ECMAScript 6, A Radical Jump Forward

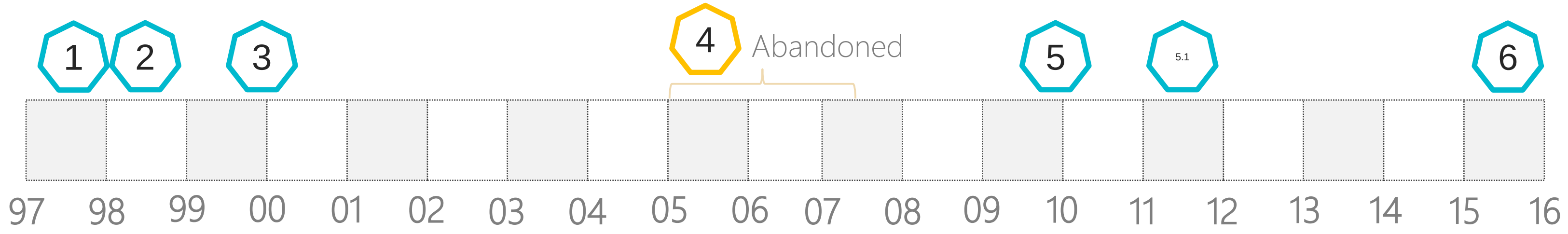
Belem

ECMAScript

- "ECMAScript" is the name under which the language more commonly known as "JavaScript" is standardized.
- Development of the ECMAScript standard is the responsibility of Technical Committee 39 (TC39) of Ecma International.
- The ECMAScript Language Specification standard is officially known as ECMA-262.

ECMAScript History

- Development at Netscape began in 1994
- ECMA-262 1st Edition June 1997
- ISO/IEC 16262 Approved April 1998
- ECMA-262 2nd Edition June 1998
- ECMA-262 3rd Edition December 1999
- ECMA-262 5th Edition December 2009
- ISO/IEC 16262 3rd Edition April 2011
- ECMA-262 5.1 Edition June 2011
- ECMA-262 6th Edition June 2015



ECMAScript 6

- ECMAScript 6, also known as ECMAScript 2015, is the latest version of the ECMAScript standard
- ES6 is a significant update to the language
- The first update to the language since ES5 was standardized in 2009
- ES6 standard for full specification of the ECMAScript 6 language
- Draft Specification for ES.next (Ecma-262 Edition 6)

ECMAScript 6

- More significant changes than ES5
- Much more new syntax
- Very, very high back compat
- ES Harmony => ES.next => ES 6 (ES Harmony is a superset of ES.next)

```
$ node -harmony
```

https://en.wikipedia.org/wiki/ECMAScript#Implementations

Transpiling

Problem: Use new features while at the same time
being slapped with the reality that their sites/apps may need
to support older browsers without such support.

Answer: Tooling, specifically a technique called transpiling
(transformation compiling).

Use a special tool to transform ES6 code into equivalent
(or close!) matches that work in ES5 environments.

Transpiling

Enhanced object literals: Shorthand property definitions

```
var foo = [1,2,3];
var obj = {
foo
};
obj.foo;
```

ES6 form

```
var foo = [1,2,3];
var obj = {
foo: foo
};
obj.foo;
```

ES5 form

Transpiling: JavaScript compiler

```
// https://babeljs.io/
// Use next generation JavaScript, today.

$ npm install -g babel
$ babel script.js

<script src="node_modules/babel-core/browser.js"></script>
<script type="text/babel">
// Your ES6 code
</script>

// https://github.com/google/traceur-compiler
// A JavaScript.next-to-JavaScript-of-today compiler

$ npm install -g traceur
$ traceur --script calc.es6.js --out calc.es5.js

// http://google.github.io/traceur-compiler/demo/repl.html
// http://www.es6fiddle.net/
```

Shims/Polyfills

- Not all new ES6 features need a transpiler.
- Polyfills (aka shims) are a pattern for defining equivalent behavior from a newer environment into an older environment.
- Syntax cannot be polyfilled, but APIs often can be.

