



→ p.2

# ‘use strict’

*striktní režim → správný režim*

```
function nonStrict() {  
    a = 2 // funguje, i pres to ze 'a' neni spravne definovano (pomoci var/let/const)  
    return a * a  
}  
  
function strictlyStrict() {  
    'use strict';  
    a = 2 // ReferenceError: a is not defined  
    return a * a  
}  
  
strictlyStrict()  
nonStrict()
```

# *Datové typy*



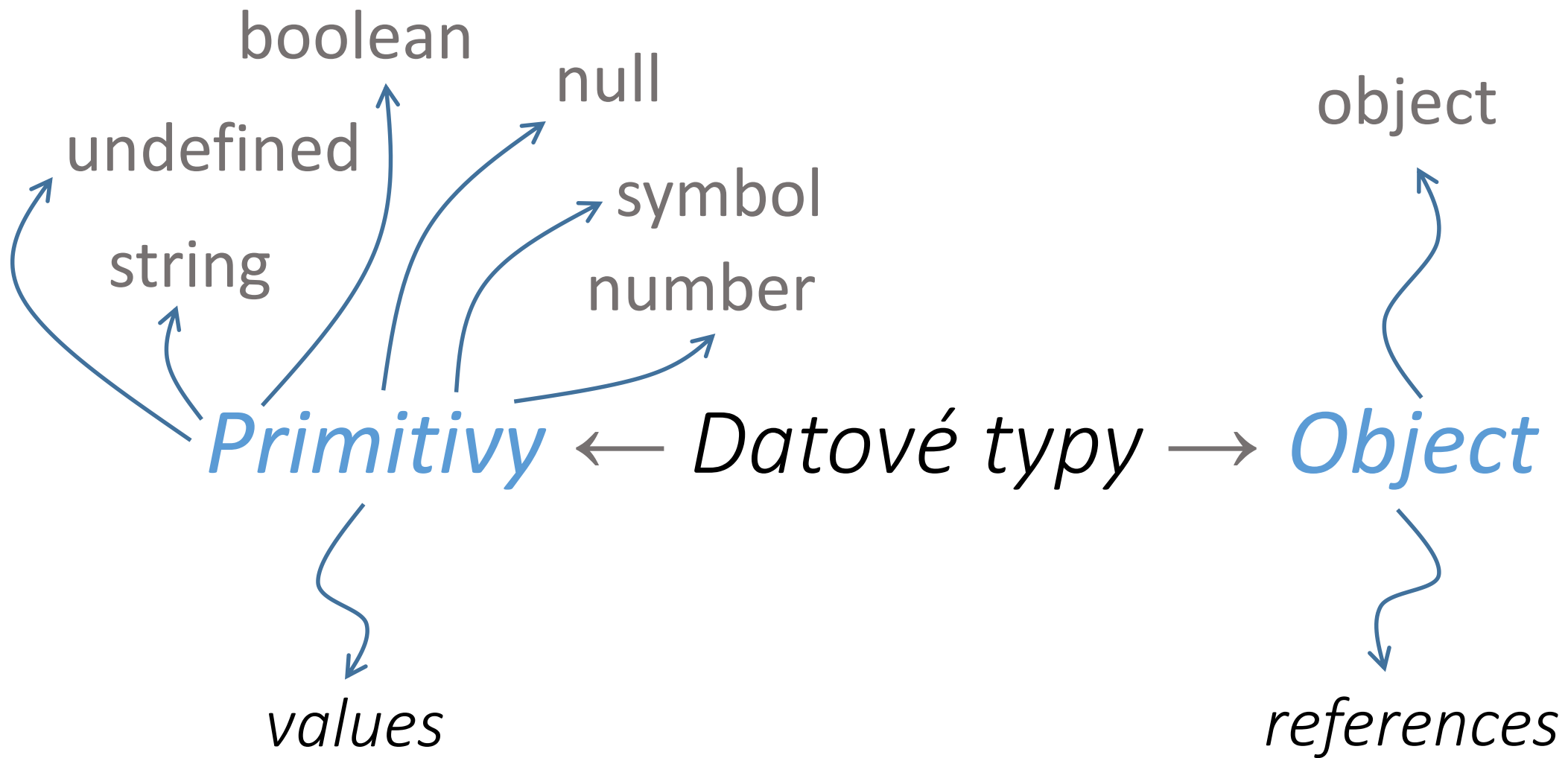
p.2 → *Datové typy*

*Primitivy* ← *Datové typy* → *Object*

*Primitivy* ← *Datové typy* → *Object*

*values* *references*

The diagram illustrates the classification of data types in JavaScript. It features three main terms in a horizontal line: *Primitivy* (in blue), *Datové typy* (in black), and *Object* (in blue). *Datové typy* is the central concept, with arrows pointing to *Primitivy* on the left and *Object* on the right. Below *Primitivy*, a blue curved arrow points down to the word *values*. Similarly, below *Object*, a blue curved arrow points down to the word *references*.



```
/*
    obj je strukturou klic: hodnota.
    obj je referenci na objekt v pameti.
    obj nelze skopirovat, lze skopirovat jenom obsah.
*/
var obj = {a: 1, b: 2}

/*
    Objekt neni skopirovan do obj2, obj2 je jenom dalsi referenci na ten samy objekt v pameti.
    To znamena, ze modifikaci hodnot obj2 bude ovlivnen i obj
    (protoze oboje odkazujou na stejny objekt v pameti).
*/
var obj2 = obj
obj2.b = 3

console.log(obj) // {a: 1, b: 3}
console.log(obj2) // {a: 1, b: 3}
```

