VYBRANÉ SÚČASTI JAVASCRIPTU

Peter Gurský

premenné a konštanty

- var
  - premenná viditeľná v celom module, aj keď je definovaná vo vnútri funkcie alebo cyklu
  - nepoužívať, ak to nie je nutné
- let + const
  - premenná/konštanta existuje iba v bloku, v ktorom bola vytvorená – tj. po najbližšie { a }
  - odporúča sa preferovať const pred let, JS funguje rýchlejšie
  - použitie pre samostatné premenné a konštanty
  - inštančným premenným sa let ani const nepíše
- readonly
  - modifikátor pre inštančné premenné tried (property)
  - zabraňuje viacnásobnému priradeniu hodnoty iba v čase kompilácie napr. Typescriptu – upozorní v IDE programátora
    - po preklade do JS už kontrolu nerobí na rozdiel od const

template strings

- výraz medzi spätnými apostrofmi ` `, ktorý je alternatívou konkatenácie
- môže byť aj viacriadkový
- ak chceme použiť nejakú premennú alebo ľubovoľný kód, ktorý vracia hodnotu, uzavrieme ju/ho do ${ }

```
const mail = ` Dobrý deň ${meno} ${priezvisko}

  Prosím o zaslanie ${1.2 * faktura.cenaBezDph} Eur do ${getDueDate(faktura)}

Ďakujem`;
```

truthy & falsy

- **falsy**: false, undefined, null, 0, "", NaN
- **truthy**: všetko, čo nie je falsy

- const myObj = otherObj || {myid:1};
  - ak otherObj je truthy, vráti otherObj
  - ak otherObj je falsy, výraz vráti objekt {myid:1};
- const val = "" && "Jano";
  - vráti "" lebo je falsy, takže výraz sa ďalej nevyhodnocuje

práca s atribútmi objektov

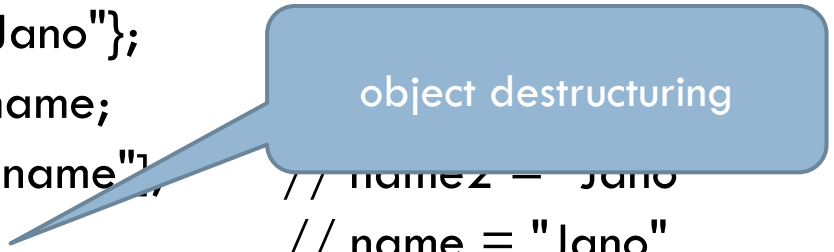
- const human = {name:"Jano"};
- const name1 = human.name; // name1 = "Jano"
- const name2 = human["name"]; // name2 = "Jano"
- const {name} = human; // name = "Jano"
- const {name: name3} = human; // name3 = "Jano"

- const reverseHuman = {[human.name]: "name"}; // {Jano: "name"}
- const reverseHuman2 = {[human["name"]]: "name"}; // {Jano: "name"}

- const human2 = {id: 5, name:"Jano"};
- const keys = Object.keys(human2); // ["id", "name"]
- const values = Object.values(human2); // [5, "Jano"]
- const entries = Object.entries(human2); // [["id", 5], ["name", "Jano"]]

práca s atribútmi objektov

- const human = {name:"Jano"};
- const name1 = human.name;
- const name2 = human["name"];
- const {name} = human;
- const {name: name3} = human;



// name2 = "Jano"
// name = "Jano"
// name3 = "Jano"

- const reverseHuman = {[human.name]: "name"}; // {Jano: "name"}
- const reverseHuman2 = {[human["name"]]: "name"}; // {Jano: "name"}

- const human2 = {id: 5, name:"Jano"};
- const keys = Object.keys(human2); // ["id", "name"]
- const values = Object.values(human2); // [5, "Jano"]
- const entries = Object.entries(human2); // [["id", 5], ["name", "Jano"]]

spread operátor

```
const myArray = [20, 30];
const myArray1 = [...myArray, 40];      // [20, 30, 40]
const myArray2 = [10, ...myArray1];   // [10, 20, 30, 40]
const [first, second, ...rest] = myArray2; // first = 10, second = 20, rest = [30, 40]
```

spread operátor

```
const myArray = [20, 30];
const myArray1 = [...myArray, 40];      // [20, 30, 40]
const myArray2 = [10, ...myArray1];   // [10, 20, 30, 40]
const [first, second, ...rest] = myArray2; // first = 10, second = 20, rest = [30, 40]
```

