The JavaScript Language (Part 4)

Robert M. Dondero, Ph.D.
Princeton University

Objectives

- We will cover:
  - A subset of JavaScript…
  - That is appropriate for COS 333…
  - Through example programs

Agenda

- **Arrays**
- Associative arrays
- Asynchronous processing: callbacks

Arrays

- See **arrays.js**

```
$ node arrays.js
[ 'Ruth', 'Gehrig', 'Jeter' ]
3
------------------------------------------------
[ 'Ruth', 'Gehrig', 'Jeter' ]
3
------------------------------------------------
[ 'Ruth', 'Mantle', 'Jeter' ]
3
------------------------------------------------
[ 'Ruth', 'Mantle', 'Jeter', 'Berra' ]
4
------------------------------------------------
[ 'Ruth', 'Mantle', 'Jeter' ]
3
------------------------------------------------
Ruth
Mantle
Jeter
------------------------------------------------
Ruth
Mantle
Jeter
------------------------------------------------
$
```

Arrays

- An **array** is:
  - An object...
  - That delegates to `Array.prototype`...
  - That has a `length` property...
  - That is maintained automatically…
  - Such that its value is always one greater than the largest integer index (or 0 if the array is empty)

Arrays

- See **linesort.js**

```
$ cat hamlet.txt
to be
or not to be,
that is
the question.
$ node linesort.js hamlet.txt
or not to be,
that is
the question.
to be
$
```

Agenda

- Arrays
- **Associative arrays**
- Asynchronous processing: callbacks

Associative Arrays

- See **assocarrays1.js**

```
$ node assocarrays1.js
{ Ruth: 'RF', Gehrig: '1B', Jeter: 'SS' }
-----------------------------------------
{ Ruth: 'RF', Gehrig: '1B', Jeter: 'SS' }
-----------------------------------------
{ Ruth: 'P', Gehrig: '1B', Jeter: 'SS' }
-----------------------------------------
{ Ruth: 'P', Gehrig: '1B', Jeter: 'SS', Maris: 'RF' }
-----------------------------------------
{ Ruth: 'P', Gehrig: '1B', Jeter: 'SS' }
-----------------------------------------
Ruth: P
Gehrig: 1B
Jeter: SS
$
```

Associative Arrays

- See **assocarrays2.js**

```
$ node assocarrays2.js
{ Ruth: 'RF', Gehrig: '1B', Jeter: 'SS' }
-----------------------------------------
{ Ruth: 'RF', Gehrig: '1B', Jeter: 'SS' }
-----------------------------------------
{ Ruth: 'P', Gehrig: '1B', Jeter: 'SS' }
-----------------------------------------
{ Ruth: 'P', Gehrig: '1B', Jeter: 'SS', Maris: 'RF' }
-----------------------------------------
{ Ruth: 'P', Gehrig: '1B', Jeter: 'SS' }
-----------------------------------------
Ruth: P
Gehrig: 1B
Jeter: SS
$
```

Associative Arrays

- The ***member access operator***
  - `object.property`
  - `property` must be a simple identifier
- The ***computed member access operator***
  - `object[property]`
  - `property` can be an arbitrary expression

Associative Arrays

To create and use an **associative array**:

```
aa = {'Ruth': 'RF', 'Gehrig': '1B', …};
…
… aa['Ruth'] …          // Computed member access operator
… aa['Ru' + 'th'] …     // Computed member access operator
… aa.Ruth …             // Member access operator
```

To create and use an **object**:

```
aa = {Ruth: 'RF', Gehrig: '1B', …};
…
… aa.Ruth …             // Member access operator
… aa['Ruth'] …          // Computed member access operator
… aa['Ru' + 'th'] …     // Computed member access operator
```

Associative Arrays

- See **concord.js**

```
$ cat hamlet.txt
to be
or not to be,
that is
the question.
$ node concord.js hamlet.txt
to: 2
be: 2
or: 1
not: 1
that: 1
is: 1
the: 1
question: 1
$
```

Agenda

- Arrays
- Associative arrays
- **Asynchronous processing: callbacks**

Async Processing: Callbacks

- Recall **linesort.js**

```
…
let data = fs.readFileSync(fileName, 'UTF-8');
…
```