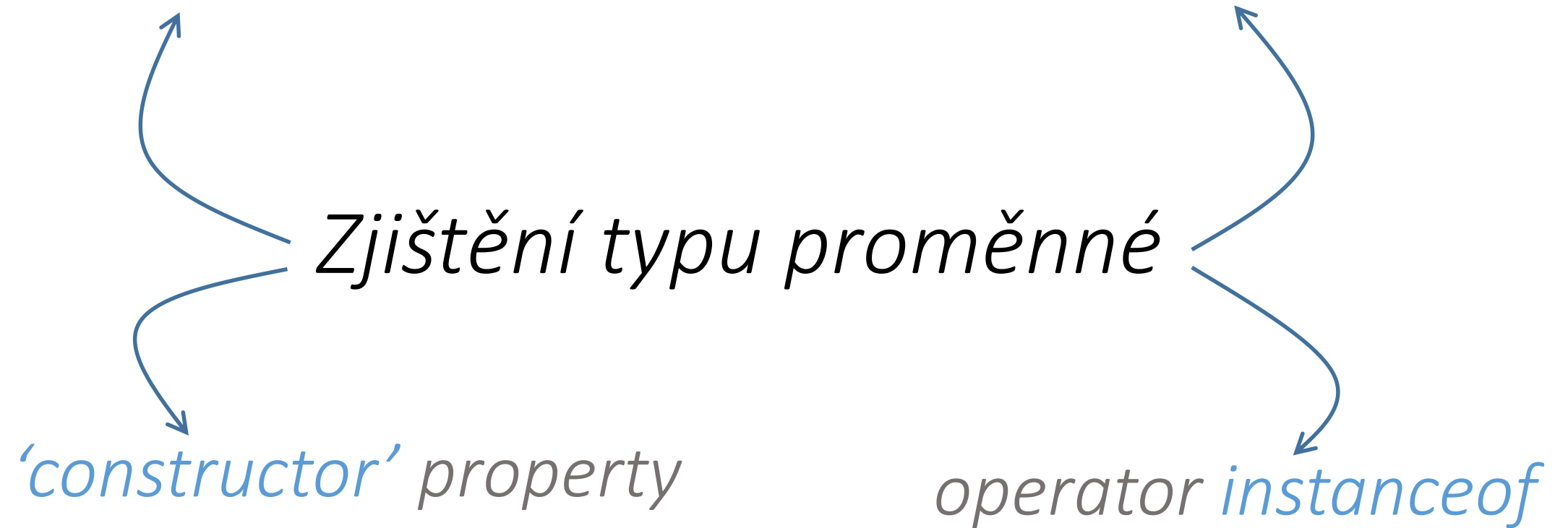


```
/*  
    S primitivy se pracuje skutečne by value,  
    to znamená že promenna sa skutočne kopíruje v pameti.  
*/  
var a = b = c = 3 // vsichni tri promenne maju svoji "trojku"  
a = 1  
b = 2  
console.log(a, b, c) // 1, 2, 3
```

# *Zjištění typu proměnné*

operator *typeof*

*[[class]]* property



*typeof* *x*

*vrací string*

*string* → 'string'

*number* → 'number'

*boolean* → 'boolean'

*symbol* → 'symbol'

*object* → 'object'

*array* → 'object'

*function* → 'function'

*null* → 'object'

```
// Primitivy
typeof 1 // 'number'
typeof 'ahoj' // 'string'
typeof null // 'object' <- pozor, null není object, ale null
typeof undefined // 'undefined'
typeof true // 'boolean'
typeof Symbol('example') // 'symbol'

// Object
typeof {} // 'object'
typeof [] // 'object', pole nemá svůj datový typ, je to pořád object
typeof function(){} // 'function', funkce jsou ale také objekty a proto správně tu má být 'object'
typeof /abc/ // 'object' regulární výrazy jsou také objekty
```

*Object.prototype.toString.call(n)*

*má přístup k [[class]]*



*Object.prototype.toString.call(n)*

*vrací jméno konstruktoru jako  
“[object Constructor]”*

```
/*
  Nasledujici funkce neslouzi ke zjisteni datoveho typu, ale ke zjisteni konstrukturu argumenta.
  Rozlisuje pouze built-in konstruktory (jako Number, RegExp a dalsi).
  Nerozlisuje custom konstruktory (napr Animal).
*/
function classOf(any) {
  return Object.prototype.toString.call(any)
}

classOf(1) // '[object Number]'

/* Pro cistejsi kod se cast vysledku da odstranit. */
function classOf(any) {
  return Object.prototype.toString.call(any).slice(8, -1)
}

classOf(1) // 'Number'
classOf([]) // 'Array'
classOf(/abc/) // 'RegExp'
```



```
/*  
    Funguje konzistentne i pro null a undefined,  
    pres to ze takove konstruktory neexistuji.  
*/  
  
classOf(null) // 'Null'  
classOf(undefined) // 'Undefined'  
  
/* Nefunguje pro custom konstruktory. */  
  
class Animal{}  
const pejsek = new Animal()  
classOf(pejsek) // 'Object' misto 'Animal'
```