Shims/Polyfills

Polyfill for Object.is(..)

```
if (!Object.is) {
Object.is = function(v1, v2) {
// test for `-0`
if (v1 === 0 && v2 === 0) {
  return 1 / v1 === 1 / v2;
}
// test for `NaN`
if (v1 !== v1) {
  return v2 !== v2;
}
// everything else
return v1 === v2;
};
}
```

https://github.com/paulmillr/es6-shim
https://github.com/Benvie/harmony-collections

Syntax

- block-scoped declarations (let, const)
- arrow functions
- object literal extensions
- template strings
- destructuring
- default parameter values, rest, spread
- symbols
- for..of loops
- unicode

Syntax: block-scoped declarations (let, const)

- block-scoped binding
- ‘let’ is the new ‘var’
- ‘const’ is single-assignment
- ‘const’ creates constants, a variable that’s read-only after its initial value is set

```
var a = 1;
{
  let a = 2;
  console.log(a);   // 2
  const b = 3;
  b = 4;            // b is read-only
}
console.log(a);     // 1
```

Syntax: arrow functions

- Function shorthand with =>
- Syntactically similar to C#, Java
- Support for expression and statement bodies
- Shorthand for common usage of functions as callbacks

```
function foo(x,y) {
  return x + y;
}

// ES6
var foo = (x,y) => x + y;


var array = [1, 2, 3];
array.forEach(function(v) {
    return console.log(v);
});

// ES6
array.forEach(v => console.log(v));
```

Syntax: object literal extensions

- Richer object literals
- Covers common scenarios like:
    - setting prototype at construction
    - Concise properties
    - concise methods
    - accessing super from method

```
var obj = {
    // __proto__
    __proto__: theProtoObj,
    // Shorthand for ‘handler: handler’
    handler,
    // Methods
    toString() {
      // Super calls
      return "d " + super.toString()
    }
};
```

Syntax: template strings

- Using \` back-tick as the delimiter
- Expressions of the form ${..} are parsed and evaluated inline
- For string construction
- Similar to string interpolation in Python
- Make it easier to construct strings with less risk of injection attacks

```
var name = "Kyle";
var greeting = `Hello ${name}!`;

// "Hello Kyle!"
console.log(greeting);
// "string"
console.log(typeof greeting);


// Prefix is used to interpret the
//     replacements and construction
GET`http://foo.org/bar?a=${a}&b=${b}
    Content-Type: application/json
    X-Credentials: ${credentials}
    { "foo": ${foo},
      "bar":
${bar}}`(myOnReadyStateChangeHandler);
```

Syntax: destructuring

```
function foo() {
  return [1,2];
}
var tmp = foo(),
  a = tmp[0], b = tmp[1];
console.log(a, b);  // 1 2

function bar() {
  return {
    x: 4,
    y: 5,
};
}
var tmp = bar(),
  x = tmp.x, y = tmp.y;
console.log(x, y);  // 3 4
```

ES5

- Binding via pattern matching
- array destructuring and object destructuring
- { x: x, .. } syntax:
  shorten syntax var { x, y } = bar();
  if the property name being matched is the same as the variable you want to declare

```
var [ a, b ] = foo();
var { x: x, y: y } = bar();
console.log( a, b );  // 1 2
console.log( x, y );  // 3 4
```

ES6

Syntax: default parameter values, rest, spread

- Callee-evaluated default parameter values
- Turn an array into consecutive arguments in a function call
- Bind trailing parameters to an array
- Write less code, replace 'arguments'

```
function f(x, y=12) {
  // y is 12 if not passed
  return x + y;
}
f(3)

function f(x, ...y) {
  // y is an Array
  return x * y.length;
}
f(3, "hello", true)

function f(x, y, z) {
  return x + y + z;
}
// Pass each elem of array as argument
f(...[1,2,3])
```

Syntax: symbols

- Allow properties to be keyed by either string (as in ES5) or Symbols
- Symbols are a new primitive type
- Should not use new with Symbol(..).
- Parameter passed to Symbol(..) is optional
- The typeof output is a new value ("symbol") that is the primary way to identify a symbol
- Originally two kinds 'private' and 'unique'.
- Private are not exposed to reflection APIs.
- Unique are exposed to reflection like normal properties.

```
var sym = Symbol("optional description");
typeof sym;  // "symbol"
sym instanceof Symbol;  // false
var symObj = Object(sym);
symObj instanceof Symbol;  // true
```

```
(function()
  // module scoped symbol
  var key = Symbol("key");

  function MyClass(privateData) {
    this[key] = privateData;
  }

  MyClass.prototype = {
    doStuff: function() {
      ... this[key] ...
    }
  };
})();

var c = new MyClass("hello")
c["key"] === undefined
```

Syntax: for..of loops

- Loops over the set of values produced by an iterator
- for..in loops over the keys/indexes in the a array
- for..of loops over the values in a

```
var a = ["a","b","c","d","e"];

for (var idx in a) {
  console.log(idx);
}
// 0 1 2 3 4

for (var val of a) {
  console.log(val);
}
// "a" "b" "c" "d" "e"
```

