ECMAScript 2015

Were you expecting something else?

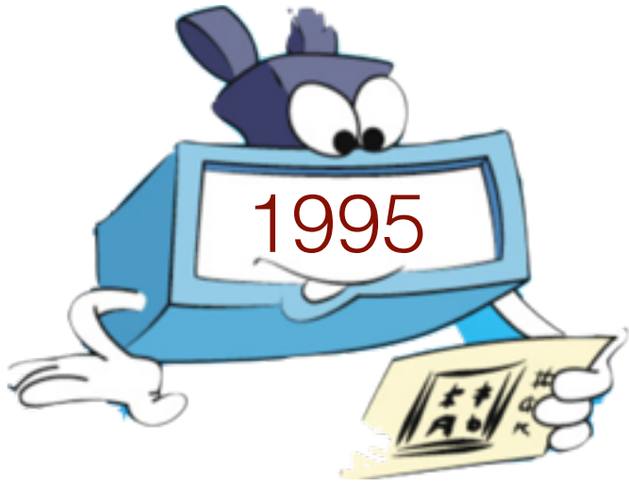
A History Lesson

A History Lesson

May - Mocha is invented in Netscape by Brendan Eich.

September - Renamed to LiveScript

December - Renamed to Javascript (Because Java was popular)



A History Lesson

1996 - JavaScript is taken to standardization in ECMA.

From now on ECMA is the spec. Javascript is an implementation (ActionScript is another implementation)

1997 - ECMA-262 (ECMAScript)

1998 - ECMAScript 2

1999 - ECMAScript 3

A History Lesson

2005 - Mozilla and Macromedia started Work on ECMAScript 4. Feature rich and a very major leap from ECMAScript 3.

Douglas Crockford (Yahoo) And Microsoft opposed the forming standard.

ECMAScript 3.1 was the compromise.

A History Lesson

2009 - Opposing parties meet in Oslo and achieve an agreement.

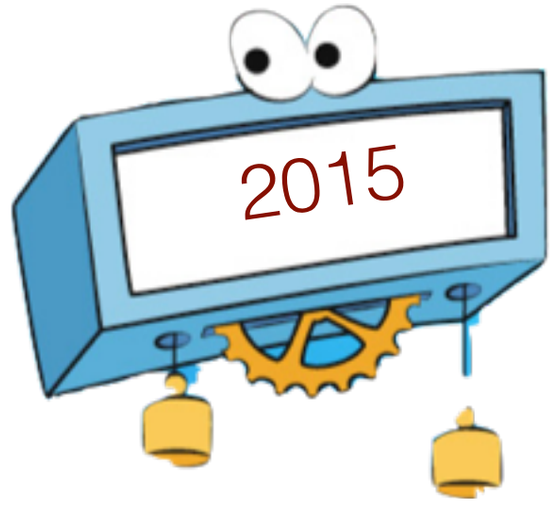
ES3.1 is renamed to ES5 (Which we have today)

In the spirit of peace and agreement, the new Javascript long term agenda is named “Harmony”

A History Lesson

2015 - ES6 (Part of the “Harmony” umbrella) will be released.

Starting with ES6 - Version names will be based on the year of release. so ES6 is ES2015 and ES7 should be ES2016



Declaration and Scoping

let - A New ~~Hope~~ Scope

var is function scope

```
if (true) {
   var x = 3;
}
console.log(x); // 3
```

let is block scope

```
if (true) {
   let x = 3;
}
console.log(x); // ReferenceError
```

A word about Hoisting

To hoist is to raise. Function declarations are hoisted which means you **can** do this with *functions* and *vars*:

```
f(); //3

function f() {
  return 3;
}
```

But you **can't** do it with *classes* or ***let*** variables

```
new X(); // TypeError
class X{}
```

Temporal dead zone

let variable declarations are not hoisted

```
console.log(x); // ReferenceError
let x = 3;
console.log(x); // 3
```

The time between block start and declaration is called “Temporal Dead Zone”. Same with ‘const’

Temporal dead zone

TDZ is based on time and **not** on location

```
function block() {
  function foo() {
    console.log(x);
  }
  foo(); // ReferenceError
  let x = 3;
  foo(); // 3
}
```

Loop scoping

With *let* You get a fresh iteration per loop

```
let vals = [];
for (let x = 0; x < 4; x += 1) {
  vals.push(() => x);
}
console.log(vals.map(x => x()));
// [0, 1, 2, 3]
```

