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 file1
to be
or not to be
that is
the question
$ node linesort.js file1
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
$
```

Uses `object.prop` instead of `assocarray[key]`

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 file1
to be
or not to be
that is
the question
$ node concord.js file1
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');
let concordance = createConcordance(data);
…
```

  - `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…
  - `fs.readFile()`
    - Reads all data from the file **asynchronously**
    - Execution proceeds before `fs.readFile()` returns

Async Processing: Callbacks

- See **linesortcallback.js**

```
$ cat file1
to be
or not to be
that is
the question
$ node linesortcallback.js file1
Doing other work
or not to be
that is
the question
to be
$
```

Async Processing: Callbacks

Node.js

JS Engine

```
…
function sortWriteLines(
    err, data) {
    JS statements;
}
…
function main() {
    JS statements;
    fs.readFile(fileName,
        'UTF-8',
        sortWriteLines);
    JS statements;
}
```

JS Event Queue

```
fs.readFile(file,enc,cb) {
    C++ statements;
    Enqueue cb(err, data)
}
```

Async Processing: Callbacks

Node.js

JS Engine

```
…
function sortWriteLines(
    err, data) {
    JS statements;
}
…
function main() {
    JS statements;
    fs.readFile(fileName,
        'UTF-8',
        sortWriteLines);
    JS statements;
}
```

JS Event Queue

```
fs.readFile(file,enc,cb) {
    C++ statements;
    Enqueue cb(err, data)
}
```

Async Processing: Callbacks

Node.js

JS Engine

```
…
function sortWriteLines(
    err, data) {
    JS statements;
}
…
function main() {
    JS statements;
    fs.readFile(fileName,
        'UTF-8',
        sortWriteLines);
    JS statements;
}
```

JS Event Queue

```
fs.readFile(file,enc,cb) {
    C++ statements;
    Enqueue cb(err, data)
}
```

Async Processing: Callbacks

Node.js

JS Engine

```
…
function sortWriteLines(
    err, data) {
    JS statements;
}
…
function main() {
    JS statements;
    fs.readFile(fileName,
        'UTF-8',
        sortWriteLines);
    JS statements;
}
```

JS Event Queue

```
fs.readFile(file,enc,cb) {
    C++ statements;
    Enqueue cb(err, data)
}
```

Async Processing: Callbacks

Node.js

JS Engine

```
…
function sortWriteLines(
    err, data) {
    JS statements;
}
…
function main() {
    JS statements;
    fs.readFile(fileName,
        'UTF-8',
        sortWriteLines);
    JS statements;
}
```

JS Event Queue

```
fs.readFile(file,enc,cb) {
    C++ statements;
    Enqueue cb(err, data)
}
```

Async Processing: Callbacks

Node.js

JS Engine

```
…
function sortWriteLines(
    err, data) {
    JS statements;
}
…
function main() {
    JS statements;
    fs.readFile(fileName,
        'UTF-8',
        sortWriteLines);
    JS statements;
}
```

JS Event Queue

```
fs.readFile(file,enc,cb) {
    C++ statements;
    Enqueue cb(err, data)
}
```

Async Processing: Callbacks

Node.js

JS Engine

```
…
function sortWriteLines(
    err, data) {
    JS statements;
}
…
function main() {
    JS statements;
    fs.readFile(fileName,
        'UTF-8',
        sortWriteLines);
    JS statements;
}
```

JS Event Queue

sortWriteLines(
err, data)

```
fs.readFile(file,enc,cb) {
    C++ statements;
    Enqueue cb(err, data)
}
```

Async Processing: Callbacks

Node.js

JS Engine

```
…
function sortWriteLines(
    err, data) {
    JS statements;
}
…
function main() {
    JS statements;
    fs.readFile(fileName,
        'UTF-8',
        sortWriteLines);
    JS statements;
}
```

JS Event Queue

```
fs.readFile(file,enc,cb) {
    C++ statements;
    Enqueue cb(err, data)
}
```

Async Processing: Callbacks

Summary of the cycle:

