Functional JavaScript #part 2

iterative functions, array manipulations, function composition

Functional JavaScript

- JavaScript is a multi-paradigm language
- It can be used to program functionally

FP idioms:

- iterative functions, which can replace loops,
- list processing
- function manipulations
- immutability
- pure functions
- branching
- … and many other things,

Can help us to keep code:

- smaller
- cleaner/readable/semantic
- testable
- reusable
- maintainable
- ….
- more fun

Pure functions

Pure functions

- **Referential transparency:** The function always gives the same return value for the same arguments. This means that the function **cannot depend on any mutable state**.
- **Side-effect free:** The function **cannot cause any side effects**. Side effects may include I/O (e.g., writing to the console or a log file), modifying a mutable object, reassigning a variable from outer scope, etc.

Pure functions

- Benefits of pure functions:
- **easier to reason about** and debug because they don't depend on mutable state
- **return value can be cached or "memoized"** to avoid recomputing it in the future.
- easier to test because there are no dependencies
- …

pure functions

function cannot **depend on any mutable state**

Bad - dependency on:

- captured *let, var*
- captured const but with with *mutable value*
- *dependency on mutable or impure function*
- *methods* are impure by definition

Ok – dependency on:

- no captured variables
- captured primitive const
- captured frozen object const
- on constant and pure function
- on immutable captured argument

pure-functions.js

```
1  //------------------------
2  function below(n, limit) {
3    return n < limit;
4  }
5  //------------------------
6  function below(n) {
7    var LIMIT = 100;
8    return n < LIMIT;
9  }
10 //------------------------
11 const LIMIT = 100;
12 function below(n) {
13   return n < LIMIT;
14 }
15 //------------------------
16 const limit = () => 100;
17 function below(n, limit) {
18   return n < limit();
19 }
20 //------------------------
21 function below(limit) {
22   return function(n) {
23     return n < limit;
24   }
25 }
26 below(100)(10);
```

impure-functions.js

```
1  //------------------------
2  function below(n, options) {
3    return n < options.limit;
4  }
5  //--------------------------------
6  var LIMIT = 100;
7  function below(n) {
8    return n < LIMIT;
9  }
10 //--------------------------------
11 const constants = { LIMIT: 100 }
12 function below(n) {
13   return n < constants.LIMIT;
14 }
15 //--------------------------------
16 let limit = () => 100;
17 function below(n, limit) {
18   return n < limit();
19 }
20 //--------------------------------
21
22
23
24
25
26
```

/samples/01-pure-functions/*.js

pure functions

A. Side-effect free

- function **must return value,** if not it is *noop* or exists to ***perform side effects***

B. No IO

- console.log,…
- http, net,…
- fs, process,…
- DOM, XHR

C. Use internally pure functions

- pure function calling impure becomes impure

```
1  // A) void function
2  // modifies by ref param instead of return
3  // performance reasons ?
4  function bcrypt_hash(sha2pass, sha2salt, out) {
5    //...
6    //...
7    out[4 * i + 3] = cdata[i] >>> 24;
8    out[4 * i + 2] = cdata[i] >>> 16;
9    out[4 * i + 1] = cdata[i] >>> 8;
10   out[4 * i + 0] = cdata[i];
11 };
12
13 // B)
14 stream.on("data", (chunk) => {
15   console.log(chunk);
16 });
17
18 // C)
19 const constants = { LIMIT: 100 }
20 function below(n) {
21   return n < constants.LIMIT;
22 }
23
24 function filterByLimit(nums) {
25   return nums.filter(below);
26 }
```

Pure vs impure functions

- Any meaningful program will contain also impure functions (Ajax call, I/O calls, check the current date, or get a random number,...). rule of thumb is to follow the 80/20 rule: 80% of your functions should be pure, and the remaining 20%, of necessity, will be impure.
- The general idea is that you write as much of your application as possible in an FP style, and then handle the UI and all forms of input/output (I/O) (such as Database I/O, Web Service I/O, File I/O, etc.) in the best way possible for your current programming language and tools

https://opensource.com/article/17/6/functional-javascript https://alvinalexander.com/scala/fp-book/pure-functions-and-io-input-output

Higher order functions

Higher order functions

Higher order function is a function that does one of or both:

- takes a **function as an argument**
- **returns function**

Higher order functions – **functions returning functions**

- transform one function somehow
  - **add functionality (vs. aspects)**
    - memoize
    - deprecate
    - …
  - change signature,
  - change return value
  - bind parameters,
  - change context,
  - curry
  - …
- combine functions
- …

Add Functionality

- Začnime tými najjednoduchšími, kde novovzniknutá funkcia má rovnakú signatúru (vstupy a výstupy) ale robí niečo navyše alebo inak
- Anatómia: funkcia vracajúca funkciu, ktorá po zavolaní zavolá originál funkciu (6-11)
- Vytvorenie novej funkcie zo starej (15-16)
- Pointa: pri vytváraní funkcie okrem pôvodnej funkcie posielame ďalšie argumenty (20) , a tie potom použijeme na implementáciu ten dodatočnej funkcionality 21,25,27,30
- Finta: aj vrátená funkcia ich môže využiť v svojom tele, lebo closure

```
1  // orig funkcia a volania
2  function sum(a, b, c) { return a + b + c };
3  let x1 = sum(1, 2, 3);
4
5  // transform funkcia
6  function transf(fn) {
7    const fn2 = function(...args) {
8      let origRet = fn.apply(this, args);
9      return origRet;
10   }
11   return fn2;
12 }
13
14 // nova funkcia a volanie
15 const sum2 = transf(sum);
16 let x2 = sum2(1, 2, 3);
17
18
19 // transform funkcia
20 function transf(fn, ...transformArgs) {
21   // ...
22   const fn2 = function(...args) {
23     // here we still see ...transformArgs
24     // because of closure
25     // ...
26     let origRet = fn.apply(this, args);
27     // ...
28     return origRet;
29   }
30   // ...
31   return fn2;
32 }
```

Deprecate

https://github.com/nodejs/node/tree/v11.x

Príklad na wrapnutie pôvodnej funkcie a **pridanie funkcionality**:

1. Nik nechce otvárať a chytať kód originál funkcii a niečo do nich dopisovať (lenivosť, bezpečnosť, zdravý rozum)
2. Takže len v exporte transformnu original funkciu „na inú“
3. A tú exportnú
4. Predtým tam mali napísane debug:debug
5. Debug bola funkcia a bude funkcia

```
// file: lib/util.js
module.exports = exports = {
  //....
  isFunction,
  isPrimitive,
  log,
  promisify,
  TextDecoder,
  TextEncoder,
  types,

  // Deprecated Old Stuff
  debug: deprecate(debug,
                   'util.debug is deprecated.
                   'DEP0028'),
  error: deprecate(error,
                   'util.error is deprecated.
                   'DEP0029'),
  //....
};
```

Deprecate

https://github.com/nodejs/node/tree/v11.x

1. Anatómia riešenia je klasická,
2. funkcia deprecate, ktorá berie inú funkciu fn ako parameter, vracia novú funkciu (deprecated), ktorú keď zavoláte z nejakými argumentami, zavolá originálnu funkciu fn() s týmito parametrami

```
// Mark that a method should not be used.
// Returns a modified function which warns once by default.
// If --no-deprecation is set, then it is a no-op.
function deprecate(fn, msg, code) {
  if (process.noDeprecation === true) {
    return fn;
  }

  if (code !== undefined && typeof code !== 'string')
    throw new ERR_INVALID_ARG_TYPE('code', 'string', code);

  let warned = false;
  function deprecated(...args) {
    if (!warned) {
      warned = true;
      if (code !== undefined) {
        if (!codesWarned[code]) {
          process.emitWarning(msg, 'DeprecationWarning', code, deprecated);
          codesWarned[code] = true;
        }
      } else {
        process.emitWarning(msg, 'DeprecationWarning', deprecated);
      }
    }
    if (new.target) {
      return Reflect.construct(fn, args, new.target);
    }
    return fn.apply(this, args);
  }

  // The wrapper will keep the same prototype as fn to maintain prototype chain
  Object.setPrototypeOf(deprecated, fn);
  if (fn.prototype) {
    // Setting this (rather than using Object.setPrototype, as above) ensures
    // that calling the unwrapped constructor gives an instanceof the wrapped
    // constructor.
    deprecated.prototype = fn.prototype;
  }

  return deprecated;
}
```