# *'constructor' property*

*nebezpečný způsob*

```
/*  
|   Da se zjisti referenci na konstruktor, nefunguje na null/undefined.  
*/  
const neco = []  
console.log(neco.constructor) // Array, odkazuje primo na funkci konstruktor  
console.log(null.constructor) // TypeError: Cannot read property 'constructor' of null  
console.log(undefined.constructor) // TypeError: Cannot read property 'constructor' of undefined  
  
/*  
|   Da se prepsat a proto neni uplne safe.  
*/  
neco.constructor = Object  
console.log(neco.constructor) // Object, ma byt spravne Array
```

# *Operator instanceof*

*nepřesný způsob*

```
/*  
    Vraci true pokud dana instance je instanci dane tridy.  
    False v opacne pripade.  
    Nejednoznacny zpusob zjisteni typu/konstruktoru promenne.  
*/  
  
/* Cislo jako primitiv */  
const n = 1  
console.log(n instanceof Number) // false  
  
/* Cislo jako objekt */  
const m = new Number(1)  
console.log(m instanceof Number) // true  
  
/* Pole */  
console.log([] instanceof Object) // true  
console.log([] instanceof Array) // true
```

# *Casting*

*převod mezi datovými typy*

*s využitím operátorů*

*s využitím konstruktorů*



```
/*
|   Prevod na cislo pomoci operatoru + nebo konstruktoru Number.
|   Konstruktor se vola bez 'new'.
*/

const n = '123'
console.log(typeof +n) // 'number'
console.log(typeof Number(n)) // 'number'

/*
|   Operator 'new' ma jinou semantiku, vytvari objekt, který obsahuje primitiv uvnitr.
*/
const nAsObject = new Number(n)
console.log(nAsObject) // [object Number]
console.log(nAsObject.valueOf()) // metoda valueOf() vraci primitiv
console.log(typeof nAsObject) // 'object'
console.log(typeof nAsObject.valueOf()) // 'number'
```



```
/*  
    Prevod na string pomoci konkatenaci se stringem  
    nebo volanim toString nebo konstruktoru String.  
*/  
  
const n = 123  
console.log('' + n) // '123'  
console.log(n.toString()) // '123'  
console.log(String(n)) // '123' <- nejcitelnejsi a nebezpecnejsi zpusob.  
  
console.log(null.toString()) // TypeError: Cannot convert object to primitive value  
console.log(String(null)) // 'null'
```

```
/*  
|   Prevod na boolean hodnotu pomoci operatoru ! nebo konstrukturu Boolean.  
*/  
  
const value = 'ahoj' // truthy hodnota, po prevodu ma byt true  
  
console.log(!!value) // dvojita negace vraci 'true'  
console.log(Boolean(value)) // 'true'  
  
const nan = NaN // NaN (not-a-number) je falsy hodnotou, po prevodu ma byt false.  
  
console.log(!!nan) // dvojita negace vraci 'false'  
console.log(Boolean(nan)) // 'false'
```

# *Autoboxing*

*možnost přistupovat na prototypové vlastnosti primitiv*

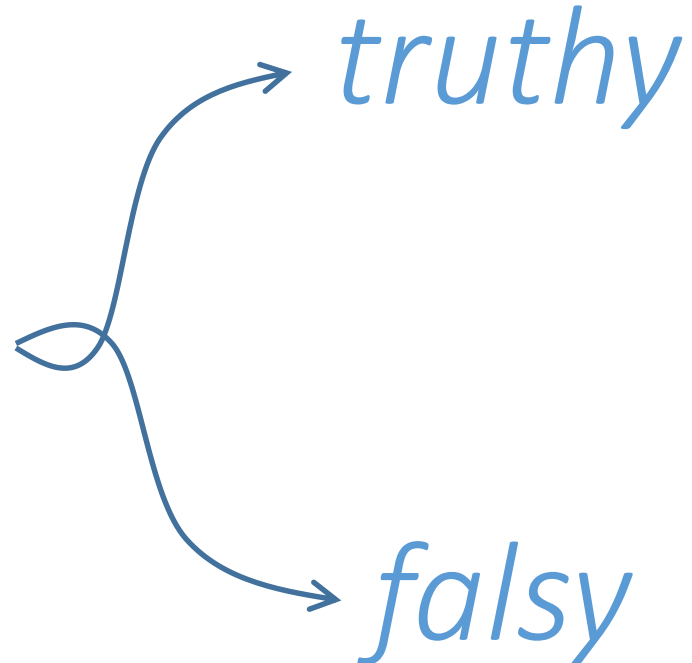
```
/*
| Pres to ze retezec je primitiv (string),
| da se na nem aplikovat objektove metody jeho konstrukturu.
*/

console.log('ahoj'.length) // 4

/*
| Na pozadi se deje neco podobneho:
*/

const temporaryObject = new String('ahoj')
const result = temporaryObject.length
console.log(result)
```