Syntax: unicode

- Non-breaking additions to support full Unicode
- Unicode code point escaping
- New RegExp mode to handle code points
- Several new libraries to support processing strings as 21bit code points
- Support building global apps in JavaScript

```
 // same as ES5.1
"𠮷".length == 2

// new RegExp behaviour, opt-in 'u'
"𠮷".match(/./u)[0].length == 2

// new form
"\u{20BB7}"=="𠮷"=="\uD842\uDFB7"

// new String ops
"𠮷".codePointAt(0) == 0x20BB7

// for-of iterates code points
for(var c of "𠮷") {
  console.log(c);
}
```

Organization

- classes
- iterators
- generators
- comprehensions
- modules
- module loaders

Organization: classes

- Simple classes
- Compiles to prototype-based OO pattern
- Formalizes common JS OO pattern into syntax
- Declarative classes covering common case

```
class Foo {
  static answer = 42;
  static cool(){ console.log( "cool" ); }
  // ..
}
Foo.answer;  // 42
```

```
class Foo {
  constructor(a,b) {
    this.x = a;
    this.y = b;
  }
  gimmeXY() {
    return this.x * this.y;
  }
}

class Bar extends Foo {
  constructor(a,b,c) {
    super( a, b );
    this.z = c;
  }
  gimmeXYZ() {
    return super.gimmeXY() * this.z;
  }
}
var b = new Bar(5, 15, 25);
b.z;  // 25
b.gimmeXYZ();  // 1875
```

Organization: iterators

- Iterator objects enable custom iteration like CLR IEnumerable or Java Iteratable
- Generalize for-in to custom iterator-based iteration with for-of
- Don’t require realizing an array
- Open up new kinds of iteration, including databases (LINQ)

```
interface IteratorResult {
  done: boolean;
  value: any;
}
interface Iterator {
  next(): IteratorResult;
}
interface Iterable {
  [Symbol.iterator](): Iterator
}
```

```
var arr = [1,2,3];
var it = arr[Symbol.iterator]();
it.next();  // { value: 1, done: false }
it.next();  // { value: 2, done: false }
it.next();  // { value: 3, done: false }
it.next();  // { value: undefined, done: true }
```

```
for (var v of it) {
  console.log( v );
}
```

```
more manual version of previous snippet’s loop:

for (var v, res; !(res = it.next())
&& !res.done; ) {
  v = res.value;
  console.log( v );
}
```

Organization: generators

- Simplify iterator-authoring using 'function*' and 'yield'
- A function declared as function* returns a Generator instance
- Generators are subtypes of Iterators
- Open up new kinds of iteration, including databases (LINQ)
- Enables 'await'-like async programming

```
interface Generator extends Iterator {
    next(value?: any): IteratorResult;
    throw(exception: any);
}
```

```
var fibonacci = {
  [Symbol.iterator]: function*() {
    var pre = 0, cur = 1;
    for (;;) {
      var temp = pre;
      pre = cur;
      cur += temp;
      yield cur;
    }
  }
}

for (var n of fibonacci) {
  // truncate the sequence at 1000
  if (n > 1000)
    break;
  print(n);
}
```

Organization: comprehensions

- Sugar over simple iterator composition
- Option to realize results as array or generator
- Compact queries over in memory data

```
// Array comprehensions
var results = [
  for(c of customers)
  if (c.city == "Seattle")
  { name: c.name, age: c.age }
]

// Generator comprehensions
var results = {
  for(c of customers)
  if (c.city == "Seattle")
  { name: c.name, age: c.age }
}
```

Organization: modules

- Language-level support for modules
- Codify patterns from popular JavaScript module loaders (AMD, CommonJS)
- Host-defined default loader
- Implicitly async model – no code executes until requested modules are available and processed

```
// point.js
module "point" {
    export class Point {
        constructor (x, y) {
            public x = x;
            public y = y;
        }
    }
}

// myapp.js
module point from "/point.js";
import Point from "point";

var origin = new Point(0, 0);
console.log(origin);
```

Organization: module loaders

- Dynamic loading
- State isolation
- Global namespace isolation
- Compilation hooks
- Nested virtualization

- Support code loading natively in the JavaScript engine

```
// Dynamic loading – 'System' is default loader
System.import('lib/math').then(function(m) {
  alert("2π = " + m.sum(m.pi, m.pi));
});

// Create execution sandboxes – new Loaders
var loader = new Loader({
  global: fixup(window) // replace 'console.log'
});
loader.eval("console.log('hello world!';)")

// Directly manipulate module cache
System.get('jquery')
System.set('jquery', Module({$: $}))
```

Collections

- maps, weakmaps
- sets, weaksets

Collections: maps, weakmaps

- Efficient Map data structures
- Maps associate some extra piece of information (the value) with an object (the key) without actually putting that information on the object itself.
- WeakMaps differs underneath in how the memory allocation (specifically its GC) works, provides leak-free object-key'd side tables
- WeakMaps take (only) objects as keys. If the object itself is GC'd, the entry in the WeakMap is also removed.

```
// Maps
var m = new Map();
m.set("hello", 42);
m.set(s, 34);
m.get(s) == 34;

// Weak Maps
var wm = new WeakMap();
wm.set(s, { extra: 42 });
wm.size === undefined
```

