zeolearn™

Exploring ES6

Outline

- What is ES6
- Why ES6
- Scoping
- Template strings
- Rest and Spread operator
- Arrow functions
- Destructuring
- Loops
- Classes
- Iterators
- Promises
- Generators
- Modules

What is ES6

zeolearn™

- ES6 is a major update to JavaScript that includes dozens of new features
- It took a long time to finish it, but ECMAScript 6, the next version of JavaScript, is finally a reality:
- It became a standard on 17 June 2015.
- Most of its features are already widely available (as documented in kangax' ES6 compatibility table).
- Transpilers (such as Babel) let you compile ES6 to ES5.

Why ES6

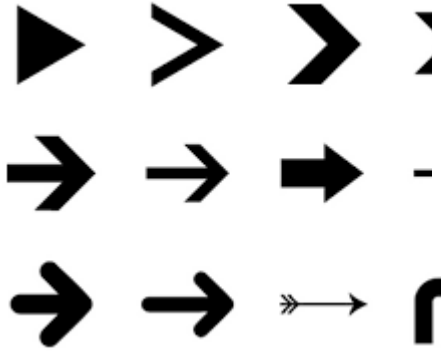
zeolearn™

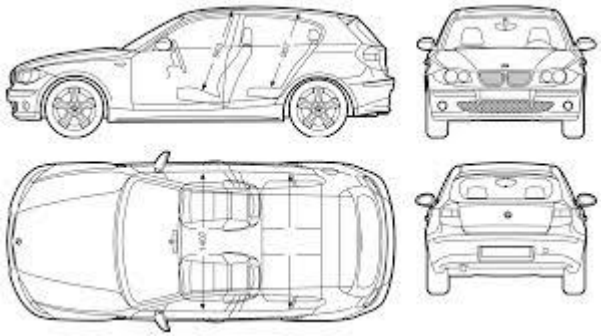
You will know it in sometime

Guess and win

zeolearn™

LET





Scoping – Block Scope

zeolearn™

- var vs let & const

```
var x = 3;

function func(randomize) {
  if (randomize) {
    var x = Math.random();
    return x;
   }
  return x;
}

func(false);
```

```
let x = 3;

function func(randomize) {
  if (randomize) {
    let x = Math.random();
    return x;
   }
  return x;
}

func(false);
```

var, let & const

- In ES5, you declare variables via var. Such variables are function-scoped, their scopes are the innermost enclosing functions
- In ES6, you can additionally declare variables via **let** and **const**. Such variables are **block-scoped**, ***their scopes are the innermost enclosing blocks***.
- **const** works like let, but creates variables ***whose values can't be changed***.


5
6
7
1
2
3
4

Template Strings

zeolearn™

With ES6, JavaScript finally gets literals for string interpolation and multi-line strings

```
function printCoord(x, y) {
  console.log('('+x+', '+y+')');
}
```

```
function printCoord(x, y) {
   console.log(`(${x}, ${y})`);
}
```

```
var HTML5_SKELETON =
'<!doctype html>\n' +
'<html>\n' +
'<head>\n' +
' <meta charset="UTF-8">\n' +
' <title></title>\n' +
'</head>\n' +
'<body>\n' +
'</body>\n' +
'</html>\n';
```

```
const HTML5_SKELETON = `
<!doctype html>
<html>
    <head>
        <meta charset="UTF-8">
        <title></title>
    </head>
    <body>
    </body>
</html>`;
```

Rest operator

- In ES5, if you want a function (or method) to accept an arbitrary number of arguments, you must use the special variable arguments
- In ES6, you can declare a rest parameter (args in the example below) via the ... operator

```
function logAllArguments() {
  for (var i=0; i < arguments.length; i++) {
     console.log(arguments[i]);
   }
}
```

```
function logAllArguments(...args) {
   for (const arg of args) {
     console.log(arg);
   }
}
```


nutella
THECYBERDOLL.TUMBLR

Spread operator

- The spread syntax allows an expression to be expanded in places where multiple arguments (for function calls) or multiple elements (for array literals) or multiple variables (for destructuring assignment) are expected.