With *var* You would get *[4, 4, 4, 4]* as a result

* we're using functions to store by reference otherwise it won't prove the point. Full Example here

const

const makes variables immutable

```
const obj = { par: 3 };
obj = 4; // TypeError
```

but it doesn't stop you from changing the object values

```
obj.par = 12; // Fine
```

you can use Object.freeze() to achieve that.

```
Object.freeze(obj);
obj.par = 10; // TypeError
```

or the Object.seal() which blocks changing object structure

```
console.seal(obj); // 17
obj.newParam = 3 // TypeError
```

String Templates

String Templates

This is syntactic sugar for string concatenation.

```
let name = 'John';
`Hi ${name},
 Did you know that 5 * 3 = ${5*3}?`
/*
"Hi John,
 Did you know that 5 * 3 = 15?"
 */
```

Anything in *${}* is evaluated
Back-ticks are used to enclose the string.
New lines are retained

Enhanced Object Literals

Shorthand assignment

basically just shortening *{x:x}* to *{x}*

```
let x = 3;
let xo =  {x};
// xo = { x: 3 }
```

equivalent to:

```
let x = 3;
let xo =  {x:x};
// xo = { x: 3 }
```

Method Definition *

a new syntax for defining methods inside an object literal

```
let foo = {
   f(x) {
      return x + 1;
   }
};
foo.f(3); // 4
```

Super

you can make super calls for inherited methods

```
let foo = {
   toString() {
      super.toString() + ' with foo!';
   }
};

foo.toString();
//[object Object] with foo!
```

in class constructors you can simply call *super()* to call on the parent constructor

Computed Properties

you can define keys that evaluate on run time inside literals

```
let ind = 100;
let foo = {
   ['question'+ ind]:
      'question number ' + ind,
   ['answer' + ind](x) {
      return ind === x;
   }
};
foo.question100; // “question number 100”
foo.answer100(100); // true
```

Symbols

Symbol

a symbol is **immutable** and **unique**

```
let s1 = Symbol('test');
let s2 = Symbol('test');
s1 === s2; // false
```

the string parameter is for console and debug.

Symbol

symbol keys are not visible

```
let  o = {
   [Symbol('hellllo')]: 7,
   'str': 6
};
Object.getOwnPropertyNames(o);
// Array [ "str" ]
JSON.stringify(o); // "{"str":6}"
```

since Symbols are hidden, Object allows reflection:

```
Object.getOwnPropertySymbols(o);
// Array [ Symbol(hellllo) ]
```

Symbol

symbols are now used by javascript, internally, with certain features. for example:

`Symbol.iterator`
`used to provide iterator function`
`Symbol.hasInstance`
`used internally for instanceof`
`Symbol.toPrimitive`
`used internally to convert to primitives`

Classes

Class

this is a class

```
class Jedi {
   constructor() {
      this.forceIsDark = false;
   }
   toString() {
      return (this.forceIsDark ? 'Join' :
         'Fear is the path to' ) +
        ' the dark side';
   }
}
```

you cannot define a property inside a class, but getters/setters can create one to the outside)

Extends

extends works as you'd expect

```
class Sith extends Jedi {
   constructor() {
      super();
      this.forceIsDark = true;
   }
}
```

super() in a ctor calls the parent ctor

Class

```
let yoda = new Jedi();
let darth = new Sith();

console.log(yoda.toString());
// Fear is the path to the dark side
console.log(darth.toString());
// Join the dark side

console.log(darth instanceof Sith); //
true
console.log(darth instanceof Jedi); //
true
```

Static

you can declare static functions

```
class Util {
   static inc(x) { return x + 1 },
   static dec(x) { return x - 1 }
}
```

```
console.log(Util.inc(3)); // 4
```

it’s the same as placing them on the prototype object

Class Get/Set

you can define get/set just like in ES5 object literals

```
class O {
   constructor() {
      this.mx = 'initial';
   }
   get x() {
      return  this.mx;
   }
   set x(val) {
      console.log('x changed');
      this.mx = val;
   }
}
```

Class Get/Set

and to the user of the class it would appear like a property

```
let o = new O();
console.log(o.x); //initial

o.x = 'newval';
// x changed
console.log(o.x); //newval
```

