ES2015 / **ES**6

Basics of modern JavaScript

Agenda

1. Evolution of JavaScript
2. Main goals of JavaScript and ES6
3. ES6 in practice (selected features)
4. Q&A

Dictionary

- **JavaScript (JS)** - a high-level, dynamic, untyped, and interpreted programming language created originally for web browsers, ECMAScript *implementation*

- **ECMA International** - an international non-profit standards organization for information and communication systems.It acquired its current name in 1994, when the European Computer Manufacturers Association (ECMA) changed its name to reflect the organization's global reach and activities

- **ECMAScript (ES)**- scripting-language *specification* standardized by Ecma International in ECMA-262 and ISO/IEC 16262. Well-known implementations of the language, such as JavaScript, JScript and ActionScript have come into wide use for client-side scripting on the Web

- ES**2015 (**ES**6)** - the newest version of ECMAScript

90s

- *1995:* Netscape creates *Mocha*
- *1995:* Mocha -> LiveScript -> **JavaScript**
- **1996: ECMA** adopts *JavaScript*
- 1997: ECMA-262 (ES1)
- 1998: ES2
- 1999: ES3 (regex, try/catch)

90s

- browser wars IE vs Netscape

- DHTML, “animate everything”
- forms validation
- visitor counters

- code had to be optimised per browser (IE vs Netscape)

2000-2004

- browser wars - IE wins
- not much going on in JS world

2005: AJAX

- Broadband Internet becomes popular
- Asynchronus server requests (AJAX) becomes popular
- renaissance of JavaScript
- countless libraries (mainly helping with AJAX requests and DOM operations)

2006-2009

- 2008: *ECMAScript4* (abandoned)
- *2009:* **ECMAScript** ~~3.1~~ **5** (strict, JSON, Reflect)
- 2009: servers welcome JavaScript: Node.js

2010-2015

- frameworks evolution, no longer just DOM & AJAX helpers

- JS packet managers: **npm**, **bower**

- solutions for keeping code in modules (node.js, CommonJS, AMD, Browserify)

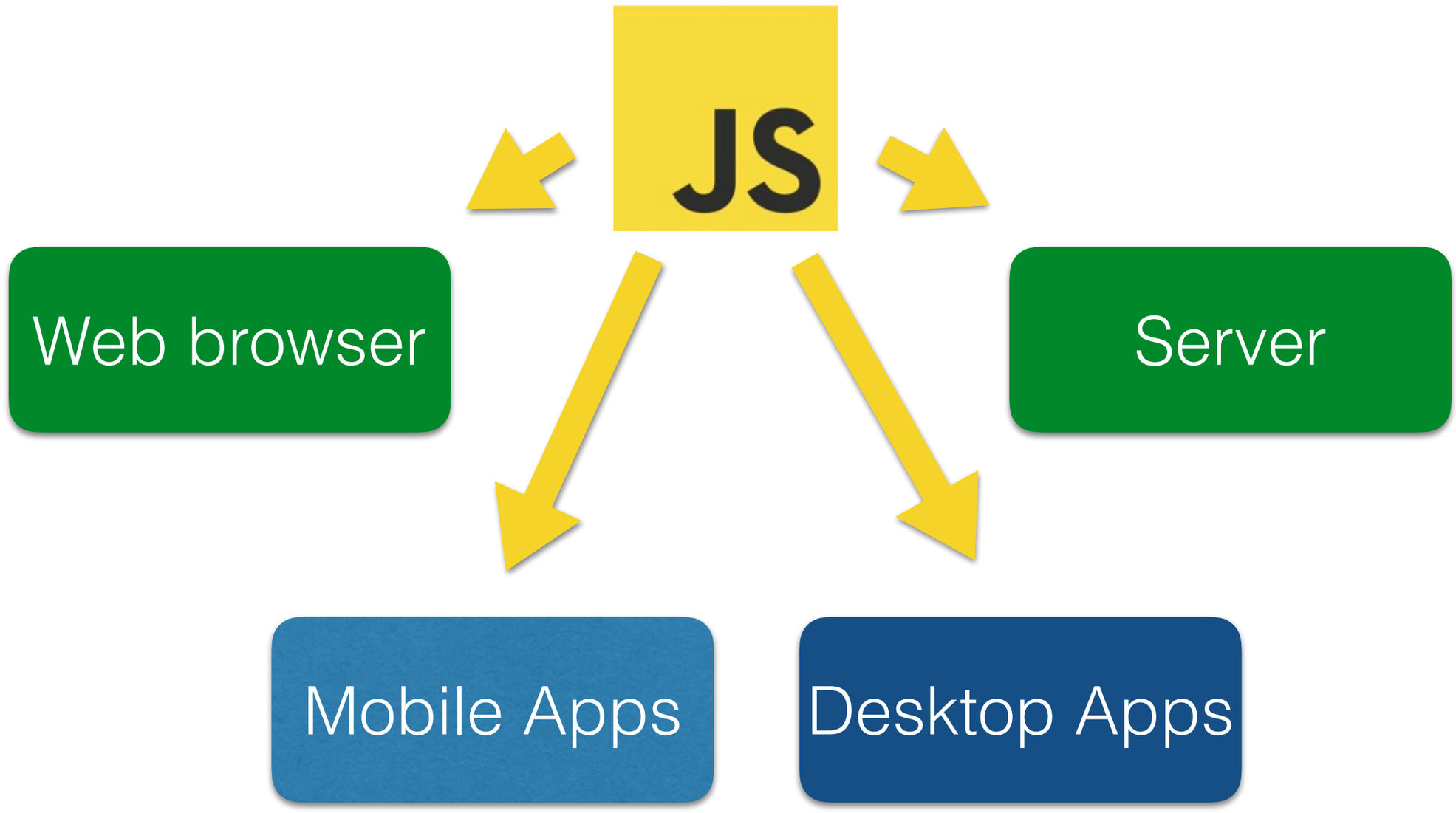
- JavaScript preprocessors (Grunt, Gulp, Webpack,…)



