Functional JavaScript

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

RECAP: 02.02-functions.pptx

Function is first-class object

Všetko čo viete spraviť s objektom, viete spraviť aj z funkciou.
Všade kde viete použiť objekt viete použiť aj funkciu

object

```
// passed as arguments to functions
calc({});

function calc(o) {
    return o.data + 10;
}


// returned as values from functions
function returnObject() {
    return {};
}
```

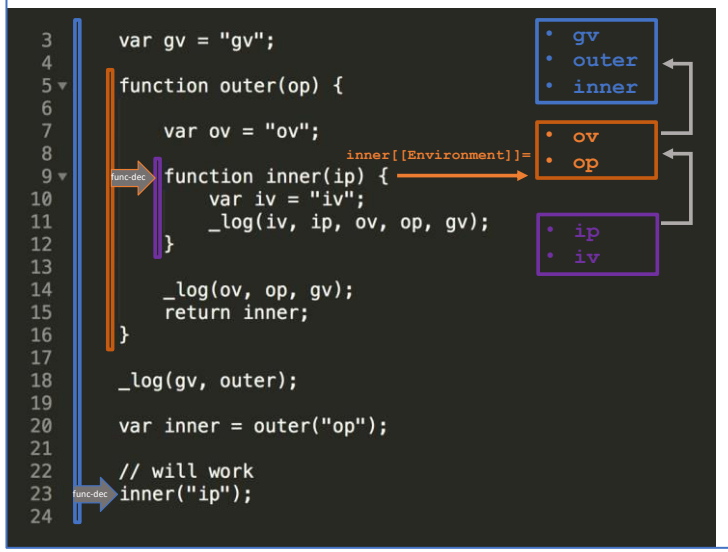
function

```
// passed as arguments to functions
calc(function() { /*...*/ });

function calc(f) {
    return f();
}


// returned as values from functions
function returnFn() {
    return function() {};
}
```

Closure



- Position of function definition and the position from where the function is executed, are generally not related
- Function defined in one scope can be called in another scopes
- Function remembers all *definition scope variables* in closer

Array iteration and processing functions

more readable array (list) iterations and transformations

Function	In	Out	loop eq.	ES
map	[], N	[], N	var [], for, push, return []	
filter	[], N	[], M<N	var [], for, if, push, return []	
reduce	[]	{}, [], whatever,…	var [], for, if, push, return {}	
reduceRight	[]	{}, [], whatever,…	var [], for (i--), if, push, return {}	
some	[]	boolean	var b, for, if return true	
every	[]	boolean	var b, for, if return false	
forEach	[]		for	
find	[] of items	item	for, if return a[i]; return;	
fill	[], item	[] of items	var [], for, push, return []	
from	[], iterable	[]	var [], for, push, return []	

JavaScript – "ugly" "for"

```
var insuredSubject;
for (var i = 0; i < insuredSubjects.length; i++) {
    var subject = insuredSubjects[i];
    if (subject._type === insuredSubject_type) {
        insuredSubject = subject;
        break;
    }
}
```

- for, while, do while
- they exist from Basic … Java
- Bad:
  - Verbose
  - Not semantic
  - Not reusable
- Good:
  - …

- keď sa pozriem na FOR cyklus neviem čo robí, lebo
  - robiť hocičo
  - veľa vecí naraz
- cyclomatic complexity
  - for, if,. for, while all nested

JavaScript - "ugly" "for"

How it is done

```
var insuredSubject;
for (var i = 0; i < insuredSubjects.length; i++) {
    var subject = insuredSubjects[i];
    if (subject._type === insuredSubject_type) {
        insuredSubject = subject;
        break;
    }
}
```

```
// Before:
// 4 vars (1 real data, 1 scoped condition, 2 help vars),
// 8 lines
// Vocabulary: 10, insuredSubject,insuredSubjects, for,
//      subject, length, i, _type, equals,
//      insuredSubject_type, break
```

What it does

```
// var desiredType = '....'; //better name ?
var insuredSubject = insuredSubjects.find(({ _type }) => _type === desiredType);
```

```
// After:
// 1 var (real data, 1 scoped condition, shell be const),
// 1 line
// Vocabulary: 5,  insuredSubject, insuredSubjects,
//                 find, type, equals, (_type)
```

