თეიმურაზ სტურუა

თეა თოდუა

ქეთევან ნანობაშვილი

ბესიკ ტაბატაძე

JavaScript-ის საფუძვლები

ნაწილი I

თბილისი

2023

თეიმურაზ სტურუა

თეა თოდუა

ქეთევან ნანობაშვილი

ბესიკ ტაბატაძე

JavaScript-ის საფუძვლები

ნაწილი I

რეკომენდირებულია სახელმძღვანელოდ
ევროპის უნივერსიტეტის სამართლის,
განათლების, ბიზნესისა და ტექნოლოგიების
ფაკულტეტის საბჭოს მიერ

თბილისი 2023

სახელმძღვანელო, JavaScript ენის ელემენტარული საფუძვლებიდან დაწყებული და რთული პრაქტიკული საკითხებით დამთავრებული, საკითხების ფართო სპექტრს მოიცავს. დეტალურად არის განხილული JavaScript-ისა და HTML-ის ურთიერთდამოკიდებულება: მონაცემთა ტიპები, ოპერაციები, გამოსახულებანი და ოპერატორები, ხდომილობები, ობიექტები და ობიექტთა თვისებები, კლასები, JavaScript HTML დოკუმენტის ობიექტური მოდელი (DOM), ასინქრონული ფუნქციები და სხვა. ასევე, მოყვანილია ვებგვერდების შექმნის მრავალი მაგალითი და შესასრულებლად გამზადებული პროგრამების კოდები JavaScript-სცენარების გამოყენებით.

სახელმძღვანელო განკუთვნილია სტუდენტების, მაგისტრებისა და ვებგვერდების შექმნით დაინტერესებული სპეციალისტებისათვის.

© „ევროპის უნივერსიტეტი", 2023

ISBN 978-9941-8-5256-5

ყველა უფლება დაცულია. ამ წიგნის არც ერთი ნაწილის (იქნება ეს ტექსტი, ფოტო, ილუსტრაცია თუ სხვა) გამოყენება არანაირი ფორმით და საშუალებით (იქნება ეს ელექ-ტრონული თუ მექანიკური), არ შეიძლება გამომცემლის წერილობითი ნებართვის გარეშე. საავტორო უფლებების დარღვევა ისჯება კანონით.

სარჩევი

რა არის JAVASCRIPT? ... **9**

შიდა სკრიპტი ... 12
გარე სკრიპტი ... 14
კოდის სტრუქტურა ... 16
კომენტარები ... 18
მკაცრი რეჟიმი - „USE STRICT“ ... 19

ცვლადები ... **21**

საკვანძო სიტყვა VAR ... 22
ცვლადის სახელი ... 26
მუდმივები ... 28

მონაცემები JAVASCRIPT-ში ... **30**

რიცხვითი ტიპი ... 30
BigInt ... 32
სტრიქონული ტიპი ... 32
ბულის (ლოგიკური) ტიპი ... 34
მნიშვნელობა NULL ... 35
მნიშვნელობა UNDEFINED ... 35
ობიექტები და სიმბოლოები ... 36
ოპერატორი TYPEOF ... 37

მონაცემების შეტანა-გამოტანა ... **39**

alert ... 39
prompt ... 39
confirm ... 41

მონაცემთა გარდაქმნები ... **44**

სტრიქონის გარდაქმნები ... 44
რიცხვითი გარდაქმნები ... 44
ლოგიკური გარდაქმნები ... 46

მათემატიკური საბაზო ოპერატორები 47
სტრიქონების შეკრება ბინარული +-ის საშუალებით 49
ბინარული + ... 50
მინიჭების ოპერატორი... 52
მინიჭების დამატებითი ოპერატორები 54
ინკრემენტი და დეკრემენტი.. 55
ბიტური ოპერატორები 57
ოპერატორი „მძიმე“ 58
შედარების ოპერატორები..59
სტრიქონების შედარება 60
სხვადასხვა ტიპის ცვლადის შედარება.................................. 62
მკაცრი შედარება 63
null-ისა და undefined-ის შედარებები 64
ლოგიკური ოპერატორი...65
|| („ან“).. 67
&& („და“)... 69
! (უარყოფა)... 71
null-თან გაერთიანების ოპერატორი ?? 72
ოპერატორების პრიორიტეტები.. 74
პირობითი გადასვლის ოპერატორები..77
ინსტრუქცია IF 77
ბლოკი else 78
ტერნარული „?“ ოპერატორი 80
რამდენიმე ტერნარული „?“ ოპერატორი 81
კონსტრუქცია SWITCH 82
ციკლის ოპერატორები 87
ციკლის ოპერატორი WHILE 87
ციკლის ოპერატორი DO...WHILE..89
ციკლის ოპერატორი FOR...91
ცვლადის ჩაშენებული გამოცხადება.................................... 92
for ციკლის ნაწილების გამოტოვება 93

ოპერატორი BREAK 95
ოპერატორი CONTINUE 96
ჭდეები break და continue ოპერატორებისათვის 97

ფუნქციები 99

მომხმარებლის ფუნქციები 99
ლოკალური ცვლადები 101
გლობალური ცვლადები 101
პარამეტრები 103
ნაგულისხმევი პარამეტრები 105
მნიშვნელობის დაბრუნება 106
ფუნქციის სახელის შერჩევა 108
ფუნქცია - კომენტარი 109
FUNCTION EXPRESSION (ფუნქციური გამოსახულება) 116
ფუნქცია „call back“ 119
Named Function Expression 121
სინტაქსი new Function 125
ფუნქცია-ისარი 127
სტანდარტული ფუნქციები 129
რეკურსიული ფუნქციები 131
რეკურსიული ფუნქციის გამოყენების მაგალითები 136

SETTIMEOUT და SETINTERVAL 144

setTimeout 144
clearTimeout 146
setInterval 147

ობიექტები 149

ლიტერალი და თვისება 150
კვადრატული ფრჩხილები 152
გამოთვლადი თვისებები 155
თვისება ცვლადისაგან 156
თვისების არსებობის შემოწმება 157

ციკლი for ... in 159
ობიექტთა თვისებების მოწესრიგება 161
ობიექტისა და ზმულის კოპირება 164
ზმულით შედარება 165

JSON ფორმატი, TOJSON მეთოდი 167

JSON.stringify 168
JSON.parse 180

გლობალური ობიექტი 184

ფუნქციის ობიექტი, NFE 187
თვისება name 187
თვისება length 189
მომხმარებლის თვისებები 190
მონაცემთა ტიპი SYMBOL 191
სიმბოლოები 191
„დაფარული“ თვისებები 193
სიმბოლოები ლიტერარულ ობიექტებში 194
სიმბოლოები და for…in ციკლი 195
გლობალური სიმბოლოები 196
Symbol.keyFor 198
სისტემური სიმბოლოები 199
ობიექტების პრიმიტივად გარდაქმნა 206
პრიმიტივების მეთოდები 213
თვისებები „გეტერები“ და „სეტერები“ 215
წვდომა თვისების დესკრიპტორები 218
ნარჩენი პარამეტრები და გაფართოების ოპერატორი 219
ნარჩენი პარამეტრები 220
გაფართოების ოპერატორი 221