*Všetchny  
hodnoty*



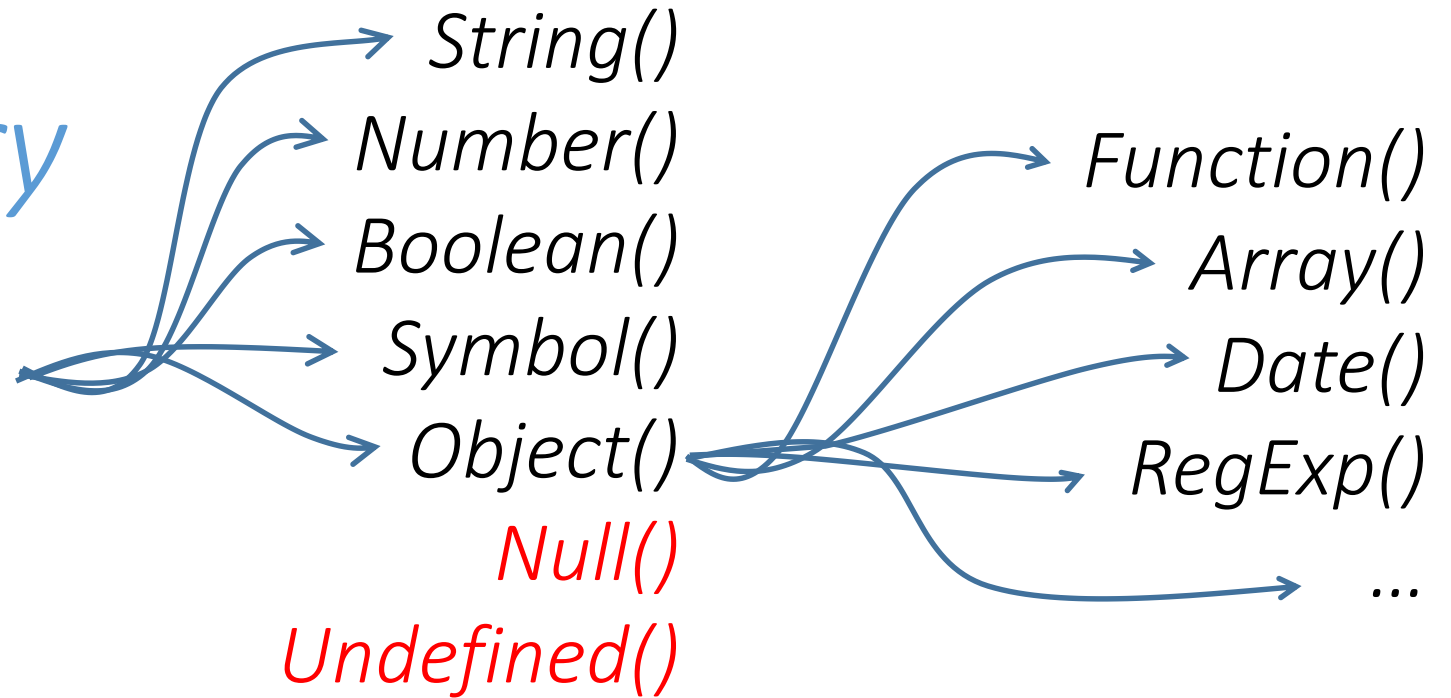
*truthy*

*falsy*

*Všechny hodnoty* { *truthy* = všechny kromě falsy  
*falsy* = null, 0, undefined, NaN, ""

```
/*  
|   Falsy hodnoty: null, undefined, NaN, 0, ''  
*/  
  
if (0) console.log('neprovedu se kdyz je to 0')  
if (NaN) console.log('neprovedu se kdyz je to NaN (not-a-number)')  
if (null) console.log('neprovedu se kdyz je to null')  
if (undefined) console.log('neprovedu se kdyz je to undefined')  
if ('') console.log('neprovedu se kdyz je to prazdny retezec')  
  
/*  
|   Truthy hodnoty jsou vsechny ostatni.  
*/  
  
if ({}) console.log('provedu se kdyz je to objekt')  
if ('ahoj') console.log('provedu se kdyz je to neprazdny retezec')  
if (-123.32) console.log('provedu se kdyz to neni 0')
```

# Konstruktory datových typů





# *Constructor()* vs *new Constructor()*

# *Constructor.prototype*

*String.prototype.slice = function() {...}*

# *Number*



p.2 → *Datové typy* → *Number*

0.1 + 0.2

===

...