  - `fs.readFileSync()`
    - Reads all data from the file **synchronously**
      - Execution does not proceed until `fs.readFileSync()` returns

Async Processing: Callbacks

- Recall **linesort.js** (cont.)
  - The more normal approach…
    - Especially when JavaScript is run in browsers…
  - `fs.readFile()`
    - Reads all data from the file **asynchronously**
    - Execution proceeds before `fs.readFile()` returns

Async Processing: Callbacks

- See **linesortcallback1.js**

```
$ cat hamlet.txt
to be
or not to be,
that is
the question.
$ node linesortcallback1.js hamlet.txt
Doing some work
Doing more work
or not to be,
that is
the question.
to be
$
```

Async Processing: Callbacks

- See **linesortcallback1.js** (cont.)

```
fs.readFile(fileName, 'UTF-8', sortWriteLines);
```

When executed by the JavaScript engine:

```
Dear Node.js,
   Please execute fs.readFile giving it
   fileName and 'UTF-8'.
   When you're finished, if no error occurred then
   please call sortWaitLines giving it null and the
   data. Or, if an error occurred, then please call
   sortWriteLines giving it an error object and null.
   Meanwhile, I'll proceed to execute the statements
   that follow.
                              Sincerely,
                              JS Engine
```

Async Processing: Callbacks

Asynchronous processing in JavaScript & Node.js:

<table>
<tr><td colspan="2">(1) JS Engine executes <code>the current function</code></td></tr>
<tr><td colspan="2">(2) While executing <code>the current function</code>, JS Engine calls <code>the slow function</code>, giving it <code>the callback function</code></td></tr>
<tr><td>(3) JS Engine continues executing <code>the current function</code> without preemption</td><td>(3) Node.js executes <code>the slow function</code></td></tr>
<tr><td>(4) JS Engine, when finished executing <code>the current function</code>, repeatedly examines JS Event Queue</td><td>(4) Node.js, when finished executing <code>the slow function</code>, adds a call of <code>the callback function</code> to JS Event Queue</td></tr>
<tr><td colspan="2">(5) JS Engine removes call of <code>the callback function</code> from JS Event Queue, and considers that function to be <code>the current function</code></td></tr>
<tr><td colspan="2">(6) Go to step (1)</td></tr>
</table>

Async Processing: Callbacks

- See **linesortcallback2.js**

```
$ cat hamlet.txt
to be
or not to be,
that is
the question.
$ node linesortcallback2.js hamlet.txt
Doing some work
Doing more work
or not to be,
that is
the question.
to be
$
```

Async Processing: Callbacks

- **Summary…**

- **Python** is:
  - Multithreaded
  - Preemptive
    - OS can context switch from one thread to another at any time

Async Processing: Callbacks

- **Summary (cont.)...**

- **JavaScript** is:
  - Event driven (not multithreaded)
  - Not preemptive
    - Each event handler executes to completion without interruption

Lecture Summary

- In this lecture we covered:
  - Arrays
  - Associative arrays
  - Asynchronous processing
    - Function callbacks

JavaScript Language Summary

- JavaScript language summary
  - C/Java-like syntax
  - Many versions
    - Transpilers used routinely
  - Dynamically typed
  - “Never fail” design philosophy

JavaScript Language Summary

- JavaScript language summary (cont.)
  - Unusual object model
    - Delegation to prototypes
    - Objects are associative arrays and vice versa
    - ES6 syntax is **much** different from pre-ES6
  - Event driven, not multi-threaded
  - Asynchronous computation is the norm

Commentary

- JavaScript is:
  - Difficult to learn
  - Difficult to use
  - Unavoidable in web applications
  - Worth learning

Lecture Series Summary

- In this lecture series we covered:
  - A subset of JavaScript…
  - That is appropriate for COS 333…
  - Through example programs
- See also:
  - **Appendix 1**: Asynchronous processing: promises
  - **Appendix 2**: Asynchronous processing: await

More Information

- The COS 333 *Lectures* web page provides references to supplementary information

Appendix 1: Asynchronous Processing: Promises

Async Processing: Promises

- **Problem**:
  - Programs using (many nested) callbacks can be difficult to understand