მონაცემთა ტიპები 226

რიცხვები 226
რიცხვების ჩაწერის წესები 226

თექვსმეტობითი, ორობითი და რვაობითი რიცხვები...... 227
toString(base).. 228
დამრგვალება 229
არაზუსტი გამოთვლები 232
isFinite და isNaN შემოწმება 234
Object.is შედარება 236
parseInt და parseFloat 236
სხვა მათემატიკური ფუნქციები.. 238
BigInt ფუნქცია... 240
მათემატიკური ოპერატორები... 241
შედარების ოპერაცია... 243
ლოგიკური ოპერაციები 243
სტრიქონები 244
სპეციალური სიმბოლოები 245
სტრიქონის სიგრძე 248
სიმბოლოებზე მიმართვა 248
ქვესტრიქონის ძებნა 250
str.indexOf... 250
includes, startsWith, endsWith 252
ქვესტრიქონის მიღება 253
str.slice(start [, end])... 253
str.substring(start [, end]).. 253
str.substr(start [, length]) 254
სტრიქონების შედარება 255

მასივი...257

მასივის გამოცხადება 258
POP/PUSH, SHIFT/UNSHIFT მეთოდები 260
მასივის ბოლოსთან მომუშავე მეთოდები......................... 261
მასივის დასაწყისთან მომუშავე მეთოდები 262
მასივის მოწყობა 263
ეფექტურობა 264
მასივის ელემენტების გამოტანა.. 266

length თვისება 267
new Array().. 269
toString.. 269
მრავალგანზომილებიანი მასივი 270
მასივის კოპირება 271
თვისება prototype 272
ობიექტ ARRAY-ის მეთოდები 273
concat 273
split და join 275
splice... 276
slice... 278
sort.. 279
reverse 282
toLocaleString, toString... 282
რიცხვითი მასივების დამუშავების ფუნქციები. 282
თარიღი და დრო.. 285
DATE ობიექტის შექმნა 286
new Date(year, month, date, hours, minutes, seconds, ms)..... 288
DATE ობიექტის მეთოდები. 289
თარიღის ავტომატური კორექტირება 294
თარიღის სხვაობის რიცხვად გადაქცევა 296
Date.now() 298
კვირის დღის ჩვენება 299
გამოყენებული ლიტერატურა 302

რა არის JavaScript?

თავდაპირველად JavaScript ვებგვერდების „გასაცოცხლებლად“ შეიქმნა. პროგრამას ამ ენაზე სკრიპტი ეწოდება. ისინი შეიძლება HTML-ში იყოს ჩაშენებული და ავტომატურად შესრულდეს ვებგვერდის ჩატვირთვისთანავე. სკრიპტები ვრცელდება და სრულდება როგორც ჩვეულებრივი ტექსტი. გაშვებისათვის მას არ ჭირდება სპეციალური მომზადება ან კომპილაცია. ეს ასხვავებს JavaScript-ს სხვა Java - ენისაგან.

როდესაც JavaScript-ი პირველად იქმნებოდა, მას სხვა სახელი ქონდა - „LiveScript“. იმ დროს Java ენა ძალზე პოპულარული იყო და ამიტომაც გადაწყდა დარქმეოდა სახელი JavaScript-ი, როგორც Java დაპროგრამების ენის „უმცროს ძმას“, რაც მის პოპულარობას გაზრდიდა. მაგრამ, თანდათან JavaScript, თავისი სპეციფიკაციით, სრულიად დამოუკიდებელი ენა გახდა, რომელსაც ECMAScript[1] ჰქვია და ამჟამად Java ენასთან არავითარი კავშირი არა აქვს.

ამჟამად, JavaScript-ს არა მარტო ბრაუზერში შეუძლია მუშაობა, არამედ სერვერზეც (SSJS) ან ნებისმიერ სხვა მოწყობილობაზეც, რომელსაც გააჩნია სპეციალური პროგრამა და რომელსაც JavaScript-ის „ძრავს“ უწოდებენ. ბრაუზერს

[1] **ECMAScript** - ეს არის ჩაშენებული, გაფართოებადი, მონაცემთა შეტანა-გამოტანის საშუალებების გარეშე დაპროგრამირების ენა, რომელიც გამოიყენება სხვა სკრიპტების ენების შესაქმნელად. სტანდარტიზებულია საერთაშორისო ორგანიზაციის ECMA-ს მიერ ECMA-262 სპეციფიკაციაში. ECMAScript WWW-ში ჩვეულებრივ კლიენტის მხარეს სცენარებისა და ყველაზე ხშირად სერვერული აპლიკაციების დასაწერად გამოიყენება. ECMAScript ენის გაფართოებას წარმოადგენენ ენები: JavaScript, JScript და ActionScript.

გააჩნია საკუთარი ძრავა, რომელსაც ზოგჯერ „JavaScript-ის ვირტუალურ მანქანას“ უწოდებენ. სხვადასხვა ძრავს სხვადასხვა „კოდური სახელწოდება“ აქვს. მაგალითად:

- V8 - Chrome-ში და Opera-ში;
- SpiderMonkey - Firefox-ში;
- „Trident“ და „Chakra“ IE-ს სხვადასხვა ვერსიისათვის;
- „ChakraCore“ Microsoft Edge-სათვის;
- „Nitro“ და „SquirrelFish“ Safari-სათვის და ა. შ.

ამ დასახელებების ცოდნა საჭიროა, რადგან ისინი ხშირად გამოიყენება სტატიებში.

ამ ძრავების მუშაობის აღწერა რთულია, მაგრამ მათი მუშაობის არსი შემდეგია:

1. ძრავა (ჩაშენებული, თუ ეს ბრაუზერია) კითხულობს სკრიპტის ტექსტს;
2. შემდეგ სკრიპტს გარდაქმნის („კომპილირებას აკეთებს“) მანქანურ ენაზე;
3. ამის შემდეგ გაუშვებს შესრულებაზე მანქანურ კოდს და საკმაოდ სწრაფად მუშაობს.

ძრავა, მუშაობის ყოველ ეტაპზე ახდენს ოპტიმიზაციას. ის მუშაობის პროცესში უკვე კომპილირებულ სკრიპტს ათვალიერებს, მასში გამავალი მონაცემების ანალიზს ახორციელებს და მიღებულ ცოდნაზე დაყრდნობით მანქანური კოდის ოპტიმიზაციას ახდენს. შედეგად, სკრიპტები ძალიან სწრაფად მუშაობს.

რითი არის JavaScript განსაკუთრებული? როგორც მინიმუმ მას სამი ძლიერი მხარე გააჩნია: 1. HTML/CSS-თან სრული ინტეგრაცია; 2. სიმარტივე; 3. ყველა ძირითადი ბრაუზერის მიერაა მხარდაჭერილი და ჩართული.