Now

- 2015: **ECMAScript 2015** (lots of features)
- since 2015 a new ES spec will be released each year
- upcoming: ES2016 (no major changes)

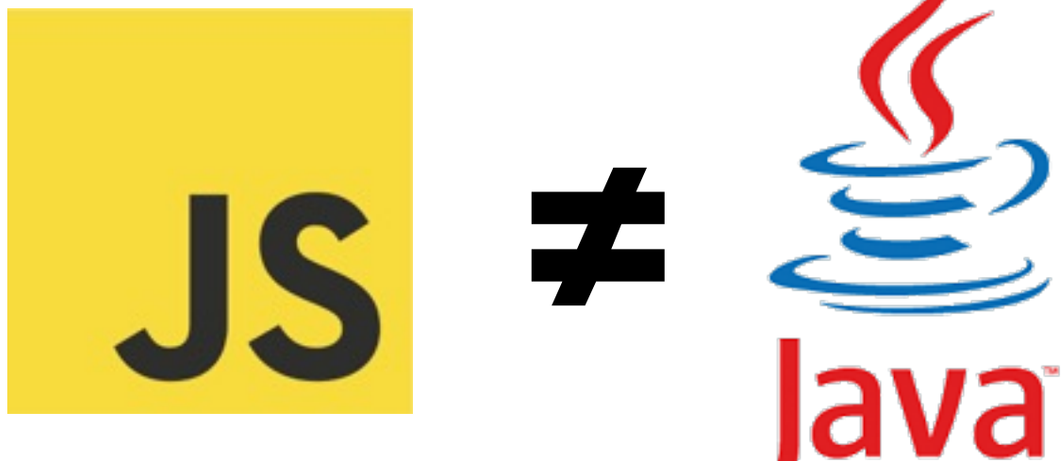
2016

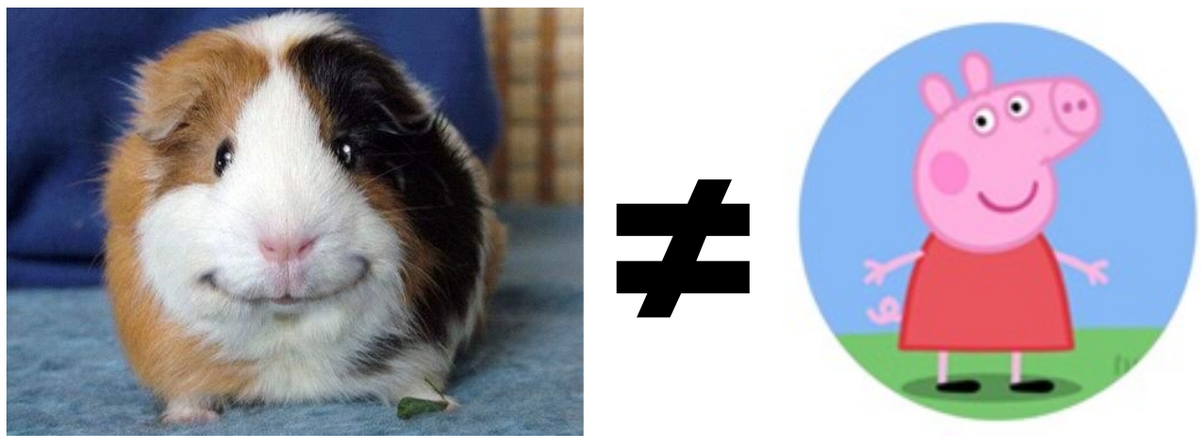


JavaScript - **pros**

- easy syntax
- functions are objects (awesome!)
- independent from any big company
- the only native web browser language
- big and vibrant community
- lots of helpful tools, libraries and frameworks

JavaScript ≠ Java





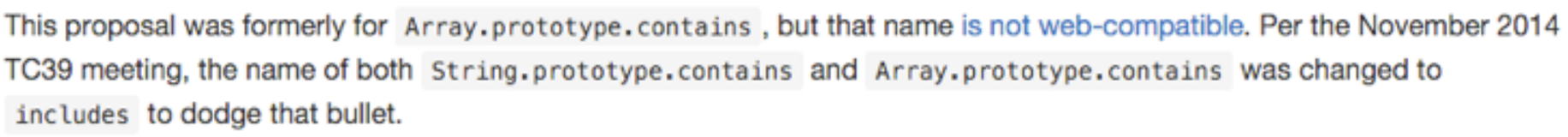
Guinea pig ≠ pig

JavaScript - **cons** (subjective list)

- **differences in comparison to Java-like languages can create confusion** (prototypes vs classes,function scope vs block scope, hoisting, +, …)

JavaScript - **cons** (subjective list)

- not many unequivocal clean code practices (each framework = new practices, enforcing bad practices can be harmful to community)


This proposal was formerly for `Array.prototype.contains`, but that name is not web-compatible. Per the November 2014 TC39 meeting, the name of both `String.prototype.contains` and `Array.prototype.contains` was changed to `includes` to dodge that bullet.

- very rapid development often makes tools and frameworks obsolete fast, it is hard to choose frameworks and tools for apps that need to be maintained for years

Main goals of ES6

- fix (*some of)* ES5 problems
- backwards compatibility (ES5 code is valid in ES6)
- modern syntax
- better suited for big applications
- new features in standard library

ES6 in practice

ES5: var

```
1  var foo = 'OUT'
2
3  {
4    var foo = 'IN'
5  }
6
7  console.log(foo) //IN
8
```

ES5: var - hoisting

```
1  var foo
2
3  foo = 'OUT'
4
5  {
6    foo = 'IN'
7  }
```

ES6: ***let*** is new *var*

```
let foo = 'OUT'

{
  let foo = 'IN'
}

console.log(foo) //OUT
