CS/COE 1520

pitt.edu/~ach54/cs1520

Advanced JavaScript and ECMAScript

ECMAScript and JavaScript

- ECMAScript is based on JavaScript, and JavaScript is based on ECMAScript
- ...

First of all, what is ECMA?

- Ecma International
  - Formerly the European Computer Manufacturers Association (ECMA)
  - A standards organization similar to ANSI or ISO

What is ECMAScript?

- Ecma standard ECMA-262
- A specification for implementing a scripting language
- Created to standardize the scripting language developed out of Netscape by Brendan Eich
- ECMA-262 tells you how to implement a scripting language
  - JavaScript documentation tells you how to use an implementation of ECMA-262

A little bit of history

- 1997: 1st edition of ECMAScript published
- 1998: 2nd edition published
- 1999: 3rd edition published
- 2007: Work on 4th edition begins
  - Due to political infighting in the working group, the contributions of the 4th edition are almost completely abandoned
- 2009: 5th edition is published
- 2015: 6th edition (aka ECMAScript 2015) published
- 2016: 7th edition (aka ECMAScript 2016) published
- 2017: 8th edition (aka ECMAScript 2017) published
- 2018: 9th edition (aka ECMAScript 2018) published
- 2019: 10th edition (aka ECMAScript 2019) published

Falling short

- How much of each standard is supported varies

Browser	ES5 Features	ES6 Features	ES7 Features	ES8+ Features
Microsoft Edge	100%	96%	100%	54%
Firefox	100%	98%	100%	77%
Google Chrome	100%	98%	100%	93%
Safari	97%	99%	100%	83%

Above and beyond

- Some browsers will support JavaScript features not outlined in ECMA-262

Experimental new features

The following features are already implemented, but only available in the Firefox Nightly channel and not yet included in a draft edition of an ECMAScript specification.

Additions to the `ArrayBuffer` object

- `ArrayBuffer.transfer()` (spec)

New TypedObject objects

- Based on Typed Objects draft, and exposed via a global `TypedObject` object, e.g. `TypedObject.StructType` & `TypedObject.ArrayType`. This feature is non-standard and not documented.

New SIMD objects

- SIMD specification draft and polyfill

New Shared Memory objects

- `SharedArrayBuffer`
- `Atomics`

ECMAScript features

- Ed. 1-3 describe the JavaScript we all know and love
  - E.g., regex support was proposed in 3rd edition
- 5th has been widely supported for quite some time
  - Several features are things that we have already been using
    - E.g., library JSON support
  - One big feature that we haven't discussed is *strict* mode

Strict mode vs. sloppy mode

- Up until now, we have been evaluating JavaScript in what is know as *sloppy* mode
  - Note this is not an official designation, but the term is used often to contrast with strict mode
- Using strict mode changes how your JavaScript code is evaluated and executed, primarily it:
  - Eliminates some JavaScript silent errors by changing them to thrown errors
  - Fixes mistakes that make it difficult for JavaScript engines to perform optimizations
    - Hence, strict mode code can sometimes be made to run faster than identical code that's not run in strict mode
  - Prohibits some syntax likely to be defined in future versions of ECMAScript

Enabling strict mode

- Either:
  - `"use strict";`

    or
  - `'use strict';`
- Appears before any other statement
- If placed before any other statement in a script, the entire script is run using strict mode
- Can also be used to set individual functions to be evaluated in strict mode by placing it before any other statements in a function

Raises errors on variable name typos

- The following will raise a `ReferenceError`:
  -
    ```
    var myVar = 12;

    mVar = 13;
    ```

Invalid assignments

- All of the following will throw a `TypeError`:
  - `var NaN = 13;`
  - `true.false = "TypeError";`
  - `"This".willbe = "A TypeError";`

No duplicate function arguments