JavaScript ერთადერთი ბრაუზერული ტექნოლოგიაა, რომელშიც ეს სამივე მხარეა თავმოყრილი. ეს მას განსაკუთრებულს ხდის. ამიტომაც არის ის ბრაუზერში ინტერფეისის შესაქმნელი ყველაზე ფართოდ გავრცელებული და ხელმისაწვდომი ინსტრუმენტი. თუმცა, JavaScript-ი საშუალებას იძლევა აპლიკაციები შეიქმნას არა მარტო ბრაუზერში, არამედ სერვერზეც, მობილურ მოწყობილობებზეც და ა. შ.

JavaScript-ის სინტაქსი ყველა ტიპის ამოცანის გადასაწყვეტად არ გამოდგება. სხვადასხვა მომხმარებელს სხვადასხვა მოთხოვნილებები გააჩნიათ. ეს ცხადია, ვინაიდან პროექტები სხვადასხვაა და მათ მიმართ მოთხოვნებიც სხვადასხვა აქვთ. ამის გამო, ბოლო დროს მრავალი ახალი ენა გამოჩნდა, რომელთა ბრაუზერში გაშვების წინ მათი JavaScript-ში გადაყვანა ხდება. თანამედროვე ხელსაწყოები გადაყვანას ძალიან სწრაფს და გამჭვირვალეს ხდის, რაც დეველოპერებს საშუალებას აძლევს კოდი სხვა ენაზე დაწერონ და ის ავტომატურად JavaScript-ში გადააკეთონ. ასეთი ენებია:

CoffeeScript - მას აქვს უფრო მოკლე სინტაქსი, რომელიც საშუალებას იძლევა დაიწეროს მარტივი და ლაკონური კოდი. ეს ჩვეულებრივ პროგრამისტებს მოსწონთ;

TypeScript „მკაცრ ტიპიზაციაზეა“ ფოკუსირებული, დამუშავების გამარტივებისა და დიდი და რთული სისტემების მხარდასაჭერად. დამუშავებულია Microsoft-ის მიერ;

Flow მსგავსი ენაა, მაგრამ სხვაგვარი. დამუშავებულია Facebook-ის მიერ;

Dart იმით გამოირჩევა, რომ მას აქვს საკუთარი ძრავა, რომელიც მუშაობს ბრაუზერის გარეთ (მაგალითად, მობილურ

აპლიკაციებში). ის თავდაპირველად Google-ის მიერ JavaScript-ის შემცვლელად იყო შემოთავაზებული, მაგრამ ამ დროისთვის საჭიროა მისი ტრანსლირება, რათა იმუშაოს ისევე, როგორც ზემოთ მოყვანილმა ენებმა;

Brython გადააქვს Python JavaScript-ზე, რაც საშუალებას გვაძლევს დაიწეროს აპლიკაციები სუფთა Python-ზე JavaScript-ის გარეშე.

არიან სხვა ენებიც. მაგრამ მაშინაც კი, თუ ამ ენებიდან ერთ-ერთს ვიყენებთ, აუცილებლად უნდა ვიცოდეთ JavaScript-ი.

შიდა სკრიპტი

JavaScript თავსდება HTML დოკუმენტის შიგნით, სადაც ის ტექსტის სახით ინახება HTML დოკუმენტთან ერთად.

JavaScript-ის ენაზე პროგრამების შესაქმნელად საკმარისია ჩვენს კომპიუტერში ჩატვირთული იყოს Notepad ან რომელიმე ცნობილი პროგრამული კოდის რედაქტორი. იმისათვის, რომ შექმნილი კოდი გავუშვათ ტესტირებაზე, როგორც უკვე აღვნიშნეთ, აუცილებელია ნებისმიერი ბრაუზერი.

JavaScript-ზე დაწერილი პროგრამა შეიძლება განთავსებული იყოს HTML-დოკუმენტის ნებისმიერ ადგილზე <script> წყვილი ტეგის საშუალებით. საცდელად დავწეროთ და შევასრულოთ ელემენტარული პროგრამა JavaScript-ზე.

გავხსნათ ტექსტური რედაქტორი და შევიტანოთ შემდეგი ტექსტი:

```
<!DOCTYPE HTML>
<html>
```

```
<body>
  <p>სკრიპტის წინ...</p>
    <script>
          alert( 'Hello, Word!' );
    </script>
  <p>...სკრიპტის შემდეგ.</p>
</body>
</html>
```

ეს პროგრამა შევინახოთ ფაილში Mag.html სახელით. ამ პროგრამის შესრულების შედეგად გაიხსნება ბრაუზერი, ხოლო მის ფონზე დიალოგური ფანჯარა წარწერით "Hello, Word!" და OK ღილაკით.

OK ღილაკზე ხელის დაჭერით ბრაუზერის ფანჯარაში გამოჩნდება შემდეგი ტექსტი:

სკრიპტის წინ...

...სკრიპტის შემდეგ.

ეს არის HTML-დოკუმენტი. ეს ფაილი შევინახოთ დისკზე გაფართოებით .htm ან .html. ამ ფაილში ჩვენ დავწერეთ HTML ენის ტეგები, ხოლო JavaScript ენაზე გამოსახულების ჩაწერა მოვახდინეთ <SCRIPT> და </SCRIPT> ტეგებს შორის. სასურველია, რომ თითოეულ სტრიქონში არ ჩავწეროთ ერთზე მეტი გამოსახულება. ახალ სტრიქონზე გადასასვლელად სტრიქონის ჩაწერა დავამთავროთ <Enter> კლავიშზე ხელის დაჭერით.

პროგრამის შესრულების შედეგის ეკრანზე გამოსატანად გამოიყენება alert() მეთოდი. მას შედეგი გამოაქვს დიალოგურ ფანჯარაში, რომელზეც გამოტანილი იქნება ფრჩხილებში მოთავსებული გამოსახულების მნიშვნელობა (ჩვენს შემთხვევაში – Hello, Word!).

გარე სკრიპტი

თუ თქვენ ბევრი JavaScript კოდი გაქვთ, შეგიძლიათ განათავსოთ იგი ცალკე ფაილში. სკრიპტის ფაილი შეიძლება ჩაერთოს HTML-ში src ატრიბუტის გამოყენებით:

```
<script src="C:\Users\user\Documents\mag.js"></script>
```

C:\Users\user\Documents\mag.js არის აბსოლუტური გზა საიტიდან სკრიპტისკენ, რაც არასწორი მიდგომაა და არ გამოიყენება. თქვენ შეგიძლიათ მიუთითოთ ფარდობითი გზა მიმდინარე გვერდიდან. მაგალითად, src="mag.js" ნიშნავს, რომ ფაილი "mag.js" არის მიმდინარე საქაღალდეში. თქვენ ასევე შეგიძლიათ მიუთითოთ სრული URL.