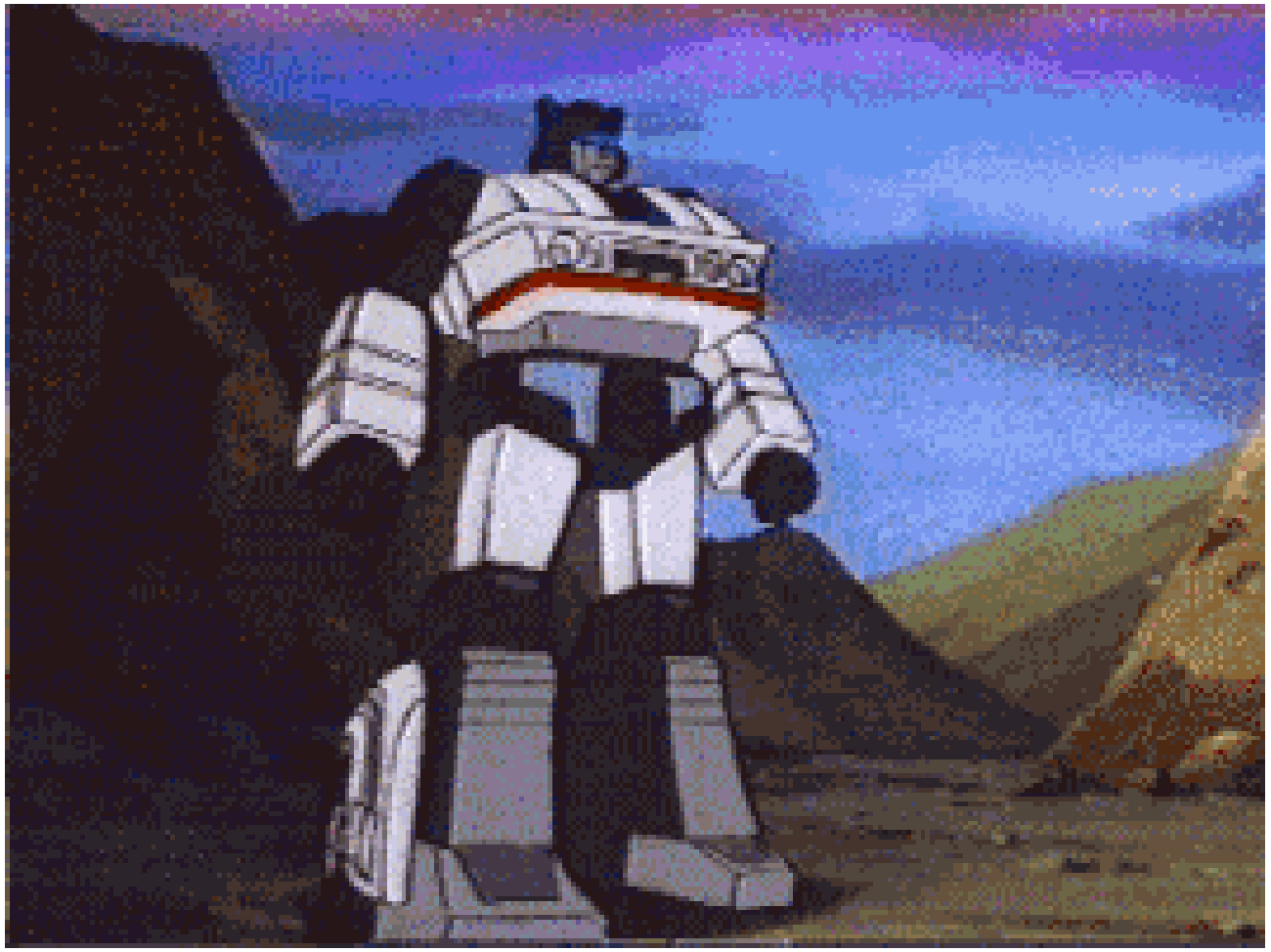
```
const x = ['a', 'b'];
const y = ['c'];
const z = ['d', 'e'];

const arr = [...x, ...y, ...z]; // ['a', 'b',
'c', 'd', 'e']
```

