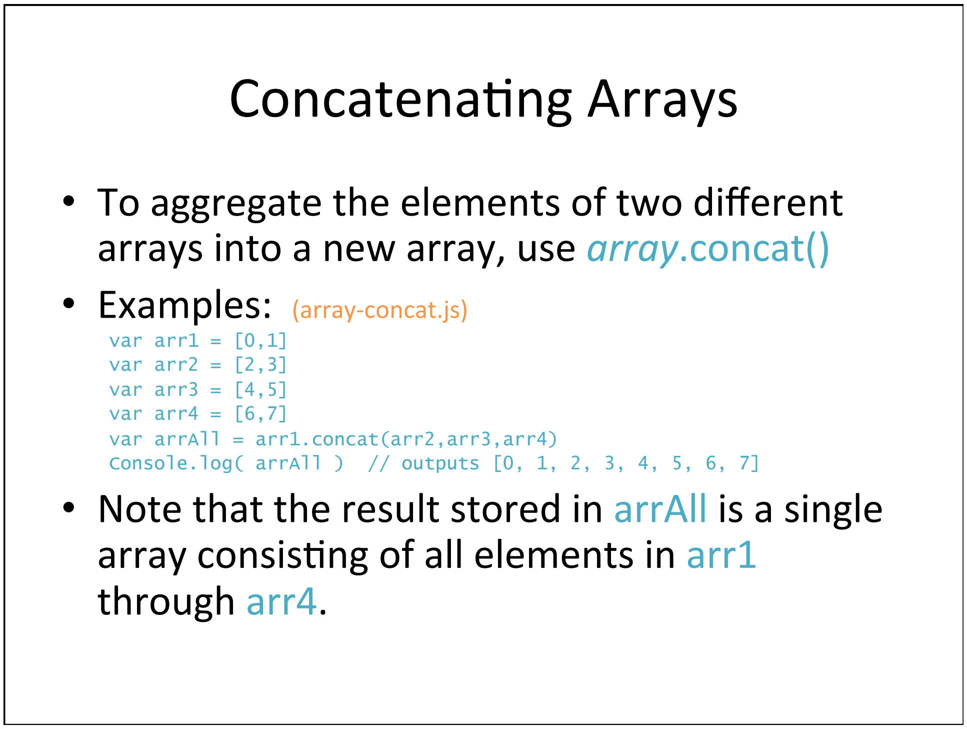


Concatenating Array Elements

- Sometimes, array elements will need to be concatenated into a single string. This can be optimally accomplished using the .join() method.
- Example: (array-join.js)

```
var arr = ['Hello', 'World']
var s1 = arr.join();       // assigns 'Hello,World'   to s1
var s2 = arr.join(', ');   // assigns 'Hello, World'  to s2
var s3 = arr.join(' - ');  // assigns 'Hello - World' to s3
var s4 = arr.join('');     // assigns 'HelloWorld'    to s4
```

- The typeof value that is returned to s1 through s4 is a string.
- *array*.join() typically runs faster in the browser than long string concatenations (e.g. 'hello' + ', ' + 'world')



Concatenating Arrays

- To aggregate the elements of two different arrays into a new array, use *array*.concat()
- Examples: (array-concat.js)

```
var arr1 = [0,1]
var arr2 = [2,3]
var arr3 = [4,5]
var arr4 = [6,7]
var arrAll = arr1.concat(arr2,arr3,arr4)
Console.log( arrAll )  // outputs [0, 1, 2, 3, 4, 5, 6, 7]
```

- Note that the result stored in arrAll is a single array consisting of all elements in arr1 through arr4.