მრავალი სკრიპტის ჩასართავად მრავალი ტეგი გამოიყენეთ:

```
<script src="mag1.js"></script>
<script src="mag2.js"></script>
…
```

როგორც წესი, HTML-ში მოთავსებულია მხოლოდ უმარტივესი სკრიპტები, ხოლო უფრო რთული სკრიპტები უმჯობესია ცალკეულ ფაილებში განთავსდეს. ცალკეული ფაილების უპირატესობა ის არის, რომ ბრაუზერი ჩამოტვირთავს სკრიპტს ცალკე და შეძლებს მის ქეშ მეხსიერებაში[2] შენახვას. სხვა გვერდები, რომლებიც იმავე სკრიპტს შეიცავს, მის ამოღებას ქეშიდან შეძლებენ. ამრიგად, ფაილი სერვერიდან მხოლოდ ერთხელ ჩამოიტვირთება. ეს ამცირებს ტრაფიკის მოხმარებას და აჩქარებს გვერდის ჩატვირთვას.

თუ სკრიპტში src ატრიბუტია გამოყენებული, სკრიპტის ტეგის შინაარსი იგნორირებული იქნება.

იმავე <script> ტეგში, თქვენ არ შეგიძლიათ ერთდროულად გამოიყენოთ src ატრიბუტი და კოდი. თქვენ

---

[2] ქეში (ინგლისური cache, ფრანგული cacher-დან - „დამალვა“) - სწრაფი წვდომის შუალედური ბუფერი, რომელიც შეიცავს ინფორმაციას. ქეშში მონაცემებზე წვდომა უფრო სწრაფად ხორციელდება, ვიდრე ორიგინალური მონაცემების მიღება უფრო ნელი მეხსიერებიდან ან დისტანციური წყაროდან, მაგრამ მისი მოცულობა მნიშვნელოვნად შეზღუდულია საწყის მონაცემთა სხვა საცავებთან შედარებით.

უნდა აირჩიოთ: ან გარე სკრიპტი <script src="...">, ან ჩვეულებრივი კოდი <script> ტეგის შიგნით.

ზემოთ განხილული ამოცანა შევცვალოთ გარე სკრიპტის გამოყენებით და შევასრულოთ:

```
<!DOCTYPE html>
<html>
<body>
  <script src="mag.js"></script>
</body>
</html>
```

ხოლო mag.js ფაილისათვის იმავე საქაღალდეში:

```
alert( 'Hello, Word!' );
```

კოდის სტრუქტურა

ინსტრუქციები ეს არის სინტაქსური კონსტრუქტები და ბრძანებები, რომლებიც ასრულებენ მოქმედებებს. ჩვენ უკვე ვნახეთ ინსტრუქცია alert ('Hello World!'), რომელიც გამოიტანს შეტყობინებას "Hello World!". ჩვენს კოდში შეიძლება გვქონდეს იმდენი ინსტრუქცია, რამდენიც გვინდა. ინსტრუქციები ერთმანეთისაგან შეიძლება გამოიყოს წერტილ-მძიმით.

მაგალითად, აქ ჩვენ ვყოფთ შეტყობინებას "Hello World" ორ ნაწილად:

```
alert( 'Hello'); alert ('Word!');
```

როგორც წესი, თითოეული ინსტრუქცია იწერება ახალ სტრიქონზე, რათა კოდი უფრო ადვილად წასაკითხი იყოს:

```
alert( 'Hello');
alert ('Word!');
```

უმეტეს შემთხვევაში, წერტილ-მძიმე შეიძლება გამოტოვოთ, თუ არის ახალ სტრიქონზე გადასვლა. ეს ასეც იმუშავებს:

```
alert( 'Hello')
alert ('Word!')
```

ამ შემთხვევაში, JavaScript სტრიქონის გადატანის ინტერპრეტირებას ახდენს, როგორც არ არსებული წერტილ-მძიმის სახით. ამას წერტილ-მძიმის ავტომატური ჩასმა ჰქვია. უმეტეს შემთხვევაში, ახალი სტრიქონი წერტილ-მძიმეს გულისხმობს. მაგრამ „უმეტეს შემთხვევაში" ეს ყოველთვის არ მუშაობს!

ზოგიერთ სიტუაციაში, ახალი სტრიქონი არ ნიშნავს წერტილ-მძიმეს. მაგალითად:

```
alert(3 +
1
+ 2);
```

კოდი ამ შემთხვევაში გამოიტანს 6-ს, რადგან JavaScript აქ არ გამოიყენებს წერტილ-მძიმეს. ინტუიციურად აშკარაა, რომ თუ სტრიქონი მთავრდება „+"-ით, მაშინ ეს არის დაუმთავრებელი გამოსახულება და ამიტომ არ არის საჭირო წერტილ-მძიმე. ამ შემთხვევაში, ყველაფერი იმუშავებს ისე, როგორც დაგეგმილი იყო.

მაგრამ არის სიტუაციები, როდესაც JavaScript-ს „ავიწყდება" წერტილ-მძიმის ჩასმა, სადაც ეს საჭიროა.

შეცდომები, რომლებიც ამ შემთხვევაში ჩნდება, საკმაოდ რთული გამოსავლენი და გამოსასწორებელია.

კომენტარები

დროთა განმავლობაში, პროგრამები უფრო და უფრო რთული ხდება. საჭიროა კომენტარების დამატება, რომლებიც აღწერს რას აკეთებს კოდი და რატომ.

კომენტარები შეიძლება განთავსდეს სკრიპტის ნებისმიერ ადგილას. ისინი გავლენას არ ახდენენ მის შესრულებაზე, რადგან ძრავა უბრალოდ უგულებელყოფს მათ.

ერთსტრიქონიანი კომენტარები იწყება ორმაგი დახრილი ხაზით //.

სტრიქონის ნაწილი //-ის შემდეგ ითვლება კომენტარად. ასეთმა კომენტარმა შეიძლება დაიკავოს მთელი სტრიქონი ან განთავსდეს ინსტრუქციის შემდეგ:

```
// ეს კომენტარი მთელ ხაზს იკავებს
alert( 'Hello')
alert ('Word!') // ეს კომენტარი მოსდევს ინსტრუქციას
```

მრავალსტრიქონიანი კომენტარები იწყება დახრილი ხაზით, რასაც მოჰყვება ვარსკვლავი /* და მთავრდება ვარსკვლავით, რასაც მოჰყვება ისევ დახრილი ხაზი */.

```
/* მაგალითი ორი შეტყობინებით.
ეს არის მრავალსტრიქონიანი კომენტარი.
*/
alert( 'Hello')
alert ('Word!')
```

კომენტარის შინაარსის იგნორირება ხდება, ამიტომ თუ კოდს /* ... */ შორის მოვათავსებთ, ის არ შესრულდება. ეს მოსახერხებელია, როდესაც კოდის ნაწილის დროებითი გამორთვა გვსურს, მაგალითად:

```
/* კოდი მოთავსებულია კომენტარში
alert('Hello')
*/
alert ('Word!')
```

კომენტარები ზრდის კოდის ზომას, მაგრამ ეს არ არის პრობლემა. მრავალი ინსტრუმენტი არსებობს, რომელიც ამცირებს კოდს სამუშაო სერვერზე განთავსებამდე. ისინი ხსნიან კომენტარებს ისე, რომ სამუშაო სკრიპტებში არ ფიგურირებდნენ. ამრიგად, კომენტარები არ აზიანებს სამუშაო კოდს.