Starší prístup:

```
const myArray1 = myArray.concat([40]);
```

alebo:

```
const myArray1 = myArray.concat(40);
```

všeobecne:

newArray = *oldArray*.concat(*value1*, *value2*, ..., *valueN*);

kde valueX je prvok alebo pole

iný príklad:

```
const letters = ['a', 'b', 'c'];
const alphaNumeric = letters.concat(1, [2, 3]); // ['a', 'b', 'c', 1, 2, 3]
```

spread operátor

```
const myArray = [20, 30];
const myArray1 = [...myArray, 40];      // [20, 30, 40]
const myArray2 = [10, ...myArray1];   // [10, 20, 30, 40]
const [first, second, ...rest] = myArray2; // first = 10, second = 20, rest = [30, 40]

const myObj = { id: 1, name: "Jano" };
const myObj1 = { ...myObj, height: 180 }; // { id: 1, name: "Jano", height: 180};
const myObj2 = { ...myObj, id: 10 };          // { id: 10, name: "Jano"};
const myObj3 = { id: 1000, ...myObj };      // { id: 1, name: "Jano"};
const myObj4 = { ...myObj1, ...myObj2 } // { id: 10, name: "Jano", height: 180};
```

spread operátor

```
const myArray = [20, 30];
const myArray1 = [...myArray, 40];      // [20, 30, 40]
const myArray2 = [10, ...myArray1];   // [10, 20, 30, 40]
const [first, second, ...rest] = myArray2; // first = 10, second = 20, rest = [30, 40]

const myObj = { id: 1, name: "Jano" };
const myObj1 = { ...myObj, height: 180 }; // { id: 1, name: "Jano", height: 180};
const myObj2 = { ...myObj, id: 10 };          // { id: 10, name: "Jano"};
const myObj3 =
const myObj4 =                                                          ght: 180};
```

Starší prístup:

```
const myObj2 = Object.assign({}, myObj, {id:10});
```

všeobecne:

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

spread operátor

```
const myArray = [20, 30];
const myArray1 = [...myArray, 40];      // [20, 30, 40]
const myArray2 = [10, ...myArray1];   // [10, 20, 30, 40]
const [first, second, ...rest] = myArray2; // first = 10, second = 20, rest = [30, 40]

const myObj = { id: 1, name: "Jano" };
const myObj1 = { ...myObj, height: 180 }; // { id: 1, name: "Jano", height: 180};
const myObj2 = { ...myObj, id: 10 };          // { id: 10, name: "Jano"};
const myObj3 = { id: 1000, ...myObj };      // { id: 1, name: "Jano"};
const myObj4 = { ...myObj1, ...myObj2 } // { id: 10, name: "Jano", height: 180};

const myFunc = (par1, ...rest) => console.log(rest, par1);
myFunc(1,2,3,4);                  // vypíše [2, 3, 4] 1
myFunc(...myArray2);          // vypíše [20, 30, 40] 10

const letters = [ ..."Jano" ];          // [ "J", "a", "n", "o" ]
const weirdObj = { ..."Jano" };   // { 0: "J", 1: "a", 2: "n", 3: "o" }
const errArray = [ ...myObj ];     // TypeError: object is not iterable
```

práca s poľom

- const people = ["Jano", "Fero", "Jožo"];
- const [first] = people; // first = "Jano"
- const [„third] = people; // third = "Jožo“
- const [first2, …rest] = people; // first2 = "Jano", rest = ["Fero", "Jožo"]
- const [last] = […people].reverse();

