Modern Javascript in GNOME

Philip Chimento (ptomato, @therealptomato)
GUADEC, July 30, 2017

Who is this talk for

Those who write code for GNOME apps written in Javascript

Those who write code for GNOME shell and GNOME shell extensions

Anyone who tried writing Javascript in GNOME but got sick of it

Javascript's shady reputation

- Truthy and falsy values
- Subclassing is terrible
- Variable hoisting
- Having both `undefined` and `null`
- `==` is weird
- `for...in` is weird
- `this` is weird

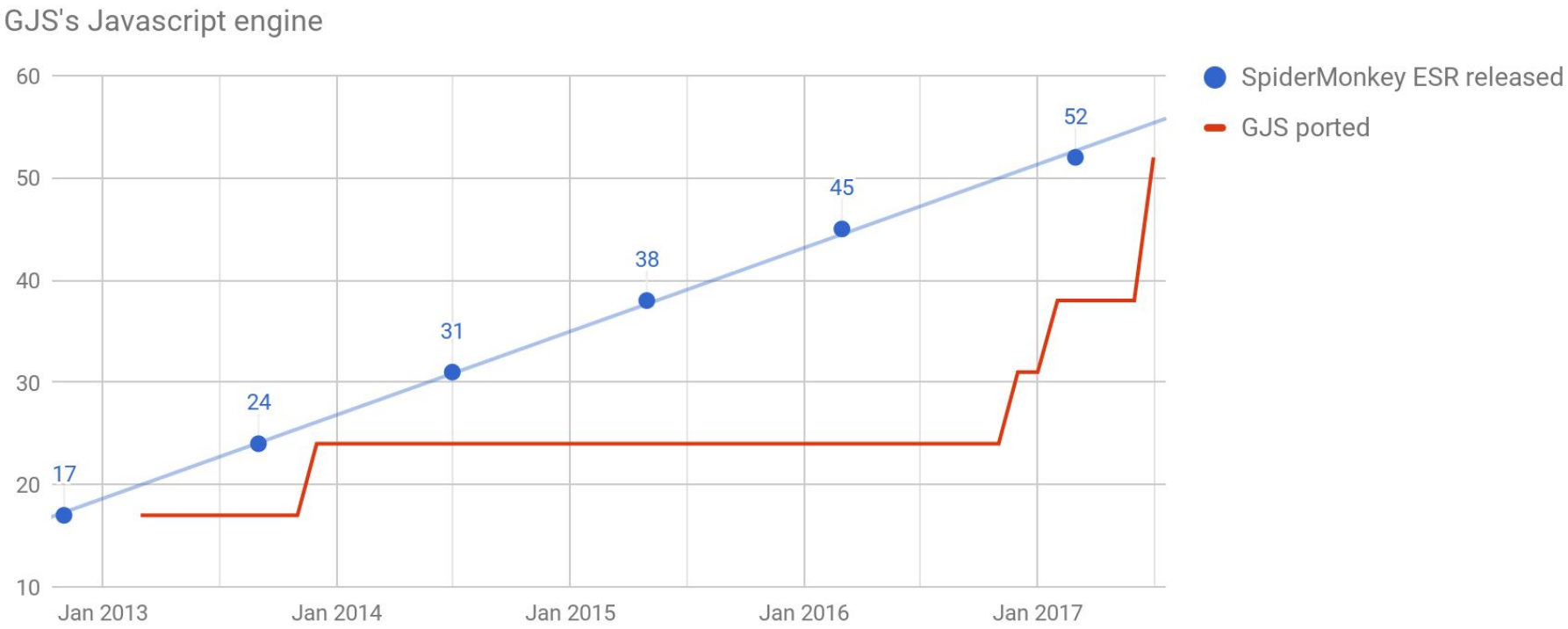
Read more about these problems:
2ality.com/2012/02/js-pitfalls.html

Standards committee

- Language stagnated for years due to public disagreements between Mozilla, Microsoft, and other stakeholders over the direction
- “The average Javascript program was one line”: `onclick="alert()"`
- ES6 published 2015, first update that addresses shortfalls in programming for the modern web
- Standards now updated and published every year
- Implementations follow rapidly

