

- History.
- A practical example illustrating various features.
- Primitive types.

Brief History

- Written by Brendan Eich within 10 days in May, 1995 at Netscape Communications. Name chosen based on marketing considerations to cash in on the popularity of Java.
- Microsoft released `jscript` in 1996. Incompatibilities introduced ("embrace and extend").
- Standardized as `EcmaScript` in June 1997 (ECMA-262).
- JavaScript got a bad reputation and was regarded as a poor programming language used for doing trivial things in the browser. Complexities caused by browser incompatibilities and the browser **Document Object Model (DOM)** were blamed on the language.
- Changed with the emergence of **Asynchronous JavaScript with Xml** in 2005.

Brief History Continued

- **JavaScript Object Notation (JSON)** popularized by Douglas Crockford emerged as a popular alternative to XML as a specification for data interchange between heterogeneous systems.
- Renaissance in js development. Browser incompatibilities and DOM complexities hidden by the use of libraries like prototype, jquery and dojo.
- Node.js released by Ryan Dahl in 2009. Popularized the use of js on the server.
- Succession of different ECMA standards: es 3, es 5. Currently, evolving as an "evergreen" language with standard updates being released yearly: es 2015 ... 2017.
- Allows use of a single programming language across the entire web stack. Most popular programming language in terms of deployments.

Language Overview

- Object-based scripting language.
- Also a functional programming language.
- Dynamically typed: variables are untyped, but values have types. Permits the use of **duck typing**.
- Initially interpreted, now compiled using techniques like runtime compilation.
- Possible to evaluate strings representing code at runtime using `eval()`.
- Allows programming using multiple paradigms: procedural, object-oriented, functional.
- Borrows concepts from Scheme, Perl and Self.
- Standard library is highly asynchronous.

Two main platforms on which JavaScript runs:

- 1 **Browser:** Platform provides interfaces to browser technologies like the *Document Object Model* DOM and storage.
- 2 **Server:** Exemplified by nodejs. Platform provides access to filesystem, processes, etc.

Example Program

- Non-trivial program to grep one-or-more files.
- Command-line nodejs program.
- Invoked with arguments specifying regex and one-or-more files.
- Both synchronous and asynchronous versions.

Edited Log of Operation

```
$ ./sync-grep.js
usage: sync-grep.js REGEX FILE...
$ ./sync-grep.js '\ ' sync-grep.js
bad regex \: SyntaxError: Invalid regular expression: /\/: \ a
$ ./sync-grep.js '\[\d\]' sync-grep.js
sync-grep.js:19:    path.basename(process.argv[1])); //@basen
sync-grep.js:23:    regex = new RegExp(process.argv[2]);
sync-grep.js:26:    abort("bad regex %s: %s", process.argv[2]
$ ./sync-grep.js '\[\d\]' sync-grep.js x
sync-grep.js:19:    path.basename(process.argv[1])); //@basen
sync-grep.js:23:    regex = new RegExp(process.argv[2]);
sync-grep.js:26:    abort("bad regex %s: %s", process.argv[2]
cannot read x: Error: ENOENT: no such file or directory, open
$ ./async-grep.js '\[\d\]' sync-grep.js #run async grep inste
sync-grep.js:19:    path.basename(process.argv[1])); //@basen
sync-grep.js:23:    regex = new RegExp(process.argv[2]);
sync-grep.js:26:    abort("bad regex %s: %s", process.argv[2]
```

Code for Synchronous Grep

In `<code/sync-grep.js>`:

```
#!/usr/bin/env nodejs
```

```
'use strict'; //@strict
```

```
const fs = require('fs'); //@modules
```

```
const path = require('path');
```

```
const process = require('process');
```

```
function abort() {
```

```
    //@complex
```

```
    console.log(...Array.prototype.slice.call(arguments));
```

```
    process.exit(1);
```

```
}
```


Commentary on Previous Code

- First Line** On Unix systems, a line starting with *hash-bang* `#!` specifies running the file through an interpreter. In this case, the interpreter is the `env` program which runs its argument `nodejs` with a specified environment. In this case no additional environment is specified; the `env` program is merely used to find `nodejs` on the user's `PATH`.
- @strict** The `'use strict'` directive makes JavaScript flag some problematic constructs. Note that syntactically the directive is merely a string.
- @modules** Inclusion of standard modules. The `require()` returns an object which is named by being assigned to a `const` identifier.

Commentary on Previous Code Continued

There is a lot worth noting in the single line following `@complex`:

- `console.error()` (and `console.log()`) take printf-style parameters; i.e. a message which may contain % format-specifiers followed by args for the format-specifiers. So for example, `console.log('hello %s', 'world')` would print `hello world`.
- The pseudo-variable `arguments` always contains the arguments of the current function. This acts like an `Array` in some contexts but is not a real `Array`.
- The `Array.prototype.slice()` is used to convert arguments to a true array.
- The `...` spread operator spreads the true arguments array into the parameters for `console.error()`.