- **Solution…**

Async Processing: Promises

- ***Promises***
  - ES6 (2015)
  - “A promise is an object that represents the result of an asynchronous computation.”
    - *JavaScript: The Definitive guide*, David Flanagan

Async Processing: Promises

- Pattern to create a promise-style function (`someFunctionP`) from a callback-style function (`someFunction`):

```
function someFunctionP(args) {
   return new Promise(
      function(resolve, reject) {
         someFunction(args,
            function (err, data) {
               if (err)
                  reject(err);
               else
                  resolve(data);
            }
         );
      }
   );
}
```

Async Processing: Promises

- See **linesortpromises1.js**

```
$ cat hamlet.txt
to be
or not to be,
that is
the question.
$ node linesortpromises1.js hamlet.txt
Doing some work
Doing more work
or not to be,
that is
the question.
to be
$
```

Async Processing: Promises

- See **linesortpromises1.js** (cont.)

```
readFileP(fileName, 'UTF-8')
   .then(sortWriteLines)
   .catch(reportError);
```

When executed by the JavaScript engine:

```
Dear Node.js,
   Please execute readFileP (and thus fs.readFile)
   giving it fileName and 'UTF-8'.
   When you're finished, if no error occurred then
   please call sportWriteLines giving it the data.
   Or, if an error occurred, then please call
   reportError giving it an error object.
   Meanwhile, proceed to execute the statements
   that follow.
                                Sincerely,
                                JS Engine
```

Async Processing: Promises

- See **linesortpromises2.js**

```
$ cat hamlet.txt
to be
or not to be,
that is
the question.
$ node linesortpromises2.js hamlet.txt
Doing some work
Doing more work
or not to be,
that is
the question.
to be
$
```

Async Processing: Promises

- Promises commentary
  - Difficult to understand the implementation
  - Often not difficult to use
  - OK for Assignment 4
  - OK for project

Appendix 2:
Asynchronous Processing: await

Async Processing: await

- **Problem**:
  - Programs using (many nested) callbacks can be difficult to understand

- **Solution…**

Async Processing: await

- ***Promises***
  - ES6 (2015)
  - “A promise is an object that represents the result of an asynchronous computation.”
    - *JavaScript: The Definitive guide*, David Flanagan

Async Processing: await

- Pattern to create a promise-style function (`someFunctionP`) from a callback-style function (`someFunction`):

```
function someFunctionP(args) {
   return new Promise(
      function(resolve, reject) {
         someFunction(args,
            function (err, data) {
               if (err)
                  reject(err);
               else
                  resolve(data);
            }
         );
      }
   );
}
```

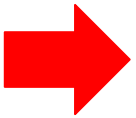
Async Processing: await

- ***Async*** and ***await***
  - ES8 (2017)
  - Allows a natural expression of callback logic
  - To some extent, isolates you from the (complicated) promise concept

Async Processing: await

- **Await and async**

```
function f(...) {
    initialCode
    f1(args)
        .then(f2)
        .catch(f3);
    finalCode
}
```



```
async function f(...) {
    initialCode
    try {
        let data = await f1(args);
        f2(data);
    }
    catch (ex) {
        f3(ex);
    }
    finalCode
}
```

Async Processing: await

- See **linesortawait1.js**

```
$ cat hamlet.txt
to be
or not to be,
that is
the question.
$ node linesortawait1.js hamlet.txt
Doing some work
Doing more work
or not to be,
that is
the question.
to be
$
```

Async Processing: await

- See **linesortawait1.js** (cont.)

```
let data = await readFileP(fileName, 'UTF-8');
```

```
Execute readFileP (and thus fs.readFile) giving it
fileName and 'UTF-8'.
Wait for fs.readFile to finish.
Meanwhile, proceed to execute the statements in the
calling function.
When fs.readFile finishes successfully, return the
data. Or, if an error occurs, throw an exception with
an err object. Either way proceed to execute the rest
of the statements in the async function.
```

Async Processing: await

- See **linesortawait2.js**

```
$ cat hamlet.txt
to be
or not to be,
that is
the question.
$ node linesortawait2.js hamlet.txt
Doing some work
Doing more work
or not to be,
that is
the question.
to be
$
```

Async Processing: await

- Await/async commentary
  - Very difficult to understand the implementation
  - Often easy to use
  - OK for Assignment 4
  - OK for project