0.1 + 0.2

===

0.30000000000000004

# *IEEE-754*

## *double precision floating point*

123   0b1111011   0173   0o173   0x7b

`1e3 === 1000`

`1e-3 === 0.001`



# Infinity, NaN

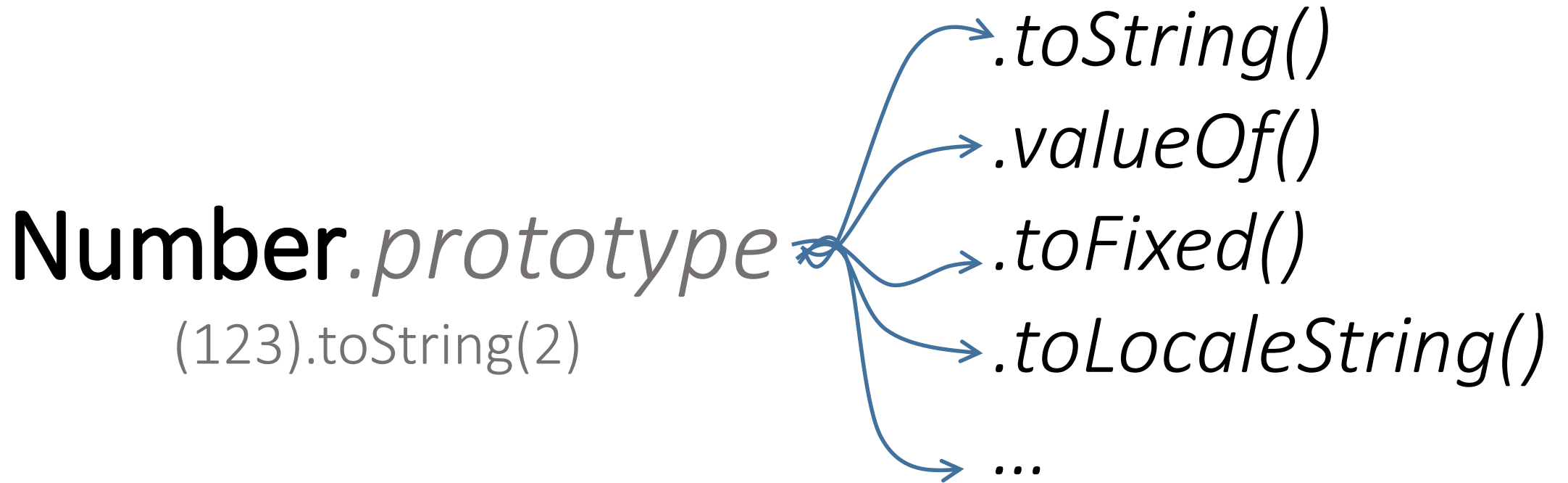
`Number.isFinite()`, `Number.isNaN()`

```
/* NaN nelze porovnavat s zadnou hodnotou (vcetne NaN). */  
console.log(NaN === NaN) // false  
const a = NaN  
console.log(a === a) // false  
  
/* Number.isNaN vraci true pokud argument je NaN. */  
console.log(Number.isNaN(NaN)) // true  
  
/* K dalsi specialni hodnote patri Infinity a -Infinity. */  
const inf = 1 / 0  
console.log(Infinity === inf) // true  
  
/* NaN, Infinity a -Infinity jsou porad hodnoty typu number. */  
console.log(typeof NaN) // number  
console.log(typeof Infinity) // number  
console.log(typeof -Infinity) // number  
  
/* Pro urceni skutecných čísel da se aplikovat metodu Number.isFinite */  
console.log(Number.isFinite(NaN)) // false  
console.log(Number.isFinite(123)) // true
```

```
/* NaN 'infikuje' vypocet bez hlaseni chyby. */  
const a = 1 / 'a' // NaN  
console.log(2 + a - 123 * 777) // NaN, zadny Error.  
  
/* Pro kazdy vypocet kde se pouzivaji uzivatelske vstupy, musite tyto vstupy osetrit. */  
function double(n) {  
    if (!Number.isFinite(Number(n))) {  
        throw Error('Argument is not valid number!')  
    }  
    return n * 2  
}  
  
console.log(double(5)) // 10  
console.log(double('a')) // Error
```