```

ES5...

```
1  'use strict'
2
3  function foo() {
4    console.log('original')
5  }
6
7  foo = function() {
8    console.log('hijacked')
9  }
10
11 foo(); //hijacked
12
```

Atom Runner: example.js

```
hijacked
```

Exited with code=0 in 0.092 seconds

…ES6: **const**

```
1  'use strict'
2
3  const foo = function() {
4    console.log('original')
5  }
6
7  foo = function() { // Error
8    console.log('hijacked')
9  }
10
11 foo();
12
```

Atom Runner: example.js

```
/Users/veedzk/es6/example.js:7
foo = function() {
    ^

TypeError: Assignment to constant variable.
    at Object.<anonymous> (/Users/veedzk/es6/
example.js:7:5)
    at Module._compile (module.js:399:26)
    at Object.Module._extensions..js (module.
js:406:10)
    at Module.load (module.js:345:32)
    at Function.Module._load (module.js:302:1
2)
    at Function.Module.runMain (module.js:431
:10)
    at startup (node.js:141:18)
    at node.js:977:3
```

Exited with code=1 in 0.072 seconds

ES5: long strings

```
var myString = 'A rather long string of English text, an error message \
                actually that just keeps going and going -- an error \
                message to make the Energizer bunny blush (right through \
                those Schwarzenegger shades)! Where was I? Oh yes, \
                you\'ve got an error and all the extraneous whitespace is \
                just gravy.  Have a nice day.'
```

```
var myString = 'A rather long string of English text, an error message ' +
       'actually that just keeps going and going -- an error ' +
       'message to make the Energizer bunny blush (right through ' +
       'those Schwarzenegger shades)! Where was I? Oh yes, ' +
       'you\'ve got an error and all the extraneous whitespace is ' +
       'just gravy.  Have a nice day.'
```