მკაცრი რეჟიმი - „use strict"

დიდი ხნის განმავლობაში, JavaScript ვითარდებოდა უკუთავსებადობის პრობლემების გარეშე. ენას დაემატა ახალი ფუნქციები, ხოლო ძველის ფუნქციონირება არ იცვლებოდა.

ამ მიდგომის უპირატესობა ის იყო, რომ არსებული კოდი აგრძელებდა მუშაობას. ნაკლოვანება არის ის, რომ JavaScript-ის შემქმნელების მიერ მიღებული ნებისმიერი შეცდომა ან არასრულყოფილი გადაწყვეტილება სამუდამოდ დარჩა ენაში.

ეს 2009 წლამდე იყო, სანამ ECMAScript 5 (ES5) გამოჩნდა. მან ენას ახალი შესაძლებლობები დაამატა და ზოგიერთი არსებული შეცვალა. იმის უზრუნველსაყოფად, რომ ძველი კოდი მუშაობდეს ისე, როგორც ადრე, ასეთი ცვლილებები

ნაგულისხმევად არ გამოიყენება. ამიტომ ცალსახად უნდა მოვახდინოთ მისი გააქტიურება სპეციალური დირექტივით: "use strict".

დირექტივა ჰგავს სტრიქონს: "use strict" ან 'use strict'. როდესაც ეს სკრიპტის დასაწყისშია, მთელი სკრიპტი მუშაობს „თანამედროვე“ რეჟიმში.

მაგალითად:

```
"use strict";

// ეს კოდი თანამედროვე რეჟიმში მუშაობს
...
```

დარწმუნდით, რომ "use strict" სკრიპტის პირველ შესრულებად სტრიქონშია განთავსებული, წინააღმდეგ შემთხვევაში მკაცრი რეჟიმი შეიძლება არ იყოს ჩართული.

არ არსებობს მკაცრი რეჟიმის გაუქმების "no use strict" ბრძანება, რომელიც ძრავას ძველ რეჟიმში დააბრუნებს. როგორც კი ჩვენ ავირჩევთ მკაცრ რეჟიმს, შემდეგ მისი შეცვლა უკვე შეუძლებელი იქნება.

ცვლადები

JavaScript აპლიკაციას ჩვეულებრივ ინფორმაციასთან მუშაობა სჭირდება. მაგალითად:

1. ინტერნეტ-მაღაზია - შეიძლება შეიცავდეს ინფორმაციას გაყიდული პროდუქტებისა და ნაყიდი პროდუქციის კალათის შესახებ;

2. ჩატი - ინფორმაცია შეიძლება შეიცავდეს მომხმარებლებს, შეტყობინებებს და სხვა.

ამ ინფორმაციის შესანახად გამოიყენება ცვლადები.

ცვლადი ეს არის მონაცემების „სახელობითი საცავი“. ჩვენ შეგვიძლია გამოვიყენოთ ცვლადები პროდუქტების, ვიზიტორების და სხვა მონაცემების შესანახად.

JavaScript-ში ცვლადის შესაქმნელად გამოიყენეთ let საკვანძო სიტყვა.

ქვემოთ მოყვანილი ინსტრუქცია ქმნის (სხვა სიტყვებით: აცხადებს ან განსაზღვრავს) ცვლადს სახელად "data":

```
let data;
```

ახლა თქვენ შეგიძლიათ ამ ცვლადს გარკვეული მნიშვნელობა მიანიჭოთ მინიჭების ოპერატორის გამოყენებით =:

```
let data;
data = 'Hello';
```

მოცემულ ცვლადთან დაკავშირებული მონაცემის მნიშვნელობა მეხსიერების ველში ინახება. ცვლადის სახელის გამოყენებით ჩვენ შეგვიძლია მასთან წვდომა მივიღოთ:

```
let data;
```

```
data = 'Hello';
alert (data); // გამოაქვს ცვლადის მნიშვნელობა
```

ჩანაწერის შესამცირებლად შესაძლებელია ცვლადის გამოცხადება და მისთვის მნიშვნელობის მინიჭება ერთ სტრიქონში მოხდეს:

```
let data = 'Hello';  // ცვლადის გამოცხადება და მნიშვნელობის მინიჭება
alert (data);
```

ჩვენ აგრეთვე შეგვიძლია ერთ სტრიქონში რამდენიმე ცვლადის გამოცხადება მოხდეს:

```
let user = 'Giorgi', age = 25, data = 'Hello';
```

ეს მეთოდი შეიძლება უფრო მოკლე ჩანდეს, მაგრამ ჩვენ არ გირჩევთ. წაკითხვისთვის უკეთესია თუ თითოეულ ცვლადს ცალკ-ცალკე ახალ სტრიქონში გამოაცხადებთ.

მრავალსტრიქონიანი ვერსია ოდნავ გრძელია, მაგრამ ადვილად წასაკითხი:

```
let user = 'Giorgi';
let age = 25;
let data = 'Hello';
```

საკვანძო სიტყვა var

ზემოთ განხილულ თავში ცვლადების გამოცხადების გზა ვისწავლეთ.

ძველ სკრიპტებში მონაცემების გამოცხადებისათვის let საკვანძო სიტყვის ნაცვლად გამოიყენებოდა var საკვანძო სიტყვა.

```
var data = 'Hello';
```

მისი საშუალებით მონაცემის გამოცხადება მოხდება, მაგრამ ცოტათი განსხვავებული სახით.

ერთი შეხედვით, var ოპერატორი იქცევა, როგორც let. მაგალითად:

```
function sayHi() {
  var phrase = "Hello!"; // ლოკალური ცვლადი, "var" "let"-ის
     მაგივრად

  alert(phrase); // Hello
}
sayHi();

alert(phrase); // შეცდომაა: phrase არ არის განსაზღვრული
```

var-ით განსაზღვრული ცვლადებისთვის არ არის ბლოკის ფარგლები შემოსაზღვრული.

var-ით განსაზღვრული ცვლადების საზღვრები შემოიფარგლება ფუნქციით ან, თუ ცვლადი გლობალურია, სკრიპტით. ასეთი ცვლადები ხელმისაწვდომია ბლოკის გარეთ.