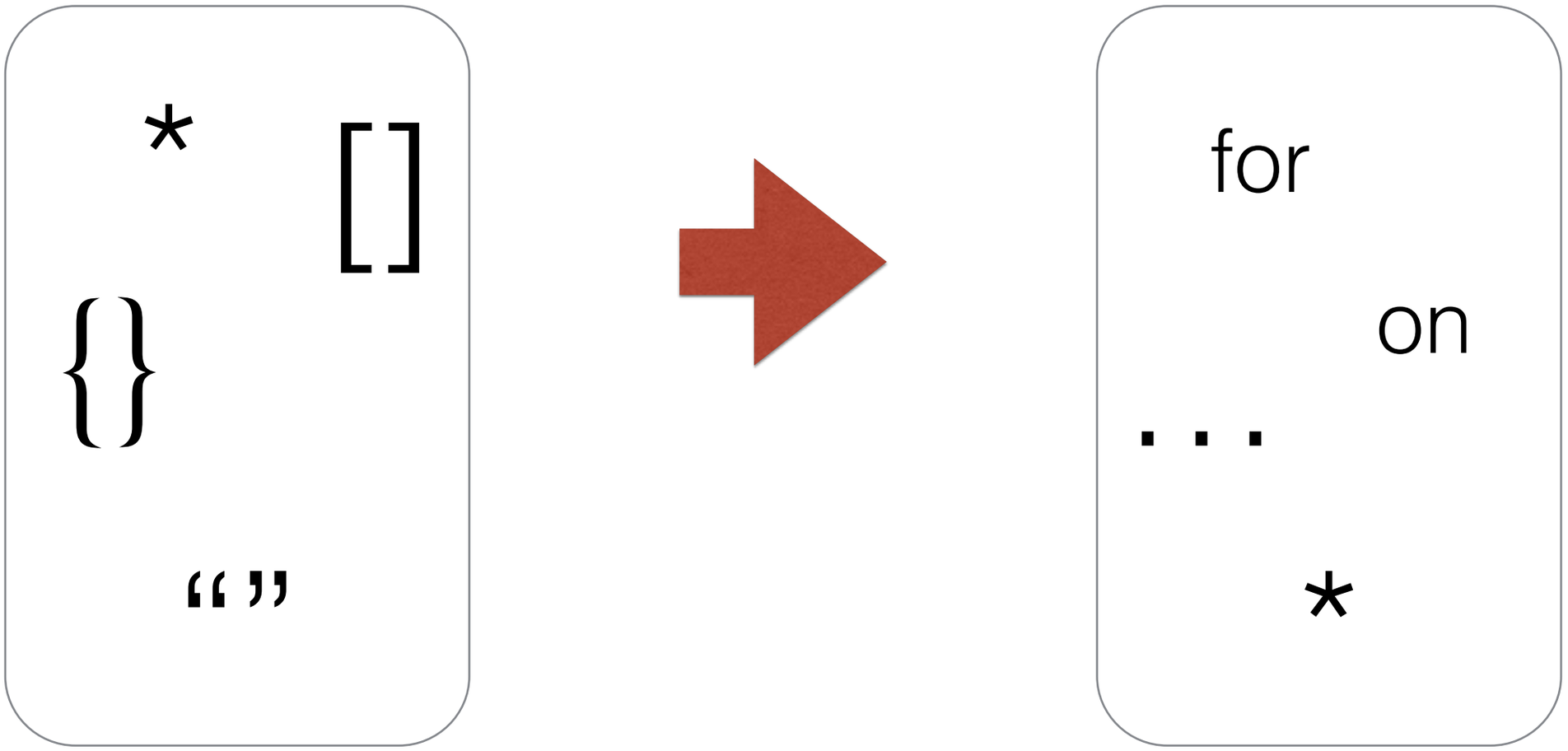
Class Hoisting

ES6 classes are **not** hoisted

```
var a = new ANumber(3); //ReferenceError

class ANumber {
  constructor(n) {
    this.x = n;
  }
}
```

The Iterator Protocol *



Sources

Consumers

The Useless Loop *

Iterating over an array using (for…in) (ES5)

```
var arr = ['a', 'b', 'c'];
for (var i in arr) {
   if (arr.hasOwnProperty(i)) {
      console.log(i);
   }
}
// '0'
// '1'
// '2'
```

The Iterator protocol *

ES6 bring the (for…of) iterator that works as you'd expect

```
for (let n of ['a', 'b', 'c']) {
  console.log(n);
}

// 'a'
// 'b'
// 'c'
```

The Iterator protocol *

other syntaxes that expect iterables:

```
// Spread Operator
[..."abcd"];
// Array ["a", "b", "c", “d"]
```

```
// Destructure Assignment
[a, b] = "xy";
b; // y

Array.from("123");
// Array [ "1", "2", "3" ]
```

The Iterator protocol *

manually iterating over an iterable object

```
let it = [1, 2, 3][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 }
```