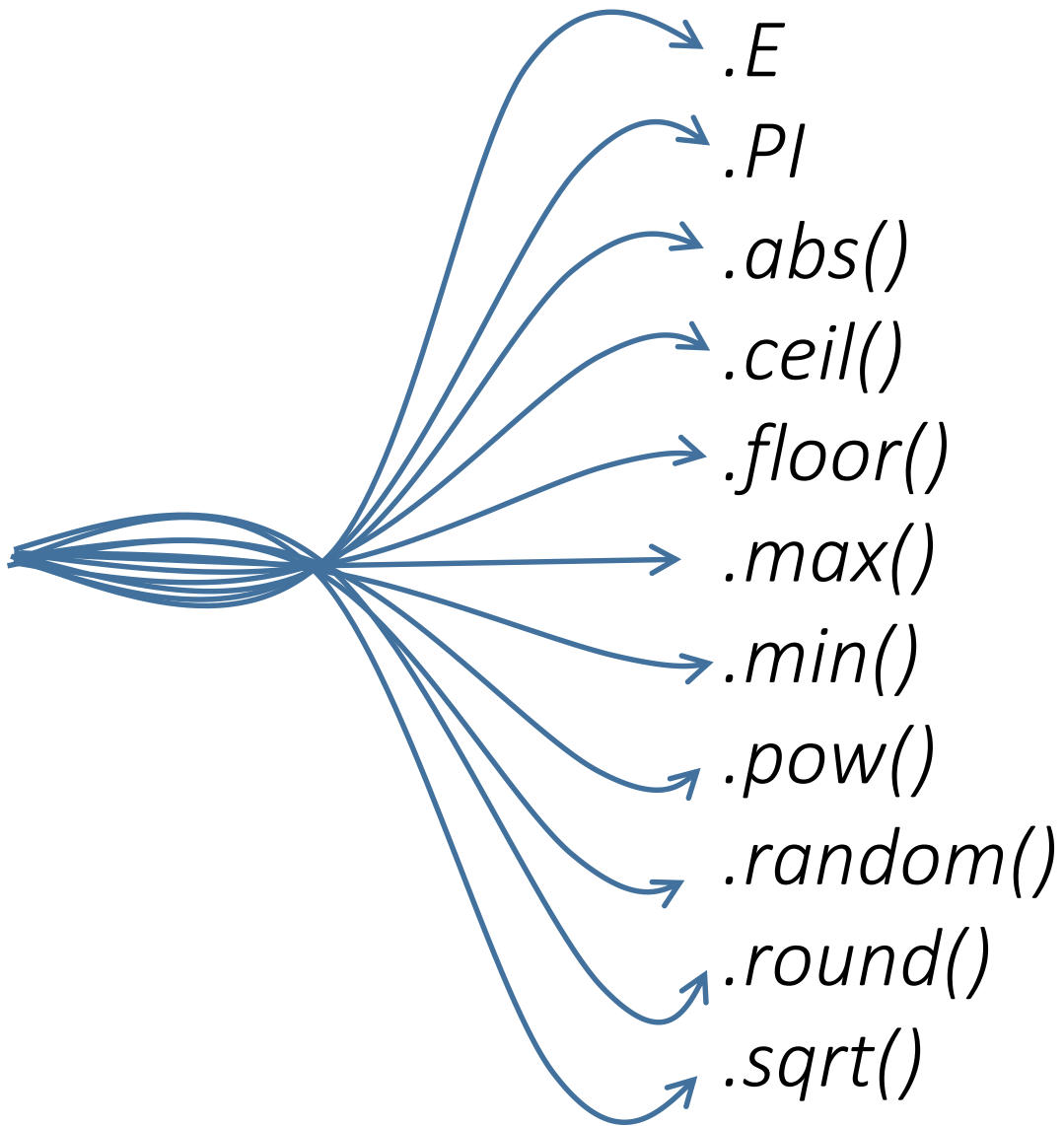
parseInt(), parseFloat()

```
/* parseInt nepodporuje floaty. */  
console.log(parseInt('123.32')) // 123  
console.log(parseFloat('123.32')) // 123.32  
  
/* parseInt podporuje ciselnou soustavu. */  
console.log(parseInt('a7', 16)) // 167
```



```
/* Number.prototype.toString */  
(123).toString() // '123'  
(123).toString(16) // '7b', 16-kova soustava  
  
/* Number.prototype.valueOf */  
new Number(123).valueOf() // 123  
  
/* Number.prototype.toFixed */  
(123.321).toFixed(5) // '123.32100', pozor, je to string  
(123.321).toFixed(3) // '123.321'  
(123.321).toFixed(1) // '123.3'  
(123.321).toFixed(0) // '123'  
(123.321).toFixed(-1) // RangeError
```

# Math





```
Math.PI // 3.1415926...
/* Absolutni hodnota. */
Math.abs(-123) // 123
/* Zaokrouhleni nahoru. */
Math.ceil(123.22) // 124
/* Zaokrouhleni dolu. */
Math.floor(123.88) // 123
/* Vraceni nejvetsiho argumentu. */
Math.max(123, 777, 12) // 777
/* Vraceni nejmensiho argumentu. */
Math.min(123, 777, 12) // 12
/* Umocnovani. */
Math.pow(2, 3) // 8
Math.pow(27, 1/3) // 3
/* Nahodne cislo v intervalu [0, 1). */
Math.random() // 0.1982387234...
/* Zaokrouhleni na cele cislo. */
Math.round(123.321) // 123
/* Odmocnina. */
Math.sqrt(169) // 13
```

*String*



p.2 → *Datové typy* → *String*

*‘abc’*

*“abc”*

*`abc`*

```
console.log('a'
+ 'b') // 'ab'

console.log('a\
b') // 'ab'

console.log('line one\nline two')

console.log(`line one
line two`)
```