Deprecate

https://github.com/nodejs/node/tree/v11.x

1. Pridaná hodnota novej funkcie oproti povodnej, spočíva v tom, že spôsobí pri zavolaní warning
2. a čaro spočíva v tom, že využije parametre s ktorými ste ju vytvárali, lebo ich má k dispozícii cez closure

```
// Mark that a method should not be used.
// Returns a modified function which warns once by default.
// If --no-deprecation is set, then it is a no-op.
function deprecate(fn, msg, code) {
  if (process.noDeprecation === true) {
    return fn;
  }

  if (code !== undefined && typeof code !== 'string')
    throw new ERR_INVALID_ARG_TYPE('code', 'string', code);

  let warned = false;
  function deprecated(...args) {
    if (!warned) {
      warned = true;
      if (code !== undefined) {
        if (!codesWarned[code]) {
          process.emitWarning(msg, 'DeprecationWarning', code, deprecated);
          codesWarned[code] = true;
        }
      } else {
        process.emitWarning(msg, 'DeprecationWarning', deprecated);
      }
    }
    if (new.target) {
      return Reflect.construct(fn, args, new.target);
    }
    return fn.apply(this, args);
  }

  // The wrapper will keep the same prototype as fn to maintain prototype chain
  Object.setPrototypeOf(deprecated, fn);
  if (fn.prototype) {
    // Setting this (rather than using Object.setPrototype, as above) ensures
    // that calling the unwrapped constructor gives an instanceof the wrapped
    // constructor.
    deprecated.prototype = fn.prototype;
  }

  return deprecated;
}
```

Deprecate

https://github.com/nodejs/node/tree/v11.x

Zvyšok kódu sa zaoberá špecialitkami

1. aby sa warningy generovali len ak sú zapnuté a len raz
2. A univerzálnymi scenármi podľa typu volania (funkcia, alebo konštruktor)
3. Zachovaním prototypu pôvodnej funkcie

```
// Mark that a method should not be used.
// Returns a modified function which warns once by default.
// If --no-deprecation is set, then it is a no-op.
function deprecate(fn, msg, code) {
  if (process.noDeprecation === true) {
    return fn;
  }

  if (code !== undefined && typeof code !== 'string')
    throw new ERR_INVALID_ARG_TYPE('code', 'string', code);

  let warned = false;
  function deprecated(...args) {
    if (!warned) {
      warned = true;
      if (code !== undefined) {
        if (!codesWarned[code]) {
          process.emitWarning(msg, 'DeprecationWarning', code, deprecated);
          codesWarned[code] = true;
        }
      } else {
        process.emitWarning(msg, 'DeprecationWarning', deprecated);
      }
    }
    if (new.target) {
      return Reflect.construct(fn, args, new.target);
    }
    return fn.apply(this, args);
  }

  // The wrapper will keep the same prototype as fn to maintain prototype chain
  Object.setPrototypeOf(deprecated, fn);
  if (fn.prototype) {
    // Setting this (rather than using Object.setPrototype, as above) ensures
    // that calling the unwrapped constructor gives an instanceof the wrapped
    // constructor.
    deprecated.prototype = fn.prototype;
  }

  return deprecated;
}
```

Memoize

- Máme dlhotrvajúcu sync/async operáciu
- Máme pure function, teda vracia to isté pri tých istých vstupoch
- Odložíme si výsledky pre dané vstupy a miesto vykonania funkcie vrátime odpamätané záznamy
- Otázka znie ako hashovat parametre (bude neskôr)
- Prinicíp: nová vrátená funkcia používa ako cache premennú z closure, vytvorenú pri jej konštrovaní

```
5
6  // can memoize only 1 param functions
7  function memoize(fn) {
8    const memo = new Map();
9
10   const memoized = function(arg) {
11     const key = arg;
12     if (memo.has(key)) { return memo.get(key) }
13
14     let origRet = fn.call(this, arg);
15
16     memo.set(key, origRet);
17     return origRet;
18   }
19   return memoized;
20 }
21
22 // orig funct and call
23 const findStudent = (name) =>
24   students.find((s) => s.name === name);
25
26 let s1 = findStudent("student1000000");
27 let s3 = findStudent("student1000000");
28
29 // new funct and call
30 const findStudent2 = memoize(findStudent);
31
32 let ss1 = findStudent2("student1000000");
33 let ss3 = findStudent2("student1000000");
```

Memoize

https://github.com/caolan/async/blob/master/lib/memoize.js

1. Klasický pattern komplikovaný o to, že riešime async funkcie z callbackom
2. Podstatne jednoduchšie z promismi, alebo async funkciami

```
function memoize(fn, hasher = v => v) {
    var memo = Object.create(null);
    var queues = Object.create(null);
    var _fn = (0, _wrapAsync2.default)(fn);
    var memoized = (0, _initialParams2.default)((args, callback) => {
        var key = hasher(...args);
        if (key in memo) {
            (0, _setImmediate2.default)(() => callback(null, ...memo[key]));
        } else if (key in queues) {
            queues[key].push(callback);
        } else {
            queues[key] = [callback];
            _fn(...args, (err, ...resultArgs) => {
                // #1465 don't memoize if an error occurred
                if (!err) {
                    memo[key] = resultArgs;
                }
                var q = queues[key];
                delete queues[key];
                for (var i = 0, l = q.length; i < l; i++) {
                    q[i](err, ...resultArgs);
                }
            });
        }
    });
    memoized.memo = memo;
    memoized.unmemoized = fn;
    return memoized;
}
```

Memoize - precvičenie

- https://www.codewars.com/kata/529adbf7533b761c560004e5/train/javascript
- TODO: najdite a vyrieste viacej zadani na memoizer na codewars a poslite mi please mailom
- TODO: napíšte si vlastný memoizer na async (promise) funkcie (napr. request(url) aj z parsingom parametrov, nesenzitívnym na poradie, aj z ukladaním na disk atď....)

Measure

1. Klasický scenár obalenia funkcie a pridania nejakej before a after implementácie

```
1  // just demo impl
2
3  function measure(fn) {
4
5    return function(...args) {
6      const hrstart = process.hrtime()
7      try {
8
9        return fn.apply(this, args);
10
11     } finally {
12       const hrend = process.hrtime(hrstart)
13       console.error(
14         'Execution time (hr): %ds %dms',
15         hrend[0], hrend[1] / 1000000
16       );
17     }
18   }
19 }
20 module.exports = measure;
```

Measure

1. Meranie úspešnosti našej memoizácie
2. Kompozícia viacerých transformačných funkcii

```
35  // how faster (naive measure)
36  const measure = require("./03-measure.js");
37  const findStudentM = measure(findStudent);
38  const findStudentMM = measure(memoize(findStudent));
39
40  //non memoised
41  findStudentM("student999999")
42  findStudentM("student999999")
43  findStudentM("student999999")
44
45  //memoized"
46  findStudentMM("student999999")
47  findStudentMM("student999999")
48  findStudentMM("student999999")
49
50  /*
51
52  $ node 02-higher-order-functions/01-returning-functio
53  Execution time (hr): 0s 89.678275ms
54  Execution time (hr): 0s 19.953699ms
55  Execution time (hr): 0s 16.645149ms
56  Execution time (hr): 0s 17.606519ms
57  Execution time (hr): 0s 0.005991ms
58  Execution time (hr): 0s 0.000937ms
```

Change function

- Zložitejšie prípady kedy chceme funkciu transformovať zahŕňajú:
  - Zmeny počtu parametrov
  - Zmeny návratových hodnôt
  - Komplexné zmeny správania
- Typické príklady:
  - z callback async funkcie spraviť funkciu vracajúcu promise
  - Zo synchrónnej funkcie spraviť async funkciu

Promisify

Ako spraviť z async cb API, async promise API

1. Meníme počet parametrov
2. Meníme návratovú hodnotu
3. Meníme správanie

```
1  // mame funkciu z N paramerami posledný je callback
2  const fs = require("fs");
3
4  // parametre 3 (path,opts,cb), returns nothing
5  fs.readdir(".", {}, (err, dirs) =>
6    console.log(err, dirs)
7  );
8
9  // a checme to volat takto
10 const readdirP = promisify(fs.readdir);
11 const dirs = readdirP(".", {})
12
13 dirs
14   .then(dirs => console.log(dirs))
15   .catch(err => console.log(err))
```

Promisify

Klasický scenár funkcia vracajúca funkciu volajúca pôvodnú funkciu. Špecialitky:

- 20 – funkcia vracia promise, resolvnutý po spustením exec funkcie
- Tá zavolá originál fn [14] z paramerami
- a našim ad hoc callbackom,[15,16], ktorý zavolá promise resolve alebo reject.

```
 7  // kostra
 8  function promisify(fn) {
 9
10    return function(...args) { // this ?
11
12      const exec = (resolve, reject) => { // this ?
13
14        fn.call(this, ...args, (err, data) => {
15          if (err) reject(err);
16          else resolve(data);
17        })
18
19      };
20      return new Promise(exec);
21    }
22  }
```

Promisify

Použitie function vs => v tomto prípade, ale aj iných patternoch, nie je estetická záležitosť, ale je podstatné pre resolving this.