JavaScript - **array extras** vs ugly “for”

mapping variants of for, to semantic methods

Function	In	Out	loop eq.	ES
map	[], N	[], N	var [], for, push, return []	
filter	[], N	[], M<N	var [], for, if, push, return []	
reduce	[]	{}, [], whatever,…	var [], for, if, push, return {}	
reduceRight	[]	{}, [], whatever,…	var [], for (i--), if, push, return {}	
some	[]	boolean	var b, for, if return true	
every	[]	boolean	var b, for, if return false	
forEach	[]		for	
find	[] of items	item	for, if return a[i]; return;	
fill	[], item	[] of items	var [], for, push, return []	
from	[], iterable	[]	var [], for, push, return []	

filter vs. for transformujeme [] N – [] M

TASK:
Nájdite študentov
čo majú nejakú 5ku

/samples/01-iteration-functions/03-students-functional.jsa
/samples/01-iteration-functions/03-students-structural.js

03-students-functional.js

```
1
2
3  var students = [{
4          name: "Marcus",
5          grades: [1, 2, 2, 5]
6      }, {
7          name: "John",
8          grades: [3, 2, 1, 1, 1, 1]
9      },
10     {
11         name: "Emilia",
12         grades: [5, 4]
13     }
14 ];
15
16 // Task: find failing student
17 // [] N -> [] M<N means filter
18 const failing = (students) =>
19     students.filter(({ grades }) =>
20         grades.some(grade => grade === 5)
21     )
22
23
24
25
26
27
28
29
30
```

03-students-structural.js

```
1
2
3  var students = [{
4          name: "Marcus",
5          grades: [1, 2, 2, 5]
6      }, {
7          name: "John",
8          grades: [3, 2, 1, 1, 1, 1]
9      },
10     {
11         name: "Emilia",
12         grades: [5, 4]
13     }
14 ];
15
16 // Task: find failing student
17 // var [], for-for-push, []
18 const failing = (students) => {
19     var failingStudents = [];
20     for (var i = 0; i < students.length; i++) {
21         var grades = students[i].grades;
22         for (var j = 0; j < grades.length; j++) {
23             if (grades[j] === 5) {
24                 failingStudents.push(students[i]);
25             }
26         }
27     }
28     return failingStudents;
29 }
30
```

JavaScript - Array Extras vs ugly “for”

- implementing changes in functional vs. structured

03-students-functional.js

```
29
30  // Changed requirement:
31  // hey, sorry I just want the names, not
32  // full data
33  // [] of full -> [] of Strings -> map, add code
34
35  const failingNames = (students) =>
36      students.filter(student =>
37          student.grades.some(grade => grade === 5)
38      )
39      // CHR: 2018_123
40      .map(({ name }) => name);
41
42
43
44
45
46
47
48
```

- it can be addition

03-students-structural.js

```
29
30  // Changed requirement:
31  // hey, sorry I just want the names, not
32  // full data
33  // for, change code
34
35  const failingNames = (students) => {
36      var failingStudents = [];
37      for (var i = 0; i < students.length; i++) {
38          var grades = students[i].grades;
39          for (var j = 0; j < grades.length; j++) {
40              if (grades[j] === 5) {
41                  //failingStudents.push(students[i]);
42                  // CHR: 2018_123
43                  failingStudents.push(students[i].name);
44              }
45          }
46      }
47      return failingStudents;
48  }
```

- it is mostly change

/samples/01-iteration-functions/03-students-functional.js a
/samples/01-iteration-functions/03-students-structural.js

reduce vs. for

03-students-functional.js

```
52
53  // Task: average grades for every student
54  // [] of students -> [] of numbers (means map)
55  // [] numbers 2 one number (means reduce)
56  const averageGrades = (students) =>
57      students.map(({ grades }) =>
58          grades.reduce((x, y) => x + y) / grades.length
59      )
60
61
62
63
64
65
66
67
68
```

03-students-structural.js

```
52
53  // Task: average grades for every student
54  //
55  //
56  const averageGrades = (students) => {
57      const sum = (arr) => {
58          var sum = 0;
59          for (var i = 0; i < arr.length; sum += arr[i++]);
60          return sum;
61      }
62      var ag = [];
63      for (var i = 0; i < students.length; i++) {
64          var grades = students[i].grades;
65          ag[i] = sum(grades) / grades.length;
66      }
67      return ag;
68  }
```

/samples/01-iteration-functions/03-students-functional.js a
/samples/01-iteration-functions/03-students-structural.js

JavaScript - best of both worlds chaining of array functions

04-chaining-vs-args-oo.js

```
1  // JavaScript (OO, chaining)
2
3  // oneliner
4  var output = input.map(f1).map(f2).map(f3);
5
6
7  // custom formatted
8  var output = input
9      .map(f1)
10     .map(f2)
11     .map(f3);
```