Removing vs. Deleting Array Elements

- **Removing** an element from an array effectively shortens the length of the array by 1. Use array.splice()

```
var fruit = ['apple', 'banana', 'orange', 'pear']

fruit.splice(2, 1)  // Removes 1 element from the array beginning at index 2
```

  - The value returned from splice is an array of elements that were removed from the array.
  - .splice() can also be used to insert elements into an array at a specified location.
- When you **delete** an array element, the array length is not affected.

```
var cars = ['civic', 'accord', 'hr-v']
delete cars[1]  // cars is now [‘civic’, undefined, ‘hr-v’]
```

- See also .push()/.pop() and .unshift()/.shift()

Functions

- Optional Arguments
  - If a function isn't expecting an argument, it will just ignore it
  - Can, and eventually will, lead to passing the incorrect number of arguments to a function
- In-Browser Example:

```
alert(‘Hello’, ‘GoodBye’, ‘Game Over’);
```

Function Syntax

- If the following is valid

```
var foo = function(){ /* code */ }
```

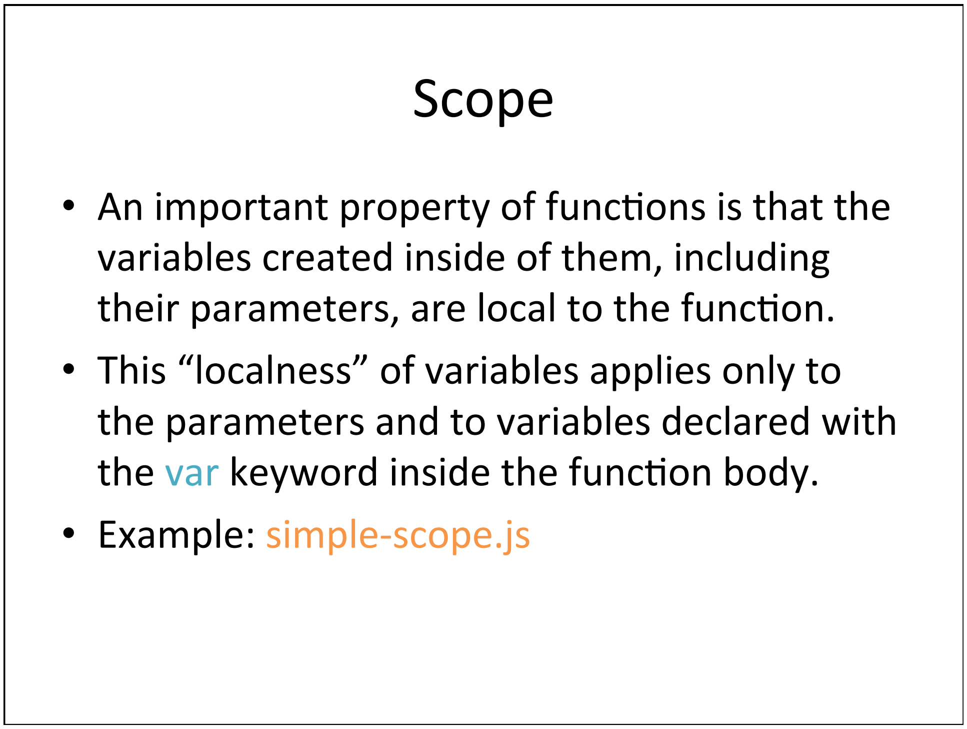
- Doesn't it stand to reason that the function expression itself can be invoked, just by putting () after it?

```
function(){ /* code */ }(); //SyntaxError:Unexpected token (
```

- When the parser encounters the function keyword in the global scope or inside a function, it treats it as a function declaration (statement), and not as a function expression, by default.
- Example: Ben-Alman.js

