ES6, ES7

L. Vilímek J.Šmolík

Ackee s.r.o

2016

In ES5, all variables are function or global scoped (you should already know that).

ES6 introduces block scope

- `let`
- `const`

Const

The `const` declaration creates a read-only reference to a value. (It does not mean the value it holds is immutable, just that the variable identifier cannot be reassigned.)

```
1 const PI = 3.141593
2 PI > 3.0     //true
3 PI = 3.14    //syntax error
```

Const & objects

```
1  const MY_OBJECT = {"key": "value"}; //assignment
2  MY_OBJECT = {"OTHER_KEY": "value"}; //fail -
   ↪  identifier reassignment
3  MY_OBJECT.key = "otherValue";       //is ok
```

Let

The `let` statement declares a block scope local variable, optionally initializing it to a value.

In sctrict-mode only.

This is wrong. Why?

```
1  var callbacks = [];
2  for (var i = 0; i <= 2; i++) {
3      callbacks[i] = function() { return i * 2; };
4  }
```

Why Let

Correct way to do this in ES5.

```
1 var callbacks = [];
2 for (var i = 0; i <= 2; i++) {
3     (function (i) {
4         callbacks[i] = function() { return i * 2; };
5     })(i);
6 }
```

Let usage

Much more convinient way in ES6 using `let`.

```
1  for (let i = 0; i < a.length; i++) {
2      let x = a[i]
3      ...
4  }
```

Scoping functions

```
{
    function foo () { return 1 }
    //foo() === 1
    {
        function foo () { return 2 }
        //foo() === 2
    }
    //foo() === 1
}
```

Classes

- ES6 classes does not add any new object-oriented inheritance models
- Classes are only syntactical sugar over existing prototype-based inheritance.

$\rightarrow$ clearer and simpler syntax.

Defining a class

```
1  class Shape {
2      constructor (id, x, y) {
3          this.id = id
4          this.move(x, y)
5      }
6      move (x, y) {
7          this.x = x
8          this.y = y
9      }
10 }
11 var shape = new Shape(1, .25, 34)
```

Constructor

- `constructor` - a special method initializing object created with `class`.
- There can be only one constructor.
- `constructor` *may* use `super` keyword to call parent constructor (we'll get to that)

Static methods

```
1  class Point {
2    constructor(x, y) {
3      this.x = x;
4      this.y = y;
5    }
6    static distance(a, b) {
7      const dx = a.x - b.x;
8      const dy = a.y - b.y;
9      return Math.sqrt(dx*dx + dy*dy);
10   }
11 }
12 var p1 = new Point(1, 2);
13 var p2 = new Point(1, 1);
14 var distance = Point.distance(p1, p2);
```

Getters/setters I

Getters/setters II

Not new in ES6, but remember:

```
1  class Rectangle {
2      constructor (width, height) {
3          this._width  = width;
4          this._height = height;
5      }
6      set width  (width)  { this._width = width                }
7      get width  ()       { return this._width                 }
8      set height (height) { this._height = height              }
9      get height ()       { return this._height                }
10     get area   ()       { return this._width * this._height }
11 }
12 var r = new Rectangle(50, 20)
13 r.area === 1000; //calls getter function
```

Computed property name getters

and new to ES6: computed property name

```
1 var expr = "foo";
2 var obj = {
3   get [expr]() { return "bar"; }
4 };
5 console.log(obj.foo); // "bar"
```

Inheritance I

Sublcassing with `extends` keyword.

```
1  class Rectangle extends Shape {
2  constructor (id, x, y, width, height) {
3      super(id, x, y);       // calls the parent
   ↪  constructor.
4      this.width  = width;
5      this.height = height;
6  }
```

- You can also extend an ES5 defined function class
- `super` in constructor appears alone and must be used before `this` is used.

Inheritance II

- `super` can be used to call parent method:
  `super.functionOnParent([arguments]);`

Arrow function expression I

```
1  (param1, param2, ..., paramN) => { statements }
2  (param1, param2, ..., paramN) => expression
3           // equivalent to:  => { return expression; }
```

- is always anonymous
- has shorter syntax (compared to function expressions)
- binds `this` value from outer scope (no `self`/`that`/`_this` or `bind` needed)

Arrow function expression II

```
1  function Person(){
2    this.age = 0;
3
4    setInterval(() => {
5      this.age++; // |this| properly refers to the
   ↪   person object
6    }, 1000);
7  }
8
9  var p = new Person();
```

Promise I

A Promise represents an operation that hasn't completed yet, but is expected in the future — *future value*

NOTE: Primary mechanisms to handle asynchronous events in Javascript are callback functions - a piece of code, that is called once an event has occurred.