04-chaining-vs-args.js

```
1  // Other language (functions only)
2
3  // oneliner
4  var output = map(map(map(input, f1), f2), f3);
5
6
7  // custom formatted
8  var output = map(
9      map(
10         map(
11             input,
12             f1
13         ),
14         f2
15     ),
16     f3
17 );
```

find vs. for

- [] -> item
- Common usecases
  - Hladame itemy z nejakou property
  - minima, maximama
  - …

```
// Before:
// 4 vars (1 real data, 1 scoped condition, 2 help vars),
// 8 lines
// Vocabulary: 10, insuredSubject,insuredSubjects, for,
//      subject, length, i, _type, equals,
//      insuredSubject_type, break

// var insuredSubject_type = '....';
var insuredSubject;
for (var i = 0; i < insuredSubjects.length; i++) {
    var subject = insuredSubjects[i];
    if (subject._type === insuredSubject_type) {
        insuredSubject = subject;
        break;
    }
}
```

```
// After:
// 1 var (real data, 1 scoped condition, shell be const),
// 1 line
// Vocabulary: 5,  insuredSubject, insuredSubjects,
//                 find, type, equals, (_type)

// var desiredType = '....'; //better name ?
var insuredSubject = insuredSubjects.find(
  ({ _type }) => _type === desiredType
);
```

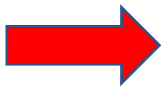
/samples/01-iteration-functions/05-find-vs-for.js

Ako funguje reduce, reduceRight

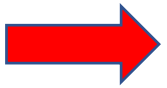
- The **reduce()** method executes a **reducer** function (that you provide) on each member of the array resulting in a single output value. but .... read more ...

```
// F) 2 items array
// returns last callback value

["a", "b"].reduce((a, c, i, arr) => {
  // callback called 2 times
  // init a 0 [ 'a', 'b' ]
  // ...    b 1 [ 'a', 'b' ]
}, "init");
```



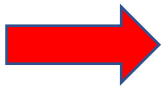
```
// A) empty array and no initial value
// TypeError: Reduce of empty array
// with no initial value ...
[].reduce((a, c, i, arr) => {
    console.log(a, c, i, arr)
});
```



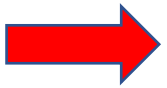
```
// B) empty array, and init value
// returns init value
[].reduce((a, c, i, arr) => {
    // callback is never called
},"init");
```

```
// C) 1 item array, and no initial value
// returns arr[0] the original item

["a"].reduce((a, c, i, arr) => {
  // callback is never called
});
```



```
// D) 1 item array, and initial value
// returns arr[0] the original item
["a"].reduce((a, c, i, arr) => {
  // callback called one time
  // 'init' 'a' 0 [ 'a' ]
}, "init");
```



```
// E) 2 item array
// returns callback ret val
["a", "b"].reduce((a, c, i, arr) => {
  // callback called one time
  // a b 1 [ 'a', 'b' ]
});
```

Reduce – načo sa používa

- [] -> anything
- Transform [] to number
  - sum, avg,...
- Transform [] to {}
  - To restructure things
  - To create groups
- Transform [] to Map
  - To create groups/counts
- Transform [] to Set
  - To create unique items
- Transform [] to []
  - „optimized“ filter/map operations
  - .
- Transform [] to several [] [] [] arrays
  - Split data to groups

- samples/01-iteration-functions/06a-reduce-number.js
- samples/01-iteration-functions/06b-reduce-object.js
- samples/01-iteration-functions/06c-reduce-map.js
- samples/01-iteration-functions/06d-reduce-set.js
- samples/01-iteration-functions/06e-reduce-array.js
- samples/01-iteration-functions/06f-reduce-arrays.js
- ....
- TODO: vecsina je nedokoncenych, mate to ako samostudium skusit si to napisat, pohladat priklady na github a na nete atd....

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/reduce

Reduce – common usecases

- **Transform [] to Map**
  - **To create groups/counts**

- **Pozrite si zdrojáky a komentáre v 08-prednaska/samples/01-iteration-functions/06c-reduce-map.js**

```
function v1() {

  // nie o moc lepsie ako for
  const classRooms = students.reduce((classRooms, student) => {
    // v reduce malokedy pouzivam semanticke nazvy
    const { classRoom } = student;
    let studentsInRoom;
    if (!(studentsInRoom = classRooms.get(classRoom))) {
      studentsInRoom = [];
      classRooms.set(classRoom, studentsInRoom);
    }
    studentsInRoom.push(student);
    return classRooms;
  }, new Map())
  // vysledok nie je array, ale map
  console.log([...classRooms]);

}


function v2() {
  // tento kod neviem stale precitat
  const classRooms = students.reduce((r, o) => {
    // ale tento reduce uz viem precitat
    const k = o.classRoom;
    let vals = r.get(k);
    vals && vals.push(o) || r.set(k, [o]);
    return r;
  }, new Map())

  // vysledok stale nie je array, ale map
  console.log([...classRooms]);
}
```