funkcie

- funkcie v javascripte sú
  - uložiteľné do premenných a konštánt
  - môžu byť použité ako prvky poľa
  - môžu byť odoslané ako parametre funkcií
  - môžu byť návratovými hodnotami funkcií
- higher-order funkcie
  - pracujú s inými funkciami – prijímajú ich ako parametre a/alebo vrátia inú funkciu

funkcie a arrow funkcie

```
const fn = function meno(param1, param2) {
  return vysledok
}
```

```
const fn = (param1, param2) => {
  return vysledok
}
```

```
const fn = (param1, param2) => vysledok
```

```
const fn2 = function meno2(param1) {
  return vysledok
}
```

```
const fn2 = (param1) => {
  return vysledok
}
```

```
const fn2 =  (param1) => vysledok
```

```
const fn2 =  param1 => vysledok
```

arrow funkcie a this

```
const people = {
  group: ["Jano", "Fero", "Jožo"],
  print: function (delay=1000) {
    setTimeout( function() {
      console.log(this.group.join(", "));
    }, delay);
  }
}
people.print();
// cannot read property 'join' of undefined
```

this je ten objekt, ktorý bude červenú funkciu naozaj spúšťať,
teda ten, ktorý je považovaný za this v setTimeout funkcii,
teda objekt, ktorý obsahuje metódu setTimeout, čo je **window** objekt

```
const people = {
  group: ["Jano", "Fero", "Jožo"],
  print: function (delay=1000) {
    setTimeout( () => {
      console.log(this.group.join(", "));
    }, delay);
  }
}
people.print();
// Jano, Fero, Jožo
```

this je ten objekt, ktorý je považovaný za this aj o úroveň vyššie, teda vo funkcii print, čo je objekt referencovaný z konštanty people

arrow funkcie a this

```
const people = {
  group: ["Jano", "Fero", "Jožo"],
  print: function (delay=1000) {
    setTimeout( () => {
      console.log(this.group.join(", "));
    }, delay);
  }
}
people.print();
// Jano, Fero, Jožo
```

this je ten objekt, ktorý je považovaný za this aj o úroveň vyššie, teda vo funkcii print, čo je objekt referencovaný z konštanty people

```
const people = {
  group: ["Jano", "Fero", "Jožo"],
  print: (delay=1000) => {
    setTimeout( () => {
      console.log(this.group.join(", "));
    }, delay);
  }
}
people.print();
// cannot read property 'join' of undefined
```

this je ten objekt, ktorý je považovaný za this aj o úroveň vyššie, teda vo funkcii print,
this vo funkcii print je ten objekt, ktorý je považovaný za this aj o úroveň vyššie, čo je **window**

immutable (nemodifikujúce) funkcie

- ľubovoľná funkcia môže mať vstupné parametre, robiť s nimi čo chce a môže vrátiť výsledok
- immutable funkcia nemodifikuje obsah vstupných parametrov
  - ak chceme vo funkcii pracovať s modifikáciou vstupného objektu alebo modifikovaný objekt vrátiť, najprv si vo funkcii vytvoríme kópiu a až tú modifikujeme
  - ak takúto funkciu zavoláme, nemusíme sa báť, že sa zmení obsah našich premenných, ktoré sme funkcii poskytli

forEach

{id:1, name: "Jano"}	{id:2, name: "Fero"}	{id:3, name: "Jožo"}	{id:4, name: "Anna"}

```
[
 {
  id: 1,
  name:"Jano"
 },
 {
  id: 2,
  name:"Fero"
 },
 {
  id: 3,
  name:"Jožo"
 },
 {
  id: 4,
  name:"Anna"
 },
]
```

Procedurálny prístup:

```
for (let value of myArray) {
  console.log(value.name);
}
```

Funkcionálny prístup:

```
myArray.forEach(function(value) {
  console.log(value.name);
});
```

```
myArray.forEach(value => console.log(value.name));
```

Funkcionálny prístup = „Spúšťame funkciu pre každý prvok poľa.“

filter

{id:1, name: "Jano"}	{id:2, name: "Fero"}	{id:3, name: "Jožo"}	{id:4, name: "Anna"}