<table>
<tr><td colspan="2">(1) JS Engine calls <code>f1(args)</code></td></tr>
<tr><td colspan="2">(2) While executing <code>f1(args)</code>, JS Engine calls <code>slowfn()</code>, giving it <code>f2</code></td></tr>
<tr><td>(3) JS Engine continues executing <code>f1(args)</code></td><td>(3) Node.js executes <code>slowfn()</code></td></tr>
<tr><td>(4) JS Engine, when finished executing <code>f1(args)</code>, repeatedly examines Queue</td><td>(4) Node.js, when finished executing <code>slowfn()</code>, adds call of <code>f2(args)</code> to Queue</td></tr>
<tr><td colspan="2">(5) JS Engine removes call of <code>f2(args)</code> from Queue
Let <code>f1</code> = <code>f2</code></td></tr>
<tr><td colspan="2">(6) Go to step (1)</td></tr>
</table>

Summary

- **Python** is:
  - Multithreaded
  - Preemptive
    - OS can context switch at any time

Summary

- **JavaScript** is:
  - Event driven (not multithreaded)
  - Not preemptive
    - Executes each event-handling function to completion without interruption
      - So (esp in browsers) event-handling functions must consume little time
      - So event-handling functions should ask container (browser or node.js) to execute slow functions asynchronously

Summary

- We have covered:
  - Arrays
  - Associative arrays
  - Asynchronous processing
    - Function callbacks

Summary

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

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

Summary

- We have 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

Appendix 1: Asynchronous Processing: Promises

Async Processing: Promises

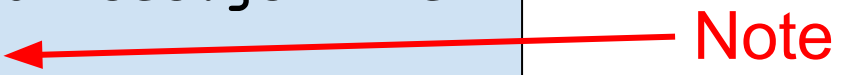
- **Problem**:
  - Programs using (many) callbacks can be difficult to understand

- **Solution…**

Async Processing: Promises

- See **linesortpromises.js**

```
$ cat file1
to be
or not to be
that is
the question
$ node linesortpromises.js file1
Doing other work
or not to be
that is
the question
to be
$
```



Async Processing: Promises

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

```
let promise1 = fs.promises.readFile(fileName, 'UTF-8')
// Let promise1 represent err and data, the future
// result of reading from fileName

let promise2 = promise1.then(sortLines)
// Let promise2 represent lines, the future result of
// calling sortLines(data)

let promise3 = promise2.then(writeLines)
// Let promise3 represent null, the future result
// of calling writeLines(lines)

promise3.catch(reportError);
// If err is not null, then call reportError(err)
```

Async Processing: Promises

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

```
fs.promises.readFile(
    fileName, 'UTF-8')
    .then(sortLines)
    .then(writeLines)
    .catch(reportError);
```

Dear Node.js:

Call `fs.promises.readFile` asynchronously, passing it `fileName`.

Then, after `readFile` is finished, call `sortLines`, passing it the value returned by `readFile` (i.e., `data`).

Then, after `sortLines` is finished, call `writeLines`, passing it the value returned by `sortLines` (i.e., `lines`).

If `readFile`, `sortLines`, or `writeLines` throws an exception, then call `reportError`, passing it the exception object.

Async Processing: Promises

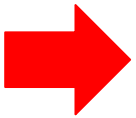
- Promises commentary
  - Difficult to understand the implementation
  - (Usually) easy to use

Appendix 2:
Asynchronous Processing: await

Async Processing: await

- **Await and async**

```
function f(...) {
    initial code
    f1(args)
        .then(f2)
        .then(f3)
        .catch(f4);
    final code
}
```



```
async function helper(args) {
    try {
        let data1 = await f1(args);
        let data2 = f2(data1);
        f3(data2);
    }
    catch (ex) {
        f4(ex);
    }
}

function f(...) {
    initial code
    helper(args);
    final code
}
```

Async Processing: await

- See **linesortawait.js**

```
$ cat file1
to be
or not to be
that is
the question
$ node linesortawait.js file1
Doing other work
or not to be
that is
the question
to be
$
```