See http://kangax.github.io/compat-table/es6/

The technical debt



Highlights of new language features

3.24

- Promises
- Generators and iterators
- String template literals
- Spread operator
- Method syntax
- Destructuring
- Symbols

3.26

- Classes
- Async / await
- Reflect
- Well-known symbols

Better JS engine

- Better garbage collection: SpiderMonkey moved from a conservative garbage collector to an exact one
- Taking advantage of other JS engine performance improvements too
- Any ideas on how to quantify this? Would be a great project for someone to do, if interested

How to modernize your code

“You don't have to deal with that anymore”

Embrace the arrow function 3.8

Never again worry about binding **`this`** to a callback: I dare say `Lang.bind` is *the* biggest pain point for newcomers to Javascript. Just use arrow functions everywhere. (And if you have to bind, use `Function.bind()`)

```
undo.connect(Lang.bind(this, function() {
    this._docs.forEach(Lang.bind(this,
function(doc) {
        docManager.addItem(doc);
    });
});
```

```
undo.connect(() => {
    this._docs.forEach(doc =>
        docManager.addItem(doc));
});
```

Ben Halpern
@bendhalpern

Sometimes when I'm writing Javascript I want to throw up my hands and say "this is bullshit!" but I can never remember what "this" refers to

3:27 PM - 20 Mar 2015

🎩/t Rory MacQueen

Use new-style classes 3.26

For non-GObject classes, don't use Lang.Class anymore. (And if you don't want to refactor your entire code all at once, you can still inherit a new style class from a `Lang.Class`.) GObject classes coming soon to master.

```
const Circle = new Lang.Class({
    Name: 'Circle',
    Extends: Point,
    _init: function(x, y, rad) {
        this.parent(x, y);
        this.radius = rad;
    },
});
```

```
class Circle extends Point {
    constructor(x, y, rad) {
        super(x, y);
        this.radius = rad;
    }
}
```

While you're at it, use method syntax 3.24

Even on old-style classes, use the shorthand method syntax for better readability.

```
startTimer: function(id) {
    this._timers[id].start();
},

elapsedTime: function(id) {
    return this._timers[id].stop();
},
```

```
startTimer(id) {
    this._timers[id].start();
}

elapsedTime(id) {
    return this._timers[id].stop();
}
```

Comma in object literals, semi or nothing in new-style classes

Stop the string concatenation madness 3.24

Use backtick strings (“template literals”) for much more readable string interpolation.

```
print('Value ' + key + ': ' + value
+ '!');

const doc = [
    '<p class="' + style + '">',
    '  ' + text,
    '</p>',
].join('\n');
```

```
print(`Value ${key}: ${value}!`);


const doc = `<p class="${style}">
  ${text}
</p>`;
```

Take advantage of new array operations 3.26

No need for all those tedious comparisons against -1.

```
if (['foo', 'bar'].indexOf(myStr)
!== -1)


let testData = [3, 3, 3, 3, 3];
```

```
if (['foo', 'bar'].includes(myStr))


let testData = Array(5).fill(3);
```

Gotcha: let syntax 3.24

This was previously a custom extension in SpiderMonkey, but is a syntax error in the ES6 standard.

```
let a = 'something';
let a = 'other thing';
```

```
let a = 'something';
a = 'other thing';
```

Gotcha: Export variables with var 3.26

Another thing that used to be a custom SpiderMonkey extension but is not allowed in the ES6 standard, is that variables declared with **`const`** and **`let`** showed up as properties on modules. Use **`var`** now.

module.js
```
const FOO = 3;
let theText = 'okay';
```

main.js
```
const Module = imports.module;
`${Module.theText} ${Module.FOO}`
```

module.js
```
var FOO = 3;
var theText = 'okay';
```

Gotcha: String.replace arguments 3.26

The three-argument form of this function was a Mozilla extension, now removed. (This will fail silently.) Use regular expression literals.

```
str = str.replace('.', '_', 'g');
```

```
str = str.replace(/\./g, '_');
```

Read more

https://hacks.mozilla.org/category/es6-in-depth/

All the stuff described there is now in GJS, except ES6 modules

Read more

Appendix available after the talk

GNOME custom stuff

GIO promise wrappers

Not committed to master yet. I'd still like to try out a few different APIs, but you can include a `wrapPromise()` implementation in your code.