1. Pôvodná funkcia môže byt metóda objektu
2. Nová funkcia môže byť použitá ako funkcia alebo ako metóda

DU:

A) fs.readdir je ozaj metóda, alebo len funkcia zavesená na “namespace“ fs

B) Ake bude this na 11 a 13 v oboch prípadoch volania ?

```
7  // kostra
8  function promisify(fn) {
9
10   return function(...args) { // this ?
11
12     const exec = (resolve, reject) => { // this ?
13
14       fn.call(this, ...args, (err, data) => {
15         if (err) reject(err);
16         else resolve(data);
17       })
18
19     };
20     return new Promise(exec);
21   }
22 }
23
24 // volanie ako funkcia
25 const readdirP = promisify(fs.readdir);
26 const dirs = readdirP(".", {})
27 dirs
28   .then(dirs => console.log(dirs))
29   .catch(err => console.log(err))
30
31 // volanie ako metoda
32 fs.readdirP = readdirP
33 const dirs2 = fs.readdirP(".", {})
34 dirs2
35   .then(dirs => console.log(dirs))
36   .catch(err => console.log(err))
```

Promisify

https://github.com/nodejs/node/blob/f607e169612ff60a00f456f3c698a6b41792a8f5/lib/internal/util.js#L268

Full verzia promisify z nodejs util

1. Nájdite si kostru z predošlého zjednodušeného príkladu
2. Zistite čo to robí navyše
3. Pokúste sa zistiť načo
4. Pokúste sa zamyslieť prečo
   - Lebo funkcia nie je len funkcia ale .....
5. Napíšte si test a pokúste sa si to oddebugovať

```
268  function promisify(original) {
269    if (typeof original !== 'function')
270      throw new ERR_INVALID_ARG_TYPE('original', 'Function', original);
271
272    if (original[kCustomPromisifiedSymbol]) {
273      const fn = original[kCustomPromisifiedSymbol];
274      if (typeof fn !== 'function') {
275        throw new ERR_INVALID_ARG_TYPE('util.promisify.custom', 'Function', fn);
276      }
277      return Object.defineProperty(fn, kCustomPromisifiedSymbol, {
278        value: fn, enumerable: false, writable: false, configurable: true
279      });
280    }
281
282    // Names to create an object from in case the callback receives multiple
283    // arguments, e.g. ['bytesRead', 'buffer'] for fs.read.
284    const argumentNames = original[kCustomPromisifyArgsSymbol];
285
286    function fn(...args) {
287      return new Promise((resolve, reject) => {
288        original.call(this, ...args, (err, ...values) => {
289          if (err) {
290            return reject(err);
291          }
292          if (argumentNames !== undefined && values.length > 1) {
293            const obj = {};
294            for (var i = 0; i < argumentNames.length; i++)
295              obj[argumentNames[i]] = values[i];
296            resolve(obj);
297          } else {
298            resolve(values[0]);
299          }
300        });
301      });
302    }
303
304    Object.setPrototypeOf(fn, Object.getPrototypeOf(original));
305
306    Object.defineProperty(fn, kCustomPromisifiedSymbol, {
307      value: fn, enumerable: false, writable: false, configurable: true
308    });
309    return Object.defineProperties(
310      fn,
311      Object.getOwnPropertyDescriptors(original)
312    );
313  }
```

Methods to functions

Príklad demonštruje

- Pridanie parametra do signatúry novej funkcie
- Zavolanie originál funkcie zo zmeneným this

```
// real world example
// change method to function

// orig
[-1, 2, -3].filter(n => n > 0);

// wanted
filter([-1, 2, 3], n => n > 0);

function method2func(fn) {
  if (typeof fn !== "function")
    throw new TypeError("expected function")
  return function(self, ...args) {
    return fn.apply(self, args);
  }
}

const filter = method2func(Array.prototype.filter);
assert.deepStrictEqual(
  // now we can call it with new syntax
  filter([-1, 2, 3], n => n > 0), [2, 3]
);
```

Functions to methods

Príklad demonštruje implementáciu funkcionality ako funkcii aj ako metód z tým, že sú kódnuté len raz:

- Funkcie sú dostuplné ako „statické“ na XString „triede“ a zároveň
- V loope sú všetky pridané na prototyp, ako wrapnuté funkcie z doplneným this do prvého parametra (toto je kód tak 10+ rokov dozadu)

```
103     XString.replaceParams = function(that, params) {
104         // new version by zagi ;-)
105         return that.replace(/\%([0-9]+)/gm, function(a, b) {
106             return params[parseInt(b, 10)];
107         });
108     };
109     XString.endsWith = function(that, search) {
110         var x = that.lastIndexOf(search);
111         return x != -1 && x == (that.length - search.length);
112     };
113     XString.capitalize = function(that) {
114         return String(that).replace(/\b[a-z]/g, function(match) {
115             return match.toUpperCase();
116         });
117     };
118
119     /*
120      The next code is equivalent for
121      XString.prototype.*=function(){return XString.*.apply(null,[String(this),otehr args]);}
122      XString.prototype.trim=function(){return XString.trim.apply(null,[String(this)]);}
123      */
124     for ( var fnName in XString) {
125         /*jshint loopfunc:true*///TODO: fix and test
126         XString.prototype[fnName] = function(method) {
127             return function() {
128                 var args = Array.prototype.slice.call(arguments);
129                 args.unshift(String(this));
130                 return method.apply(null, args);
131             };
132         }(XString[fnName]);
133     }
```

Higher order functions – function as argument

Can exist because:

- functions are first-class elements

Usage:

- callback of async task
  - toto už sme si ukázali v async prednáškach
- parametrized algorithms
  - napríklad array funkcie z FP prednášky
  - ďalšie príklady (builders, …), Array.from, Array.fill
- parametrized iterations
  - repeat, until, whilst
- transforming function
  - predchádzajúca sekcia, ale parameter nie sú hodnoty ale funkcie)

parametrized algorithms – max/best example

```
// finding maximum

Math.max([10, 30, 20]); // (-) can work only on numbers
```

NO PLEASE !!!!!

takto sa v JS nekoduje

```
// ale co ak chceme nieco taketo ?
// max({ age: 10 }, { age: 30 }, { age: 20 }) //=> {age:30}

function maxAge(people) {
  let max = -Infinity;
  let imax;
  for (let i = 0; i < people.length; i++) {

    if (people[i].age > max) {
      max = people[i].age;
      imax = i;
    }
  }
  return imax != null ? people[imax] : undefined;
}

maxAge([{ age: 10 }, { age: 30 }, { age: 20 }]) // => {age:30}
```

```
// a teraz nieco taketo
// max({ speed: 10 }, { speed: 30 }, { speed: 20 })

function maxSpeed(cars) {
  let max = -Infinity;
  let imax;
  for (let i = 0; i < cars.length; i++) {

    if (cars[i].speed > max) {
      max = cars[i].speed;
      imax = i;
    }
  }
  return imax != null ? cars[imax] : undefined;
}

maxSpeed([{ speed: 10 }, { speed: 30 }, { speed: 20 }]) // => {speed:30}
```

parametrized algorithms – max/best example

```
// finding maximum

Math.max([10, 30, 20]); // (-) can work only on numbers
```

```
// ale co ak chceme nieco taketo ?
// max({ age: 10 }, { age: 30 }, { age: 20 }) //=> {age:30}
```

```
function maxAge(people) {
  let max = -Infinity;
  let imax;
  for (let i = 0; i < people.length; i++) {

    if (people[i].age > max) {
      max = people[i].age;
      imax = i;
    }
  }
  return imax != null ? people[imax] : undefined;
}
```

```
maxAge([{ age: 10 }, { age: 30 }, { age: 20 }]) // => {age:30}
```

extract hardcoded parts as params = functions

```
function max(value, compare, items) {
    return items.reduce((best, curr) => {
      var bestV = value(best);
      var currV = value(curr);
      return (bestV === compare(bestV, currV)) ?
        best :
        curr;
    });
}

let people = [{ age: 10 }, { age: 30 }, { age: 20 }];

max((o) => o.age, (v1, v2) => v1 > v2 ? v1 : v2, people)

max((o) => o.age, Math.max, people)

let cars = [{ speed: 10 }, { speed: 30 }, { speed: 20 }];
max((o) => o.speed, Math.max, cars)
```

parametrized algorithms – max/best example

```
function max(value, compare, items) {
    return items.reduce((best, curr) => {
      var bestV = value(best);
      var currV = value(curr);
      return (bestV === compare(bestV, currV)) ?
        best :
}       curr;
    });

let people = [{ age: 10 }, { age: 30 }, { age: 20 }];

max((o) => o.age, (v1, v2) => v1 > v2 ? v1 : v2, people)
max((o) => o.age, Math.max, people)

let cars = [{ speed: 10 }, { speed: 30 }, { speed: 20 }];
max((o) => o.speed, Math.max, cars)
```

2 functions extracted and needed in each call
compare can be defaulted to Math.max, but still
….. can we do better ? best ?

```
function best(better, items) {
      return items.reduce((best, curr) => {
        return better(best, curr) ?
          best :
          curr;
      });
}

best((o1, o2) => o1.age > o1.age, people)

best(({ speed1 }, { speed2 }) => speed1 > speed2, cars)

best((s1, s2) => s1.length < s2.length, ["short", "a", "the"]);
```

```
const best = (better, items) => items.reduce(
  (best, curr) => better(best, curr) ? best : curr
)
```

max/best example recap

Before:

```
function maxAge(people) {
  let max = -Infinity;
  let imax;
  for (let i = 0; i < people.length; i++) {

    if (people[i].age > max) {
      max = people[i].age;
      imax = i;
    }
  }
  return imax != null ? people[imax] : undefined;
}
```

```
function maxSpeed(cars) {
  let max = -Infinity;
  let imax;
  for (let i = 0; i < cars.length; i++) {

    if (cars[i].speed > max) {
      max = cars[i].speed;
      imax = i;
    }
  }
  return imax != null ? cars[imax] : undefined;
}
```