Promises are not about replacing callbacks, but provide an *trustable intermediary* to manage callbacks.

```
1 new Promise(...).then(...).then(...).catch(...);
```

Promise lifecycle I

A Promise can only have one of two possible resolution outcomes

- fullfilled
- rejected

Promise can be resolved only once and promise once resolved becomes an immutable object.

```
1 promise = new Promise((resolve, reject) => {
2     if (ok) {
3         return resolve(true);
4     }
5     reject(new Error('Not ok'));
6 });
7 promise.then(console.log).catch(console.error);
```

Promise lifecycle II

Promise constructor takes a single function, which is called immediately and receives two control functions as arguments. Usually named *resolve* and *reject*.

- If you call `reject`, the promise is rejected and any value passed to `reject` function is the reason of rejection.
- If you call `resolve` with no value, or any non-promise value, the promise is fullfilled.
- If you call `resolve` and pass another promise, this promise adopts the state of the passed promise.

Promise lifecycle III

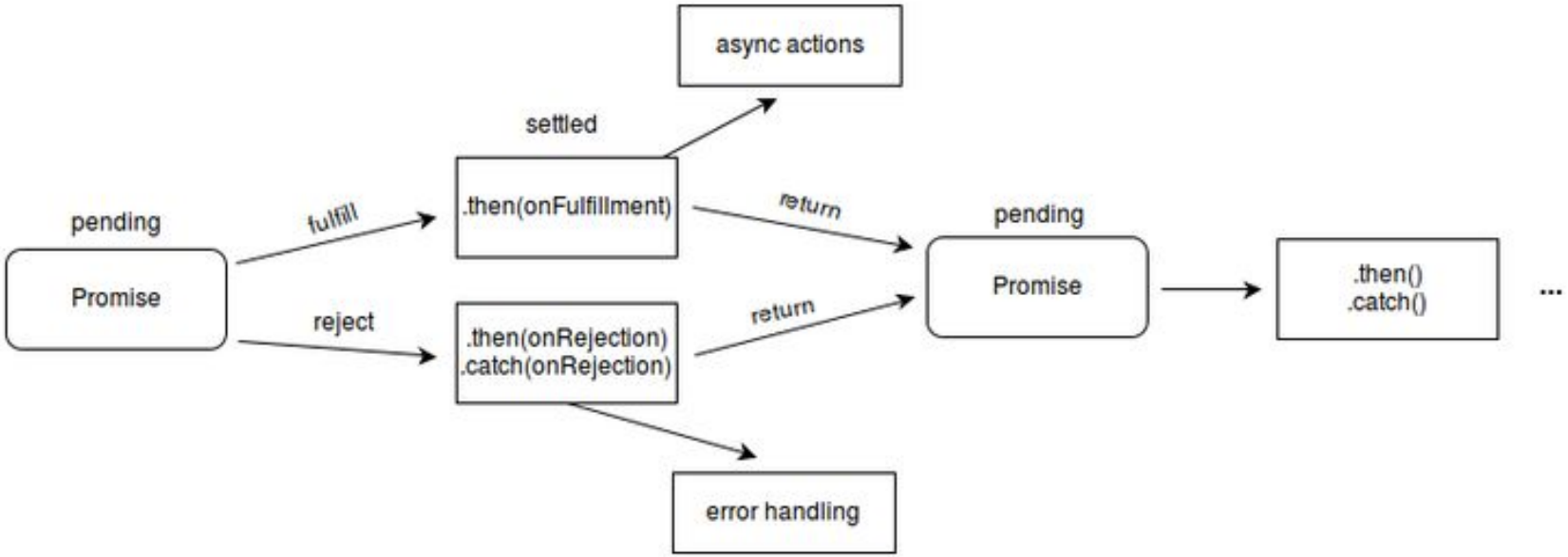

Figure: Promise lifecycle

Chaining I

Promises have `then` method, that accepts one or two callbacks.

1. first is treated as the handler to call if promise was fulfilled successfully,
2. second (if present) is treated as the handler to call if promise is rejected explicitly or any error is caught during execution

Shorthand for `then(null, errorHandler)` is `catch(handleRejection)`. Both of `then` and `catch` create a new promise.

Note on *thenebles*: *Theneble* is a promise-like object, that have a `then` method implemented. Thenebles, if not a genuine promise, are not to be trusted, whether in interface promises provide, error handling etc.