მაგალითად:

```
if (true) {
  var test = true; // let-ის ნაცვლად გამოვიყენეთ var-ი
}
```

```
alert(test); // true, ცვლადი არსებობს if ბლოკის გარეთაც
```

ვინაიდან var უგულებელყოფს ბლოკებს, ჩვენ მივიღეთ test გლობალური ცვლადი.

და თუ var ტესტის ნაცვლად გამოვიყენებდით let-ს, მაშინ test ცვლადი იქნებოდა ლოკალური მხოლოდ if ბლოკში:

```
if (true) {
  let test = true;
}

alert(test); // Error: test is not defined
```

ანალოგიურად, ციკლისათვის var არ განსაზღვრავს ლოკალურ ცვლადს:

```
for (var i = 0; i < 10; i++) {
  // ...
}

alert(i); // 10
```

თუ კოდის ბლოკი არის ფუნქციის შიგნით, მაშინ var ხდება ამ ფუნქციის ლოკალური ცვლადი.

თუ კოდის ბლოკში ერთსა და იგივე ცვლადს ორჯერ გამოვაცხადებთ let ოპერატორის საშუალებით, მოგვცემს შეცდომას.

```
let user;
let user; // SyntaxError: 'user' has already been declared
```

var ოპერატორი უშვებს ცვლადის განმეორებით გამოცხადებას. განმეორებითი გამოცხადება იგნორირებული იქნება.

```
var user = "გიორგი";

var user; // არაფერს არ აკეთებს, შეცდომა არ არის

alert(user); // გიორგი
```

თუ დამატებით მივანიჭებთ მნიშვნელობას, მაშინ ცვლადი მიიღებს ახალ მნიშვნელობას:

```
var user = "გიორგი";

var user = "ანა";

alert(user); // ანა
```

var ოპერატორით ცვლადის გამოცხადება მუშავდება ფუნქციის შესრულების დასაწყისში (ან სკრიპტის გაშვების დროს, თუ ცვლადი გლობალურია).

სხვა სიტყვებით რომ ვთქვათ, var-ით ცვლადები გამოცხადებულად განიხილება ფუნქციის შესრულებისას თავიდანვე, მიუხედავად იმისა, თუ სად არის რეალურად განთავსებული ფუნქციაში მათი გამოცხადება (იმ პირობით, რომ ისინი არ არიან ერთმანეთში ჩადგმულ ფუნქციაში).

ცვლადის გამოცხადება მუშავდება ფუნქციის შესრულების დასაწყისში („ამოცურდება“), მაგრამ მნიშვნელობის მინიჭება

ყოველთვის ხდება კოდის იმ სტრიქონში, სადაც ისაა მითითებული.

ცვლადის სახელი

JavaScript-ს ცვლადის სახელთან დაკავშირებით ორი შეზღუდვა აქვს:

1. ცვლადის სახელი უნდა შეიცავდეს მხოლოდ ასოებს, ციფრებს ან სიმბოლოებს $ (დოლარის ნიშანი) და _ (ქვედა ხაზი);

2. პირველი სიმბოლო არ უნდა იყოს რიცხვი.

სწორი სახელების მაგალითები:

```
let UserName;
let test123;
```

თუ სახელი შეიცავს რამდენიმე სიტყვას და სიტყვები მიჰყვება ერთმანეთის მიყოლებით, ყოველი შემდეგი სიტყვა სასურველია იწყებოდეს დიდი ასოებით, მაგალითად: myVeryLongName.

საინტერესო ის არის, რომ დოლარის ნიშანი "$" და ქვედა ხაზი "_" ასევე შეიძლება გამოყენებულ იქნას ცვლადის სახელებში. ისინი ჩვეულებრივი სიმბოლოებია, ასოებივით, განსაკუთრებული მნიშვნელობის გარეშე. ეს სახელები შეიძლება ასე იყოს გამოყენებული:

```
let $ = 1; // გამოცხადებულია ცვლადი სახელით "$"
let _ = 2; // აქ კი ცვლადი სახელით "_"

alert ($ + _); // შედეგი იქნება 3-ის ტოლი
```

ცვლადის გამოცხადების დროს არსებითი მნიშვნელობა აქვს რეგისტრს, ცვლადები სახელით Name და name - ეს ორი სხვადასხვა ცვლადია. არა ლათინური ასოების გამოყენება დაშვებულია, მაგრამ არ არის რეკომენდებული. შეიძლება გამოყენებული იყოს ნებისმიერი ენა, მაგალითად:

```
let სახელი;
სახელი='Hello, Word!';
```

ცვლადის სახელად არ შეიძლება გამოყენებული იყოს ენის დარეზერვირებული (საკვანძო) სიტყვები. დარეზერვირებული სიტყვები მოცემულია 1-ელ ცხრილში.

ცხრილი 1

abstract	continue	false	in	short	true
boolean	debugger	final	instanceof	super	try
break	default	finally	int	switch	typeof
byte	delete	float	let	Synchro-nized	var
case	do	for	long	this	void
catch	double	function	native	throw	while
char	else	goto	new	volatile	with
class	export	if	null	transient	yield
const	extends	import	return		

ზოგჯერ ცვლადის სახელის პირველ სიმბოლოდ იყენებენ ასოს, რომელიც ამ ცვლადის მონაცემთა (მნიშვნელობის) ტიპს მიუთითებს: c – სტრიქონული (character), n – რიცხვითი (number), b – ლოგიკური (boolean), o – ობიექტი (object), a – მასივი (array).

პროგრამაში ცვლადის შექმნა რამდენიმე ხერხით შეიძლება:

ცვლადის შექმნა მკაცრი გამოყენების გარეშეც შეიძლება, მაგრამ ძველ ვერსიებში ტექნიკურად შესაძლებელი იყო ცვლადის შექმნა, უბრალოდ მნიშვნელობის მინიჭებით let-ის გამოყენების გარეშე. ეს ვარიანტი ახლაც მუშაობს.

შენიშვნა: ქვემოთ მაგალითში არ არის მოცემული ცვლადის შექმნა მკაცრი გამოყენებით.

```
num = 5; // თუ ცვლადი "num" აქამდე არ არსებობდა, იქმნება ავტომატურად
alert (num); // გამოიტანს შედეგს 5-ს
```

ასეთ შემთხვევაში სჯობს მაინც გამოვიყენოთ ცვლადის შექმნის მკაცრი რეჟიმი, რადგან თუ ცვლადი წინასწარ არ არის განსაზღვრული, ამან შეიძლება ზოგიერთ ბრაუზერში შეცდომა გამოიწვიოს.

მუდმივები

კონსტანტის გამოსაცხადებლად, let-ის ნაცვლად გამოიყენება const. const-ით გამოცხადებულ ცვლადებს მუდმივა ეწოდება. მათი შეცვლა შეუძლებელია. ამის გაკეთების მცდელობა გამოიწვევს შეცდომას, მაგალითად:

```
const pi=3.14;
```

თუ პროგრამისტი დარწმუნებულია, რომ ცვლადი არასოდეს შეიცვლება და აქვს ამის გარანტია, მაშინ მას შეუძლია ეს მუდმივა გამოაცხადოს const-ის გამოყენებით.