Functions - default params

- In ES6 functions can accept default params
- Default params MUST be right most params

```
function foo(x, y) {
  x = x || 0;
  y = y || 0;
}
```

```
function foo(x=0, y=0) {
}
```



Destructuring

- *Destructuring* is a convenient way of extracting multiple values from data stored in (possibly nested) objects and Arrays
- It can be used in locations that receive data (such as the left-hand side of an assignment).

```
const obj = { first: 'Jane', last: 'Doe' };
const {first: f, last: l} = obj;
// f = 'Jane'; l = 'Doe'
```

```
// {prop} is short for {prop: prop}
const {first, last} = obj;
// first = 'Jane'; last = 'Doe'
```

```
const iterable = ['a', 'b'];
const [x, y] = iterable;
// x = 'a'; y = 'b'
```

```
const arr = ['a', 'b'];
for (const [index, element] of arr.entries()) {
  console.log(index, element);
}
// Output:  0 a   1 b
```

Function named params

- Using Object Destructuring you can simulate named parameters

```
function selectEntries({ start=0,
                         end=-1,
                         step=1 }) {
}

selectEntries({ start: 0, end: -1 });

                         end=-1,
                         step=1 }) {
}

selectEntries({ start: 0, end: -1 });
```


SWIFT
SWIFT

Object Literals - IMPROVEMENTS

- Short-hand props and methods
- Computed keys

```
var obj = {
  foo: function () {
},
bar: function () {
  this.foo();
},
}
```

```
const obj = {
  foo() {
},
bar() {
  this.foo();
},
}
```

From function expressions to arrow functions

zeolearn™

- In current ES5 code, you have to be careful with this whenever you are using function expressions.

```
function UiComponent() {
   var _this = this; // (A)
   var button = document.getElementById('myButton’);
   button.addEventListener('click', function () {
      console.log('CLICK’);
     _this.handleClick(); // (B)
   });
}
```

```
function UiComponent() {
   var button = document.getElementById('myButton’);
   button.addEventListener('click', () => {
      console.log('CLICK’);
      this.handleClick(); // (A)
   });
}
```

Arrow Functions

zeolearn™

- Arrow functions are especially handy for short callbacks that only return results of expressions.

```
var arr = [1, 2, 3];
var squares = arr.map(function (x) {
        return x * x
});
```

```
const arr = [1, 2, 3];
const squares = arr.map(x => x * x);
```

for -> forEach() -> for-of

zeolearn™

- A for loop has the advantage that you can break from it, forEach() has the advantage of conciseness.
- In ES6, the for-of loop combines both advantages:

```
var arr = ['a', 'b', 'c'];
for (var i=0; i<arr.length; i++) {
   var elem = arr[i];
   console.log(elem);
}

arr.forEach(function (elem) {
   console.log(elem);
});
```

```
const arr = ['a', 'b', 'c'];
for (const elem of arr) {
console.log(elem);
}

for (const [index, elem] of arr.entries()) {
console.log(index+'. '+elem);
}
```

zeolearn™


FAMILY ROOM
SOLID BEARING
STAIR
BEDROOM
CLOSE-UP EXIST. WINDOW W/ FACE BRICK TO MATCH EXIST.
CLOSE-UP EXIST. WINDOW W/ CONC. BL.
KITCHEN
TOP OF ROOF
EAVES
SECOND FLOOR
FIRST FLOOR
FINISHED GRADE
BASEMENT
STORAGE/ UTILITIES
EXISTING PORCH
GARAGE
25'-0"
1 1/2HR FIRE RATED CEILING ABOVE (C1 TYPE)