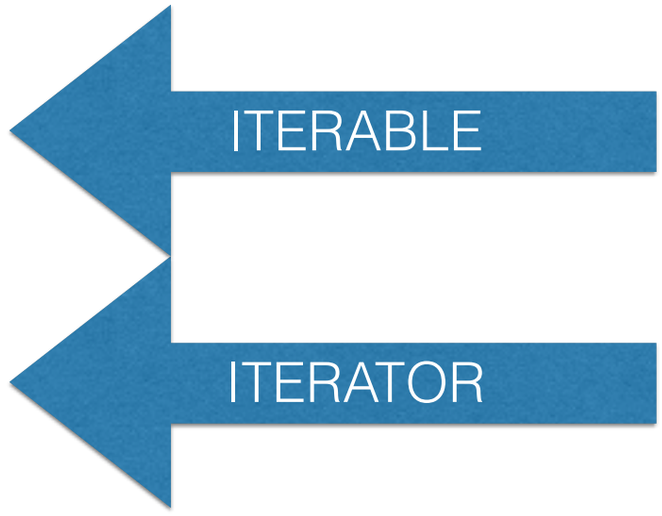
The Iterator protocol *

Iterable data sources

Arrays	Maps (Not WeakMaps)	Dom queries
Strings	Sets (Not WeakSets)	Arguments

The Iterator protocol *

Anything can be a source for iteration:




```
function gen(n) {
  return {
      [Symbol.iterator]() {
        let i = 0;
        return {
          next() { return {
            done: (i > n) ? true : false,
            value: i++ };
          }
        };
      }
  };
}
```

The Iterator protocol

Anything can be a source for iteration:

```
for (let n of gen(10) {
  console.log(n);
}
//1
//2
//...
//10
```

generators are a great way to create iterables. We'll get to that later

Libraries

String

String got some new functions

```
"Hello".startsWith('Hell');
"Goodbye".endsWith('bye');
"Jar".repeat(2); // JarJar
"abcdef".includes("bcd");
```

Firefox still uses *contains* instead of *includes*.
That should change

Number

New Number constants and methods

```
Number.EPSILON
Number.MAX_SAFE_INTEGER
Number.MIN_SAFE_INTEGER
```

```
Number.isNaN()
Number.isFinite()
Number.isInteger()
Number.isSafeInteger()
Number.parseFloat()
Number.parseInt()
```

Array.from()

from array-like objects (objects that have indices and length)

```
let arrayLike = {
   0: 'zero',
   1: 'one',
   2: 'two',
   3: 'three',
   'length' : 4
};

Array.from(arrayLike);
// Array ["zero", "one", "two", "three"]
```

Reminder…

genTen is a custom iterable that generates 0 to 10

```
function *
        [Symbol.iterator]() {
            done: (i > n) ?
            value: i++ };
        }
    };
  };
}
```

Array.from()

from iterables

```
Array.from(gen(6));
// Array [ 0, 1, 2, 3, 4, 5, 6 ]
```

second parameter is a mapping function

```
Array.from(gen(6), x => x*x);
//Array [ 0, 1, 4, 9, 16, 25, 36 ]
```

Array.of()

a better way to create arrays

```
Array.of(1, 2, 4, 3, 4);
//Array [ 1, 2, 4, 3, 4 ]
```

you should use Array.of over Array constructor because:

```
new Array(2.3);
//RangeError: invalid array length
```

Array.prototype.*

Array.prototype.keys()

```
['a','b','c'].keys()
// Array Iterator {  }
[...['a','b','c'].keys()]
// Array [ 0, 1, 2 ]
```

Array.prototype.entries()

```
Array.from(['a','b','c'].entries())
// [[0,"a"],[1,"b"],[2,"c"]]
```

Notice how keys() and entries() return an iterator and not an array

Array.prototype.*

Array.prototype.find()

```
[4, 100, 7].find(x => x > 5);
// 100
```

Array.prototype.findIndex()

```
[4, 100, 7].findIndex(x => x > 5);
// 1
```

Array.prototype.fill()

```
(new Array(7)).fill(2).fill(3, 2, 5);
// Array [ 2, 2, 3, 3, 3, 2, 2 ]
```

Object

Object.assign()

```
let x = {a:1};
Object.assign(x, {b:2});
x; // {a:1, b:2}
```

object.is() checks if two values are the same

```
Object.is('y', 'y'); // true
Object.is({x:1}, {x:1}); // false
Object.is(NaN, NaN); // true
```

different than === in (+0 !== -0) and (NaN === NaN)

also - doesn't coerce values (as == does)

Maps/Sets

Map

ES5 objects are not maps. ES6 introduces real maps

construction

```
var m = new Map([
   [1, ‘first']
]);
m;
// Map { 1: "first"}
```

Map

ES5 objects are not maps. ES6 introduces real maps

objects can be keys

```
var m = new Map([
   [1, 'first'],
   [{}, ‘second']
]);
m;
// Map { 1: "first", Object: "second"}
```

Map

ES5 objects are not maps. ES6 introduces real maps

functions can be key

```
var m = new Map([
   [1, 'first'],
   [{}, 'second']
]);
m.set(x => x+1, ‘third');
// Map { 1: "first", Object: "second",
function (): "third"}
```