Collections: sets, weaksets

- A set is an a collection of unique values (duplicates are ignored)
- add(..) method takes the place of the set(..) method
- No get(..) method since you don't retrieve a value from a set
- WeakSets are sets where the value is weakly held, GC can remove the entry if it's the last reference to that object.

```
// Sets
var s = new Set();
s.add("hello").add("goodbye").add("hello");
s.size === 2;
s.has("hello") === true;

// Weak Sets
var ws = new WeakSet();
ws.add({ data: 42 });
```

Meta Programming

- proxies
- tail call optimization (TCO)

Meta Programming: proxies

- Proxies can intercept and customize various low-level operations on objects
- Enable creation of objects with full range of behaviors available to host object
- Support interception, object virtualization, logging/profiling, etc.
- Proxy semantics restricted to “proxy” objects, not general interception

```
//Define object to be intercepted
var engineer = { name: 'Tom Kai', salary: 50 };

//Define handling program
var interceptor = {
  set: function (receiver, property, value) {
    console.log(property, 'is changed to',
value);
    receiver[property] = value;
  }
};

//Create proxy to detect
engineer = Proxy(engineer, interceptor);

//Trigger proxy by modifications
engineer.salary = 60;
//Console output: salary is changed to 60
```

Meta Programming: tail call optimization (TCO)

- Calls in tail-position are guaranteed to not grow the stack unboundedly.
- Makes recursive algorithms safe in the face of unbounded inputs.
- Provides a better compilation target for languages that depend on tail call optimization
- This optimization can only be applied in strict mode.

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

// Stack overflow in most implementations today,
// but safe on arbitrary inputs in ES6
factorial(100000)
```

Async Flow Control

- promises

Async Flow Control: promises

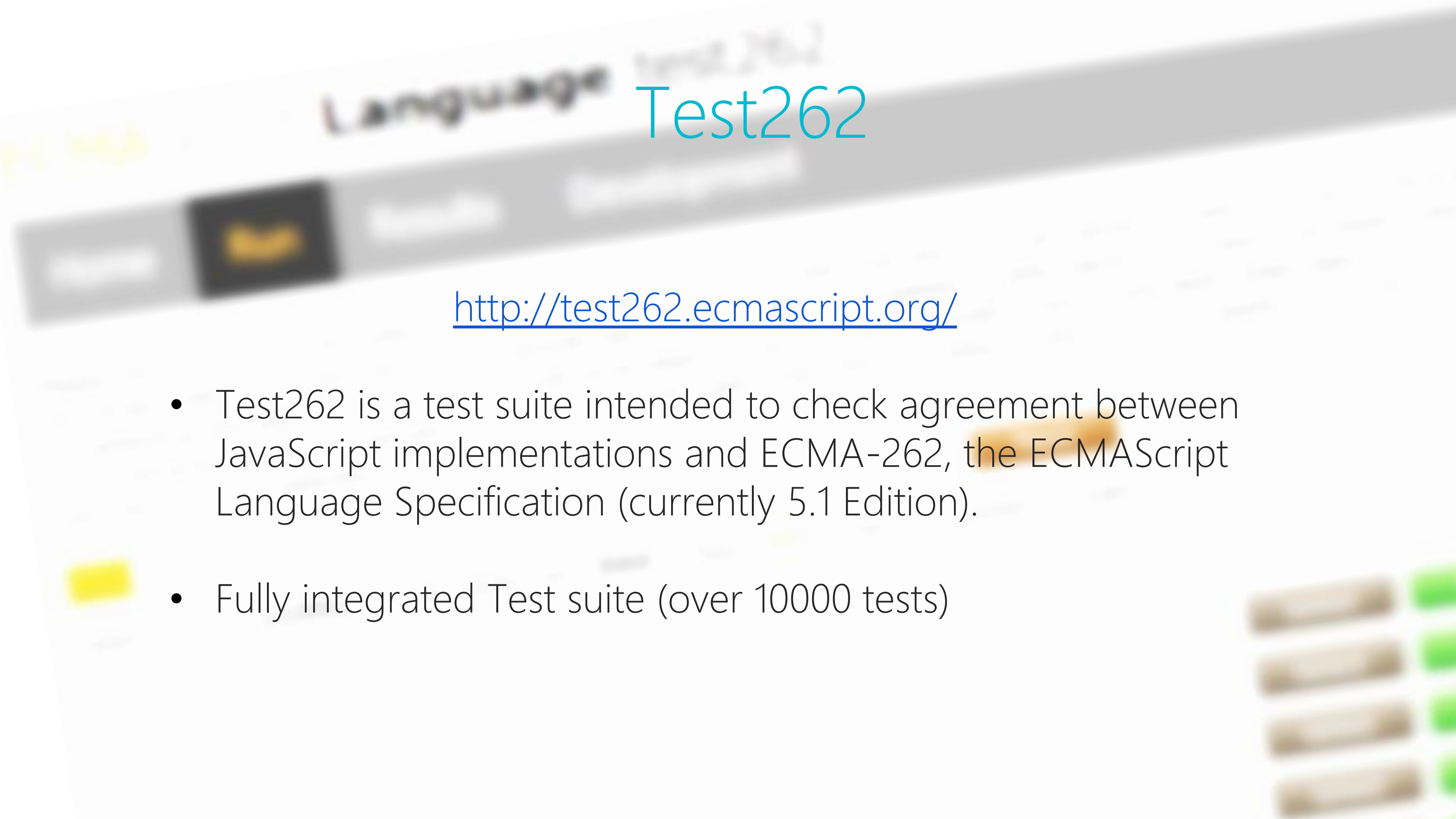
- Callbacks are not fully sufficient for asynchronous tasks
- Promises addresses one of the major shortcomings of callbacks: lack of trust in predictable behavior
- Promises represent the future completion value from a potentially-async task, normalizing behavior across sync and async boundaries