After:

```
const best = (better, items) => items.reduce(
  (best, curr) => better(best, curr) ? best : curr
)
```

```
best((s1, s2) => s1.length < s2.length, ["short", "a", "the"]);
```

```
maxAge([{ age: 10 }, { age: 30 }, { age: 20 }]) // => {age:30}

maxSpeed([{ speed: 10 }, { speed: 30 }, { speed: 20 }]) // => {speed:30}
```

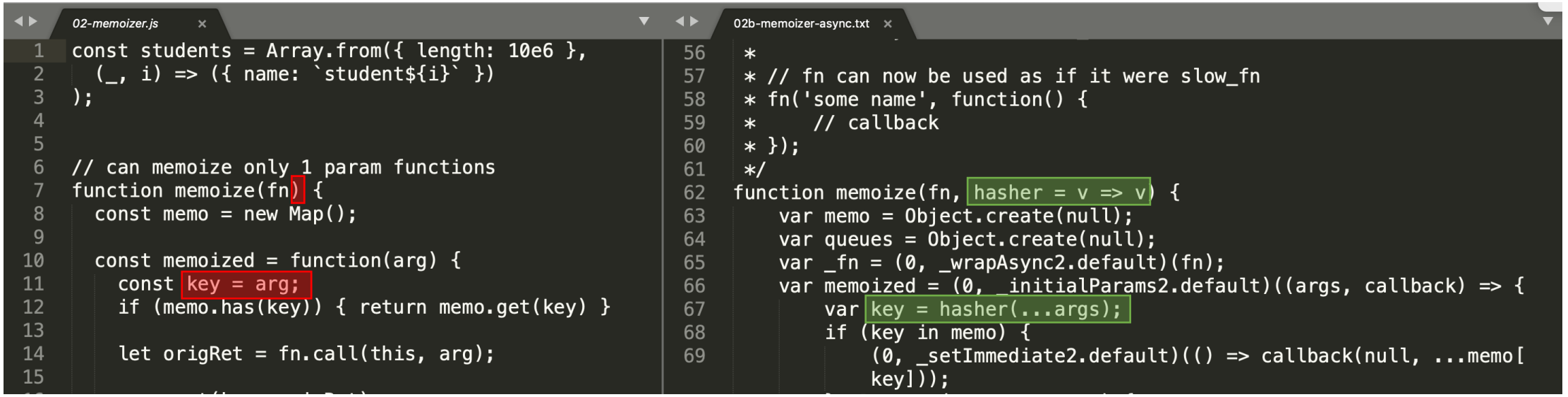
```
best((o1, o2) => o1.age > o1.age, people)

best(({ speed1 }, { speed2 }) => speed1 > speed2, cars)
```

max/best example recap

- Poradie parametrov ?
- Defaultne hodnoty ?
  - Áno aj funkcia môže mat defaulty v signatúre metódy

Parametrized memoizer



02-memoizer.js

```
const students = Array.from({ length: 10e6 },
  (_, i) => ({ name: `student${i}` })
);


// can memoize only 1 param functions
function memoize(fn) {
  const memo = new Map();

  const memoized = function(arg) {
    const key = arg;
    if (memo.has(key)) { return memo.get(key) }

    let origRet = fn.call(this, arg);
```

02b-memoizer-async.txt

```
 *
 * // fn can now be used as if it were slow_fn
 * fn('some name', function() {
 *     // callback
 * });
 */
function memoize(fn, hasher = v => v) {
    var memo = Object.create(null);
    var queues = Object.create(null);
    var _fn = (0, _wrapAsync2.default)(fn);
    var memoized = (0, _initialParams2.default)((args, callback) => {
        var key = hasher(...args);
        if (key in memo) {
            (0, _setImmediate2.default)(() => callback(null, ...memo[
            key]));
```

- Na dva riadky sparametrizujeme napríklad nás memoizer s predošlých príkladov

Parametrized, ale by function !!!

1. Ak máte len jeden prípad uniqBy, je moré ok
2. Ak ale potrebujete uniq by niečo iné, tak to nerobte tým červeným COPY PASTE spôsobom
3. Orange je dobre, ak vám stači unique podľa jednej property
4. Ale čo keď chcete podľa viacerých props alebo podľa „length“, nepridávajte ďalšie parametre a zamyslite sa či nespravíte miesto X parametrov funkciu
5. A keď spravíte to zelené tak skúste backward compatible (interne funcia, na vstupe string)
6. A keď už otravujete z funkciami na vstupde dajte im defaulty

Je to spravidla postupný refactor, nie „first look design“

02-uniqueBy.js

```
1  // BAD:
2  const uniqueName = (arr) => {
3    return [...new Set(arr.map((a) => a.name))];
4  }
5  const uniqueAge = (arr) => {
6    return [...new Set(arr.map((a) => a.age))];
7  }
8
9  // BETTER: value as param
10 const unique = (arr, key) => {
11   return [...new Set(arr.map((a) => a[key]))];
12 }
13
14 // EVEN BETTER: function as param
15 const unique = (arr, key) => {
16   const hash = typeof key === "string" ?
17     o => o[key] :
18     key;
19   return [...new Set(arr.map(hash))];
20 }
21
22 // EVEN BETTER: function as param + default
23 const unique = (arr, key = (x) => x) => {
24   const hash = typeof key === "string" ?
25     (item) => item[key] : key
26   return [...new Set(arr.map(hash))];
27 }
```

Parametrized, ale by function !!!

1. Ak máte len jeden prípad uniqBy, je moré ok
2. Ak ale potrebujete uniq by niečo iné, tak to nerobte tým červeným COPY PASTE spôsobom
3. Orange je dobre, ak vám stači unique podľa jednej property
4. Ale čo keď chcete podľa viacerých props alebo napr. podľa „length“, nepridávajte ďalšie parametre a zamyslite sa či nespravíte miesto X parametrov funkciu
5. A keď spravíte to zelené tak skúste backward compatible (interne funkcia, na vstupe string)
6. A keď už „otravujete“ z funkciami na vstupe dajte im defaulty

```
let names = uniqueName(students);
let ages = uniqueAges(students);

let names = unique(students, "name");
let ages = unique(students, "ages");

let names = unique(students, "name");
let ages = unique(students, (({ age, gender }) =>
    JSON.stringify({ age, gender }
));

let ages = unique([1, 2, 2]);
```

Je to spravidla postupný refactor, nie „first look design“

parametrized generation, iterations

- from RX JS: generate(
  ```
  0,                  // start with this value
  x => x < 10,    // condition: emit as long as a value is less than 10
  x => x*2          // iteration: double the previous value
  )
  ```
- async.js
  - doWhilst, whilst, until, doUntil

- Hlavnou výhodou je, že iteration, condition a ostatné prvky daného loopu sú funkcie (a môožu byť async), čo sa Vám z for, while atď nepodarí napísať.
- Konštrukcie z funkciami sú podstatne “flexibilnejšie”

parametrized algorithms

Načo je to dobré ?

- Miesto vymýšlania mien funkcii a wrapovania funkcionality,
- Vymýšlame názvy replacerov, filtrov, comparatorov a reusujeme ich
- Máme väčšiu dynamiku a flexibilitu,
- …

- `replaceSpaces(str)`
- `removeNewLines`
- `removeEmplyLines`
- `sortById`
- `sortCaseInsensitive`
- `findOldest(arr)`
- `serializeNonPrivate(obj)`

- `str.replace(spaces,"-")`
- `str.replace(newlines,"")`
- `str.replace(newlines,removeEmpty)`
- `arr.sort(byId)`
- `arr.sort(caseInsensitive)`
- `arr.find(oldest)`
- `json.stringify(o, nonPrivate)`

transforming params, filtering arguments

```
// initial motivation:
// implement max() similar to Math.max()
// but be sure that the mathod does not fail
// (does not return NaN if some parameter is not convertible to number)
```

```
// max implementations
// this would be manual code, not bad
// however let's try other approach
const maxNumber = (...args) => Math.max(...args.filter(isNumber));

// implementation using transforming function
const maxNumber = filterArgs(Math.max, isNumber);
```

see also: lodash (underscore)

```
_.overArgs(func, [transforms=[_.identity]])
```

source npm package

Creates a function that invokes `func` with its arguments transformed.