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/reduce

Reduce – common usecases

- **Transform [] to Map**
  - **To create groups/counts**
- Toto je V3, taká nejaká varianta v strede toho ako by sa to dalo priemerne napísať

```
function v3() {

  // toto uz precitam rychlo (ale nepaci sa mi tam...
  const classRooms = students //..ten new Map()
    .reduce(groupByClassRoom, new Map())

  // a len ked chcem lustim toto
  function groupByClassRoom(r, o) {
    // ale tento reduce uz viem precitat
    const k = o.classRoom; //toto je jedina ad hoc vec
    let vals = r.get(k);
    vals && vals.push(o) || r.set(k, [o]);
    return r;
  };

  // vysledok stale nie je array, ale map
  console.log([...classRooms]);

}
```

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/reduce

Reduce – common usecases

- **Transform [] to Map**
  - **To create groups/counts**
- V4 extrémna varianta ako to teraz píšem ja v poslednej dobe
- A ešte si ukážeme nabudúce, ako to zuniverzálniť, keď propName nebude string ale funkcia

```
function v4() {

  // zabijem 3 mochy jednou ranou (toto si rozpiste na 3 kroky)
  // a) mam chainable reduce
  // b) mam reduce v chaine bez uvadzania init value
  // c) mam univerzalny gropBy podla akelkovek property
  // d) mam garantovanu konverziu na [] nakonci

  // otazka znie co ne chaining za to stoji
  // miesto kodovania groupBy(arr,prop)

  const classRooms = students
    .reduce(...groupBy("classRoom")) //b)
    // a)
    .map(([classRoom, {length}]) => ({ classRoom, length }))

  function groupBy(propName) {
    // returns pair of reducer and init value
    // to be used as reduce(...myReducer)
    // c)
    const reducer = (r, o, i, { length }) => {
      if (i == 0) r = new Map(); //d
      const k = o[propName];
      let vals = r.get(k);
      vals && vals.push(o) || r.set(k, [o]);
      return i == length - 1 ? [...r] : r; //d
    };
    const init = []; //d
    return [reducer, init];
  }
  // vysledok array
  console.log(classRooms);
}
```

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/reduce

forEach is almost evil as for loop

```
/**
 * Get an object with all currently loaded rules
 * @returns {Map} All loaded rules
 */
getAllLoadedRules() {
    const allRules = new Map();

    Object.keys(this._rules).forEach(name => {
        const rule = this.get(name);

        allRules.set(name, rule);
    });
    return allRules;
}
```

- What is correct functional equivalent of this ? Reduce
- Why they did not write it as reduce ?
  - Because another OO evil => ***this***
  - and because array.reduce has no thisArg

sample: code is from ESLint source code

array extras vs. for loop iterating arrays

- máme sparse arrays
- a su dva sposoby iterovania nad array, podla pouziteho API
  - over **index** from 0 to length-1 (vsecky itemy)
    - ostatne
  - over **defined** array keys (iba definovane itemy)
    - Array.prototype.forEach()
    - Array.prototype.indexOf()
    - Array.prototype.lastIndexOf()
    - Array.prototype.filter()
    - Array.prototype.map()* – but returns length
    - Array.prototype.every()
    - Array.prototype.some()
    - Array.prototype.reduce()
    - Array.prototype.reduceRight()

Excel je tu: /02-prednasky/js-api-doc.xlsx
https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array

A	B	C	D	E
	API	**Spec**	**Mut**	**sparse**
	Array.length	1		index
	Array.name			
	Array.prototype	1		
Construct	Array.prototype.constructor()			index
	Array.of(element0[, element1[, ...[, elementN]]])	6		index
	Array.from(arrayLike[, mapFn[, thisArg]])	6		index*
	Array.fill(value[, start[, end]])	6	y	index
Types	Array[@@species]	6		
	Array.isArray()	5.1		
Iterate	Array.prototype.length	1	y	index
	Array.prototype.keys()	6		index
	Array.prototype.entries()	6		index
	Array.prototype.forEach()	5.1		in
	Array.prototype[@@iterator]	6		index
	Array.prototype[@@unscopables]	6		
modify add	Array.prototype.concat()	3	n	index
	Array.prototype.push()	3	y	index
	Array.prototype.unshift()	3	y	index
modify remove	Array.prototype.pop()	3	y	index
	Array.prototype.shift()	3	y	index
modify	Array.prototype.splice()	3	y	index
	Array.prototype.copyWithin()	6	y	index
search	Array.prototype.indexOf()	5.1		in
	Array.prototype.lastIndexOf()	5.1		in
	Array.prototype.find()	6		index
	Array.prototype.findIndex()	6		index
	Array.prototype.includes()	7		index
	Array.prototype.every()	5.1		in
	Array.prototype.some()	5.1		in
extract	Array.prototype.filter()	5.1		in
	Array.prototype.slice()	3		index
Transform	Array.prototype.map()	5.1		in
	Array.prototype.reduce()	5.1		in
	Array.prototype.reduceRight()	5.1		in
	Array.prototype.reverse()	1	y	index
	Array.prototype.sort()	1	y	index
	Array.prototype.join()	1		index
	Array.prototype.toString()	1		index
	Array.prototype.toLocaleString()	8		index

array extras vs. for loop

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

let arr = [0, 1];
arr[99] = 99;