Chaining II

```
1  var fakePromise = {
2      then: function(cb) {
3          cb('Gotcha!');
4      }
5  }
```

Useful Promise API I

- `Promise.resolve(value)`. Returns a Promise object that is resolved with the given value.
- `Promise.reject(reason)`. Returns a Promise object that is rejected with the given reason.
- `Promise.all(iterable)`. Returns a promise that either resolves when all passed promises have resolved or rejects as soon as one of the passed promises rejects.
- `Promise.race(iterable)`. Returns a promise that resolves o rejects as soon as any of passed promises resolves or rejects.

Useful Promise API II

```
1  var promise1 = Promise.resolve('value');
2  var promise2 = Promise.reject(new Error('reject
   ↪  error'));
3  Promise.all([promise1, promise2])
4  .then(promised => { //array of resolved values in
   ↪  same order as promises passed
5      console.log(promised[0]);
6  }).catch(err => { //at least one promise was rejected
   ↪  and error immediatly catched
7      console.error(err);
8  });
```

Map I

The Map object is a simple key/value map. Any value (both objects and primitive values) may be used as either a key or a value.
API:

- `size` - number of values
- `clear()` - removes all key/value pairs
- `entries()` - Iterator obj for all key/values
- `keys()/values()`
- `set(key, value)`
- `get(key)`
- `has(key)`

Map II

- `delete(key)`
- `forEach(callbackFn)`

Weak Map

- Primitive datatypes as keys are not allowed.
- Does not prevent entries to be garbage collected if no other reference on key object is kept

Set, Weak Set

Stores unique values of any type.

API similar to Map.

WeakSet: similar properties as WeakMap.

Generators I

aka `function*`.

`function*` declaration defined a generator function, returning a Generator object.

```
1  function* idMaker(){
2    var index = 0;
3    while(index < 3)
4      yield index++;
5  }
6
7  var gen = idMaker();
8
```

Generators II

```
9  console.log(gen.next().value); // 0
10 console.log(gen.next().value); // 1
11 console.log(gen.next().value); // 2
12 console.log(gen.next().value); // undefined
```

Object.assign I

Copies values from all sources to target object and returns target object.

```
Object.assign(target, ...sources)
```

Typical usage: cloning objects without modifying source, passing defaults.

```
var defaults = {
  param1: true,
  otherParams: {
    otherParam1: 13,
    otherParam2: 'Invincible'
  }
}
```

Object.assign II

```
8
9   var options = {
10    otherParams: {
11      otherParam1: 42
12    }
13  };
14
15  var params = Object.assign({}, defaults, options);
```

Results in:

Object.assign III

```
1  {
2      otherParams: {
3          otherParam1: 42
4      },
5      param1: true
6  }
```

Spread operator

Spread operator (...) allows an expression to be expanded in multiple places:

- function calls

```
myFunction(...iterableObj); //calls function as
↪ myFunction(iterableObj[0], iterableObj[1]...)
```

- array literals

```
[...iterableObj, 4, 5, 6] //resulting in an array
↪ of [4,5,6] prefixed with all elements in
↪ iterableObj
```

- destructuring assignment

```
[a, b, ...iterableObj] = [1, 2, 3, 4, 5]; //a:=1,
↪ b:=2, iterableObj[0]:=3, iterableObj[1]:=4
↪ ...
```

Rest argument

Rest argument in functions

```
1 function f (x, y, ...a) {
2     return (x + y) * a.length
3 }
4 // f(1, 2, 3, 4, 5) === 9
```

ES7

ES2016 / ES7

- next evolution of ECMA standard
- still in progress

Object.values/Object.entries

```
Object.values(obj)
```

Returns object's values. (for consistency with `Object.keys`)

```
Object.entries(obj)
```

Returns array of tuples (arrays) [key, value] of an object.

Array.prototype.includes

Includes function added to array prototype - returs true/false whether param has been found in an array.