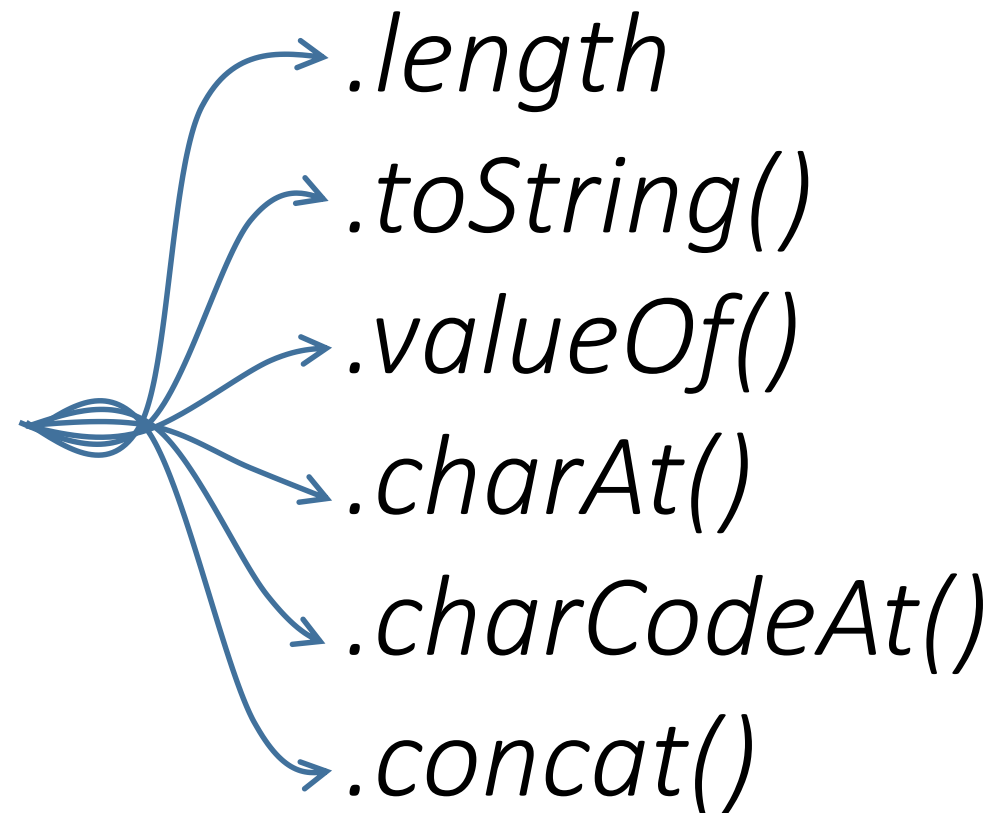
`'\n\t\\'`



`'\x61' === 'a'`  
`'\u0061' === 'a'`

# String.prototype

'asd'.charCodeAt(1)



```
'ahoj'.length // 4

'ahoj'.toString() === 'ahoj'.valueOf()

'ahoj'.charAt(2) // 'o'
'ahoj'[2] // 'o'
'ahoj'.charAt(5) // ''
'ahoj'[5] // undefined

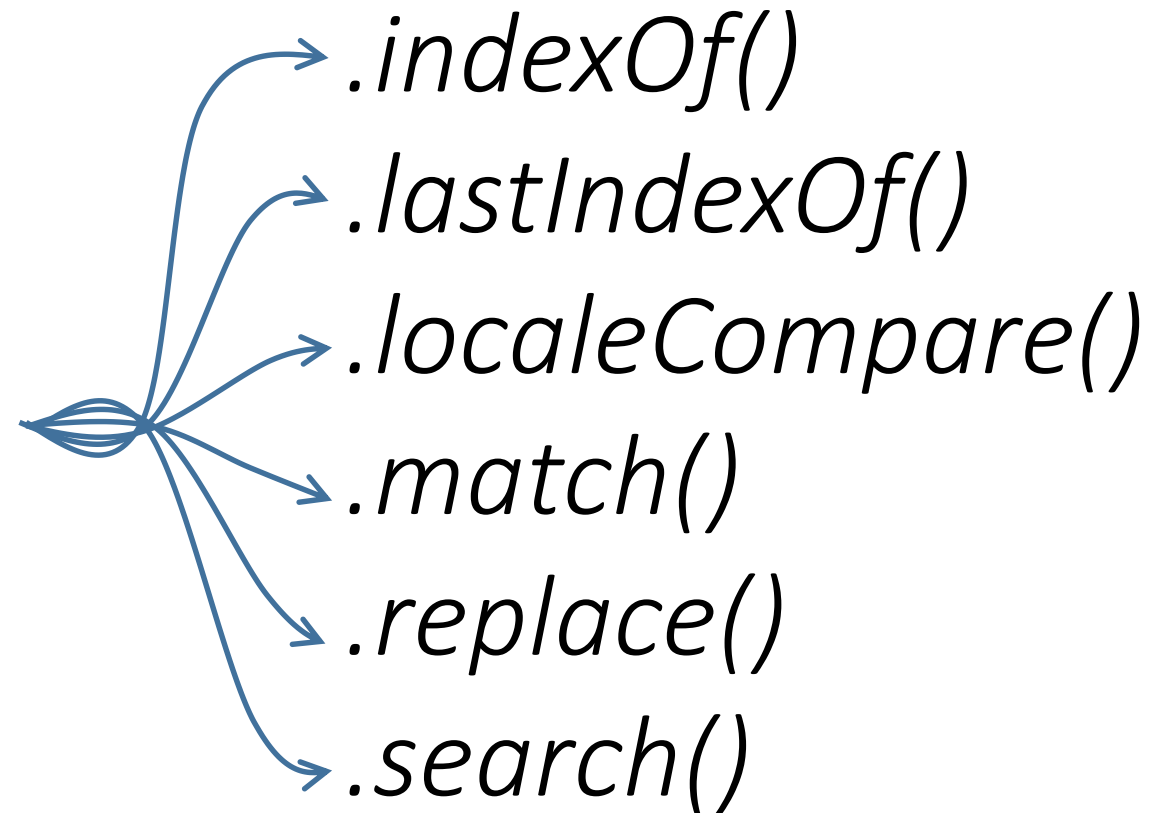
'ahoj'.charCodeAt(2) // 111, String.fromCharCode(111) === 'o'

'a'.concat('h', 'oj') // 'ahoj'
```