//[ 0, 1, <97 empty items>, 99 ]

// .length and for loop
let i;
for (i = 0; i < arr.length; i++) {
  // 100 times
}
assert(i === 100);

// @@iterator, and arr.values()
// values for each index in the array.
var k = 0;
for (let v of arr.values()) {
  k++;
}
assert(k === 100);
var k = 0;
for (let v of arr.keys()) {
  k++;
}
// array.from(arrayLike with length and index)
assert(Array.from(arr).length === 100);
assert(arr.concat(arr).length === 200);
assert(arr.findIndex((a) => a === undefined) === 2);
assert(arr.includes(undefined) === true);
```

```
var j = 0
arr.forEach((a, i, arr) => {
  // 3 times;
  j++;
});
assert(j === 3);

assert(arr[50] === undefined);
var hasUndef = arr.some((a) => a === undefined);
assert(hasUndef === false);

var defined = arr.filter(() => true);
assert(defined.length === 3);

var m = 0;
var mapped = arr.map((i) => { m++; return i; });
assert(mapped.length, 100);
assert(m === 3);

assert(arr.indexOf(undefined) === -1);
assert(arr.lastIndexOf(undefined) === -1);
```

Syntax

- 1. param - callback je funkcia
  - Existujúca JS/API funkcia
  - Reusnuta funkcia
  - Inline arrow/function
- 2. param – thisArgs, len u niektorych funkcii
  - Možnosť použiť metódu ako callback
  - Možnosť poslať dodatočný „parameter“ do bežnej funkcie (faster then scope lookup)

samples/01-iteration-functions/00-syntax.js

00-syntax.js

```
/*
plna syntax ma 3 parametre callbacku
a u niektorych metod aj thisArg ako 2 param metody

items.map(function(item, i, items) { ... }, thisArg);

*/

// vecsinou potrebujete len item
// takze caste su skratene formy
// len z prvym parametrom

// potrebujem v vypocte len item
[1, 2, 3].map((item) => item * 10);

// potrebujem len nejake properties z itemu
[1, 2, 3].map(({ name, age }) => `${name} is ${age}`);

// niekedy potrebujete aj index i
[1, 2, 3].map((item, i) => item / i);

//niekedy cely array
[1, 2, 3].map((n, i, ns) => n / ns.length);
[1, 2, 3].map((n, i, {length}) => n / length);

// niekedy potrebujete ako callback
// pouzit nie funkciu
// ale metodu z ineho objektu
let limiter = {
  _lim: 1.5,
  lim(n) { return Math.min(this._lim, n) }
};
// potrebujete si poslat this, lebo ...
[1, 2, 3].map(limiter.lim, limiter);

// samozrejme sa snazite v 90% pripadov poslat tam
// pomenovanu/reusnutu/existujucu funkciu
```

Syntax - often unused

- **thisArg,** map, filter, some, every – signature ma 2 param
  - benefits: using methods as callbacks
  - passing extra value (avoiding scope lookup)
- **3rd param of callback**, the array itself

01a-syntax-better.js

```
/*
plna syntax ma 3 parametre callbacku
treba ich pouzivat, a nie zbytocne lookupovat
z upper scope
*/

const items = [1, 2, 3];

var avgs = items.map(n => n / items.length);

var avgs = items.map((n, i, items) => n / items.length);

var avgs = items.map((n, _, { length }) => n / length);

/*
plna syntax funkcie ma 2 parametre,
druhy je thisArg,
treba ho pouzivat, a nie zbytocne lookupovat
z upper scope
(samozrejme nemozete uz pouzit arrow)

*/
const obj = { a: 1, b: 2, c: 3 };

Object.keys(obj).forEach(k => obj[k] = obj[k] * 10);

Object.keys(obj).forEach(function(k) {
  this[k] = this[k] * 10
}, obj);
```

samples/01-iteration-functions/01a-syntax-better.js

semantic code
”real world” samples

https://www.codewars.com/kata/**equal-sides-of-an-array**/train/javascript

You are going to be given an array of integers. Your job is to take that array and find an index N where the sum of the integers to the left of N is equal to the sum of the integers to the right of N. If there is no index that would make this happen, return `-1`.

rahul