From constructors to classes

zeolearn™

- In ES6, classes provide slightly more convenient syntax for constructor functions

```
function Person(name) {
    this.name = name;
}
Person.prototype.describe = function () {
return 'Person called '+this.name;
};
```

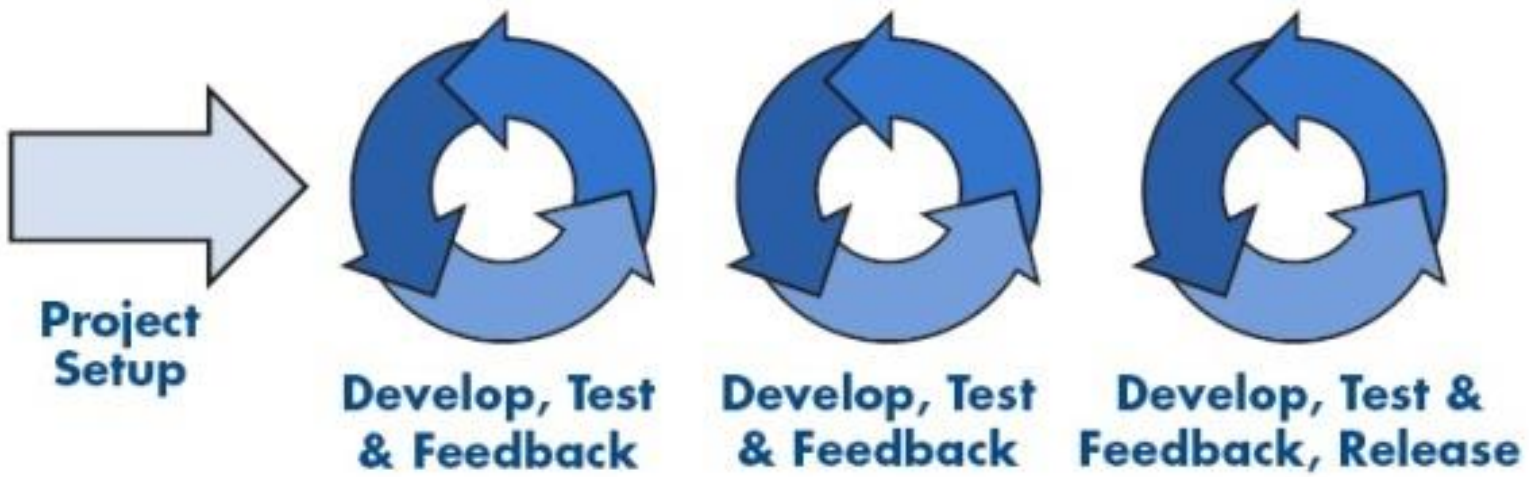
```
class Person {
  constructor(name) {
    this.name = name;
  }
  describe() {
    return 'Person called '+this.name;
  }
}
```

Inheritance

zeolearn™

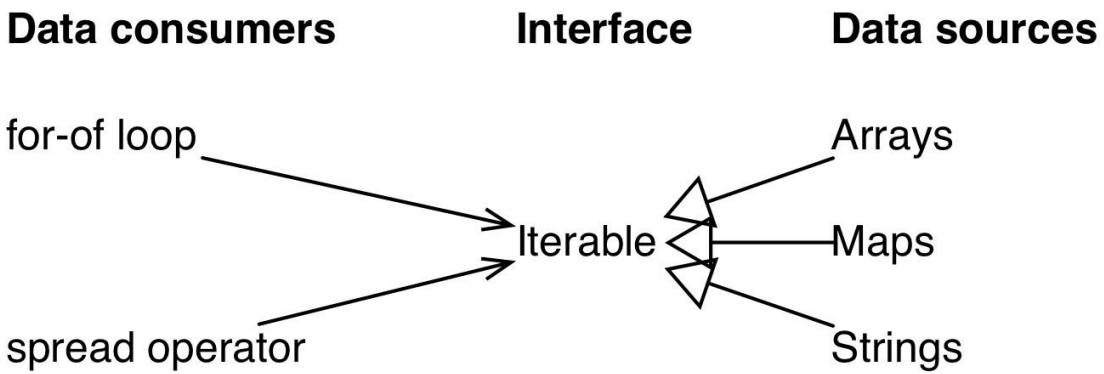
```
function Employee(name, title) {
   Person.call(this, name); // super(name)
   this.title = title;
}

Employee.prototype = Object.create(Person.prototype);
Employee.prototype.constructor = Employee;
Employee.prototype.describe = function () {
  return Person.prototype.describe.call(this) // super.describe()
  + ' (' + this.title + ')';
};
```

```
class Employee extends Person {
 constructor(name, title) {
  super(name);
  this.title = title;
 }
 describe() {
   return super.describe() + ' (' + this.title + ')’;
 }
}
```