Code for Synchronous Grep Continued

```
function main() {  
  if (process.argv.length < 4) { //@argv  
    abort('usage: %s REGEX FILE...',  
          path.basename(process.argv[1])); //@basename  
  }  
  let regex; //@let  
  try {  
    regex = new RegExp(process.argv[2]);  
  }  
  catch(err) {  
    abort("bad regex %s: %s", process.argv[2], err);  
  }  
  grep(regex, process.argv.slice(3));  
}
```

Commentary on Previous Code

@argv `process.argv[]` contains the program's command-line arguments. `argv[0]` contains the path to the interpreter, i.e. the path to the `nodejs` executable; `argv[1]` contains the path of the JavaScript file being run, i.e. the path to `sync-grep.js` file. The remaining arguments are the actual arguments provided to the program. In this case, a REGEX and at least one FILE name argument are required.

@let The modern way of declaring variables in JavaScript is using `let`. Does not have the surprises associated with the older `var` declarations.

@basename Returns the last component of its path parameter.

Code for Synchronous Grep Continued

```
function grep(regex, files) {  
  for (const file of files) { //@for-of  
    try {  
      const contents = fs.readFileSync(file).toString();  
      let lineN = 1;  
      for (const line of contents.split('\n')) {  
        if (line.match(regex)) { //@regex  
          console.log("%s:%i: %s", file, lineN, line);  
        }  
        lineN++;  
      }  
    }  
  }  
}
```

Code for Synchronous Grep Continued

```
    catch (err) { //@exception
        console.error("cannot read %s: %s", file, err);
    }
} //for
} //grep()

main();
```

Commentary on Previous Code

@for-of The modern way to loop through elements of an array in order is **for** (variable **of** array) { ... }

@regex `line.match(regex)` returns "true" iff some contents in `line` matched the regular expression `regex`.

@exception The catch will trigger if an exception occurs. JavaScript automatically scopes the `err` variable in `catch(err)` to only the catch-block.

Asynchronous Code

Most modern computer systems allow execution of code while waiting for external events like I/O completion. Some alternatives:

- ❶ Blocking synchronous I/O with explicit concurrency constructs like threads or processes. Problems with synchronizing access to shared data.
- ❷ Asynchronous I/O with a single thread of execution with an event loop which runs event handlers when events occur. Each event handler **runs to completion** before the next event handler is run by the event loop. Reduces synchronization problems; no synchronization problems while an event handler is running but need to handle synchronization between event handlers.

JavaScript prefers (2).

Asynchronous Grep

- Only change from code for synchronous grep are the `abort()` and `grep()` functions; rest of code is identical and not discussed further.
- When a file is open'd, it is passed a callback event handler which should handle both success and failure of the open. The `open()` call will return immediately before the file is open'd; the event handler will be run when the status of the file open is known.
- The code uses nodejs's `readline` module. Normally used for reading from a terminal but can also be used to read from files.
- The code uses explicit callback event handlers for `readline` completing reading of a line or encountering an error.

Code for abort()

In `<code/async-grep.js>`:

```
function abort(...args) { //@rest-args
  console.log(...args); //@spread-args
  process.exit(1);
}
```

@rest-args If last formal parameter is prefixed with a `...`, then that parameter becomes an array. In the example, `...args` will set `args` to an array containing all the arguments to `abort()`.

@spread-args If a variable which is an array is prefixed with a `...` in a function call, then that array gets spread into the call such that each element is a separate argument in the call.

Code for Asynchronous Grep

```
function grep(regex, files) {  
  for (const file of files) {  
    fs.open(file, 'r', function(err, fd) { //@open  
      if (err) {  
        console.error("cannot read file %s: %s",  
                      file, err);  
      }  
      else {  
        const rl = readline.createInterface({  
          input: fs.createReadStream(file, {fd: fd}),  
          crlfDelay: Infinity  
        });  
      }  
    }  
  }  
}
```

Code for Asynchronous Grep Continued

```
let lineN = 1;
rl.on('line', (line) => { //@line
  if (line.match(regex)) {
    console.log("%s:%i: %s",
                file, lineN, line);
  }
  lineN++; //@closure
});
rl.on('error', function() { //@error
  console.error("cannot read %s: %s",
                file, err);
});
}
}); //fs.open()
} //for
```

Commentary on Previous Code

- @open** The callback takes two arguments: an error object `err` which is "true" if the open fails and a file descriptor `fd` which will contain a descriptor for the file if the `open()` succeeds. Note the use of an anonymous function to specify the callback.
- @line** The 'line' event fires for each line read and the event handler is run. Note the use of JavaScript's fat-arrow notation to specify the callback.
- @closure** The `lineN++` within the callback is referring to the `lineN` variable defined outside the callback. It is able to do so because JavaScript supports *closures*.
- @error** The 'error' event fires if an error is occurred while reading a line.