```
//Create promise
var promise = new Promise(function(resolve,
reject) {
    // Do some async operations
    if ( /* success */ ) {
        resolve("Stuff worked!");
    } else {
        reject(Error("It broke"));
    }
});
//Bind handling program
promise.then(function(result) {
          //promise success
    console.log(result); // "Stuff worked!"
}, function(err) {
          //promise failed
    console.log(err); // Error: "It broke"
});
```

API Additions

- `Array:`
  - `Static functions of(..), from(..)`
  - `Prototype functions like copyWithin(..), fill(..)`
- `Object: Static functions like is(..), assign(..)`
- `Math: Static functions like acosh(..), clz32(..)`
- `Number:`
  - `Static properties like Number.EPSILON`
  - `Static functions like Number.isFinite(..)`
- `String:`
  - `Static functions like String.fromCodePoint(..), String.raw(..)`
  - `Prototype functions like repeat(..), includes(..)`

Most of these additions can be polyfilled (see ES6 Shim).

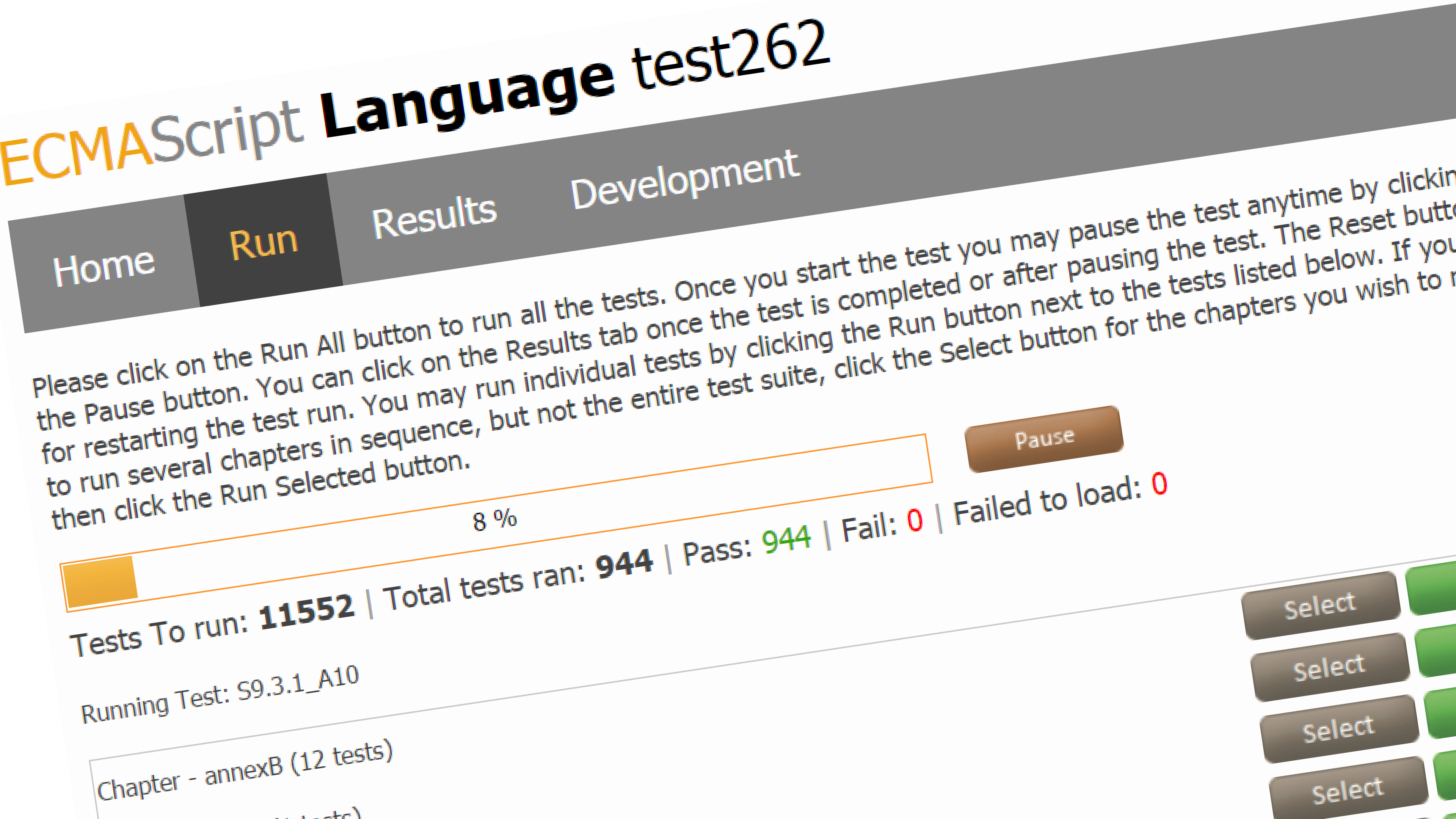


Test262

http://test262.ecmascript.org/

- Test262 is a test suite intended to check agreement between JavaScript implementations and ECMA-262, the ECMAScript Language Specification (currently 5.1 Edition).