Project Setup
Develop, Test & Feedback
Develop, Test & Feedback
Develop, Test & Feedback, Release

Iterators

- **The iterable protocol**
  - allows JavaScript objects to define or customize their iteration behavior, such as what values are looped over in a for..of construct
- **The iterator protocol**
  - defines a standard way to produce a sequence of values (either finite or infinite).



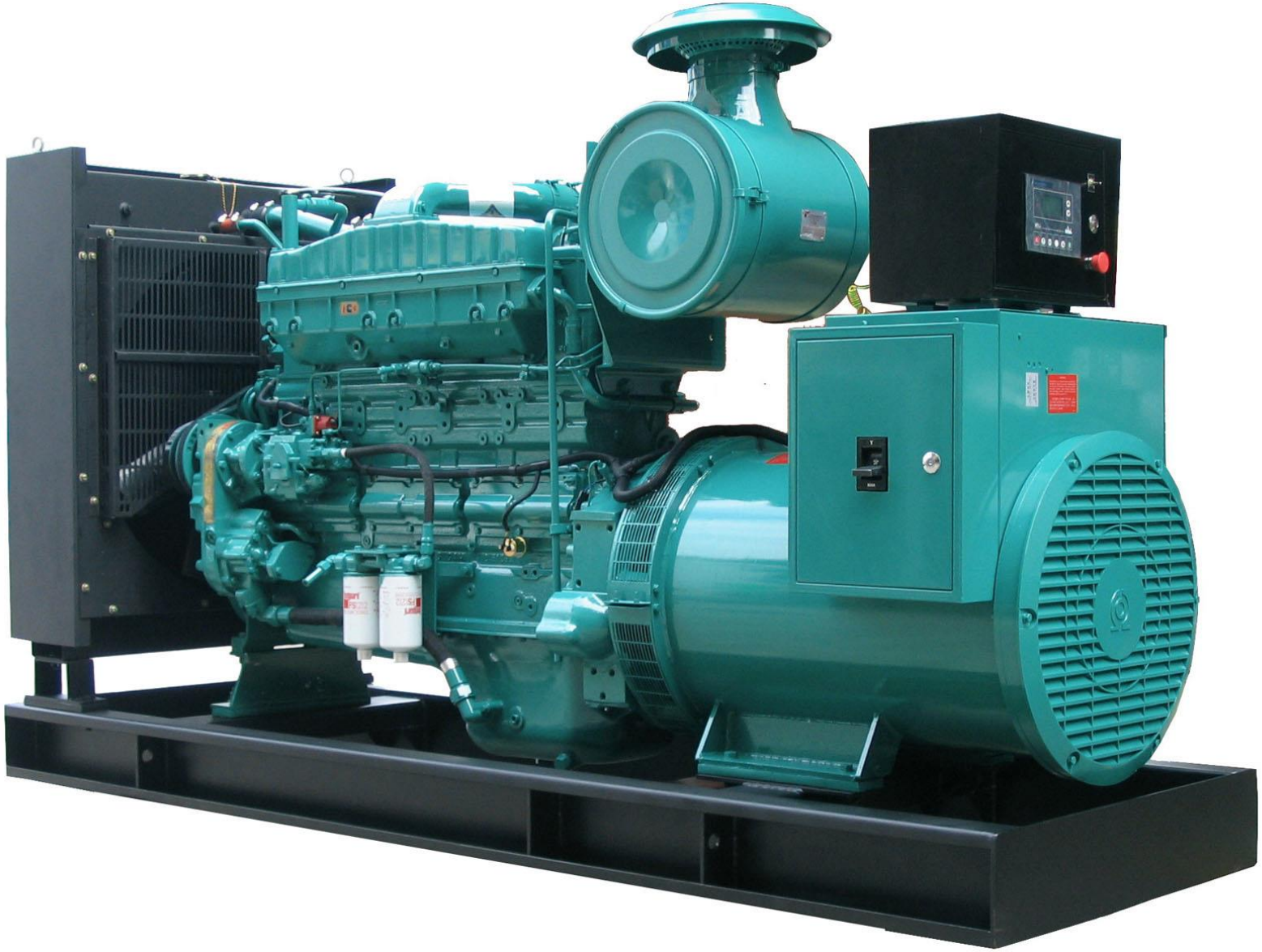

```
const arr = ['a', 'b', 'c'];
const iter = arr[Symbol.iterator]();
```

```
iter.next()
{ value: 'a', done: false }
iter.next()
{ value: 'b', done: false }
iter.next()
{ value: 'c', done: false }
iter.next()
{ value: undefined, done: true }
```