```
const sum = (a, from, to) => a.slice(from, to).reduce((a, b) => a + b, 0)
const findEvenIndex = a => a.findIndex((el, i) => sum(a, 0, i) === sum(a, i + 1));
```

```
function findEvenIndex(arr)
{
    // since arrays are not needed on output
    // leaving only sums
    const sum=(a)=>a.reduce((a,b)=>a+b,0);
    let a1sum=0;
    let a2sum=sum(arr);
    for(i=0;i<arr.length;i++){
      a1sum+=arr[i-1]||0;
      a2sum-=arr[i]||0;
      if(a1sum===a2sum) return i;
    }
    return -1;
}
```

ainthek, perf. oriented solution

TODO performance: spočítajte zložitosť algoritmov

Programovanie z iterative functions (array extras)

- Program pozostáva zo sekvencie transformácii
  A. Buď preto lebo z povahy zadania je to ozaj sekvencia transformácii
  B. Alebo preto lebo sa to tak lahsie (postupne) kóduje
  C. Alebo preto lebo sa nám to pri zmenách bude ľahšie preskladávať

```
const people = [];

people
  .map(localDatesToUTC)
  .map(calculateAge)
  .reduce(...splitByAge(18))
  .map(([kids, adults]) => [
    kids.find(oldest),
    adults.find(youngest)
  ])
  .flat()
  .map(assignStartNumbers)
  .forEach(startRun)


// this kind of code "can be" much easier to
// - explain verify with customer/boss/team
// - to reason about
// - to restructure

// avoid too deep nesting
// avoid too complicated lambdas
// can have some simple lambdas
```

- Ak si spomenie na async prednášku aj tam sme sa snažili program sekvencie/paralelizmy zapísať podobne deklaratívne
- A ešte sa s tým stretneme u streamov

Objects as data

- Všetko co sme si ukazovali funguje na []
- Na {} nemáme takéto extras, ale ak chceme dokážeme object properties skonvertovat na [] of entries a nazad [] of entries na {}
- Z prednášky o objektoch by ste mali poznať aspoň 3 metódy na to prvé

A) Object.????() -> []

B) Object.fromEntries([]) -> {} (draft 2019)

function composition

composition - definitions

- In computer science, **function composition is an act or mechanism to combine simple functions to build more complicated ones.**
- Like the usual composition of functions in mathematics, the result of each function is passed as the argument of the next, and the result of the last one is the result of the whole.
- In mathematics, function composition is the pointwise application of one function to the result of another **to produce a third function**
- **composition** of functions deals with **finite data** and **sequential execution**
- when speaking about **infinite data** (e.g. **streams**) and possibly **parallel** processing we call it **pipeline** (don't confuse with pipe() on other slides)

```
const compose = (f1, f2) => value => f1( f2(value) );

(f1, f2, f3, f4...) => value => f1( f2(f3(f4(value) )));
```

ES2015 ArrowFunctionExpression:

```
() => () => result;
```

```
function(){return function(){return result;};}
```

https://medium.com/@dtipson/creating-an-es6ish-compose-in-javascript-ac580b95104a

Composition styles inline, adhoc, generic

```
// 1 - 3 * O(n)
let out1 = input.map(f1).map(f2).map(f3);

// 2 - inline, anonymous, O(n)
let out2 = input.map((item) => f3(f2(f1(item))));

// 3 - named, O(n), do "not use made up names"
const f123 = (item) => f3(f2(f1(item)));
let out3 = input.map(f123);

// 4 - "generic" compose
let out4 = input.map(compose(f3, f2, f1));

// 5 - override of array.map syntax
let out5 = input.map([f3, f2, f1]);
```

samples/02-function-composition/01-composition-styles.js

1. readable
reused,
slow 400ms

2. readable,
reused
fast 120ms

3. readable,
reused
fast 120ms

4. readable,
reused
fast 153ms
slow 300ms *

5. readable
reused,
non standard

Composition styles – performance view

- Môžete sa pohrať sami, naivný test (nie ozajstný banchmarking) ale na predstavu to postačuje
- Stále sa bavíme o tom, že všetko je kompromis medzi čitateľnosťou, pisateľnosťou, performance, bezpečnosť, flexibilita, robusnosť....