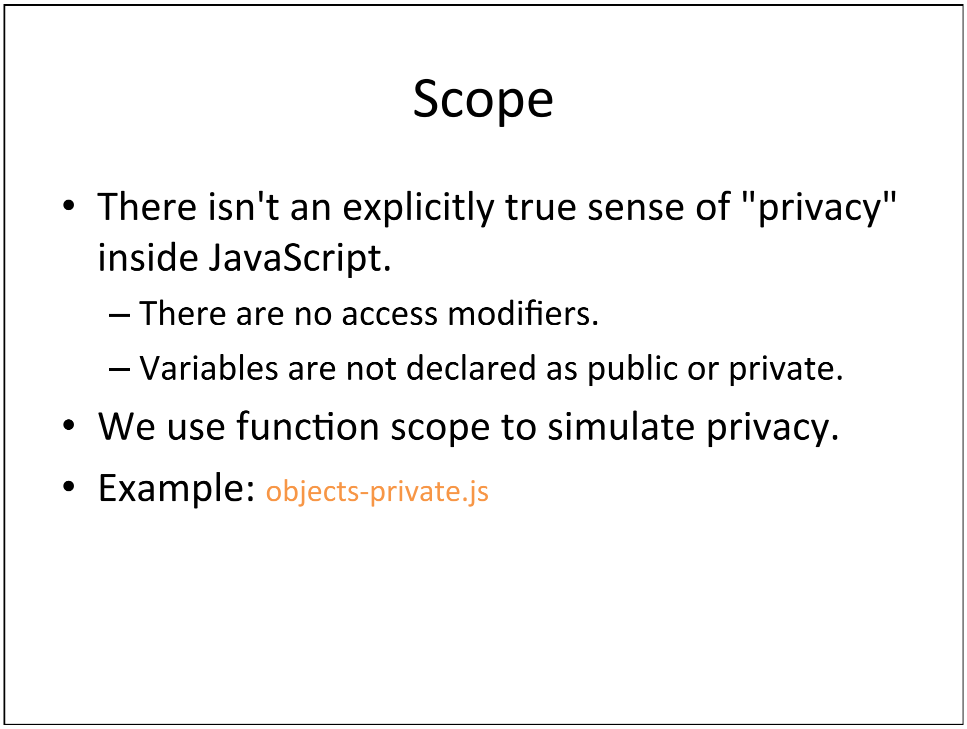
Immediately Invoked Function Expression (IIFE)

- The most widely accepted way to tell the parser to expect a function expression is just to wrap it in parenthesis.
  - In JavaScript, parenthesis can't contain statements.
- Example: Ben-Alman.js
- http://benalman.com/news/2010/11/immediately-invoked-function-expression/



Scope

- An important property of functions is that the variables created inside of them, including their parameters, are local to the function.
- This “localness” of variables applies only to the parameters and to variables declared with the var keyword inside the function body.
- Example: simple-scope.js



Scope

- There isn't an explicitly true sense of "privacy" inside JavaScript.
  - There are no access modifiers.
  - Variables are not declared as public or private.
- We use function scope to simulate privacy.
- Example: objects-private.js

Closures

- What happens to local variables when the function call that created them is no longer active?
  - Accessing variables outside of the immediate lexical scope creates a closure.
  - In practice, a closure is formed when a nested function is defined inside of another function, allowing access to the outer functions variables.
- The incrementCounter function is an example of a closure in the objects-private.js example

Closures

- The most popular type of closure is the module pattern.
- Example: closures-module-pattern.js