Map

ES5 objects are not maps. ES6 introduces real maps

key equality is === (+NaN is equal to each other)

```
var m = new Map([
   [1, 'first'],
   [{}, 'second']
]);
m.set(x => x+1, 'third')
 .set({}, 'fourth');
// Map { 1: "first", Object: "second",
function (): "third", Object: "fourth" }
```

notice how using {} twice creates 2 different keys

Map

let's go over some basic Map api

```
var key = {};
var m = new Map([
   [key, 'first'], [1, 'second']
]);
```

```
m.get(key); // 'first'
m.has({}); // false
m.delete(key); // true
m.size; // 1
m.clear();
m.size; // 0
m.forEach((val, key) => { … });
```

Map

there are several ways to iterate over a map

```
let m =
new Map([...'abcd'].map(x=>[x, x + x]));
JSON.stringify([...m]);
// "[["a","aa"],["b","bb"],["c","cc"],
["d","dd"]]"
JSON.stringify([...m.keys()]);
// "["a","b","c","d"]"
JSON.stringify([...m.values()]);
// "["aa","bb","cc","dd"]"
JSON.stringify([...m.entries()]);
// "[["a","aa"],["b","bb"],["c","cc"],
["d","dd"]]"
```

WeakMap

WeakMap is a Map which doesn't prevent garbage collection

- you can't iterate over it
- no clear() method
- doesn't have size property
- keys must be objects

WeakMap is good for hiding private class members

Set

Set is a collection of distinct objects

```
let s = new Set(['red', 'blue']);
s.add('yellow'); // adds yellow
s.add('red'); // doesn't add red
s.has('blue'); // true
s.delete('blue'); // true
s.size; // 2

s; // Set ["red", "blue"]
[...s]; // Array [ "red", "blue" ]
s.clear();
```

Set

sets are iterable

```
JSON.stringify([...s.values()]);
// "["red","blue"]"
```

for the sake of symmetry with maps, this works:

```
JSON.stringify([...s.entries()]);
// "[["red","red"],["blue","blue"]]"
JSON.stringify([...s.keys()]);
// "["red","blue"]"
```

WeakSets

WeakSet exists but it doesn't allow iteration. so you can't do a whole lot with it.
You can mark objects (check if in set or not)

Generators

function*

```
function* genFour() {
  yield 1;
  yield 2;
  yield 3;
  return 4;
}
let four = genFour();
```

the function suspends on every yield and allows other code to run until the next time it resumes

Consuming

generators are both an iterator and an iterable

as an iterator:

```
four.next();
// Object { value: 1, done: false }
four.next();
// Object { value: 2, done: false }
four.next();
// Object { value: 3, done: false }
four.next();
// Object { value: 4, done: true }
```

Consuming

generators are both an iterator and an iterable

as an iterable:

```
[...genFour()];
// Array [ 1, 2, 3, 4 ]
Array.from([...genFour()]);
// Array [ 1, 2, 3, 4 ]
```

Here is an example consuming an iterator step by step

	`three.next();`
`yield 1;`	`// {value:1, done:false}`
	`three.next();`
`yield 2;`	`// {value:2, done:false}`
	`three.next();`
`yield 3;`	`// {value:3, done:false}`
	`three.next();`
`return 4;`	`// {value:4, done:true}`
	`three.next();` `// Object { value: undefined, done: true }`

yield*

yield* yields every yield inside a called generator

```
function* flatten(arr) {
  for (let x of arr) {
    if (x instanceof Array) {
      yield* flatten(x);
    } else {
      yield x;
    }
  }
}
let t = flatten([1, 2, [3, 4]]);
// Array [1, 2, 3, 4]
```

Arrow Functions

The Basics

```
function inc(x) {
   return x + 1;
}
```

is equivalent to:

```
let inc = x => x + 1;
```

2 parameters:

```
let inc = (x, y) => x + y;
```

no parameters

```
let inc = () => 7;
```

The Basics

```
function inc(x) {
   return x + 1;
}
```

more than 1 statement

```
let inc = (x) => {
   console.log(x);
   return 7;
}
```

Lexical this

Arrow functions capture the this value of the enclosing context. In other words, no more `that`, `self` or `bind(this)`

```
this.x = 7;
setTimeout(() =>
   console.log(this.x),
   2000);
// wait for it....

// 7
```

Promises

Incentive

```
requestFromServer(function (err, res) {
  if (err) {
    // do something with error
  } else {
    // do something with result
    doSomethingAsync(function () {
      andAnotherAsyncThing(function () {
        andAnotherAsyncThing(function () {
          andAnotherAsyncThing(function () {
            andAnotherAsyncThing(function () {
              andAnotherAsyncThing(function () {
                andAnotherAsyncThing(function () {
                  // CALLBACK PYRAMID OF DOOM!!!
                });
              });
            });
          });
        });
      });
    });
  }
});
```