```
$ npx mocha 01b-composition-styles-perf.spec.js


  function compositions
    ✓ map(f1).map(f2).map(f3) (287ms)
    ✓ ad-hoc inline - map(()=f3(f2(f1()))) (100ms)
    ✓ ad-hoc extracted - map(f123)) (99ms)
    ✓ generic compose (POC) - map(compose([f1,f2,f3])) (98ms)
    ✓ generic compose functional - map(compose(f1,f2,f3)) (123ms)
    ✓ generic compose (old good for cycle) - map(df.compose(f1,f2,f3)) (113ms)


  6 passing (831ms)
```

2019-javascript/08-prednaska/samples/02-function-composition/01b-composition-styles-perf.spec.js

Composition – compose and pipe

function composition is an act or mechanism to combine simple functions to build more complicated ones.

2 general compositions:

- **compose** – left-to-right
- **pipe** – right-to-left

compose.js

```
1  // compose - left-to-right
2  const a = (v) => `a(${v})`
3  const b = (v) => `b(${v})`
4  const c = (v) => `c(${v})`
5
6  // var horror
7  var rc = c("test");
8  var rb = b(rc);
9  var r0 = a(rb);
10
11 // not a composition
12 // does not return function but value
13 var r0 = a(b(c("test")));
14
15 // manual composition
16 // returns function
17 const abc1 = (v) => a(b(c(v)));
18 let r1 = abc1("test");
19
20 // generic composition
21 const compose = (...fns) =>
22   x => fns.reduceRight((v, f) => f(v), x);
23
24 const abc2 = compose(a, b, c);
25 let r2 = abc1("test");
26
27
28
```

```
a(b(c(test)))
a(b(c(test)))
a(b(c(test)))
```

pipe.js

```
1  // pipe - right-to-left
2  const a = (v) => `a(${v})`
3  const b = (v) => `b(${v})`
4  const c = (v) => `c(${v})`
5
6  // var horror
7  var ra = a("test");
8  var rb = b(ra);
9  var r0 = c(rb);
10
11 // not a composition
12 // does not return function but value
13 var r0 = c(b(a("test")));
14
15 // manual composition
16 // returns function
17 const cba1 = (v) => c(b(a(v)));
18 let r1 = cba1("test");
19
20 // generic composition
21 const pipe = (...fns) =>
22   x => fns.reduce((v, f) => f(v), x);
23
24 const cba2 = pipe(a, b, c);
25 let r2 = cba2("test");
26
27
28
```

```
c(b(a(test)))
c(b(a(test)))
c(b(a(test)))
```

samples/02-function-composition/02a-compose.js
samples/02-function-composition/02b-pipe.js

other composition

- compose, pipe – well known, well named
- and, or, ….

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

- TODO: find libraries having other ad hoc compositions, study and compare codes

Why to use composition

- **Instead of jamming** functions within functions **or creating a bunch of intermediate variables**, let's pipe all the things!
- Those who prefer functional programming would tell you that the **compose is more declarative**. Instead of telling you *how* the function works, it tells you *what* it does!
- Functional composing **can be a lot of fun** as seen in the previous examples plus leads to more elegant and reusable code if applied correctly.

- **clog:** logs and returns a value
- **path:** retrieves a value from an object
- **parseJSON:** parses JSON
- **httpGet:** makes an HTTP GET request for the provided URL and returns the response object (see note)

```
const url = 'https://api.icndb.com/jokes/random';

compose(
  clog,
  path(['value', 'joke']),
  parseJSON,
  httpGet
)(url);

// => "There are no steroids in baseball. Just players Chuck Norris
has breathed on."
```

https://medium.com/@wpcarro/stop-writing-for-loops-compose-7d4ac507bd36

Functional vs. OO, do we have to choose ?

- Keep it sane

- **function is basic (JS) construct**
- functions to methods
- methods to functions

09-graph-functional.js

```
// from book [1], Chapter 6: Recursion
var influences = [
    ['Lisp', 'Smalltalk'],
    ['Lisp', 'Scheme'],
    ['Smalltalk', 'Self'],
    ['Scheme', 'JavaScript'],
    ['Scheme', 'Lua'],
    ['Self', 'Lua'],
    ['Self', 'JavaScript']
];
const first = ([first, ...rest]) => first;
const rest = ([first, ...rest]) => rest;
const construct = (head, tail) => [head].concat(tail);
const second = ([first, second, ...rest]) => second;
const isEmpty = (arr) => arr.length === 0;
const isEqual = (a, b) => a == b;
const cat = (a, b) => a.concat(b);
const contains = (arr, a) => ~arr.indexOf(a);
const rev = (arr) => [].concat(arr).sort();

function nexts(graph, node) {
    //console.log(arguments);
    if (isEmpty(graph)) return [];
    var pair = first(graph);
    var from = first(pair);
    var to = second(pair);
    var more = rest(graph);
    if (isEqual(node, from))
        return construct(to, nexts(more, node));
    else
        return nexts(more, node);
}
function depthSearch(graph, nodes, seen) {
    if (isEmpty(nodes)) return rev(seen);
    var node = first(nodes);
    var more = rest(nodes);
    if (contains(seen, node))
        return depthSearch(graph, more, seen);
    else
        return depthSearch(
            graph,
            cat(nexts(graph, node), more),
            construct(node, seen)
        );
};
console.log(
    depthSearch(influences, ['Smalltalk'], [])
);
```

09-graph-js.js