Also, there are two concepts from GLib async I/O that don't map to promises: I/O priority, and cancellation. I haven't figured out how these should look yet.

```
try {
    let [, contents] = await wrapPromise(file,
        'load_contents_async',
        'load_contents_finish');
    print(contents);
    let info = await wrapPromise(file,
        'query_info_async',
        'query_info_finish',
        'standard::*',
        Gio.FileQueryInfoFlags.NONE,
        GLib.PRIORITY_DEFAULT);
    print(info.get_size(), 'bytes');
} catch (err) {
    logError(err, 'Something failed');
}
loop.quit();
```

GIO promise wrappers

It would be even better to automatically return a promise if no callback was passed in. This is the API that I eventually want.

That requires introspecting the finish function automatically. (Bug 623635)

```
try {
    let [, contents] =
        await file.load_contents_async();
    print(contents);
    let info = await file.query_info_async(
        'standard::*',
        Gio.FileQueryInfoFlags.NONE,
        GLib.PRIORITY_DEFAULT);
    print(info.get_size(), 'bytes');
} catch (err) {
    logError(err, 'Something failed');
}
loop.quit();
```

GObject classes

Subject of my last blog post.
This API still in draft status as I work out problems.

```
var MyClass = GObject.registerClass({
    GTypeName: 'MyClass',
    Properties: {
        'foo': GObject.ParamSpec.string('foo',
            'Foo', 'Description of foo',
            GObject.ParamFlags.READWRITE |
                GObject.ParamFlags.CONSTRUCT,
            ''),
    },
}, class MyClass extends GObject.Object {
    _init(props={}) {
        this._foo = 'default';
        super._init(props);
    }

    get foo() { return this._foo }
    set foo(value) { this._foo = value; }
});
```

GObject classes

This is the API I eventually want.

It requires language features which are not in Javascript yet.

Could be available to transpiled code already.

```
@GObject.registerClass
class MyClass extends GObject.Object {
    static [GObject.GTypeName] = 'MyClass'

    _init(props={}) {
        this._foo = 'default';
        super._init(props);
    }

    @GObject.property.string({
        flags: GObject.ParamFlags.CONSTRUCT,
    })
    get foo() { return this._foo }
    set foo(value) { this._foo = value; }
});
```

