The JavaScript Language (Part 4)

Copyright © 2026 by

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
- Asynchronous processing: promises

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
- Asynchronous processing: promises

Associative Arrays

- Associative arrays
  - Bad news: There are none!
  - Good news: Objects serve the purpose

Associative Arrays

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

Associative Arrays

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
```

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
```

Associative Arrays

- See **assocarrays.js**

```
$ node assocarrays.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 **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**
- Asynchronous processing: promises

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 **linesortcallback.js**

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

Async Processing: Callbacks

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

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

Informally translated to English:

```
Dear JavaScript,
   Asynchronously execute fs.readFile() giving it
   fileName, 'UTF-8', and sortWriteLines.
   When it's finished:
      If no error occurred then execute
      sortWriteLines() giving it null and the data.
      Or, if an error occurred, then execute
      sortWriteLines() giving it an error object and
      null.
   Meanwhile, proceed to execute the statements
   that follow.
```

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

- **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

Agenda

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

Async Processing: Promises

- Slightly more elaborate line sorting program, synchronous version…

Async Processing: Promises

- See **linesorttofile.js**

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

Async Processing: Promises

- Asynchronous version using callbacks…

Async Processing: Promises

- See **linesorttofilecallback.js**

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

Async Processing: Promises

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

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

Informally translated to English:

```
Dear JavaScript,
   Asynchronously execute fs.readFile() giving it
   fileName, 'UTF-8', and sortWriteLines.
   When it's finished:
      If no error occurred then execute
      sortWriteLines() giving it null and the data.
      Or, if an error occurred, then execute
      sortWriteLines() giving it an error object and
      null.
   Meanwhile, proceed to execute the statements
   that follow.
```

Async Processing: Promises

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

```
fs.writeFile(fileName, outData, 'UTF-8', finish);
```

Informally translated to English:

```
Dear JavaScript,
   Asynchronously execute fs.writeFile() giving it
   fileName, uutData, 'UTF-8', and finish.
   When it's finished:
      If no error occurred then execute
      finish() giving it null.
      Or, if an error occurred, then execute
      finish() giving it an error object.
   Meanwhile, proceed to execute the statements
   that follow (but there are none).
```

Async Processing: Promises

- **Problem:**
  - Callback calls callback calls callback …
  - Difficult to write and read

Async Processing: Promises

- **Solution:** ***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

- **Callback mechanism**
  - Your code calls *slow function*, giving it your *callback function*
  - *Slow function* does work
  - When work is finished, *slow function* calls your *callback function*
- **Promise mechanism**
  - Your code calls *slow function*
  - Slow function returns *Promise object*
  - Your code registers *callback function(s)* on *Promise object*
  - *Slow function* does work
  - When work is finished, *Promise object* calls your *callback function(s)*

Async Processing: Promises

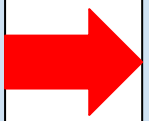
Callback pattern

```
function f2(data) {...}

function f3(err) {...}

function cb(err, data) {
   if (err)
      f3(err);
   else
      f2(data);
}

f1(args, cb);
```



Promises pattern

```
function f2(data) {...}

function f3(err) {...}

f1p(args).then(f2).catch(f3);
```

`f1` is a callback-style function

`f1p` is a promise-style version of `f1`

Async Processing: Promises

- See **linesorttofilepromises.js**

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

Async Processing: Promises

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

```
fs.promises.readFile(process.argv[2], 'UTF-8')
    .then(sortLines)
    .then(writeFileAdapt)
    .then(finish)
    .catch(reportError);
```

Informally translated to English…

Async Processing: Promises

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

```
Dear JavaScript,
   Asynchronously execute fs.promises.readFile() giving
   it fileName and 'UTF-8'.
   When it's finished, execute sortLines() giving
   it the data returned by the previous function.
   When it's finished, execute fs.promises.writeFile()*
   giving it the data returned by the previous function.
   When it's finished, execute finish() giving it the
   data (nothing) returned by the previous function.
   If anything goes wrong, execute reportError() giving
   it an error object.
   Meanwhile, proceed to execute the statements
   that follow.
```

* Adapter required

Async Processing: Promises

- **Solution:** ***`await`*** and ***`async`*** keywords
  - ES8 (2017)
  - Allows a natural expression of asynchronous logic
  - To some extent, isolates you from the (complicated) promise concept

Async Processing: Promises

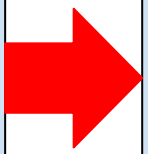
Callback pattern

```
function f2(data) {...}

function f3(err) {...}

function cb(err, data) {
   if (err)
      f3(err);
   else
      f2(data);
}

f1(args, cb);
```



Await/async pattern

```
function f2(data) {...}

function f3(err) {...}

async function f(args) {
   try {
      let data = await f1p(args);
      f2(data);
   }
   catch (err) {
      f3(err);
   }
}

f(args);
```

`f1` is a callback-style function

`f1p` is a promise-style version of `f1`

Async Processing: Promises

- See **linesorttofileawait.js**

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

Aside

- 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);
            }
         );
      }
   );
}
```

Aside

- See **linesorttofileawait2.js**

```
$ cat hamlet.txt
to be
or not to be,
that is
the question.
$ node linesorttofileawait2.js hamlet.txt hamletb.txt
Doing more work
Finished
$ cat hamletb.txt
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

Lecture Summary

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

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
    - Via callbacks
    - Via promises

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

More Information

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