ES6: **Template strings**

```
1  `Prosty string.`
2
3  // Multiline strings
4  `ES5 tego
5   nie potrafi.`
6
7  // Interpolate variable bindings
8  let name = "Jan",
9      language = "JavaScript",
10     phrase = `${name} uwielbia ${language}!`
11
12 // Unescaped template strings
13 String.raw`W ES5 "\n" przechodzi do nowej linii`
```

ES6: Object declarations


```
let foo = 'foo'

let es5_object = {
  classicParam: foo,
  foo: foo,
  doStuff: function() {
    return 'function call'
  }
}

let es6_object = {
  classicParam: foo,
  foo,
  doStuff() {
    return 'function call'
  }
}
```

ES6: Classes

```
class Animal {
  constructor(type = 'animal') {
    this.type = type
  }

  get type() {
    return this._type
  }

  set type(val) {
    this._type = val.toUpperCase()
  }

  makeSound() {
    console.log('Making animal sound')
  }
}

let a = new Animal()
console.log(a.type) //ANIMAL
```

```
class Cat extends Animal {
  constructor(){
    super('cat')
  }

  makeSound() {
    super.makeSound()
    console.log('Meow!')
  }
}

let b= new Cat()
console.log(b.type) //CAT
```

ES6: Classes

```
class Animal {
  constructor(type = 'animal') {
    this.type = type
  }

  get type() {
    return this._type
  }

  set type(val) {
    this._type = val.toUpperCase()
  }

  makeSound() {
    console.log('Making animal sound')
  }
}

let a = new Animal()
console.log(a.type) //ANIMAL
```

```
class Cat extends Animal {
  constructor(){
    super('cat')
  }

  makeSound() {
    super.makeSound()
    console.log('Meow!')
  }
}

let b= new Cat()
console.log(b.type) //CAT
```

ES6: Setters & getters

```
class Animal {
  constructor(type = 'animal') {
    this.type = type
  }

  get type() {
    return this._type
  }

  set type(val) {
    this._type = val.toUpperCase()
  }

  makeSound() {
    console.log('Making animal sound')
  }
}

let a = new Animal()
console.log(a.type) //ANIMAL
```

```
class Cat extends Animal {
  constructor(){
    super('cat')
  }

  makeSound() {
    super.makeSound()
    console.log('Meow!')
  }
}

let b= new Cat()
console.log(b.type) //CAT
```

ES6: Default params

```
class Animal {
  constructor(type = 'animal') {
    this.type = type
  }

  get type() {
    return this._type
  }

  set type(val) {
    this._type = val.toUpperCase()
  }

  makeSound() {
    console.log('Making animal sound')
  }
}

let a = new Animal()
console.log(a.type) //ANIMAL
```

```
class Cat extends Animal {
  constructor(){
    super('cat')
  }

  makeSound() {
    super.makeSound()
    console.log('Meow!')
  }
}

let b= new Cat()
console.log(b.type) //CAT
```

ES5 recap: **map**

```
1  let arr = [1, 2, 3]
2
3  let duplicatedArr = arr.map(function(el) {
4    return el * 2
5  }) // [2, 4, 6]
```

```
10  let duplicatedArr = []
11  for (let i=0; i< arr.length; i++) {
12    duplicatedArr.push(arr[i] * 2)
13  }
```

ES5 recap: **filter**

```
1  let arr = [1, 2, 3]
2
3  let evenArr = arr.filter(function(el){
4    return el % 2 === 0
5  }) // [2]
```

```
12 let evenArr = []
13 for (let i=0; i< arr.length; i++) {
14   if (arr[i] % 2 === 0){
15     evenArr.push(arr[i])
16   }
17 }
18
```

ES5 recap: **reduce**

```
1  let arr = [1, 2, 3]
2
3  let sum = arr.reduce(function(sumSoFar, el){
4    return sumSoFar + el
5  }, 0) // 6
```

```
12 let sum = 0
13 for (let i=0; i< arr.length; i++) {
14   sum = sum + arr[i]
15 }
```