- Fully integrated Test suite (over 10000 tests)


ECMAScript **Language** test262

Please click on the Run All button to run all the tests. Once you start the test you may pause the test anytime by clickin
the Pause button. You can click on the Results tab once the test is completed or after pausing the test. The Reset butt
for restarting the test run. You may run individual tests by clicking the Run button next to the tests listed below. If you
to run several chapters in sequence, but not the entire test suite, click the Select button for the chapters you wish to r
then click the Run Selected button.

8 %

Tests To run: **11552** | Total tests ran: **944** | Pass: 944 | Fail: 0 | Failed to load: 0

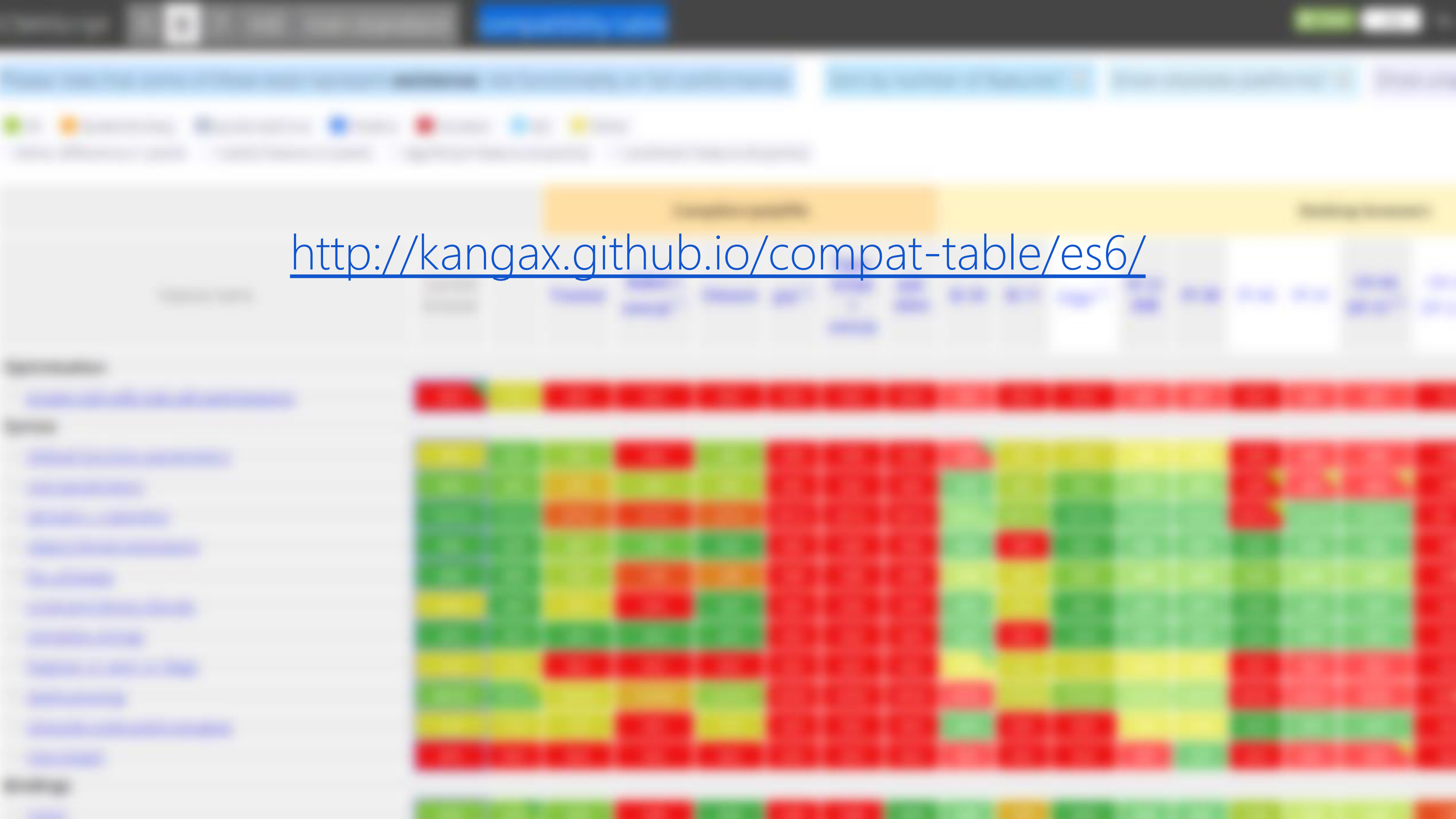
Running Test: S9.3.1_A10

Chapter - annexB (12 tests)

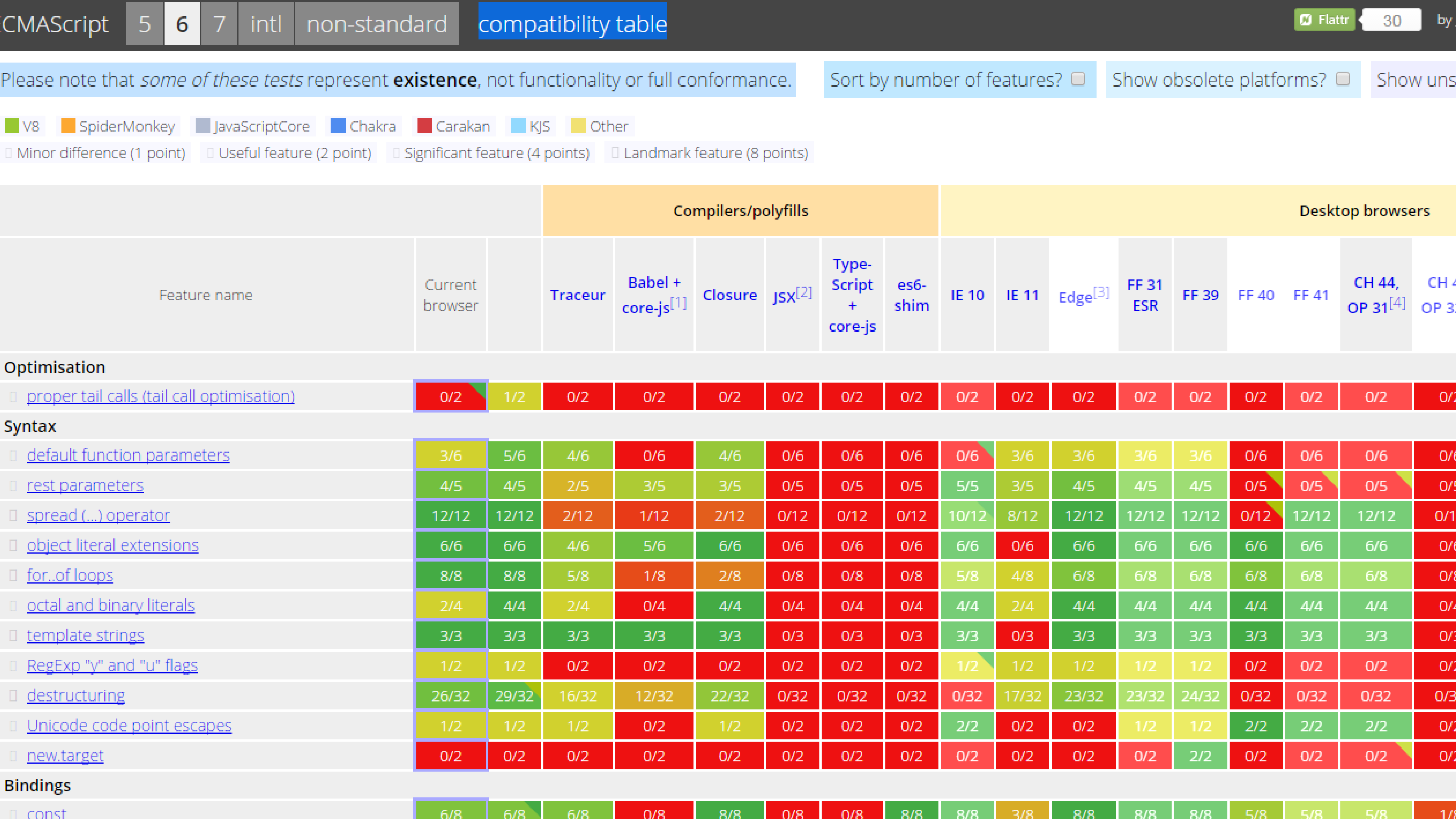
ES Feature Tests

- Feature test ES6+ JavaScript syntax and APIs.
- Load native ES6+ code in ES6+ compliant browsers and transpiled code in older browsers.
- https://featuretests.io/

ES6 Compatibility Table