- výstup: nové pole s niektorými prvkami zo vstupného poľa
- vnútorná funkcia zaradí prvok do výstupného poľa tým, že pre tento prvok vráti true (truthy)

```
const result = myArray.filter(value => value.id %2 === 1);
```

result:

{id:1, name: "Jano"}	{id:3, name: "Jožo"}

ak má arrow funkcia iba jeden príkaz jeho hodnota sa vráti bez toho, aby sme písali return

map

{id:1, name: "Jano"}	{id:2, name: "Fero"}	{id:3, name: "Jožo"}	{id:4, name: "Anna"}

- výstup: pole prvkov, ktoré vráti vnútorná funkcia
- výstupné pole má vždy rovnakú dĺžku ako vstupné

```
const result = myArray.map(value => value.name);
```

result:

"Jano"	"Fero"	"Jožo"	"Anna"

ekvivalentná funkcia cez object destructuring:

```
const result = myArray.map(({name}) => name);
```

reťazenie filter + map

{id:1, name: "Jano"}	{id:2, name: "Fero"}	{id:3, name: "Jožo"}	{id:4, name: "Anna"}

- Vypíšte mená z objektov s párnym id

```
const result = myArray.filter(value => value.id %2 === 0)
                      .map(value => value.name);
```

filter vráti:

{id:2, name: "Fero"}	{id:4, name: "Anna"}

map vráti:

"Fero"	"Anna"

reduce

{id:1, name: "Jano"}	{id:2, name: "Fero"}	{id:3, name: "Jožo"}	{id:4, name: "Anna"}

- výstup: jeden objekt / hodnota
- vnútorná funkcia dostane na vstup okrem hodnoty z poľa aj medzivýsledok (accumulator)

```
const result = myArray.reduce((accumulator, value) => accumulator + value.id, 100);
```

result: 110

postupné volania: (100, {id:1,name:"Jano"}) => 100 + 1 // vráti 101

(101, {id:2,name:"Fero"}) => 101 + 2 // vráti 103

(103, {id:3,name:"Jožo"}) => 103 + 3 // vráti 106

(106, {id:4,name:"Anna"}) => 106 + 4 // vráti **110**

iniciálna hodnota medzivýsledku

reduce na bežné poľové úlohy

- v nasledovných príkladoch použijeme reduce iba s jedným parametrom – bez iniciálnej hodnoty – ako iniciálna sa použije prvá hodnota poľa a vnútorná funkcia sa volá až od druhého prvku
- nájdenie maxima (ok)/minima (ok)/priemeru (zle)
  - [8,4,9,1].reduce((acc, value) => (acc<value) ? value : acc);
  - [8,4,9,1].reduce((acc, value) => (acc>value) ? value : acc);
  - [8,4,9,1].reduce((acc, value) => acc + value/4);

postupné volania: (8, 4) => 8 + 4/4 // vráti 9

(9, 9) => 9 + 9/4 // vráti 11.25

(11.25, 1) => 11.25 + 1/4 // vráti 11.5

reduce na vytváranie objektu

{id:1, name: "Jano"}	{id:2, name: "Fero"}	{id:3, name: "Jožo"}	{id:4, name: "Anna"}

chceme: { ids: [1, 2, 3, 4], names: ["Jano", "Fero", "Jožo", "Anna"]}

```
const result = myArray
   .reduce( (acc, value) => ({ ids: [ ...acc.ids, value.id ],
                               names: [ ...acc.names, value.name ]
                             }), { ids: [], names: [] });
```

```
const result = myArray
   .reduce( (acc, value) => ({ ids: [ ...(acc.ids || [] ), value.id ],
                               names: [ ...(acc.names ||[] ), value.name ]
                             }), {});
```

Array.reduceRight()