```
function foo(a, b, a, a) {
    console.log(a);
    console.log(b);
    console.log(a);
    console.log(a);
}
foo(1, 2, 3, 4);
```

The `with` operator is prohibited

- cool

Paving the way for future ECMAScripts

- The following are treated as reserved words in strict mode:
  - `implements`
  - `interface`
  - `let`
  - `package`
  - `private`
  - `protected`
  - `public`
  - `static`
  - `yield`

Note that this is only a brief look at strict

- There are many more changes that are made to how using strict mode with affect the running of your JavaScript code

strict scripts vs strict functions

- Be very cautious with making a script strict…
  - Consider concatenating two scripts together:
    - sloppy_script + strict_script
      - Result will be sloppily evaluated
      - The `"use strict";` from the strict_script will no longer come before the first statement
    - strict_script + sloppy_script
      - Result will be treated as strict!
      - Could result in errors from strict evaluation of sloppy code!

ECMAScript 6 (2015)

- A huge update to the language
- Alot of new language features were added
- We'll review some of them here

Arrow function

- Probably already saw these in the `map()`, `reduce()`, and `filter()` documentation
- Succinct, anonymous function definitions:
  - `a => a + 1`
  - `(a, b, c) => { return a + b + c; }`
  - `(a) => { return a + 1; }`
  - `(a, b, c) => { console.log(a); console.log(b); console.log(c); }`

Python detour: lambda expressions

- `lambda x: x ** 2`
- `lambda x, y: x + y`

- `lambda PARAMS: EXPR`

Template strings

- Defined with backticks (` not ' or ")
- `Can span
  multiple lines`
- var a = 1;
  var b = 2;
  var s = `Can reference vars like ${a} and ${b}`;
- var t = `Can include expressions line ${a + b}`;

`let` and `const`

- Both alternatives to var for variable declaration
- `const` variables cannot be reassigned
  - Note that this does not mean values are immutable…
- `let` allows you to declare variables limited in scope to the block, statement, or expression where they're used
-
```
var a = 1;
var b = 2;
if (a === 1) {
   var a = 11;
   let b = 22;
   console.log(a);  // 11
   console.log(b);  // 22
}
console.log(a); // 11
console.log(b); // 2
```

`for … of` and iterables

- ES6 instroduces iterators, iterables, and a for loop syntax for iterables
-
```
let iterable = [10, 20, 30];
for (let value of iterable) {
   value += 1;
   console.log(value);
}
```

`for … of` vs `for … in`

- Both are valid in JavaScript
- `for … in` iterates through the enumerable properties of an object in an arbitrary order
- `for … of` iterates over an iterable object

Generators

```
function* fib() {
    let p = 0, c = 1;
    while (true) {
        yield c;
        let temp = p;
        p = c;
        c = p + temp;
    }
}
```

Default parameters

- ```
  function foo(x, y=4) {
     return x + y;
  }
  f(3) == 7  // true
  ```
  ```

Rest parameters

```
function bar(x, ...y) {
   for (let i of y) {
      console.log(i);
   }
}
bar("not logged", "first", "last");
```

Spread operator

- ```
  function baz(a, b, c) {
     console.log(a);
     console.log(b);
     console.log(c);
  }
  baz(...[1, 2, 3]);
  ```
  ```

this

- We've seen `this` before
  - When a function is called as a constructor (i.e., after `new`), `this` refers to the object being constructed
  - E.g.:
    -
```
function TV(brand, size, injacks, outjacks) {
    this.brand = brand;
    this.size = size;
    this.jacks = new Object();
    this.jacks.input = injacks;
    this.jacks.output = outjacks;
}
```

Similar use in object methods

```
function show_properties() {
    document.write("Here is your TV: <br />");
    document.write("Brand: ", this.brand,"<br />");
    document.write("Size: ", this.size, "<br />");
    document.write("Input Jacks: ");
    document.write(this.jacks.input, "<br />");
    document.write("Output Jacks: ");
    document.write(this.jacks.output, "<br />");
}
my_tv.display = show_properties;
```

`this` in an event handler

- When a function is used as an event handler, its `this` is set to the element the event fired from
  -
```
function makeRed() {
    this.style.backgroundColor = "#FF0000";
}
let d = document.getElementById("theDiv");
d.addEventListener("click", makeRed, true);
```

`this` outside of a function

- Outside of any function, `this` refers to the global object
  -
```
console.log(this === window);  // true
a = 37;
console.log(window.a);         // 37
this.b = "MDN";
console.log(window.b)          // "MDN"
console.log(b)                 // "MDN"
```

`this` inside a regular function call

- The value of this depends on strict vs sloppy evaluation
  - ```
    function foo() {
        return this;  // === window
    }
    ```
  - ```
    function bar() {
        "use strict";
        return this;  // === undefined
    }
    ```

`call()` and `apply()`

- Both are methods of function objects
  - Set the value of this to be used in a call to the function
    -
```
function add(b, c) {
    return this.a + b + c;
}
var o = {a: 1};
add.call(o, 2, 3);
add.apply(o, [2, 3]);
```

bind()

- `bind()` creates a new function with a set value for `this`
  -
```
function test() {
    return this.a;
}
var f = test.bind({a: "foo"});
var b = f.bind({a: "bar"});
var o = {a: 42, test: test, f: f, b: b};
console.log(o.a, o.test(), o.f(), o.b());
```

`this` in arrow functions

- The value of `this` in an arrow function is set to the value of `this` present in the context that defines the arrow function
  -
```
var obj = {bar: function() {
                var x = (() => this);
                return x;
            }
        };
var fn = obj.bar();
fn()     // ???
var fn2 = obj.bar;
fn2()()  // ???
```

Consider the following code