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



CMAScript | 5 | 6 | 7 | intl | non-standard | compatibility table

Flattr 30 by

Please note that *some of these tests* represent **existence**, not functionality or full conformance.

Sort by number of features? ☐ Show obsolete platforms? ☐ Show uns

🟩 V8 🟧 SpiderMonkey ▨ JavaScriptCore 🟦 Chakra 🟥 Carakan 🟦 KJS 🟨 Other

Minor difference (1 point) Useful feature (2 point) Significant feature (4 points) Landmark feature (8 points)

			Compilers/polyfills							Desktop browsers								
Feature name	Current browser		Traceur	Babel + core-js[1]	Closure	JSX[2]	Type-Script + core-js	es6-shim	IE 10	IE 11	Edge[3]	FF 31 ESR	FF 39	FF 40	FF 41	CH 44, OP 31[4]	CH 4 OP 3
Optimisation																	
proper tail calls (tail call optimisation)	0/2	1/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/
Syntax																	
default function parameters	3/6	5/6	4/6	0/6	4/6	0/6	0/6	0/6	0/6	3/6	3/6	3/6	3/6	0/6	0/6	0/6	0/
rest parameters	4/5	4/5	2/5	3/5	3/5	0/5	0/5	0/5	5/5	3/5	4/5	4/5	4/5	0/5	0/5	0/5	0/
spread (...) operator	12/12	12/12	2/12	1/12	2/12	0/12	0/12	0/12	10/12	8/12	12/12	12/12	12/12	0/12	12/12	12/12	0/1