09-cvicenie/src/filterArgs.js

```
module.exports = (orig, ...filters) =>
  (...args) => orig(...args.filter(and(...filters)));
```

```
const and = (f1, ...fns) => x => !!fns
  .reduce((r, fn) => r = r && fn(x), f1(x));
```

Immutability

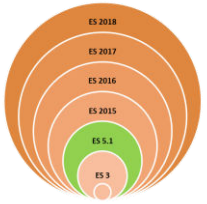
Immutability

- Immutability is essential in FP and VOP
- Immutability can be **forced by language** or we have to rely on **coding discipline**
- In JS:
  - By default things are mutable in JS
  - A lot of build-in APIs are designed as mutable
  - with FP trends, JavaScript is taking evolution towards immutability as well

Immutability support in JavaScript

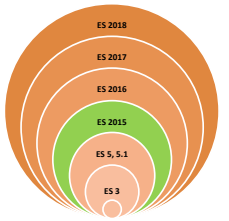
- primitive types are immutable
- object is mutable (by default),
- all other data structures are objects so they are mutable as well
- ES 5.1
  - **nonwritable properties** (Object.defineProperty, Object.create)
  - **Object.freeze**, …
  - strict mode
- ES 2015
  - **const**
  - destructuring, spread
- ES 2018
  - rest and spread for Objects as well

ES 5. 5.1 (*2009, 2011*)



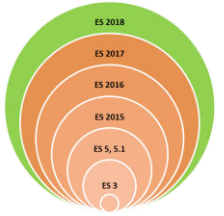
- Array functions (filter, map, reduce,...)
- Function.bind
- Object immutability (freeze, seal, ...)
- Object properties defineProperty(),getOwnPropertyDescriptor (), keys()
- Object.create
- getters, setters
- JSON API
- strict mode
- ...

ES 2015 (*6th*)



- Promise, Generators
- arrow functions
- destructuring assignment, spread operator
- rest and default parameters
- Template Strings
- block scope variables (let, const)
- for..of loop and iterators on build ins
- Object.* - setPrototypeOf, assign, is() SameValue equality,
- Symbols
- Classes
- Number.* non coercing functions
- String codePoints
- Map, Set
- Proxy, Reflect

ES 2018 (*9th*)



- RegExp
  - Named Group Capture
  - Lookbehind Assertions
  - Unicode Property Escapes
- Promises
  - Promise.prototype.finally()
  - for-await-of
- rest and spread for Objects
- Shared memory Atomics
- Tagged Template literal restriction removed
- ...

Immutability – shared state

- orthodox function programming – no shared state
- real JS programs - we often see and use state
  - captured variables in closures, module variables,shared reused modules, objects,....

- **Shared state** is *fine* if it is immutable
- **Mutable state** is *fine* if it is not shared.

Shared state is fine if it is **immutable,**
Mutable state is fine if it is **not shared**.

Čo máme považovať za ***state,*** *kedy je a nie je* ***shared****, čo* to znamená ***mutable***

State:

- variable with value, function call argument

Shared:

- strictly : visible by more than one function
- visible outside of module
- ES6 module live binding
- visible by scope or passed as argument
- inherited by prototype chain
- returned by function

Mutable variable:

- non const variable can be reassigned value (let, var)
- default object member (writable)
- FunctionDeclaration
- non const FunctionExpression or ArrowFunctionExpression

Mutable value:

- non primitive data types – Object (Array, ….) with mutable members (props, values) or mutator methods

State:

- variable with value, function call argument

Not Shared:

- strictly: no shared variables – parametrized/bound functions
- private to module, not exported
- CJS copied
- …
- own ”private” property
- ….

Immutable variable:

- const
- nonwritable object member
- …
- const FunctionExpression or ArrowFunctionExpression

Immutable value:

- primitive data types, frozen objects, special userland structures

How to achieve immutability and produce changed values

Immutable variable:

- const
- nonwritable object member
- …
- const FunctionExpression or ArrowFunctionExpression

Immutable value:

- primitive data types,
- frozen objects
- special userland structures (maps, sets)

Returning clones:

- copy object to change property value
- clone array to add, remove or modify items
- …
- …

Shared/mutated variable

Asi najklasickejší príklad:

https://eslint.org/docs/rules/no-loop-func

```
function createJobs() {
  var jobs = [];
  for (var i = 0; i < 3; i++) {
    jobs[i] = function job() {
      console.log(i);
    }
  }
  return jobs;
}

var jobs = createJobs();

jobs[0]();
jobs[1]();
jobs[2]();

/* prints:

3
3
3

asi nie to co by sme cakali */
```

Shared/mutated variable

Čo je tam zle ?

Z FP pohľadu všetko

Ako to riešiť ?

Shared:

strictly : visible by more than one function

Not Shared:

strictly: no shared variables – parametrized alebo bound functions

```
function createJobs() {
  var jobs = [];
  var i;                          // 1.) shared lebo var scope
  for (i = 0; i < 3; i++) {       // 3.) mutated
    jobs[i] = function job() {
      console.log(i);             // 2.) visible, scope
    }
  }
  return jobs;
}

var jobs = createJobs();

jobs[0]();                        // 4.) i je accessible (lebo closure)
jobs[1]();
jobs[2]();

/*
                                  // 5.) "unexpected"

3
3
3

*/
```

Shared/mutated variable

Not Shared:

no shared variables

- parametrized
- alebo bound functions
- let

V samploch mate príklady aj z vysvetlením a komentármi: z pohľadu FP je IMHO najčistejší bind

01-var-in-for-loop.js

01b-var-in-for-loop.js

02-closure-in-for-loop.js

02b-closure-in-for-loop.js

02c-bind-in-for-loop.js

03-let-in-for-loop.js

04-map-fill.js

Shared/mutated variable

Jedno z možných riešení, pozrite si tie ostatné, všetko záleží na konkrétnom prípade, čitateľnosti, pisateľnosti, atď....

```
1  // nemam sa spoliehat na ziadny scope
2  // mam mat funkcie co ocakavaju i ako parameter
3
4  const job = (i, p1, p2) => console.log(i, p1, p2);
5
6  const createJobs = (length) =>
7    // nemam citat scopovany premennu (var, let) ale mam to dostat
8    // ako parametre (fill, from, map) i v callbacku
9    Array.from({ length }, (_, i) =>
10     // novu funkciu z prednasatvenymi parametrami
11     // mam vyrabat cez bind
12     job.bind(null, i));
13
14
15 const jobs = createJobs(3);
16
17 jobs[0]();
18 jobs[1]();
19 jobs[2]();
20
21 jobs[0]("a", "b");
```

Object manipulations

Objects as values

and common operations on objects in JS FP

List and objects manipulations

- List (Array) and List of structures (Objects) are essential for FP
- A lot of app logic can be written as sequential/chained manipulation of **Arrays of Objects**
- How to perform following operation
  - ideally in non mutating way
  - with build in API and syntax
  - with *userland* libs

- Object operations
  - modify/add property
  - remove property
  - whitelist properties
  - blacklist properties
  - clone object (shallow/deep)
  - merge objects (shallow/deep)
  - traversing objects

/samples/04-objects-in-fp/01-object-manipulations

add/set property

mutating: = operator, Object.assign()

```
const assert = require("assert");

var students = [
  { name: "Marcus", grades: [1, 2, 2, 5] },
  { name: "John", grades: [3, 2, 1, 1, 1, 1] },
  { name: "Emilia", grades: [5, 4] }
];

// what is wrong with this code ?
var fiitStudents = students.map((s) => {
  s.school = "FIIT";
  return s;
});

// what is wrong with this code ?
var fiitStudents = students.map((s) =>
  Object.assign(s, { school: "FIIT" })
);

assert(fiitStudents !== students,
  "array is not mutated, OK");

assert(fiitStudents[0].school === "FIIT",
  "items in new array have school, OK");

assert(students[0].school === "FIIT",
  "but also original students have school, BAD");

assert(students[0] === fiitStudents[0],
  "because both arrays point to same object");
```

non mutating: spread properties, Object.assign({})

- The Rest/Spread Properties for ECMAScript adds spread properties to object literals. It copies own enumerable properties from a provided object onto a new object.
- Shallow-cloning (excluding prototype) or merging of objects is now possible using a shorter syntax than Object.assign().
- The **Object.assign()** method is used to copy the values of all enumerable own properties from one or more source objects to a target object. It will return the target object.

```
// non mutation map
var fiitStudents = students.map((s) => ({
  ...s,
  school: "FIIT"
}));

// non mutation map
var fiitStudents = students.map((s) =>
  Object.assign({}, s, { school: "FIIT" })
);

assert(fiitStudents !== students,
  "array is not mutated, OK");

assert(fiitStudents[0].school === "FIIT",
  "items in new array have school, OK");

assert(students[0].school === undefined,
  "original students do not have school, OK");

assert(students[0] !== fiitStudents[0],
  "arrays do not point to same object, OK");
```

remove/clear property

mutating delete, Object.assign()

```
const assert = require("assert");

var students = [
  { name: "Marcus", grades: [1, 2, 2, 5] },
  { name: "John", grades: [3, 2, 1, 1, 1, 1] },
  { name: "Emilia", grades: [5, 4] }
];

// what is wrong with this code ?
var anonymousStudents = students.map((s) => {
  delete s.name;
  return s;
});

// what is wrong with this code ?
var anonymousStudents = students.map((s) =>
  Object.assign(s, { name: undefined })
);

assert(anonymousStudents !== students,
  "array is not mutated, OK");

assert(anonymousStudents[0].name === undefined,
  "items in new array have no names, OK");

assert(students[0].name === undefined,
  "but also original students have names deleted, BAD");

assert(students[0] === anonymousStudents[0],
  "because both arrays point to same object");
```

non mutating rest properties, Object.assign({})

```
// non mutating omit of property
var anonymousStudents = students.map((s) => {
  let { name, ...others } = s;
  return { ...others };
});

// non mutating 'clear' of property
var anonymousStudents = students.map((s) =>
  Object.assign({}, s, { name: undefined })
);

assert(anonymousStudents !== students,
  "array is not mutated, OK");

assert(anonymousStudents[0].name === undefined,
  "items in new array have no names, OK");

assert(students[0].name === "Marcus",
  "original students have names, OK");

assert(students[0] !== anonymousStudents[0],
  "because both arrays point to DIFFERENT object, OK");
```

whitelisting, blacklisting

- non mutating: rest/spread properties
- **blacklisting** – do not include specified properties
- **whitelisting** – include only specified properties