```
try {
    const result = doSomething(initArgs);
    const newResult = doSomethingElse(result);
    const endResult = doThirdThing(newResult);
    console.log(`Got the end result: ${endResult}`);
} catch(error) {
    failureCallback(error);
}
```

Async calls lead to the following:

- First, redefine all function headers:
  - `function doSomething(args, onSuccess, onFail);`
  - `function doSomethingElse(args, onSuccess, onFail);`
  - `function doThirdThing(args, onSuccess, onFail);`
- Each will need to set `onSuccess` as the handler for the success of the asynchronous event, and `onFail` for the handler of the failure of the asynchronous event
- Next…

The callback "pyramid of doom"

```
doSomething(intArgs, function(result) {
    doSomethingElse(result, function(newResult) {
        doThirdThing(newResult, function(endResult) {
            console.log('End result: ' + endResult);
        }, onFail);
    }, onFail);
}, onFail);
```

What a mess!

- This motivates the need for *Promises*
  - Promises represent the eventual completion or failure of an asynchronous event
  - A Promise acts as a proxy for a value that is initially unknown
    - Because we're still waiting to determine its value
  - Essentially an IOU
    - Whenever the asynchronous event "pays up" a call to an onSuccess handler can be made with the result
    - If the asynchronous event fails to produce a valid value, an onFail handler can be called

If we modify doSomething to use promises...

```
doSomethingProm(initArgs).then(function(result) {
    return doSomethingElse(result);
})
.then(function(newResult) {
    return doThirdThing(newResult);
})
.then(function(endResult) {
    console.log("Got the end result: " + endResult);
})
.catch(onFail);
```

Even clearer with the use of arrow functions

- ```
  doSomethingProm(initArgs)
  .then(result => doSomethingElse(result))
  .then(newResult => doThirdThing(newResult))
  .then(endResult => {
     console.log("Got the end result: " + endResult);
  })
  .catch(failureCallback);
  ```
  ```

`doSomethingProm()` returns a Promise

- Not the result of the original `doSomething()`
  - But an IOU for the result of `doSomething()`
- `.then()` and `.catch()` are methods of Promises
  - `.then(onSuccess, onFail)`
    - Arguments to `.then()` are optional (as we've seen), but we can supply both an `onSuccess` and an `onFail`
  - `.catch(onFail)`
    - Similar to `.then()`, but without the option for `onSuccess`
    - Like calling `.then()` with only the 2nd parameter
  - Each of these also return Promises

Promise chains

- ```
  doSomethingProm(initArgs)
  .then(result => doSomethingElse(result))
  .then(newResult => doThirdThing(newResult))
  .then(endResult => {
     console.log("Got the end result: " + endResult);
  })
  .catch(failureCallback);
  ```
  ```

Promise chains

- ```
  p1 = doSomethingProm(initArgs)
  p2 = p1.then(result => doSomethingElse(result))
  p3 = p2.then(newResult => doThirdThing(newResult))
  p4 = p3.then(endResult => {
     console.log("Got the end result: " + endResult);
  })
  p5 = p4.catch(failureCallback);
  ```
  ```

Error propagation

- If an error is encountered, you end up skipping down the chain until an onFail handler is found, e.g.:
  - A .then() with 2 arguments
  - A .catch()
- Note that you can continue to chain after a .catch()
  - Assume p, a, b, and c are functions, and p returns a promise
    - `p().then(a).catch(b).then(c)`
      - If an error is thrown in a, both b and c will still execute
        - Why?

fetch

- An API to perform HTTP requests using Promises
-
```
fetch("http://example.com/movies.json")
.then(function(response) {
   return response.json();
})
.then(function(myJSON) {
   console.log(myJSON);
});
```

Using `fetch`

- A failing HTTP status (e.g., 404, 500), will not cause the promise to reject!
  - The HTTP request still completed
  - Only rejects on network errors
  - Need to check the `ok` attribute of the response
    - or the `status` attribute to get the actual code
    - `ok` attribute is boolean
      - `true` for status 200-299
      - `false` otherwise

A post example

```
return fetch(url, {
    method: "POST",
    credentials: "same-origin",
    headers: {
        "Content-Type": "application/x-www-form-urlencoded"
    },
    body: "key1=val1&key2=val2"
})
.then(response => response.json())
.catch(error => console.error(`Fetch Error =\n`, error));
```

Common Promise mistakes

- ```
  doSomething().then(function(result) {
     doSomethingElse(result)
     .then(newResult => doThirdThing(newResult));
  }).then(() => doFourthThing());
  ```
  ```

Back to other ES6 contributions...

- Tail call optimization
  -
```
function factorial(n, acc = 1) {
    'use strict';
    if (n <= 1) return acc;
    return factorial(n - 1, n * acc);
}
```

Note that browser support is required to use ES6

- Widely supported now
  - But not soon after launch
    - Solution?
      - *Transpile* ES6 code into ES5 code

ECMAScript 7 (2016)

- Added an exponentiation operator
  - `**`
- And an `includes()` method to arrays