- rovnaký princíp ako reduce, len vstupné pole sa spracúva z prava doľava.
- [[0, 1], [2, 3], [4, 5]].reduceRight((acc, cur) => [...acc, cur]);
  - vráti [[4, 5], [2, 3], [0, 1]]

flat

- aplikuje spread operátor na vnútorné polia
- ak má uvedený parameter aplikuje spread operátor na vnútorné polia vnútorných polí rekurzívne do až úrovne hodnoty parametra

```
const myArray1 = [[1, 2], [3, 4], 5].flat(); // [1,2,3,4,5]
const myArray2 = [1,2,[3,[4,5]]].flat(); //[1,2,3,[4,5]]
const myArray3 = [1,2,[3,[4,5]]].flat(2); // [1,2,3,4,5]
const myArray4 = [1,2,[3,[4,[5,6]]]].flat(2); // [1,2,3,4,[5,6]]
const myArray4 = [1,2,[3,[4,[5,6]]]].flat(3); // [1,2,3,4,5,6]
```

Zložitejšie vstupné objekty

```
const genres = [ {
    id: 1,
    name: "Action",
    movies: [
      { title: "Die Hard", rating: 8.2 },
      { title: "Terminator", rating: 6.5 },
      { title: "The Avengers", rating: 7.1 }
    ]}
  ,
  {
    id: 2,
    name: "Thriller",
    movies: [
      { title: "The Sixth Sense", rating: 9.2 },
      { title: "The Others", rating: 5.8 }
    ]
  }];
```

- chceme jeden zoznam všetkých filmov so žánrami
- máme pole 2 žánrov
- v každom žánri máme pole filmov
- stratégia: urobíme pole polí filmov a aplikujeme flat operátor.
- z každého prvku poľa žánrov chceme vyrobiť pole jeho filmov – použijeme map

Zložitejšie vstupné objekty

```
const genres = [ {
    id: 1,
    name: "Action",
    movies: [
      { title: "Die Hard", rating: 8.2 },
      { title: "Terminator", rating: 6.5 },
      { title: "The Avengers", rating: 7.1 }
    ]}
  ,
  {
    id: 2,
    name: "Thriller",
    movies: [
      { title: "The Sixth Sense", rating: 9.2 },
      { title: "The Others", rating: 5.8 }
    ]
  }];
```

```
const result = genres
                .map(genre => genre.movies);
```

```
[
  [ { title: "Die Hard“, rating: 8.2 },
    { title: "Terminator", rating: 6.5 },
    { title: "The Avengers", rating: 7.1}
  ], [
    { title: "The Sixth Sense", rating: 9.2 },
    { title: "The Others", rating: 5.8 }
  ]
]
```

flat + map = flatMap

```
const genres = [ {
    id: 1,
    name: "Action",
    movies: [
      { title: "Die Hard", rating: 8.2 },
      { title: "Terminator", rating: 6.5 },
      { title: "The Avengers", rating: 7.1 }
    ]}
  ,
  {
    id: 2,
    name: "Thriller",
    movies: [
      { title: "The Sixth Sense", rating: 9.2 },
      { title: "The Others", rating: 5.8 }
    ]
  }];
```

```
const result = genres
            .map(genre => genre.movies)
            .flat();
```

```
const result = genres
            .flatMap(genre => genre.movies);
```

```
[
    { title: "Die Hard“, rating: 8.2 },
    { title: "Terminator", rating: 6.5 },
    { title: "The Avengers", rating: 7.1},
    { title: "The Sixth Sense", rating: 9.2 },
    { title: "The Others", rating: 5.8 }
]
```

Chceme dodať každému filmu žáner

```
const genres = [ {
    id: 1,
    name: "Action",
    movies: [
      { title: "Die Hard", rating: 8.2 },
      { title: "Terminator", rating: 6.5 },
      { title: "The Avengers", rating: 7.1 }
    ]}
  ,
  {
    id: 2,
    name: "Thriller",
    movies: [
      { title: "The Sixth Sense", rating: 9.2 },
      { title: "The Others", rating: 5.8 }
    ]
  }];
```

- namiesto vrátenia pôvodného poľa filmov, každý film poľa namapujeme na film so žánrom

```
const mapMovies = (movies, genName) =>
  movies.map( movie =>
                    ({ ...movie, genre: genName }));

const result = genres
  .flatMap( genre => mapMovies(genre.movies,
                                genre.name);
```

```
const result = genres
  .flatMap(genre => genre.movies
            .map(movie =>
              ({...movie, genre: genre.name})
            )
  );
```