Promises

- Promises are an alternative to callbacks for delivering the results of an asynchronous computation

```
function asyncFunc() {
  return new Promise(
  function (resolve, reject) {
    if(success)
      resolve(result);
    else
      reject(error);
   });
}

asyncFunc()
.then(result => { })
.catch(error => { });
```



Generators

- *Generators* are functions that can be paused and resumed (think cooperative multitasking or coroutines), which enables a variety of applications.
- The Generator object is returned by a generator function and it conforms to both the iterable protocol and the iterator protocol.

```
function* genFunc() {
// (A)
  console.log('First’);
  yield;
  console.log('Second');
}

const genObj = genFunc();

genObj.next();
// Output: First
genObj.next();
// output: Second
```

Why generators – cleaner async

zeolearn™

```
function fetchJson(url) {
   return fetch(url)
          .then(request => request.text())
          .then(text => {
             return JSON.parse(text);
          })
         .catch(error => {
          console.log(`ERROR: ${error.stack}`);
         });
}
```

```
const fetchJson = co.wrap(function* (url) {
try {
  let request = yield fetch(url);
  let text = yield request.text();
  return JSON.parse(text);
}
catch (error) {
  console.log(`ERROR: ${error.stack}`);
}
});
```

Modules

- JS till ES6 do not have native module support
- In ES6 there are third party implementations like SystemJS and AMD
- ES6 has built-in support for modules.
- Unfortunately,no JavaScript engine supports them natively, yet. But tools such as babel, webpack or jspm let you use ES6 syntax to create modules, making the code you write future-proof.

```
//------ lib.js ------
export const sqrt = Math.sqrt;
export function square(x) {
return x * x;
}
export function diag(x, y) {
return sqrt(square(x) + square(y));
}

//------ main1.js ------
import { square, diag } from 'lib';
console.log(square(11)); // 121
console.log(diag(4, 3)); // 5
```

Check your knowledge

- Which keyword is used to define block scoped variables??
- A variable address is a constant and points to address object. Is it possible to amend the address object properties after making it a constant?
- Return statement is default in arrow functions??
- Which protocol implemented data structures can be used with for of loop?
- Name the method of iterator object
- Generator functions can be resumed by calling_____method
- yield is optional in generator methods
- rucsnotcrto – descramble
- Name concepts which uses ... operator?
- Name syntax for array destructring
- Every construct is exportable by default in a module
- “this” is autobind in arrow functions
- Promises are replacement for generators?