ჩვეულებრივი პრაქტიკაა მუდმივების გამოყენება ძნელად დასამახსოვრებელი მნიშვნელობებისთვის, რომლებიც ცნობილია სკრიპტის შესრულებამდე.

ასეთი მუდმივების სახელების ჩასაწერად ცდილობენ გამოიყენონ ზედა რეგისტრის ასოები და ქვედა ხაზი.

მაგალითად, მოდით გავაკეთოთ მუდმივები სხვადასხვა ფერისთვის "თექვსმეტობით ფორმატში":

```
const COLOR_RED = "#FF0000";
const COLOR_GREEN = "#00FF00";
const COLOR_BLUE = "#0000FF";
```

მონაცემები JavaScript-ში

JavaScript-ში მნიშვნელობა ყოველთვის ეხება გარკვეული ტიპის მონაცემებს. მაგალითად, ეს შეიძლება იყოს სტრიქონი ან რიცხვი. JavaScript-ში მონაცემთა რვა ძირითადი ტიპია. ზოგადად განვიხილოთ თითოეული მათგანი.

JavaScript-ში ცვლადი შეიძლება შეიცავდეს ნებისმიერ მონაცემს. მაგალითად, პირველად შეიძლება ის იყოს სტრიქონი, ხოლო მეორეში - რიცხვი:

```
let st = "Hello";
st = 123456;
```

ასეთი ქმედება შეცდომას არ გამოიწვევს.

ეს ნიშნავს, რომ პროგრამირების ენაში არსებობს მონაცემთა ტიპები, მაგრამ ცვლადები მიბმული არ არის არცერთ მათგანთან.

რიცხვითი ტიპი

რიცხვითი მონაცემთა ტიპი (number) წარმოადგენს როგორც მთელ, ასევე მცურავმძიმიან რიცხვებს. რიცხვებზე ბევრი ოპერაცია სრულდება, როგორიცაა გამრავლება - *, გაყოფა - /, შეკრება - +, გამოკლება - - და ა.შ.

```
let num = 417;
num = 2.345;
```

ჩვეულებრივი რიცხვების გარდა, არსებობს ეგრეთ წოდებული „სპეციალური რიცხვითი მნიშვნელობები“,

რომლებიც მიეკუთვნება რიცხვითი ტიპის მონაცემებს: Infinity, -Infinity და NaN.

Infinity წარმოადგენს მათემატიკურ უსასრულობას ∞. ეს არის სპეციალური მნიშვნელობა, რომელიც აღემატება ნებისმიერ რიცხვს.

ის ჩვენ შეგვიძლია მივიღოთ ნულზე გაყოფის შედეგად ან პირდაპირ მივანიჭოთ Infinity მნიშვნელობა, მაგალითად:

```
alert( 1 / 0 ); // შედეგად მიიღება Infinity
alert( Infinity ); // აქ პირდაპირ ვანიჭებთ Infinity მნიშვნელობას
```

NaN (Not a Number) ნიშნავს გამოთვლით შეცდომას. ეს არის არასწორი ან განუსაზღვრელი მათემატიკური ოპერაციის შედეგი. ნებისმიერი ოპერაცია NaN-ზე აბრუნებს NaN-ს. თუ მათემატიკური გამოსახულებაში სადმე არის NaN, მაშინ მისი მონაწილეობით გამოთვლების შედეგიც იქნება NaN. მაგალითად:

```
alert( "არა რიცხვითი გამოსახულება" / 2 ); // NaN, ასეთი გაყოფა შეცდომაა
alert( NaN / 2 + 5 ); // შედეგიც იქნება NaN
```

მათემატიკური ოპერაციები JavaScript-ში არის „უსაფრთხო“. ჩვენ შეგვიძლია ყველაფერი გავაკეთოთ: გავყოთ ნულზე, მივიჩნიოთ არა რიცხვითი სტრიქონები რიცხვებად და ა.შ. სცენარის მუშაობა ამით არასოდეს შეჩერდება ფატალური შეცდომით. უარეს შემთხვევაში შესრულების შედეგად მივიღებთ NaN-ს.

სპეციალური რიცხვითი მნიშვნელობები ასევე მიეკუთვნება რიცხვით ტიპს. რა თქმა უნდა, ეს არ არის რიცხვები ამ სიტყვის ჩვეულებრივი გაგებით.

დაწვრილებით რიცხვების შესახებ იხილეთ თავი „მონაცემთა ტიპები“.

BigInt

JavaScript-ში „რიცხვის“ ტიპი არ შეიძლება იყოს (2^{53}-1)-ზე (ე.ი. 9007199254740991) დიდი რიცხვი ან -(2^{53}-1)-ზე ნაკლები უარყოფითი რიცხვებისთვის. ეს ტექნიკური შეზღუდვა გამოწვეულია მათი შიდა წარმოდგენით. უმეტეს შემთხვევაში ეს საკმარისია. მაგრამ ზოგჯერ ჩვენ გვჭირდება მართლაც გიგანტური რიცხვები, მაგალითად, კრიპტოგრაფიაში ან დროის აღრიცხვისათვის მიკროწამებში.

BigInt ტიპი დაემატა JavaScript-ს, რათა შესაძლებელი ყოფილიყო ნებისმიერ მთელ რიცხვებთან მუშაობა.

BigInt მნიშვნელობის შესაქმნელად, რიცხვის ბოლოს უნდა დაამატოთ n, მაგალითად:

```
const big = 1234567890123456789012345678901234567890n;
```

ამჟამად, BigInt მხარს უჭერს მხოლოდ Firefox, Chrome, Edge და Safari ბრაუზერები.

სტრიქონული ტიპი

სტრიქონული ცვლადის (string) მნიშვნელობა JavaScript-ში მოთავსებული უნდა იყოს ბრჭყალებში. JavaScript-ში გამოიყენება როგორც ორმაგი (") ასევე ერთმაგი (' - აპოსტროფი)

ბრჭყალები. ამ ორ ბრჭყალს შორის, გამოყენების თვალსაზრი-სით, არავითარი სხვაობა არ არის.

```
let str1 = "Hello";
let str2 = 'Hello';
```

განსაკუთრებული მნიშვნელობა ენიჭება რიცხვითი და სტრიქონული მონაცემების გარჩევას. თუ ციფრებისაგან ჩაწერილი მონაცემი მოთავსებულია ბრჭყალებში, მაშინ იგი მიეკუთვნება სტრიქონული ტიპის მონაცემს და მასზე არითმეტიკული ოპერაციების ჩატარება არ შეიძლება. აქვე უნდა შევნიშნოთ, რომ სტრიქონს "", რომელიც არ შეიცავს არც ერთ სიმბოლოს, ეწოდება ცარიელი, ხოლო სტრიქონი, რომელშიც არის ერთი ჰარი (ინტერვალი) მაინც (მაგალითად, " "), არ არის ცარიელი.