Chceme dodať každému filmu žáner

```
const genres = [ {
    id: 1,
    name: "Action",
    movies: [
      { title: "Die Hard", rating: 8.2 },
      { title: "Terminator", rating: 6.5 },
      { title: "The Avengers", rating: 7.1 }
    ]}
  ,
  {
    id: 2,
    name: "Thriller",
    movies: [
      { title: "The Sixth Sense", rating: 9.2 },
      { title: "The Others", rating: 5.8 }
    ]
  }];
```

```
const result = genres
  .flatMap(genre => genre.movies
            .map(movie =>
              ({...movie, genre: genre.name})
            )
   );
```

```
[
    { title: "Die Hard“, rating: 8.2,
      genre: "Action" },
    { title: "Terminator", rating: 6.5,
      genre: "Action" },
    { title: "The Avengers", rating: 7.1,
      genre: "Action" },
    { title: "The Sixth Sense", rating: 9.2,
      genre: "Thriller" },
    { title: "The Others", rating: 5.8,
      genre: "Thriller" }
]
```

Stratégie na spracovanie polí zložitých elementov

- forEach
  - ak nemáme žiaden výstup
- map/flatMap
  - ak výsledok závisí len od hodnoty elementu poľa, alebo nadradených štruktúr v ktorých je toto pole
  - ak máme na vstupe aj výstupe rovnaký počet elementov
  - ak sa potrebujeme vnoriť do podštruktúr elementov
- reduce/reduceRight
  - ak výsledok kombinuje hodnoty z viacerých elementov t.j. súčty, priemery, maximá, ak chceme iba rôzne hodnoty
  - ak je vo výsledku menej elementov ako dĺžka vstupného poľa a nedá sa použiť filter

Všetky parametre pre vnútornú funkciu

- [10,20].forEach((value, index, array) => console.log(value, index, array));
  - výpis:
    - 10 0 [10, 20]
    - 20 1 [10, 20]
  - index: index v poli s hodnotou value
  - array: referencia na vstupné pole
- filter, map, reduce aj reduceRight majú tiež voliteľné parametre index a array

Funkcie nad poľami - rekapitulácia

- forEach
  - [1, 2, 3].forEach(value => console.log(value)); // void
- filter
  - [1, 2, 3].filter(value => value>1); // [2, 3]
- map
  - [1, 2, 3].map(value => value + 10); // [11, 12, 13]
- flat
  - [[1, 2], 3].flat(); // [1, 2, 3]
- flatMap
  - [10, 20, 30].flatMap((value,index) => [index,value]); // [0, 10, 1, 20, 2, 30]
- reduce
  - [1, 2, 3].reduce((acc,value) => [acc, value], 5); //[[[5, 1], 2], 3]
- reduceRight
  - [1, 2, 3].reduceRight((acc,value) => [acc, value], 5); //[[[5, 3], 2], 1]

Ďalšie poľové funkcionálne operátory

- [2,4,6].find(value => value > 2); // 4
  - prvá hodnota pre ktorú vráti true
- [2,4,6].findIndex(value => value > 2); // 1
  - index prvej hodnoty pre ktorú vráti true
- [2,4,6].every(value => value % 2 == 0); //true
  - či pre každú hodnotu vráti true
- [2,4,6].some(value => value % 2 == 0); //false
  - či existuje aspoň jedna hodnota, pre ktorú vráti true