- blacklisting : destructure object, name blacklisted properties explicitly, and spread …rest to returned object
- whitelisting: destructure wanted properties (do not mention others) and return object with property shorthand syntax

```
const assert = require("assert");

var students = [
  { name: "Marcus", surname: "M", grades: [1, 2, ], rank: 1 },
  { name: "John", surname: "Doe", grades: [3, 2, ], rank: 2 },
  { name: "Emilia", surname: "Black", grades: [5, 4], rank: 2 }
];

// blacklisting
var anonymousStudents = students.map((s) => {
  let { name, surname, ...allowed } = s;
  return { ...allowed };
});

// whitelisting
var namesOnly = students.map((s) => {
  let { name, surname } = s;
  return { name, surname };
});

// both can be done as part of parameter destructuring

// blacklist by param destructuring
var anonymousStudents = students.map(
  ({ name, surname, ...allowed }) => ({ ...allowed })
);
var namesOnly = students.map(
  ({ name, surname }) => ({ name, surname })
);


console.log(anonymousStudents)
console.log(namesOnly)
```

cloning objects

- shallow copy
- deep copy

- there is no Object.clone() overridable on objects like in Java
- we have several mechanisms to do it, depending on purpose of clone, with respect to different properties (own, inherited, setters, getters, enumerable, etc..)
- here are only few often used
  - Spread, assign, JSON
- JS has no mechanism for deep copy

```
describe("POC - cloning objects", () => {

  it("object spread", () => {

    var o = demoObject();
    var c = {
      ...o
    };

    assert(c !== o);
    assert(c.enumerable === 1);
    assert(c[s] === 1);
    assert(c.nonenumerable === undefined);
    assert(c.inherited === undefined);

  });
  it("Object.assign({})", () => {

    var o = demoObject();
    var c = Object.assign({}, o); ///!!! {}

    assert(c !== o);
    assert(c.enumerable === 1);
    assert(c[s] === 1);
    assert(c.nonenumerable === undefined);
    assert(c.inherited === undefined);

  });
  it("JSON.parse(JSON.stringify())", () => {

    var o = demoObject();
    var c = JSON.parse(JSON.stringify(o));

    assert(c !== o);
    assert(c.enumerable === 1);
    assert(c[s] === undefined);
    assert(c.nonenumerable === undefined);
    assert(c.inherited === undefined);

  });
})
```

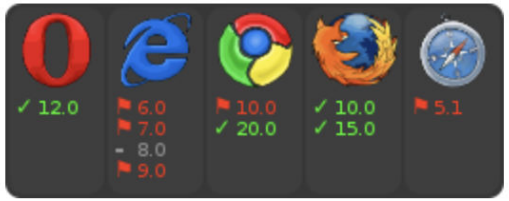
samples/04-objects-in-fp/02-cloning-objects/objects.spec.js

Deep Clone

- JSON.parse(JSON.stringify) as good trick
- or userland libraries, my suggestion is npm traverse, usable for other tasks as well

traverse

Traverse and transform objects by visiting every node on a recursive walk.



.clone()

Create a deep clone of the object.

```
test('clone', function (t) {
    var obj = { a : 1, b : 2, c : [ 3, 4 ] };
    var res = traverse(obj).clone();
    t.same(obj, res);
    t.ok(obj !== res);
    obj.a ++;
    t.same(res.a, 1);
    obj.c.push(5);
    t.same(res.c, [ 3, 4 ]);
    t.end();
});

test('cloneT', function (t) {
    var obj = { a : 1, b : 2, c : [ 3, 4 ] };
    var res = traverse.clone(obj);
    t.same(obj, res);
    t.ok(obj !== res);
    obj.a ++;
    t.same(res.a, 1);
    obj.c.push(5);
    t.same(res.c, [ 3, 4 ]);
    t.end();
});
```

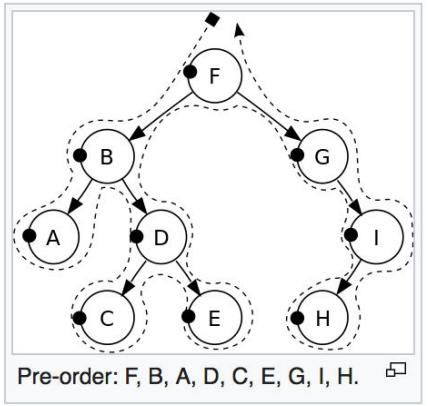
bonus: nastudujte testy k deep clone a doplnte test case (ak neni) na nested object, nie len array, **3body za accepted PR**

Object (tree) Traversal

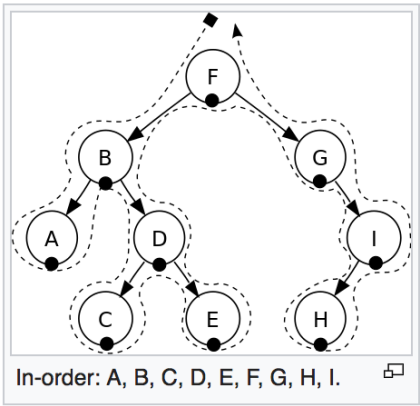
Trees are often represented as objects with properties in JS

Traversal - basics

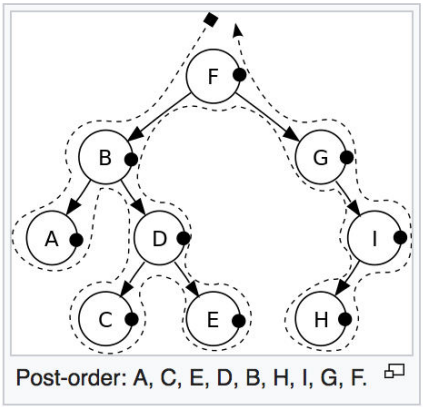
- process of **visiting** (checking and/or updating) each node in a **tree data structure**, exactly once.
- Such traversals are classified by the order in which the nodes are visited



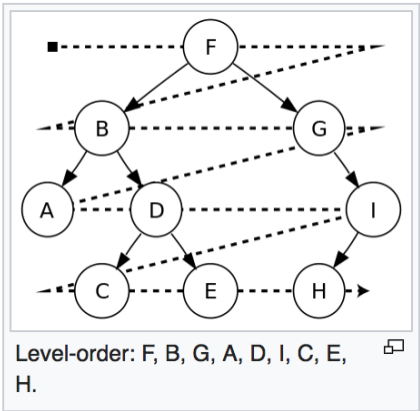
Pre-order: F, B, A, D, C, E, G, I, H.



In-order: A, B, C, D, E, F, G, H, I.



Post-order: A, C, E, D, B, H, I, G, F.



Level-order: F, B, G, A, D, I, C, E, H.

```
{
    F: {
        B: {
            A: "A",
            D: {
                C: "C",
                E: "E"
            }
        },
        G: {
            I: {
                H: "H"
            }
        }
    }
}
```

images from: https://en.wikipedia.org/wiki/Tree_traversal

Traversal

- ES nor node.js has no build in Tree data structure
- Options:
  - JS objects
  - custom or userland Tree implementations
- ES has no traversal algorithms build in, except for:
  - JSON.parse, JSON.stringify
  - custom impl. or *userland* libraries must be used

JSON.parse and JSON.stringify (reviver and replacer)

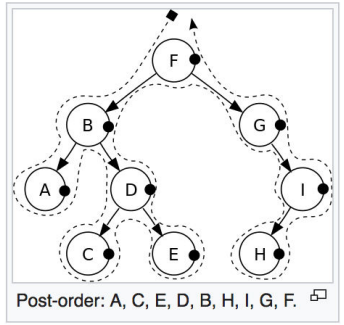
- These are the basic traversal impls, available out of box.
- Also JSON is the first/last place to perform transformations, and to avoid useless object creation, memory consumption, redundant traversal and other smells after deserialization or before serialization to JSON.
- However these traversals are not usable for general usage only for serialization and deserialization
- The **JSON.parse()** method parses a JSON string, constructing the JavaScript value or object described by the string. An optional **reviver function** can be provided to perform a transformation on the resulting object before it is returned.

```
JSON.parse(text[, reviver])
```

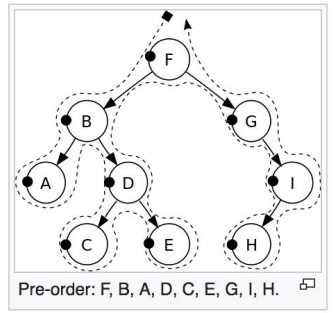
- The **JSON.stringify()** method converts a JavaScript value to a JSON string, optionally replacing values if a **replacer function** is specified or optionally including only the specified properties if a replacer array is specified.

```
JSON.stringify(value[, replacer[, space]])
```

```
{
    F: {
        B: {
            A: "A",
            D: {
                C: "C",
                E: "E"
            }
        },
        G: {
            I: {
                H: "H"
            }
        }
    }
}
```



Post-order: A, C, E, D, B, H, I, G, F.