Chaining async function calls used to look like this

Using Promises

```
requestFromServer()
.then(function (res) { return res2;})
.then(function (res2) { return res3;})
// ...
.then(function (res3) { return res4;})
.catch(function (err) { …act…;});
```

once an error is thrown, it is passed through the chain until it meets the first catch

Flattening

if a **thenable** object is returned inside a **then()** block, the param to the next then will already be the result of that object

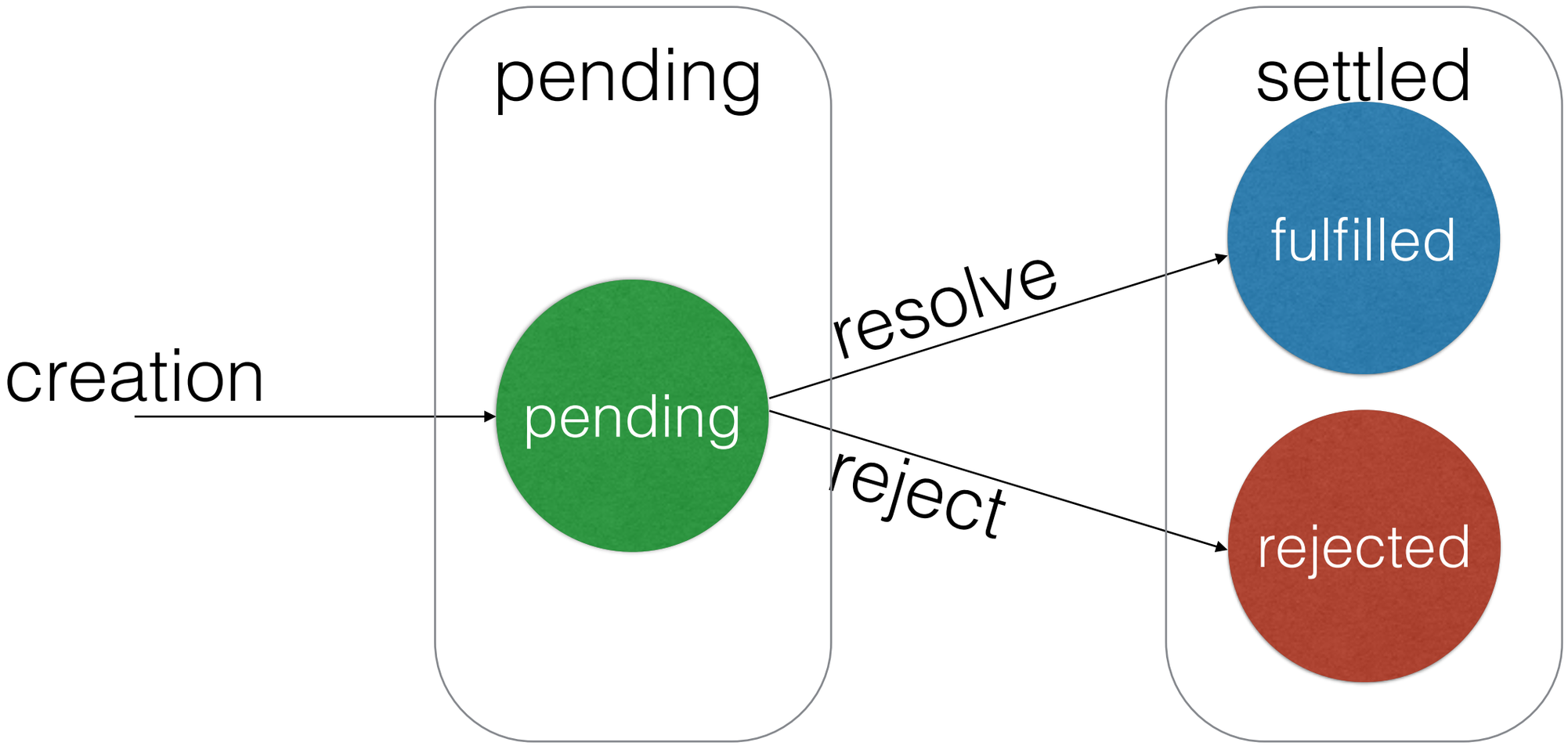
```
requestFromServer()
.then(function (res) {
   return readFile(res);
}).then(function (file) {
   // do something with file
});
```

notice the second then gets a *file* and not the promise object returned by readFile(res)