Developer tools

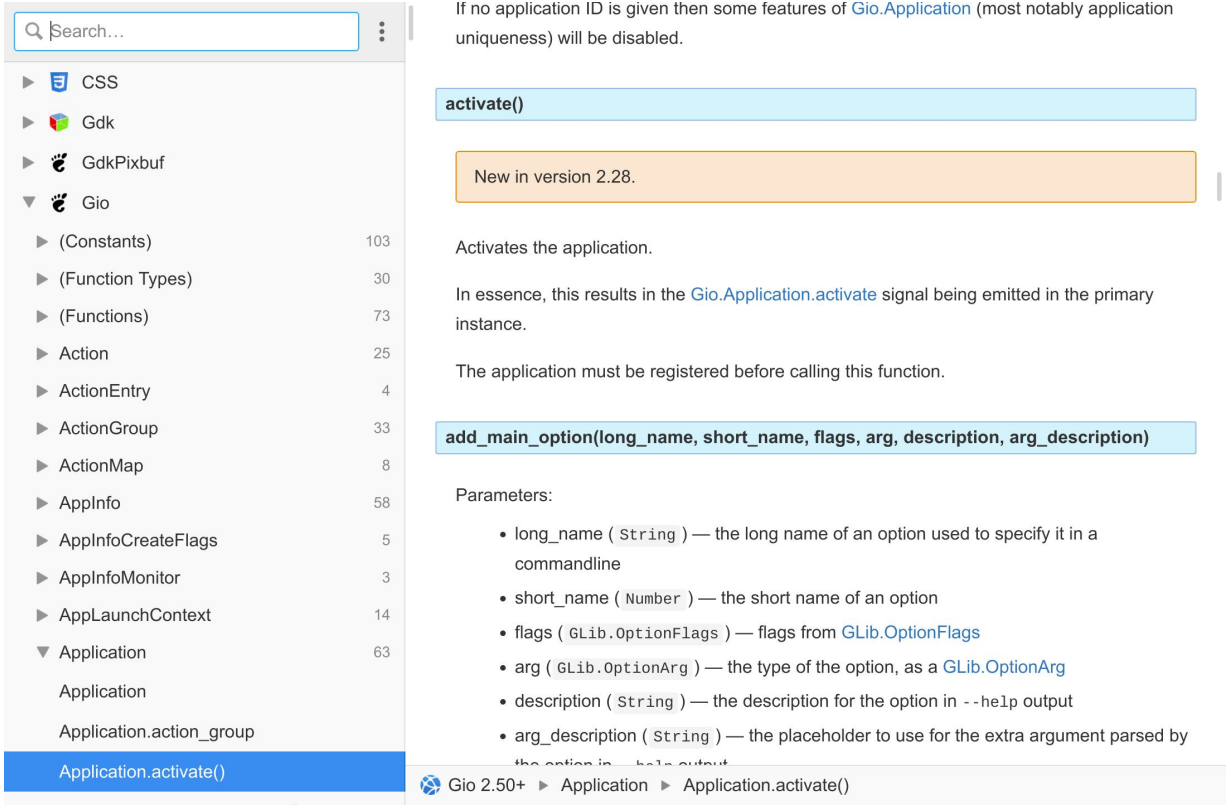
Documentation

Bookmark devdocs.baznga.org

Participate and file issues at github.com/ptomato/devdocs

Also use (and donate to support) devdocs.io

Big thank you to everyone who helped find a permanent host and create a Docker container