Array.from

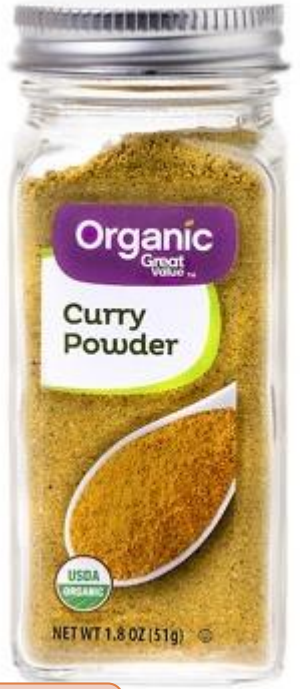
- Array.from("4567", value => value+1);
  - ["41", "51", "61", "71"]
- Array.from("4567", value => parseInt(value)+1);
  - [5, 6, 7, 8]
- Array.from([1,2,3,4], value => value * 2);
  - [2, 4, 6, 8]
- Array.from({ length: 10 }, (value, i) => 1 + i * 2);
  - [1, 3, 5, 7, 9, 11, 13, 15, 17, 19]

pure function - čistá funkcia

- všetko čo potrebuje, získa cez vstupné parametre, žiadne iné hodnoty nečíta
- vždy vracia hodnotu alebo funkciu
- nikdy nemení cudzie hodnoty (napr. globálne premenné, stav aplikácie, DOM model,...)
- je immutable – nemení vstupné parametre
- výhody:
  - nezávisle otestovateľné
  - jednoduchá kompozícia s inými čistými funkciami

curry

- **curried function** je higher-order funkcia, ktorá si berie viacero parametrov vždy po jednom
  - ak máme napr. 3 parametre, curried funkcia si vezme jeden parameter a vráti funkciu, ktorá si vezme druhý parameter a vráti funkciu, ktorá si vezme tretí parameter a vráti výsledok
- každá čiastočná funkcia vracia unárnu funkciu, teda funkciu, ktorá prijíma jeden parameter




```
const sum = (a, b) => a + b;
console.log(sum(2,3)); //5
```

```
const currySum = a => b => a + b;
console.log(currySum(2)(3)); //5
```

- z curried funkcie môžeme vyrobiť jej špecializované verzie
  - const inc = currySum(1);
    - console.log(inc(8)); // 9
  - const inc10 = currySum(10);
    - console.log(int10(8)); //18

ekvivaletne: const inc = b => 1 + b;

obrázok z https://www.walmart.com/ip/Great-Value-Organic-Curry-Powder-1-8-oz/50597542

zreťazenie a kompozícia čistých funkcií

```
const getName = (company) => company.name;
const uppercase = (string) => string.toUpperCase();
const get6Characters = (string) => string.substring(0, 6);
const reverse = (string) => [...string].reverse().join('');

const result = reverse(get6Characters(uppercase(getName({ name: 'Šíravu ľuďom' }))));
    // 'Šíravu ľuďom' => 'ŠÍRAVU ĽUĎOM' => 'ŠÍRAVU' => 'UVARÍŠ'
```

```
const pipeline = [getName, uppercase, get6Characters, reverse];
const result = input => pipeline.reduce((acc, fn) => fn(acc), input)({ name: 'Šíravu ľuďom' });
```

zdroj: https://www.freecodecamp.org/news/pipe-and-compose-in-javascript-5b04004ac937/

zreťazenie a kompozícia čistých funkcií

```
const getName = (company) => company.name;
const uppercase = (string) => string.toUpperCase();
const get6Characters = (string) => string.substring(0, 6);
const reverse = (string) => [...string].reverse().join('');

const result = reverse(get6Characters(uppercase(getName({ name: 'Šíravu ľuďom' }))));
    // 'Šíravu ľuďom' => 'ŠÍRAVU ĽUĎOM' => 'ŠÍRAVU' => 'UVARÍŠ'
```

```
const compose = (...fns) => fns.reduce((f, g) => (...args) => f(g(...args)));
```

```
const result2 = compose(
                    reverse,
                    get6Characters,
                    uppercase,
                    getName
                )({ name: 'Šíravu ľuďom' });
```

posledný parameter v compose je funkcia, ktorá môže prijať ľubovoľný počet parametrov, ostatné vždy práve jeden

zdroj: https://www.freecodecamp.org/news/pipe-and-compose-in-javascript-5b04004ac937/