- that we won't need to check for it like `array.indexOf(value) > -1`

```
1 Array.includes(val)
```

Exponential operator **

** operator.

```
1  x ** y === Math.pow(x, y)
```

Async functions

With async functions, you can wait on a promise, for it's resolved value in a non blocking way with await.

```
1 getJSON('story.json').then(function(story) {
2 addHtmlToPage(story.heading);
3 ...
4
5 let story = await getJSON('story.json');
6 addHtmlToPage(story.heading);
7 ...
```

Rest/spread properties

Rest/spread properties with objects.

```
let { x, y, ...z } = { x: 1, y: 2, a: 3, b: 4 };
x; // 1
y; // 2
z; // { a: 3, b: 4 }
```

Decorators I

Allows modifying classes and properties at design time.
Descriptor function definition:

```
function readonly(target, name, descriptor){
  descriptor.writable = false;
  return descriptor;
}
```

and usage:

```
class Person {
  @readonly
  name() { return `${this.first} ${this.last}` }
}
```

Object observable

```
1 var obj = {};
2 Object.observe( obj, function(changes)
  ↪ {console.log(changes);} );
3 obj.name = "hemanth";
4 //[ { type: 'new', object: { name: 'hemanth' }, name:
  ↪ 'name' } ]
```

Typed objects

Portable, memory safe, efficient and structured access to allocated data. Automatic coercion.

```
1  var Point = new StructType({
2    x: int32,
3    y: int32
4  });
5  var point = new Point({
6    x: 42,
7    y: 420
8  });
```

SIMD I

SIMD (Single Instruction Multiple Data) operations. Allows you to perform an operation of several values at once.

```
1 var a = SIMD.Float32x4(1.0, 2.0, 3.0, 4.0);
2 var b = SIMD.Float32x4(5.0, 10.0, 15.0, 20.0);
3 var c = SIMD.Float32x4.add(a,b);
```