Unit testing

Jasmine: "behaviour-driven" testing framework for Javascript

Describe expectations in natural-ish language

Get the tools at github.com/ptomato/jasmine-gjs

Jasmine docs at jasmine.github.io

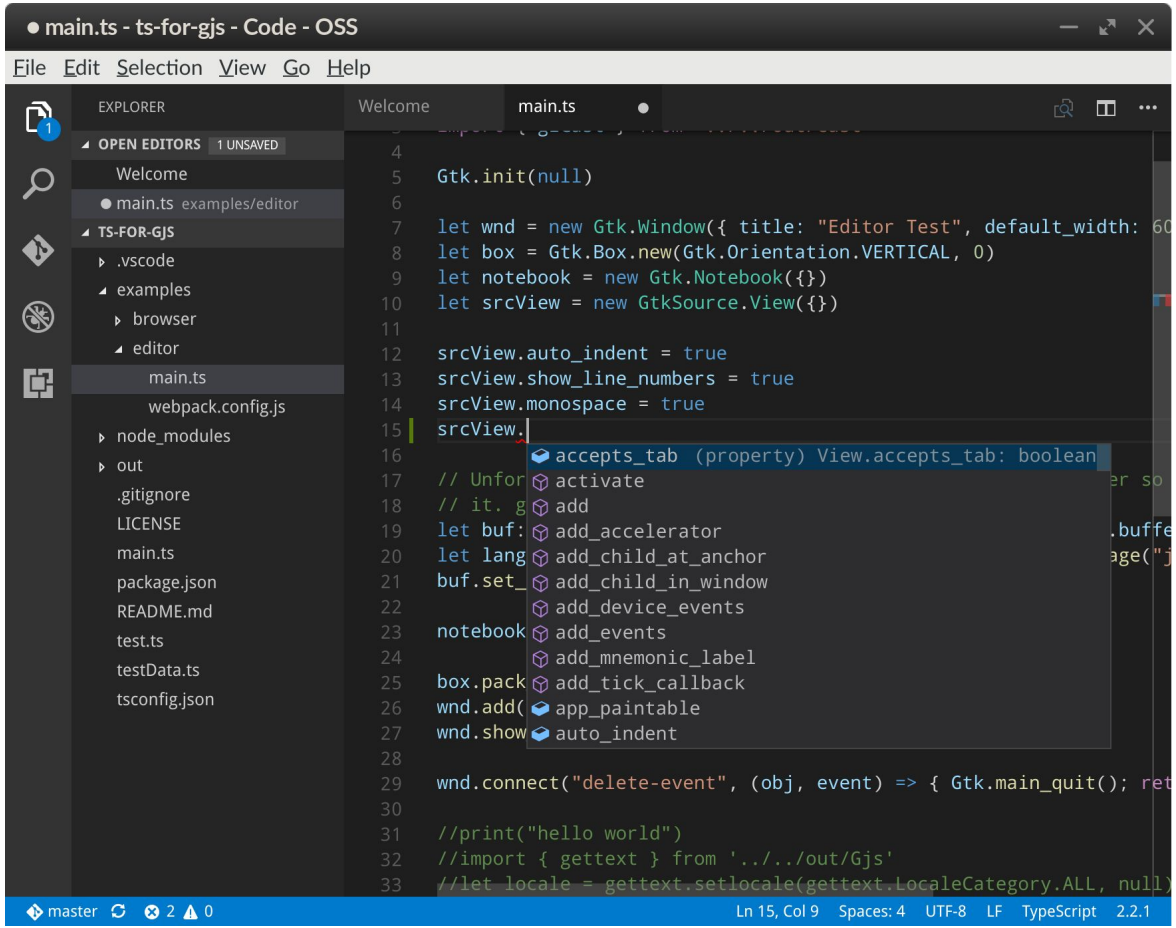
```
describe('Frobulator', function () {
    let f;

    beforeEach(function () {
        f = new Frobulator();
    });

    it('deals gracefully with null', function () {
        expect(() => f.dealWith(null))
            .not.toThrow();
    });
});
```

Type safety

Type safety with TypeScript (Sam Jansen)

Get the tools at github.com/sammydre/ts-for-gjs

Use with Visual Studio Code flatpak



Debugging and profiling

Nothing to show yet.

Acknowledgements

ENDLESS

Image reuse

Public domain

All screenshots were taken by me. Screenshots and presentation licensed under CC BY-ND 4.0.

Unconference session

Monday morning

- Packaging SpiderMonkey in GNOME and in downstream distros
- State of developer tools
- Integration of ES6 classes with GObject types
- Promise adaptors for GIO async functions

Wednesday

General hacking

Questions

Appendix 1: New stuff you're likely to use

Classes 3.26

```
class Circle extends Point {
    constructor(x, y, radius) {
        super(x, y);
        this.radius = radius;
    }

    area() {
        return this.radius ** 2 * Math.PI;
    }

    get radius() { return this._radius; }
    set radius(radius) {
        if (!Number.isInteger(radius))
            throw new Error("Must be an integer.");
        this._radius = radius;
    }
}
```

Nothing changed about Javascript's prototype inheritance system, it just is more pleasant to work with

Shorthand for methods (see next slide)

Method and object shorthand

3.24

```
{
  ...

  jsonify(result) {
    let lim = _getLimit();
    return {result, lim};
  },


  *[Symbol.iterator]() {
    yield* [1, 2, 3];
  },
}
```

Works in both classes and object literals.

Previously was an ungainly syntax with the function keyword.

Previously was the redundant but often-seen {result: result, lim: lim}

Also works with generators and computed property names.