object literal extensions	6/6	6/6	4/6	5/6	6/6	0/6	0/6	0/6	6/6	0/6	6/6	6/6	6/6	6/6	6/6	6/6	0/
for..of loops	8/8	8/8	5/8	1/8	2/8	0/8	0/8	0/8	5/8	4/8	6/8	6/8	6/8	6/8	6/8	6/8	0/
octal and binary literals	2/4	4/4	2/4	0/4	4/4	0/4	0/4	0/4	4/4	2/4	4/4	4/4	4/4	4/4	4/4	4/4	0/
template strings	3/3	3/3	3/3	3/3	3/3	0/3	0/3	0/3	3/3	0/3	3/3	3/3	3/3	3/3	3/3	3/3	0/
RegExp "y" and "u" flags	1/2	1/2	0/2	0/2	0/2	0/2	0/2	0/2	1/2	1/2	1/2	1/2	1/2	0/2	0/2	0/2	0/
destructuring	26/32	29/32	16/32	12/32	22/32	0/32	0/32	0/32	0/32	17/32	23/32	23/32	24/32	0/32	0/32	0/32	0/3
Unicode code point escapes	1/2	1/2	1/2	0/2	1/2	0/2	0/2	0/2	2/2	0/2	0/2	1/2	1/2	2/2	2/2	2/2	0/
new.target	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	2/2	0/2	0/2	0/2	0/
Bindings																	
const	6/8	6/8	6/8	0/8	8/8	0/8	0/8	8/8	8/8	3/8	8/8	8/8	8/8	5/8	5/8	5/8	1/

ES7?

- Object.observe
- Async/await
- Weak References
- Parallel JavaScript
- Decorators
- Value objects (int64 and bignum)
- Multi-Threading ?
- Traits ?

References

- http://www.ecma-international.org/ecma-262/6.0/
- https://github.com/lukehoban/es6features
- http://google.github.io/traceur-compiler/demo/repl.html

Thank you!