# String.prototype

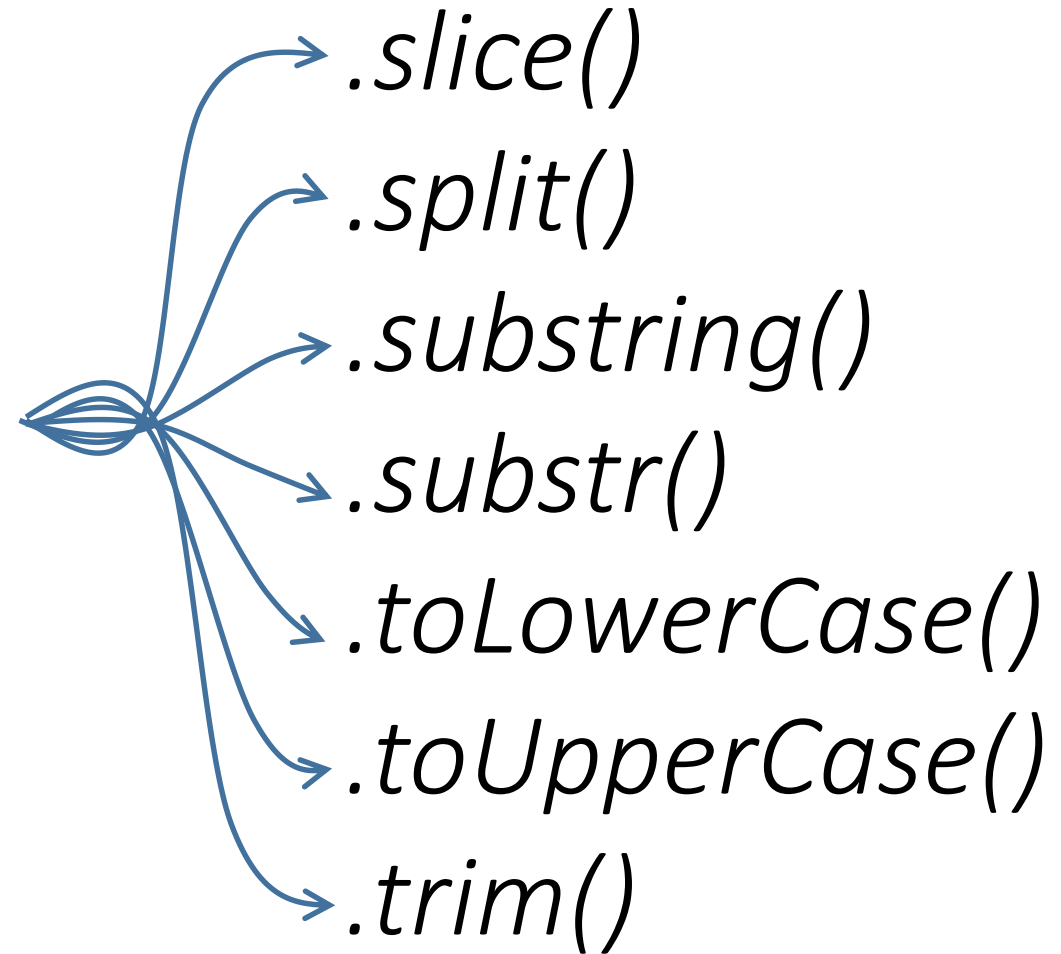
'asd'.charCodeAt(1)



```
'ahoj'.indexOf('o') // 2, pocita se od 0  
'ahoj'.indexOf('x') // -1, neni nalezeno  
'kamarad'.indexOf('a', 2) // 3, hleda od znaku na pozici 2  
  
'č'.localeCompare('d', 'cs-CZ') // -1, č jde pred d.  
'a'.localeCompare('a') // 0, jde o stejny znak  
'z'.localeCompare('a') // 1, z jde po a  
  
// metody match, replace a search podrobneji rozebereme pozdeji
```

# String.prototype

'asd'.charAt(1)



```
'kamarad'.slice(2, 5) // 'mar'
'kamarad'.slice(2, 2) // ''
'kamarad'.slice(2, -2) // 'mar'

'kamarad'.split('a') // ['k', 'm', 'r', 'd']

'kamarad'.substring(2, 5) === 'kamarad'.substring(5, 2)
// prvni arg je nejmensi cislo ze dvou, druhy – nejvetsi ze dvou,
// neumi zaporne argumenty (prevadi na 0)

'kamarad'.substr(2, 3) // 'mar'

'KaMaRaD'.toLowerCase() // 'kamarad'
'KaMaRaD'.toUpperCase() // 'KAMARAD'

'   kamarad   '.trim() // 'kamarad'
```

**String**.*fromCharCode()*

```
String.fromCharCode(97, 104, 111, 106) // 'ahoj'
```

“3” > “299999”

# *Boolean*



p.2 → *Datové typy* → *Boolean*



*true, false*

*new Boolean*

```
if (new Boolean(false)) {  
    console.log(`  
        Podminka bude splnena protoze false je obalen objektem a objekt je truthy hodnota.  
    `)  
}  
  
if (new Boolean(false).valueOf()) {  
    console.log(`  
        Podminka nebude splnena, protoze valueOf cte skutecnou hodnotu (primitiv) daneho objektu.  
    `)  
}  
  
// NEPOUZIVAT new Boolean(...)
```

*Úkoly* → [bit.ly/2XvFkTp](https://bit.ly/2XvFkTp)

// end