SIMD II

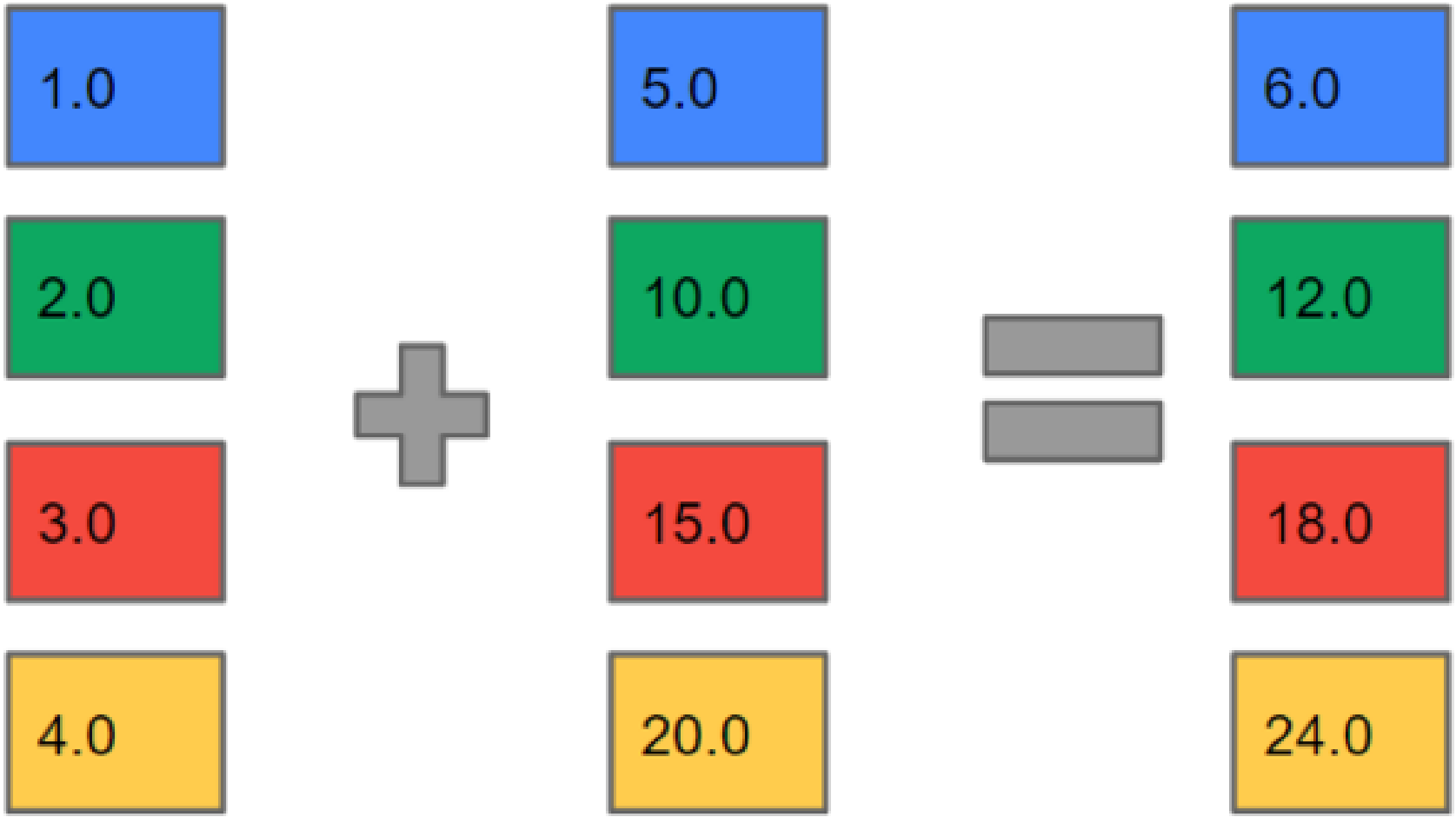

Figure: SIMD add

SIMD I

```
1  function average(list) {
2    var n = list.length;
3    var sum = SIMD.Float32x4.splat(0.0);
4    for (var i = 0; i < n; i += 4) {
5      sum = SIMD.Float32x4.add(sum,
6
   ↪   SIMD.Float32x4.load(list, i));
7    }
8    var total = SIMD.Float32x4.extractLane(sum, 0) +
9                SIMD.Float32x4.extractLane(sum, 1) +
10               SIMD.Float32x4.extractLane(sum, 2) +
11               SIMD.Float32x4.extractLane(sum, 3);
```

SIMD II

```
12      return total / n;
13  }
```

SIMD III

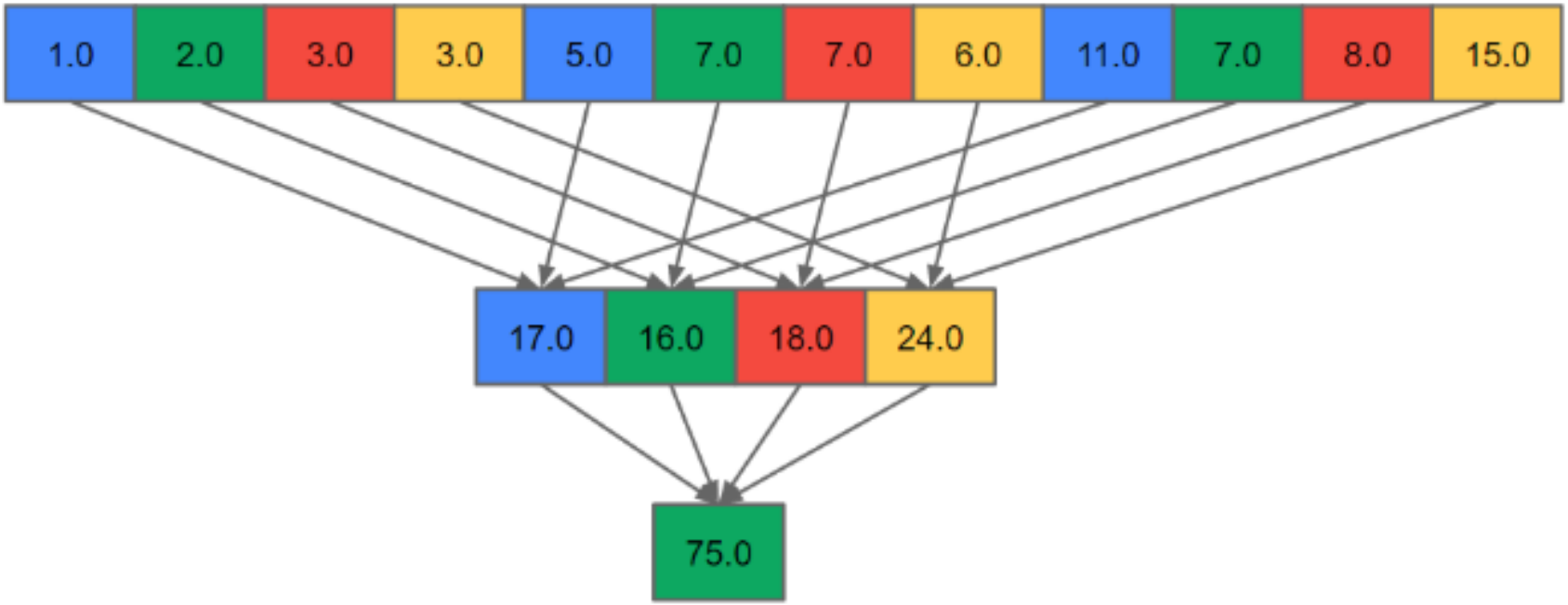

Figure: SIMD average