Template strings 3.24

```
gjs> `multiline\n
.... string`
"multiline
string"
```

Fancy backtick string literals

```
gjs> `${greeting}, ${name}`
"Hello, Philip"
```

...with variable substitution

```
gjs> `E = ${m * c ** 2} J`
"E = 89875517873681760 J"
```

...and interpolation

Promises 3.24 and async functions 3.26

```
wait3Seconds()
.then(() => print('I waited 3 seconds!'))
.then(wait3Seconds)
.then(() => print('Another 3 seconds!'))
.catch(() => print('Oh no, a waiting error!'))
```

Asynchronous operations made easy.

```
try {
    await wait3Seconds();
    print('I waited 3 seconds!');
    await wait3Seconds();
    print('Another 3 seconds!');
} catch(e) {
    print('Oh no, a waiting error!');
}
```

Same as the previous example! Uses promises under the hood, only a bit nicer syntax.

Note: Requires running a GLib main loop.

Spread operator 3.24

```
gjs> let args = [1, 2, 3];
gjs> myFunc(0, ...args, 4, ...[5]);
gjs> let [a, ...b] = args;
gjs> b
2,3
```

We already had the spread operator in array literals, but now you can do it in function calls and destructuring assignment too.

```
gjs> a = [0, 1];
gjs> a.push(...b);
4
gjs> a
0,1,2,3
```

This can be used for a more idiomatic way of appending one array to another (previously you had to do this with `apply`.)

Generator functions 3.24

```
function* leafnodes(file) {
  let enumerator = file.enumerate_children('standard::*',
    0, null);
  let info;
  while ((info = enumerator.next_file(null))) {
    let child = enumerator.get_child(info);
    if (info.get_file_type() === Gio.FileType.DIRECTORY)
      yield* leafnodes(child);
    else
      yield child.get_basename();
  }
}
```

SpiderMonkey had some nonstandard generator functions in the past, but now it has ES6-compliant ones.

They work like Python's, and you can do cool stuff with them.

Symbols 3.24

```
gjs> let o = {};
gjs> o[Symbol.iterator] = blah;
```

Hard to explain concisely, but serve the same purpose as Python's double-underscore methods.

```
gjs> const Magic = Symbol('Magic');
gjs> function getMagic(obj) {
....   return obj[Magic] || -1;
.... }
gjs> o[Magic] = 42;
42
gjs> getMagic(o);
42
```

You can also define your own symbols.

Iterator protocol 3.24

```
function infinite() {
  let index = 0;

  return {
    next() {
      return { value: index++,
        done: false };
    },
  };
}
```

Iterators are not only returned from generators, but any object can be one.

```
gjs> let o = {};
gjs> [...o]
TypeError: o[Symbol.iterator] is not a function
gjs> o[Symbol.iterator] = ''[Symbol.iterator]
gjs> [...o]
[,o,b,j,e,c,t, ,O,b,j,e,c,t,]
```

You can make any object iterable.

New Array methods 3.24

```
gjs> Array(5).fill(3);
3,3,3,3,3
gjs> [1,2,3,4,5].find(i => i >
3.5);
4

gjs> Array.from('foo');
F,o,o
gjs> Array.from(arguments);

gjs> [1,2,3,4,5].includes(4);
true
```

3.26

Filling in some useful missing operations in the standard library!

...nicer than Array.prototype.slice.call

More: Array.copyWithin(), Array.fill(), Array.find(), Array.findIndex(), Array.of(), Array.entries(), Array.keys()

New String methods 3.24

```
gjs> String.fromCodePoint(
0x1f408, 0x1f4a8)
"□💨"

gjs> '💩'.codePointAt(0);
128169

gjs> '\u1E9B\u0323'
.normalize('NFKD')
"ṩ"

gjs> 'foobar'.includes('foo') 3.26
true
```

Better support for dealing with Unicode characters!

String tags 3.24

```
gjs> String.raw`multiline\n
.... string`
"multiline\nstring"

gjs> DBusInterfaceInfo`
.... <node>
.... <interface name="foo">
.... </interface>
.... </node>`
[boxed instance proxy
GIName:Gio.DBusInterfaceInfo
jsobj@0x7f2d8bf70b50
native@0xf69ef0]
```

This tag doesn't exist, but it *could*

You can process template strings by sticking a "tag" right in front of them. The builtin "String.raw" tag is like Python's "r", ignores escapes.