Pre-order: F, B, A, D, C, E, G, I, H.

JSON.parse(text[, reviver]). Reviver will receive nodes it the following order

```
postOrder: ['A', 'C', 'E', 'D', 'B', 'H', 'I', 'G', 'F', null],
```

JSON.stringify(value[, replacer[, space]]). Replacer will receive nodes it the following order:

```
preOrder: [null, 'F', 'B', 'A', 'D', 'C', 'E', 'G', 'I', 'H'],
```

Traversal

- separate **traverse** algorithm and **visitor method**
- implementation is usually **recursive**
- based on **"object properties" APIs** (see objects lesson)
- sample implementation simulating postOrder on JS object tree

```
// postOrder
function traverse(obj, visitor, key = null, parent = null) {
  if (typeof obj === "object")
    Object.keys(obj).forEach(function(key) {
      traverse(this[key], visitor, key, this);
      visitor(key, this[key], this);
    }, obj);
  if (key === null)
    return visitor(null, obj, null);
}


traverse({ a: 1, b: 2, c: { d: 3, e: 4 } },
  (k, v, o) => console.log("k:", k, "v:", v, "o:", o)
);


/*
k: a v: 1 o: { a: 1, b: 2, c: { d: 3, e: 4 } }
k: b v: 2 o: { a: 1, b: 2, c: { d: 3, e: 4 } }
k: d v: 3 o: { d: 3, e: 4 }
k: e v: 4 o: { d: 3, e: 4 }
k: c v: { d: 3, e: 4 } o: { a: 1, b: 2, c: { d: 3, e: 4 } }
k: null v: { a: 1, b: 2, c: { d: 3, e: 4 } } o: null
*/
```

Traversal

- implementing something else ? call appropriate **traverse** with **change visitor function**
- example of deepFreeze
- freeze – postOrder alebo preOrder ?
- can **visitor** be improved ?

```
traverse({ a: 1, b: 2, c: { d: 3, e: 4 } },
  (k, v, o) => console.log("k:", k, "v:", v, "o:", o)
);



var o = Object.freeze({ a: 1, b: 2, c: { d: 3, e: 4 } });
o.c.d = 888; //freeze is shallow

var o = traverse({ a: 1, b: 2, c: { d: 3, e: 4 } },
  (_, v) => Object.freeze(v)
);
o.c.d = 999; //will fail now



/*
k: a v: 1 o: { a: 1, b: 2, c: { d: 3, e: 4 } }
k: b v: 2 o: { a: 1, b: 2, c: { d: 3, e: 4 } }
k: d v: 3 o: { d: 3, e: 4 }
k: e v: 4 o: { d: 3, e: 4 }
k: c v: { d: 3, e: 4 } o: { a: 1, b: 2, c: { d: 3, e: 4 } }
k: null v: { a: 1, b: 2, c: { d: 3, e: 4 } } o: null
*/
```

npm traverse

- 3rd party library
- preOrder with hooks
- forEach, map, reduce for even more semantic operations on traversal
- instead of params uses this context, with all infos (key, value, parent, level, isLeaf)

https://github.com/substack/js-traverse

```
'use strict';
const traverse = require("traverse");

//Pre-order:
traverse.forEach({ a: 1, b: 2, c: { d: 3, e: 4 } },
  print);

// Post-order:
traverse.forEach({ a: 1, b: 2, c: { d: 3, e: 4 } }, function() {
  this.post((n) => print.call(n));
});


function print() {
  console.log(
    "k:", this.key,
    "v:", this.node,
    "o:", this.parent && this.parent.node
  );
};

/*
k: a v: 1 o: { a: 1, b: 2, c: { d: 3, e: 4 } }
k: b v: 2 o: { a: 1, b: 2, c: { d: 3, e: 4 } }
k: d v: 3 o: { d: 3, e: 4 }
k: e v: 4 o: { d: 3, e: 4 }
k: c v: { d: 3, e: 4 } o: { a: 1, b: 2, c: { d: 3, e: 4 } }
*/
```

Traversal libraries

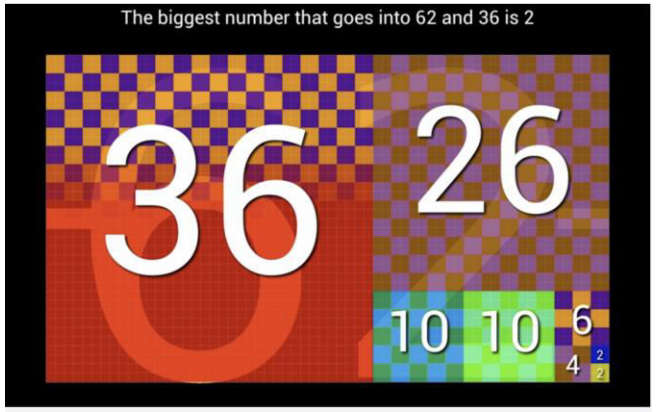
other interesting libs:

- https://github.com/nervgh/recursive-iterator
  - **iterator api** for vertical, horizontal (preOrder, levelOrder, seems to have no support for postOrder), nedaju si pokoj ty zastancovia pull API a for cyklov ;-))
  - Sample: 04-objects-in-fp/03-traversal/03-npm-recursive-iterator.js
- https://github.com/ainthek/js-tree-travesal
  - my old project with skeleton impls. of pre,post,in order algs. and interesting usecases of JSON stringify() and parse()
- …

Recursion

Recursion - divide and conquer

- Rekurzívna funkcia je taká, ktorá volá sama seba až kým.....
- Principle: divide and conquer
- Break the problem to smaller parts until you can find solution
- Aké kachličky mam kúpiť keď kúpeľňa ma 26 m a kuchyňa ma 36 metrov a chcem ich mat rovnaké (Euclidus alg for GCD)




```
function countDown(n) {
  if (n == 0) return; // conquer
  else {
    countDown(n - 1); // devide
  }
}
countDown(10);
```

```
function tileSize(a, b) {
  // ak su miestnosti rovnake
  if (a === b) return a; // conquer
  // ak je jedna vecsia ako druha
  if (a > b) return tileSize(a - b, b);
  if (b > a) return tileSize(a, b - a);
}
```

Recursion - merge sort

- classic example
- array is split to small sorted pieces and the reconstructing by merging those already sorted pieces back

```
function mergeSort(a) {

  if (a.length <= 1) return a;

  let [left, right] = halfs(a);
  left = mergeSort(left);
  right = mergeSort(right);

  return merge(left, right);
}

function halfs(a) {
  const h = ~~(a.length / 2);
  return [a.slice(0, h), a.slice(h)];
};

function merge(left, right) {
  var result = [],
    il = 0,
    ir = 0;
  while (il < left.length && ir < right.length) {
    result.push(left[il] < right[ir] ?
      left[il++] :
      right[ir++]
    );
  }
  return result
    .concat(left.slice(il))
    .concat(right.slice(ir));
}
```

end

```
MS [ 1, 5, 3, 9, 6, 4, 8 ]
MS [ 1, 5, 3 ]
MS [ 1 ]
MS [ 5, 3 ]
MS [ 5 ]
MS [ 3 ]
MI [ 5 ] [ 3 ]
MO: [ 3, 5 ]
MI [ 1 ] [ 3, 5 ]
MO: [ 1, 3, 5 ]
MS [ 9, 6, 4, 8 ]
MS [ 9, 6 ]
MS [ 9 ]
MS [ 6 ]
MI [ 9 ] [ 6 ]
MO: [ 6, 9 ]
MS [ 4, 8 ]
MS [ 4 ]
MS [ 8 ]
MI [ 4 ] [ 8 ]
MO: [ 4, 8 ]
MI [ 6, 9 ] [ 4, 8 ]
MO: [ 4, 6, 8, 9 ]
MI [ 1, 3, 5 ] [ 4, 6, 8, 9 ]
MO: [ 1, 3, 4, 5, 6, 8, 9 ]
```

call order

Recursion

- Everything that can be done with for loop can be done with recursion
- Not everything that can be done with recursion can be done (easily) by for loop
  - **tree** traversal or building
  - recursion may help in **async**
    - there are no for/do/while loops for async tasks
  - and other …..

```
var items = [
  { id: "JavaScript", parent: null },
  { id: "Operators", parent: "JavaScript" },
  { id: "Buildins", parent: "JavaScript" },
  { id: "Array", parent: "BuildIns" },
  { id: "Set", parent: "BuildIns" },
  { id: "Functions", parent: "JavaScript" },
  { id: "FuncDec", parent: "Functions" },
  { id: "FuncExpr", parent: "Functions" },
  { id: "ArrowFuncExpr", parent: "Functions" }
];
```

```
var expected = {
  JavaScript: {
    Operators: {},
    Buildins: {
      Array: {},
      Set: {}
    },
    Functions: {
      FuncDec: {},
      FuncExpr: {},
      ArrowFuncExpr: {}
    }
  }
}
```

```
var arr2obj = function(data, parent) {
  return data
    .filter(d => d.parent === parent)
    .reduce((r, d) => {
      r[d.id] = arr2obj(data, d.id);
      return r;
    }, {});
}
```

Recursion – other topics

Types of recursion

- Single recursion and multiple recursion
- Indirect recursion
- Anonymous recursion
- Structural versus generative recursion

Recursive programs

- Recursive procedures
- Recursive data structures (structural recursion)

Tail-recursive functions

Order of execution

Shortcut rule (master theorem)

Implementation issues

- Wrapper function
- Short-circuiting the base case
- Hybrid algorithm

Recursion versus iteration

- Expressive power
- Performance issues
- Stack space
- Vulnerability
- Multiply recursive problems
- Refactoring recursion

…….