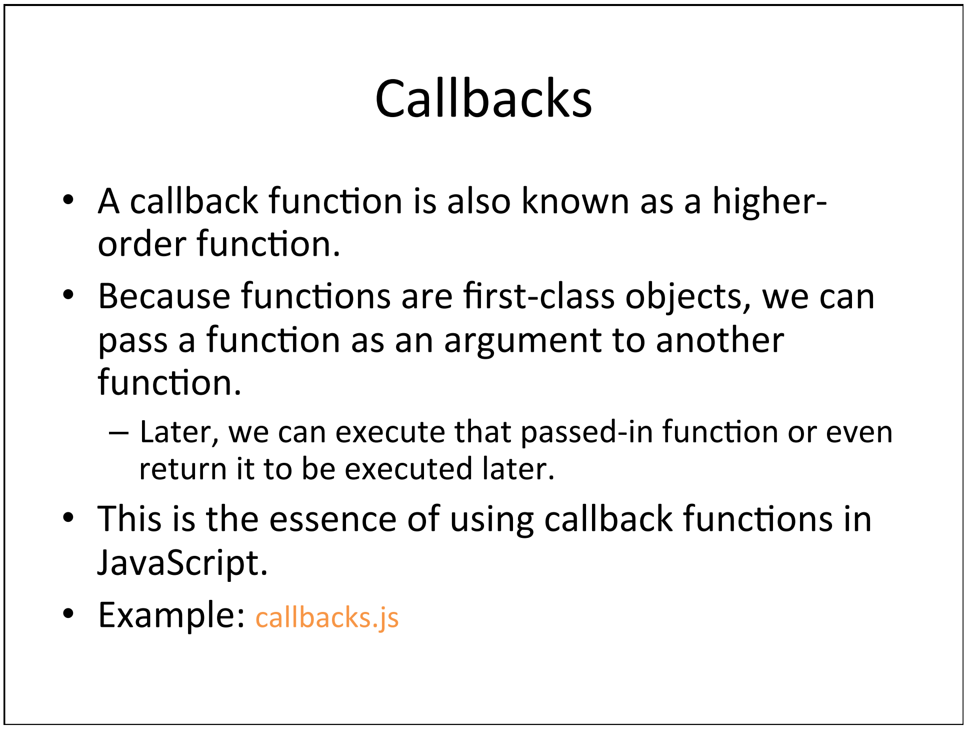
Context

- Context is most often determined by how a function is invoked.
  - When a function is called as a method of an object, this is set to the object the method is called on
- Example: context.js
- The same principle applies when invoking a function with the new operator to create an instance of an object.
  - When invoked in this manner, the value of this within the scope of the function will be set to the newly created instance

Adding Properties and Methods to Objects

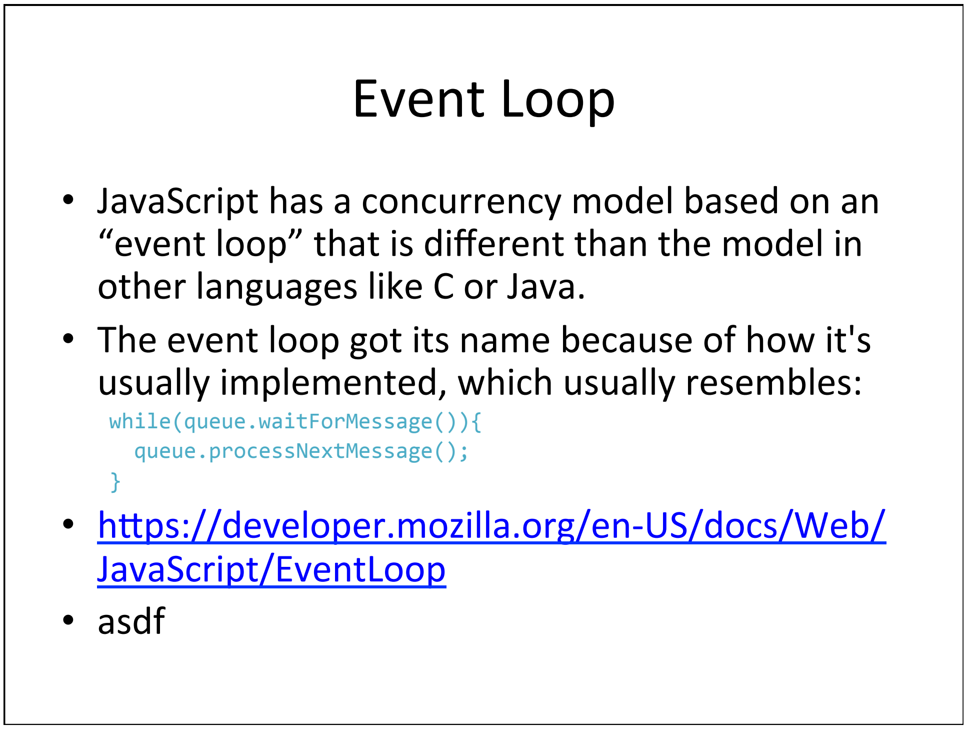
- Sometimes you want to add new properties (or methods) to an existing object.