You can define your own tags, and the return values don't have to be strings. This is a powerful way of defining DSLs, if you like that sort of thing.

Well-known symbols 3.26

```
gjs> ({[Symbol.toStringTag]:'ion'})
[object ion]
```

Equivalent of Python's double underscore methods; customize behaviour of an object

Symbol.iterator was already in 3.24

3.26 added many more

```
gjs> class Foo {
.... static [Symbol.hasInstance](){
....   return true;
.... }
.... }
gjs> [] instanceof Foo
true
```

Appendix 2: New stuff you're less likely to use

Reflect 3.26

```
function Dana() {
  return new Proxy(this, {
    get: function (obj, name) {
      if (Reflect.has(obj, name))
        return Reflect.get(obj, name);
      log(`there is no ${name}, only
Zuul`);
    },
  });
}
```

Reflect performs Javascript operations on objects.

Good for metaprogramming, together with Proxies.

🎩/t

www.keithcirkel.co.uk/metaprogramming-in-es6-part-3-proxies

WeakSet 3.24

```
gjs> let s = new WeakSet();
gjs> s.add(someObject);
[object WeakSet]
gjs> s.has(someObject);
true
```

WeakSet is a Set whose members can be garbage collected.

Joins the already-existing WeakMap, Set, and Map to form the ES6 "keyed collections"

ES6 Internationalization API 3.24

```
gjs> let formatter = new Intl.NumberFormat('de-DE',
.... { style: 'currency', currency: 'EUR' }
.... );
gjs> formatter.format(123456.789);
"123.456,79 €"
```

Read more about it on MDN

```
gjs> new Date(Date.now()).toLocaleString('pt-BR',
{weekday: 'long'})
"terça-feira"
```

Also, toLocaleString() and related methods all got new "locales" and "options" extra arguments

New Math methods 3.24

```
gjs> Math.hypot(3,4)
5
```

hypot() was always my favourite from NumPy, it sped lots of calculations up by moving operations into C...

Full list is acosh(), asinh(), atanh(), cbrt(), clz32(), cosh(), expm1(), fround(), hypot(), log10(), log1p(), log2(), sign(), sinh(), tanh(), trunc()

Numbers and floating point

3.24

```
gjs> Number.EPSILON
2.220446049250313e-16

gjs> Number.MAX_SAFE_INTEGER
9007199254740991
gjs> Number.MIN_SAFE_INTEGER
-9007199254740991
gjs> Number.isSafeInteger(Math.pow(2, 53))
false
```

Can be useful as min and max values for 64-bit GObject properties, since `GLib.MININT64` and `GLib.MAXINT64` are not "safe"

```
gjs> Number.parseInt('beef', 16)
48879
gjs> Number.parseFloat('3.14')
3.14
```

Now preferred over the global `parseInt()` and `parseFloat()`

Binary and octal literals 3.24

```
gjs> 0b11001001
201
gjs> 0o755
493
```

Probably you won't use this too often...

```
gjs> 0755
(...deprecation warning)
493
```

The old octal literals still work too, but will complain. Also they don't work in strict mode

Misc new standard library stuff 3.24

Map.forEach()

Set.forEach()

Object
.assign()
.getOwnPropertySymbols()
.setPrototypeOf()
.getOwnPropertyDescriptors() 3.26
.values()

ArrayBuffer.isView()

Date.toString() 3.26

Proxy
.handler.isExtensible
.revocable()
.getPrototypeOf()
.setPrototypeOf() 3.26

Generators return()

RegExp
.flags
.global
.ignoreCase
.multiline
.sticky
.toString() 3.26

ES2017 proposed features 3.26

```
gjs> 'foo'.padStart(10, '-')
"-------foo"
```

Free yourself from the tyranny of leftpad! (But use with care, it might not become standard JS in this form.)

More proposed features implemented: String.padEnd(), Object.entries(), Intl.DateTimeFormat.formatToParts()

You can play around with **WebAssembly** if you compile SpiderMonkey yourself!

Exponentiation operator 3.26

```
let E = m * c ** 2;
```

Sure, whatever