ES6: **Arrow functions**

```
function(a,b) {
  return a + b
}

(a, b) => {
  return a + b
}

(a, b) => a + b
```

```
function(a){
  return a
}

a => a + a
```

ES6: **Arrow functions**

```
1  let arr = [1, 2, 3]
2  let duplicatedArr = arr.map(el => el * 2)                    //[2,4,6]
3  let evenArr = arr.filter(el => el % 2 === 0)                 //[2]
4  let sum = arr.reduce( (sumSoFar, el) => sumSoFar + el, 0)  //6
5
```

Arrow functions: **this**

```
function($http){
  this.data = 'old'

  this.updateData = function() {
    var that = this
    $http.get('http://example.com').then(funtion(newData){
          that.data = newData
      })
  }

 }
```

```
function($http){
  this.data = 'old'

  this.updateData = function() {
    $http.get('http://example.com')
      .then(newData => this.data = newData)
  }

}
```

ES5: **for … in**

- best practice: **avoid that loop**

```
var a = [];
a[5] = 5;
for (var x in a) {
    // Shows only the explicitly set index of "5", and ignores 0-4
}
```

```
for (var prop in obj) {
  if( obj.hasOwnProperty( prop ) ) {
    console.log("obj." + prop + " = " + obj[prop]);
  }
}
```

ES6: **for … of**

```
1  let arr = [1, 2, 'three', 'cztery']
2
3  for (let el of arr){
4    console.log(el)
5  }
6
7  // Wynik:
8  // 1
9  // 2
10 // three
11 // cztery
```

- for … of can iterate not only over arrays
- *Homework:* Iterators in ES6

Asynchronus programming

- common in JS (animations, server requests, etc.)
- Classic solution: **callback**
- **Problem:** only one callback per async task

```
const update = function(callback) {
    setTimeout(()=> callback('slow data'), 5000)
}

update(slowData => {
  //process slowData
})
```

Async programming, ES5

- **Problem:** Nested functions create messy code

```
1  //callback hell
2  function getCompanyFromOrder(orderId) {
3    fetchOrder(orderId, function(order){
4      fetchUser(order.userId, function(user)){
5        fetchCompany(user.companyId, function(company){
6            //zrób coś z firmą
7        })
8      }
9    })
10 }
```

Async programming, ES5

- Second try: Listeners
- **Problem:** no reaction when async function ends before listener registers, often hard to debug

```
updater.on('done', (event, slowData) => {
  //process slowData
})
```

ES6: **Promise**

- object that keeps a result of an async function (waiting, resolved, rejected)

- fixes earlier problem with listeners, since callback is called even if async function completed the task earlier