Creating Promises

```
new Promise(function (resolve, reject) {
   if (testSomething()) {
      resolve(resolutionValue);
   } else {
      reject(errorValue);
   }
});
```

State of a Promise



once a promise has been settled, all calls to resolve() or reject() will have no effect

Promise.all()

Promise.all() resolves when all operations and values passed are resolved.

```
Promise.all(
  ['1', 'abcde',readFile('notes.txt')])
.then(function (res) {
   console.log(res);
});

// Array [ "1", "abcde", "file content" ]
```

read more (good stuff): 2ality

Destructuring

The Basics

You're used to structuring source objects in ES5

```
let a = {x:1, y:2};
```

in ES6, you can destructure the target

```
let {x:x} = a; // = {x:1, y:2}
console.log(x); // 1
```

notice we only took x's value

The Basics

remember: param key is the source, value is the target.

```
let a = {x:1, y:2};
let {x:x, y:z} = a;
console.log('x=' + x); // 1
console.log('z=' + z); // 2
```

also, the shorthand for {x:x} is {x}

```
let a = {x:1, y:2};
let {y} = a;
console.log('y=' + y); // 2
```

The Basics - Objects

nesting works

```
let a = {x: { y: 3 }};
let {x: { y: z}} = a;
console.log('z=' + z); // 3
```

any object property can be read:

```
let {length: len} = [1,2,3,4];
console.log(len); // 4
```

primitives are coerced to objects

```
let {length: len} = ‘abcd‘;
console.log(len); // 4
```

The Basics - Arrays

with arrays it's similar

```
let [a, [b]] = [3, [5]];
console.log(b); // 5
```

destructuring to array will iterate on the source

```
function* genInf() {
  for (let i =0;;i++) yield i;
}
let [a,b,c] = genInf();
// a === 0, b === 1, c === 2
```

The Basics - Arrays

another example with regular expressions

```
let [all, prefix, host] =
/(http[s]?):\/\/([^\/]*)/
.exec('http://www.wix.com/my-
account');

console.log(prefix); // rest
console.log(host); // www.wix.com
```

Default Values

default value is used if match on src is undefined (either missing or actual value)

```
let [a,b = 3, c = 7] = [1, undefined];
// a === 1, b === 3, c === 7
```

works on both arrays and objects

```
let {x:x = 1 ,y:y = 2,z:z = 3} =
    { x: undefined, z: 7};
// x === 1, y === 2, z === 7
```

Default Values

default values are lazily evaluated

```
function con() {
   console.log('TEST');
   return 10;
}
```

```
let [x = con()] = [];
// TEST
// x === 10

let [x = con()] = [5];
// x === 5
```

Default Values

default values evaluation is equivalent to a list of let declarations

this fails

```
let [x = y, y = 0] = [];
// ReferenceError
```

this works

```
let [x = 7, y = x] = [];
// x === 7, y === 7
```

Rest Operator

rest (...) operator allows to extract remaining values into array

```
let [,, ...y] = [1,2,3,4,5];
console.log(y); // Array [3, 4, 5]
```

notice we omitted value to skip them. (Elision)

Destructuring Parameters

function parameters are destructured

```
let reverse = ([x, ...y]) =>
   (y.length > 0)?[...reverse(y), x]:[x];

reverse([1,2,3,4,5,6]);
// [6,5,4,3,2,1]
```

the equivalent to the parameter destructuring we did is:

```
let [x, ...y] = [1,2,3,4,5,6];
```

notice we used the spread (…) operator which flattens an array

Spread

flatten arrays

```
[1, 2, ...[3, 4]] // Array [1, 2, 3, 4]
```

use arrays as function parameters

```
Math.max(-5, ...[2, 5, 4], 1); // 5
```

Spread

spread iterables into array or params

```
function* squares(n) {
  for (let i = 1; i < n; i += 1) {
    yield Math.pow(i,2);
  }
}
```

reminder: generators are iterable

```
[...squares(10)];
// [1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Modules

Modules

Modules allow you to load code on demand (async) and to provide a level of abstraction

```
//******  Shape.js  *******
export default class Shape {
   getColor() { /*...*/ }
}

//******  main.js  *******
import Shape from 'Shape';
let shape = new Shape();
```

this method exports a single default value for the module

Modules

you can also export by name

```
//******  Circle.js *******
export function diameter(r) {
   return 2 * Math.PI * r;
}
export let area = r =>
   Math.pow(Math.PI*r, 2);
```

Modules

and then import by name