zreťazenie a kompozícia čistých funkcií

```
const getName = (company) => company.name;
const uppercase = (string) => string.toUpperCase();
const get6Characters = (string) => string.substring(0, 6);
const reverse = (string) => [...string].reverse().join('');

const result = reverse(get6Characters(uppercase(getName({ name: 'Šíravu ľuďom' }))));
    // 'Šíravu ľuďom' => 'ŠÍRAVU ĽUĎOM' => 'ŠÍRAVU' => 'UVARÍŠ'
```

```
const compose = (...fns) => fns.reduce((f, g) => (...args) => f(g(...args)));
```

```
const result2 = compose(
                    reverse,
                    get6Characters,
                    uppercase,
                    getName
                )({ name: 'Šíravu ľuďom' });
```

- postupné hodnoty akumulátora f:
  - reverse
  - (...args)=>reverse(get6Characters(...args))
  - (...args)=>reverse(get6Characters(uppercase(...args)))
  - (...args)=>reverse(get6Characters(uppercase(getName(...args))))

zdroj: https://www.freecodecamp.org/news/pipe-and-compose-in-javascript-5b04004ac937/

zreťazenie a kompozícia čistých funkcií

```
const getName = (company) => company.name;
const uppercase = (string) => string.toUpperCase();
const get6Characters = (string) => string.substring(0, 6);
const reverse = (string) => [...string].reverse().join('');

const result = reverse(get6Characters(uppercase(getName({ name: 'Šíravu ľuďom' }))));
    // 'Šíravu ľuďom' => 'ŠÍRAVU ĽUĎOM' => 'ŠÍRAVU' => 'UVARÍŠ'
```

```
const compose = (...fns) => fns.reduce((f, g) => (...args) => f(g(...args)));
```

```
const pipe = (...fns) => fns.reduceRight((f, g) => (...args) => f(g(...args)));
```

```
const result2 = compose(
                    reverse,
                    get6Characters,
                    uppercase,
                    getName
                )({ name: 'Šíravu ľuďom' });
```

```
const result3 = pipe(
                    getName,
                    uppercase,
                    get6Characters,
                    reverse
                )({ name: 'Šíravu ľuďom' });
```

zdroj: https://www.freecodecamp.org/news/pipe-and-compose-in-javascript-5b04004ac937/

zreťazenie a kompozícia čistých funkcií

Nechce sa vám písať funkcie
compose a pipe?
v knižnici **Lodash** existujú funkcie
fungujúce rovnako, len s inými
názvami a vstupom je pole funkcií:

flow([...fns]) ~ pipe(...fns)
flowRight([...fns]) ~ compose(...fns)

pipe a compose sú v
Lodash ako aliasy

zdroj: https://www.freecodecamp.org/news/pipe-and-compose-in-javascript-5b04004ac937/

výpis priebežných hodnôt v pipe a compose

vyrobíme si curried funkciu na výpis

```
const trace = label => value => {
  console.log(`${ label }: ${ value }`);
  return value;
};
```

```
pipe(
        getName,
        trace('after getName'),
        uppercase,
        trace('after upprecase'),
        get6Characters,
        trace('after get6Characters '),
        reverse,
        trace('after reverse'),
)({ name: 'Šíravu ľuďom' });
```

dodali sme label, prišla nám funkcia, ktorá vezme výsledok predchádzajúcej funkcie v pipe do parametra value, použije ho a vráti ho bez zmeny pre ďalšiu funkciu v pipe

```
after getName: Šíravu ľuďom
after upprecase: ŠÍRAVU ĽUĎOM
after get6Characters: ŠÍRAVU
after reverse: UVARÍŠ
```

všeobecnejšie cez tap funkciu

vyrobíme si curried funkciu tap

```
const tap = f => value => {
  f(value);
  return value;
};
```

```
pipe(
        getName,
        tap(value => console.log(`after getName: ${ value }`)),
        uppercase,
        tap(value => console.log(`after upprecase: ${ value }`)),
        get6Characters,
        tap(value => console.log(`after get6Characters: ${ value }`)),
        reverse,
        tap(value => console.log(`after reverse: ${ value }`)),
)({ name: 'Šíravu ľuďom' });
```