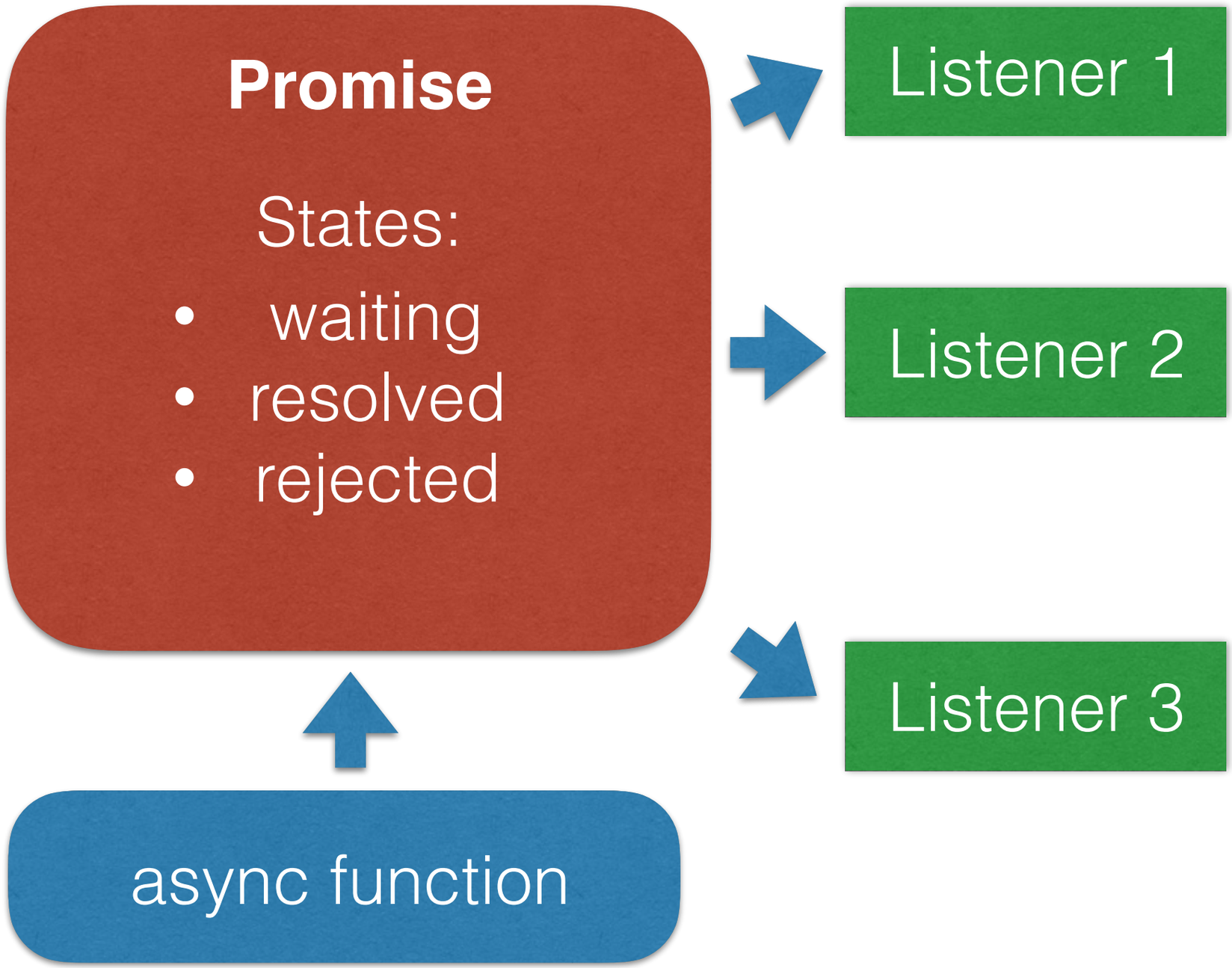
- allows to return (Promise) objects and work with them, even if async function is still not completed (better code readability)

- “promises” that it will get resolved

- uses Observer pattern to populate the result

- flattens nested promises to avoid “callback hell”

Promises


Promise
States:
• waiting
• resolved
• rejected
Listener 1
Listener 2
Listener 3
async function

Promises

```
1  const update = function() {
2      let promise = new Promise((resolve,reject) => {
3        setTimeout(()=> resolve('slow data'), 5000)
4      })
5      return promise
6  }
7
8  update().then(
9    slowData => {
10     //process slowData
11   },
12   error => {
13     //handle error
14   })
```

Callback hell, ES5

```
1  //callback hell
2  function getCompanyFromOrder(orderId) {
3    fetchOrder(orderId, function(order){
4      fetchUser(order.userId, function(user)){
5        fetchCompany(user.companyId, function(company){
6          //zrób coś z firmą
7        })
8      }
9    })
10 }
```

Promises

```
1  // fetchOrder() returns Promise
2  // fetchUser() returns Promise
3  // fetchCompany() returns Promise
4
5  const getCompanyFromOrder = function(orderId) {
6
7    let promise = fetchOrder(orderId)
8      .then(order => fetchUser(order.userId))
9      .then(user  => fetchCompany(user.companyId))
10
11   return promise
12 }
13
14 getCompanyFromOrder().then(company => {
15   //zrób coś z firmą
16 })
17
```

Modules (ES5)

- **IIFE** (Immediately Invoked Function Expression)
- controls variable exposure

```
1  (function(){
2    'use strict'
3
4    var foo = 'foo'
5  })()
6
7  foo // Error: foo is not defined
8
```

Modules (ES6) in browser

employee.js

```
export class Employee {
  constructor(name) {
    this._name = name
  }

  get name() {
    return this._name
  }

  work() {
    return `${this._name} pracuje`
  }
}

```

example.js

```
import {Employee} from './employee'

let e = new Employee('Jaś')
e.work() //Jaś

```

- use tools such as Browserify / Webpack,
- native ES6 modules are not yet implemented

What we have learned?

- let/const
- template strings
- new ways to declare objects
- classes
- map, filter, reduce (ES5)
- arrow functions
- for … of
- Promises
- Modules

Other ES6 features

- Proxy
- Iterators
- Generators
- Symbols
- Map/Set, WeakMap/WeakSet
- extended standard library (Number, Math, Array)