```
// How I would implement it (probably) as 'functional JS'
var influences = [
    ['Lisp', 'Smalltalk'],
    ['Lisp', 'Scheme'],
    ['Smalltalk', 'Self'],
    ['Scheme', 'JavaScript'],
    ['Scheme', 'Lua'],
    ['Self', 'Lua'],
    ['Self', 'JavaScript']
];



const nexts = (graph, node) => graph
    .filter(([from, to])=> from === node)
    .map(([from, to])=> to)



function depthSearch(graph, nodes, seen = []) {
    if (!nodes.length) return [].concat(seen).reverse();
    var [node, ...more] = nodes;
    return ~seen.indexOf(node)
        ? depthSearch(graph, more, seen)
        : depthSearch(
            graph,
            nexts(graph, node).concat(more),
            seen.concat(node)
        );
};


console.log(
    depthSearch(influences, ['Smalltalk'])
);
```

JavaScript - too much functional ?
Alebo čo sa stane, keď to “preženiete”.

- is this still JavaScript ?

how many JS people

  - will have to read it,
  - and will understand ?

- Do you want to study
  - custom APIs (vocabularies) or use
  - “idiomatic JS” (for common oneliners) ?
- is recursion really “best” for this structure parsing ?

- is functional really best for this ? (internal data structure hiding)
  - which functions are really reusable ?

JavaScript - from functions to object+methods

09-graph-js.js

```
const nexts = (graph, node) => graph
    .filter(([from, to]) => from === node)
    .map(([from, to]) => to)

function depthSearch(graph, nodes, seen = []) {
    if (!nodes.length) return [].concat(seen).reverse();
    var [node, ...more] = nodes;
    return ~seen.indexOf(node) ?
        depthSearch(graph, more, seen) :
        depthSearch(
            graph,
            nexts(graph, node).concat(more),
            seen.concat(node)
        );
};
```

```
depthSearch(influences, ['Smalltalk']);
```

09-graph-oo.js

```
const nexts = (graph, node) => graph
    .filter(([from, to]) => from === node)
    .map(([from, to]) => to)

function depthSearch(graph, nodes, seen = []) {
    if (!nodes.length) return [].concat(seen).reverse();
    var [node, ...more] = nodes;
    return ~seen.indexOf(node) ?
        depthSearch(graph, more, seen) :
        depthSearch(
            graph,
            nexts(graph, node).concat(more),
            seen.concat(node)
        );
};
// one of possible implementations of object
// the oldest most traditional one)
// how to quickly change functions to OO.methods
function ArrayGraph(graph) {
    this.graph = graph;
}
ArrayGraph.prototype.depthSearch = function(nodes) {
    return depthSearch(this.graph, nodes, []);
}



var graph = new ArrayGraph(influences);
graph.depthSearch(['Smalltalk']);
```

JavaScript - from methods to functions

10-methods.js

```
1
2  var arr = [1, 2, 3];
3  const transform = (item, i, items) =>
4      item * i * items.length
5
6  // Natural JS style, OO, METHODS
7  arr.map(transform)
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
```

10-functions.js

```
1
2  var arr = [1, 2, 3];
3  const transform = (item, i, items) =>
4      item * i * items.length
5
6  // but you want FUNCTIONAL SYNTAX
7  //map(arr, transform)
8
9
10 // actually JS provides "all methods as functions"
11 // magic: object.method + function.call(this, arg1,arg2)
12 Array.prototype.map.call(arr, transform)
13
14 // 'Array.prototype.map.call' === your wanted 'map'
15 // so now only "alias" the methods
16
17 const map = (arr, ...args) =>
18     Array.prototype.map.call(arr, ...args)
19 const filter = (arr, ...args) =>
20     Array.prototype.filter.call(arr, ...args)
21
22 // and here we go, functional style
23 map(arr, transform)
24 filter(arr, x => x < 2)
```

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, 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
```

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

Functional vs. OO, do we have to choose ?

- Funkcia je základ
  - Nie len v JS
- Ak máte rozumné funkcie vždy ich dokážete
  - Prilepiť na objekt
  - Konvertnúť z metód na funkcie
  - Wrapnúť
  - Bindnuť im parametre
  - Zmeniť this....
  - ....
- Snaha o pure/impure podľa toho čo kódujete
- Základ je nepísať 10ky riadkov štrukturovaného balastu, ale pekne kódovať funkciami

Zvyšok na budúce.....