Recursion - Stack space

- JS function are executed in the stack
- recursion means filling up stack with the same function
- stack has limited size and recursion can consume (fill up) the stack
- other FP languages/runtimes not based on stack

```
$ node -e 'function f() { return f(); }; f();'
[eval]:1
function f() { return f(); }; f();
           ^

RangeError: Maximum call stack size exceeded
    at f ([eval]:1:11)
    at f ([eval]:1:23)
    at f ([eval]:1:23)
    at f ([eval]:1:23)
    at f ([eval]:1:23)
    at f ([eval]:1:23)
    at f ([eval]:1:23)
    at f ([eval]:1:23)
    at f ([eval]:1:23)
    at f ([eval]:1:23)
    at f ([eval]:1:23)
```

Recursion – tail call optimization

- **Implicit tail call optimization (**TCO) **– part of** ES 2015 - you can make some function calls without growing the call stack
- **not implemented** (only Safari, XS6, DUK) http://2ality.com/2015/06/tail-call-optimization.html
- **ECMAScript 6 Proper Tail Calls in WebKit(PTC)** https://webkit.org/blog/6240/ecmascript-6-proper-tail-calls-in-webkit/
- **Syntactic Tail Calls (STC) – proposal** https://github.com/tc39/proposal-ptc-syntax/blob/master/README.md

For these reasons, the V8 team strongly support denoting proper tail calls by special syntax. There is a pending TC39 proposal called syntactic tail calls to specify this behavior, co-championed by committee members from Mozilla and Microsoft. We have implemented and staged proper tail calls as specified in ES2015 and started implementing syntactic tail calls as specified in the new proposal. The V8 team plans to resolve the issue at the next TC39 meeting before shipping implicit proper tail calls or syntactic tail calls by default. You can test out each version in the meantime by using the V8 flags `--harmony-tailcalls` and `--harmony-explicit-tailcalls`.

https://v8.dev/blog/modern-javascript

partial and curry

Object.bind

Bind (v pohľadu FP) používame na to, aby sme z existujúcej funkcie z 5 parametrami urobili funkciu z 3 parametrami

A. Vieme tie parametre v inom čase a inom kuse programu ako budeme volať finálnu funkciu
B. Vytvárate konkrétnejšie verzie všeobecného API
C. Chceme krajší/sémantickejší zápis

Existujúca funkcia môže byť

- Build in API (Array.prototype, Object.is)
- Cudzie API

- Bind vs. inline arrow
  - Odkedy máme arrows aj tak všetci píšu arrow, ale to nie je krajší/sémantickejší zápis, tým riešia iba ten bod A.
- Navyše bind vie doplniť iba parametre z predu, takže API musí byť vhodne napísané

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_objects/Function/bind

partially applied functions pomocou bind

B) The next simplest use of bind() is to make a function with pre-specified initial arguments.

A. Vytvárate konkrétnejšie verzie všeobecného API
B. Chceme krajší/sémantickejší zápis

Tak ako v OO: pôvodná trieda mala 10 parametrov, nová špecializácia ma mat len 3, ostatne si nastavíte vy v podedencovi....

```
1  function pad(left, right, string) {
2
3    return left + string + right;
4
5  }
6  //----------------------------------------
7
8  pad("'", "'", "quoted");
9  pad("\t", "", "tabed");
10
11 //----------------------------------------
12
13 const indent = pad.bind(null, "\t")
14 indent("", "tabed")
15
16 //----------------------------------------
17
18 const quote = pad.bind(null, "'", "'")
19 quote("quoted")
20
21 //----------------------------------------
22
23 const jsLine = indent.bind(null, ";");
24 jsLine("jsLine");
25
26 //----------------------------------------
27 const voidLine=jsLine.bind(null,"void(0)")
28
29 voidLine()
30 voidLine("foo")
31
32 //----------------------------------------
```

Object.bind

- Reducing „code noise“ with bind
- Žltou je zaznačený noise v business kóde, šedivou je označený noise v implementačnom kóde
- Predstavte si z to žlte číta analytik na leveli abstrakcie 1 a to šedivé je vaša implementácia v inom module.

```
1   const grades = [1, 5, 3, 5, 4]
2
3   grades.filter((g) => 5 === g)
4
5   grades.filter((g) => Object.is(5, g))
6
7   var is = (c) => (v) => Object.is(c, v)
8   grades.filter(is(5));
9
10  var is = (c) => Object.is.bind(null, c)
11  grades.filter(is(5))
12
13  var is5 = Object.is.bind(null, 5)
14  grades.filter(is5)
```

Curry

Samoštúdium:

- Building new Functions By giving functions arguments
  - https://www.youtube.com/watch?v=m3svKOdZijA
- Otázka na skúšku:
  - Aký je hlavný rozdiel medzi curry a partially applied function ?

- Definiície (tiež sa líšia):
  - Currying – decomposition of function with multiple arguments into a chain of single-argument functions....
  - Partial application of function – pass to function less arguments than it has in its declaration...

- Moja definícia:
  - Partial function musím zavolať, aj keď ju bindnem zo všetkými parametrami
  - Curry sa mi zavolá pri dodaní posledného parametra

Functional JavaScript (Summary)

- JavaScript is a multi-paradigm language
- It can be used to program functionally (specially with “modern JS”)
- But (my opinion, my current “functional JS POV”), so far on covered topics
  - use **functional concepts for business**, keep the rest “as needed” (procedural, declarative)
  - use **functional concepts for functional problems**
  - use **chaining** (nicer syntax, more readable programs)
  - use (parameter) **destructuring**, rest parameters, defaults (less bloated code, less need for low level FP primitives, branching)
  - use (study) **JS syntax**, if exists whenever possible, do **not hide known JS** under unknown libs
  - **do not implement low level functional features** by reusing functional features (e.g use raw loops to implement _ranges, etc…, beware performance, O(n), call stack price, memory)
  - do **not implement** functional low level features, **use libs** (eg. _underescore.js), use only what needed, more and more is replaced by standard JS syntax)
  - use **arrow functions only for inlines** (I like hoisting, more readable programs, top down reading, code first, then functions, function(){}, vs const=()=>{})
  - do not follow blindly FP concepts eg. “using functions instead of values” f(f()) vs f(v), we can have f(funOrValue)
  - **use recursion only where appropriate** for “problem solving”, **reduce is almost always fine** if not needing quick exits
  - implement **mappers, filters, reducers**, instead of implementing “whole methods”


_aux

JS is OO, why not benefit ? lazy and memoize

- memoize is concept for functions
- but we have object and properties
  - properies have getters
  - properties can be redefined
  - properties can be memoized (as static property)

Collection pipelines

- https://martinfowler.com/articles/collection-pipeline/#NestedOperatorExpressions
- *Collection pipelines are a programming pattern where you organize some computation as a sequence of operations which compose by taking a collection as output of one operation and feeding it into the next. (Common operations are filter, map, and reduce.) This pattern is common in functional programming, and also in object-oriented languages which have lambdas. This article describes the pattern with several examples of how to form pipelines, both to introduce the pattern to those unfamiliar with it, and to help people understand the core concepts so they can more easily take ideas from one language to another.*

Some deep thoughts

- **JavaScript doesn’t have immutability built in**, spread operator and external libs can easily help in implementing it though. If immutability is about rules, then everything is in the developer hands. If you see *the value of value* then from this moment the mutation should be gone in your application.
- However the problem is in bad habits, **bad habits mostly from object oriented programming**. Bad practices are in our minds, under our skin. Bad practices like setters in the object, methods mutating the object state, object as state, huge polymorphic objects and so on, **these things should be considered as wrong and never used again**.
- **Change your way of thinking**, see *the value of value*. Choosing immutability is a big step towards **predictable,** more efficient and **less buggy programs**.

https://medium.com/dailyjs/the-state-of-immutability-169d2cd11310

Why mutable is bad

- The worst scenario takes place when some structure is used in separate parts of the application, then any mutation in one component can create a bug in the other one. Such bug is really hard to track because the source of the problem lies out of the scope of the failure. This is an example of a bad side effect.

- TODO: nieco o moduloch ze su singeltony a ze su kesovane a ked ho pouziju dvaja a niekto ho prekonfiguruje aj tvj zacne blbnut, to je asi nahorsia ukazka celej ten veci co poznam

functional-programming-jargon

- https://github.com/hemanth/functional-programming-jargon