  myObj.foobar = ‘some value’;
- Sometimes you want to add new properties (or methods) to all existing objects of a given type.
  - You cannot add a new property to a prototype the same way as you add a new property to an existing object, because the prototype is not an existing object.
- Sometimes you want to add new properties (or methods) to an object prototype.
  - The JavaScript prototype property also allows you to add new methods to an existing prototype.
- Example: prototypes.js, context-new-keyword.js



Callbacks

- A callback function is also known as a higher-order function.
- Because functions are first-class objects, we can pass a function as an argument to another function.
  - Later, we can execute that passed-in function or even return it to be executed later.
- This is the essence of using callback functions in JavaScript.
- Example: callbacks.js

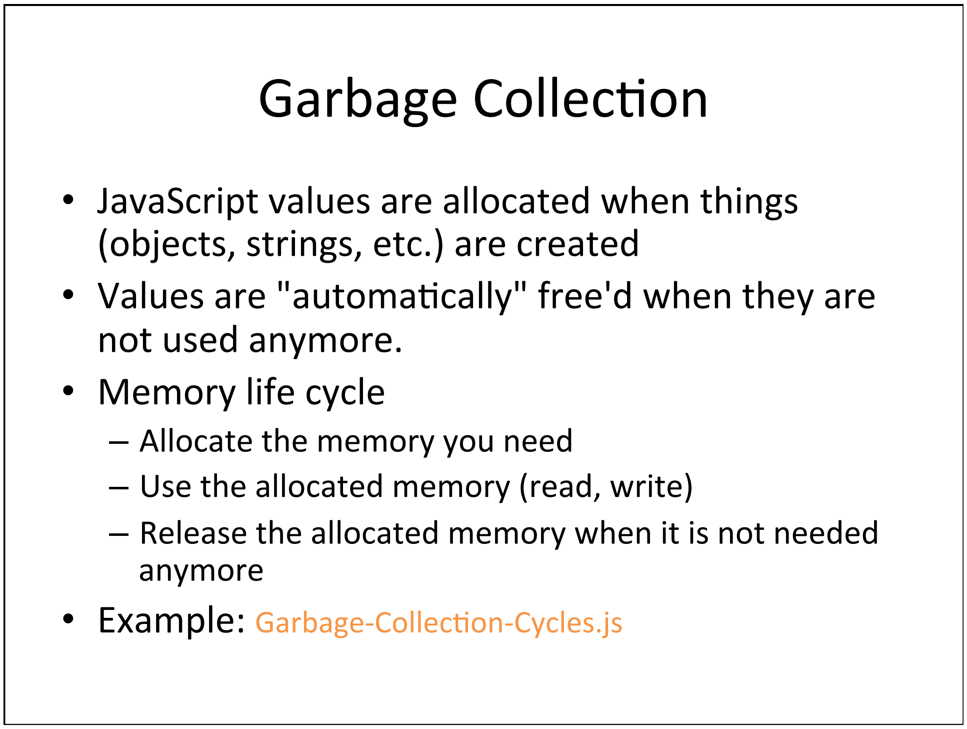


Event Loop

- JavaScript has a concurrency model based on an “event loop” that is different than the model in other languages like C or Java.
- The event loop got its name because of how it's usually implemented, which usually resembles:

```
while(queue.waitForMessage()){
  queue.processNextMessage();
}
```

- https://developer.mozilla.org/en-US/docs/Web/JavaScript/EventLoop
- asdf



Garbage Collection

- JavaScript values are allocated when things (objects, strings, etc.) are created
- Values are "automatically" free'd when they are not used anymore.
- Memory life cycle
  - Allocate the memory you need
  - Use the allocated memory (read, write)
  - Release the allocated memory when it is not needed anymore
- Example: Garbage-Collection-Cycles.js