```
//******  main.js  *******
import {area as circleArea, diameter}
   from 'Circle';
import area as cubeArea from 'Cube';
import * as tri from 'Triangle';

circleArea(5);
cubeArea(4);
tri.area(3);
```

Modules

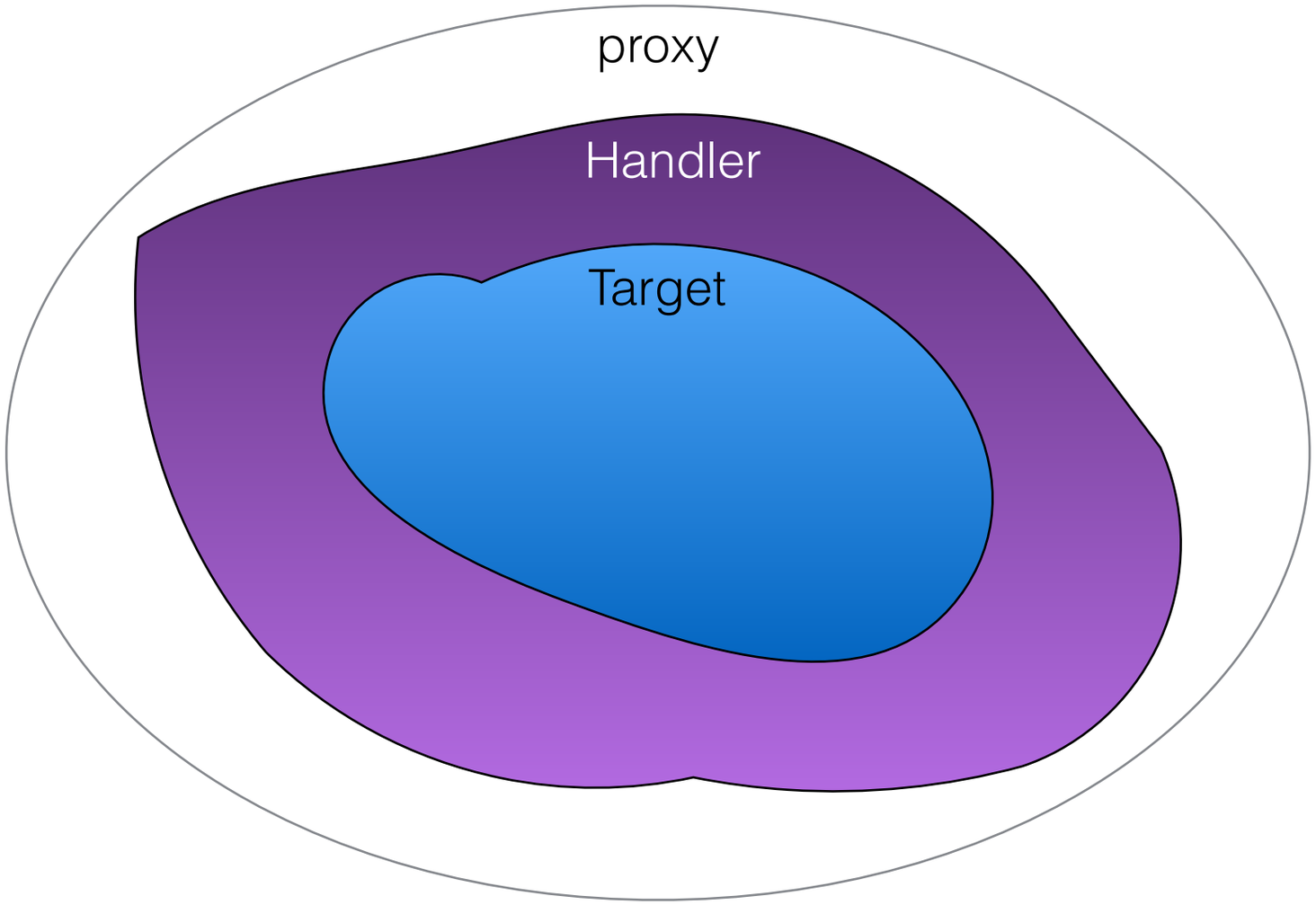
- modules structure is statically defined.
- dependencies can be resolved statically.
- you can still load modules programatically if you choose

```
System.import('module')
.then(module => {})
.catch(error => {});
```

Proxy

Proxy

Proxy is an object that allows you to trap an operation and manipulate it.



Proxy

for example, here we trap the function application

```
let target = function () {};
let handler = {
   apply(target, thisArg, argsList) {
      return 42;
   }
}
let p = new Proxy(target, handler);

console.log(p()); // 42
```

Proxy

and here we trap a get operation on an object property

```
let target = { a: 7 };
let handler = {
   get(target, name) {
      return target[name] + 10;
   }
}
let p = new Proxy(target, handler);

console.log(p.a); // 17
console.log(p.b); // NaN
```

the end

whaddya wanna know?