ზოგიერთ ენას, როგორიცაა C და Java, აქვს ცალკე ტიპი ერთი სიმბოლოს შესანახად, როგორიცაა "A" ან "&". C და Java-ში ეს არის char. JavaScript-ში ერთი სიმბოლოსთვის ცალკეული მონაცემთა ტიპი არ არსებობს, მხოლოდ სტრიქონის ტიპი. სტრიქონი შეიძლება შეიცავდეს ნულ სიმბოლოს (იყოს ცარიელი), ერთ სიმბოლოს ან ბევრ სიმბოლოს.

JavaScript ენაზე პროგრამის დაწერის პროცესში პროგრამისტმა ყურადღება უნდა მიაქციოს მონაცემთა ტიპებს. წინააღმდეგ შემთხვევაში, ინტერპრეტატორი შეცდომას არ დააფიქსირებს და ეცდება მითითებული ოპერაციის შესასრულებლად ეს მონაცემი ამა თუ იმ ტიპს მიაკუთვნოს.

იმ შემთხვევაში, თუ ტექსტში საჭიროა სპეციალური სიმბოლოების გამოტანა, გამოიყენება სიმბოლო „\“ (საწინააღმდეგოდ დახრილი ხაზი).

მაგალითად, თუ გვსურს ბრჭყალებთან ერთად გამოვიტანოთ შემდეგი სახის ტექსტი: სააქციო საზოგადოება „აზოტი“, მაშინ ეს ტექსტი ასე უნდა ჩაიწეროს: "სააქციო საზოგადოება \"აზოტი\" ".

იგივე ამოცანა შეიძლება გადაწყდეს, თუ გარე და შიგა ბრჭყალებად გამოვიყენებთ სხვადასხვა ტიპის ბრჭყალებს (ერთმაგი და ორმაგი). მაგალითად, 'სააქციო საზოგადოება "აზოტი" '.

შენიშვნა:

1. სტრიქონული მონაცემების შემოსაზღვრისათვის გამოყენებული ბრჭყალები ერთი და იმავე ტიპის უნდა იყოს და დაწყვილებული;

2. ერთი ტიპის ბრჭყალებში მოთავსებული სტრიქონის შიგნით შეიძლება მხოლოდ სხვა ტიპის ბრჭყალები გამოვიყენოთ (წინააღმდეგ შემთხვევაში ბრაუზერი მოგვცემს შეტყობინებას შეცდომის შესახებ, ან არასწორად აღიქვამს მონაცემებს).

დაწვრილებით სტრიქონული მონაცემების შესახებ იხილეთ თავი „მონაცემთა ტიპები“.

ბულის (ლოგიკური) ტიპი

ლოგიკურ ანუ ბულის (boolean) ტიპის მონაცემს შეუძლია მიიღოს მხოლოდ ორი მნიშვნელობა: true (ჭეშმარიტი) ან false (მცდარი); ჩვეულებრივ, ეს მნიშვნელობები მიიღება ორი გამოსახულების შედარების ან ლოგიკური ოპერაციების (ლოგიკური უარყოფა – NOT, ლოგიკური „და“ – AND, ლოგიკური „ან“ – OR) ჩატარების შედეგად. ლოგიკური ტიპის

მონაცემებს ხშირად ბულის ტიპის მონაცემების სახელწოდებით მოიხსენიებენ, რომელიც ინგლისელი მათემატიკოსის ჯონ ბულის საპატივცემულოდ ეწოდა. მან ლოგიკური სიდიდეებისათვის შექმნა ე.წ. ბულის ალგებრა.

მნიშვნელობა null

სპეციალური მნიშვნელობა null არ მიეკუთვნება არც ერთ ზემოთ აღწერილ ტიპს. ის ქმნის ცალკეულ ტიპს, რომელიც შეიცავს მხოლოდ ერთ null მნიშვნელობას:

```
let age = null;
```

JavaScript-ში null არ „მიანიშნებს არარსებულ ობიექტზე“ ან „ნულოვანი მაჩვენებელი“, როგორც ეს ზოგიერთ სხვა ენაშია. ეს მხოლოდ სპეციალური მნიშვნელობაა, რომელიც წარმოადგენს „არაფერს“, „ცარიელ მნიშვნელობას“ ან „უცნობ მნიშვნელობას“.

ზემოთ მოყვანილი კოდი მიუთითებს, რომ age ცვლადის მნიშვნელობა უცნობია.

მნიშვნელობა undefined

სპეციალური მნიშვნელობა undefined ასევე განუსაზღვრელი მნიშვნელობაა. ის აყალიბებს ტიპს თავის თავისგან ისე, როგორც null. undefined ნიშნავს, რომ "მნიშვნელობა არ არის მინიჭებული".

თუ ცვლადი გამოცხადებულია, მაგრამ არ არის მინიჭებული რაიმე მნიშვნელობა, მაშინ მისი მნიშვნელობა იქნება undefined. მაგალითად:

```
let age;

alert(age); // გამოიტანს მნიშვნელობას "undefined"
```

ტექნიკურად ჩვენ შეგვიძლია undefined მნიშვნელობა ნებისმიერ ცვლადს მივანიჭოთ:

```
let age = 123;
// ცვლადის მნიშვნელობა შევცვალოთ undefined-ით
age = undefined;

alert(age); // გამოიტანს მნიშვნელობას "undefined"
```

მაგრამ ამის გაკეთება რეკომენდებული არ არის. როგორც წესი, null გამოიყენება ცვლადისთვის „ცარიელი“ ან „უცნობი“ მნიშვნელობის მინიჭებისთვის, ხოლო undefined გამოიყენება იმის შესამოწმებლად, მინიჭებული აქვს თუ არა ცვლადს მნიშვნელობა.

ობიექტები და სიმბოლოები

object (ობიექტი) ტიპი განსაკუთრებულია. ყველა სხვა ტიპს უწოდებენ „პრიმიტიულს“, რადგან მათი მნიშვნელობები შეიძლება იყოს მხოლოდ მარტივი მნიშვნელობები (იქნება ეს სტრიქონი, რიცხვი თუ სხვა რამ). ობიექტები ინახავს მონაცემთა კოლექციებს ან უფრო რთულ სტრუქტურებს. ობიექტებს ენაში მნიშვნელოვანი ადგილი უკავია და განსაკუთრებულ ყურადღებას მოითხოვს.

symbol (სიმბოლო) ტიპი გამოიყენება ობიექტებში უნიკალური იდენტიფიკატორების შესაქმნელად.

ოპერატორი typeof

ოპერატორი typeof აბრუნებს არგუმენტის ტიპს. ეს გამოიყენება, მაშინ როდესაც გვსურს სხვადასხვა მნიშვნელობების ტიპის დადგენა, ან უბრალოდ გვსურს შემოწმება.

მას აქვს ორი სინტაქსური ფორმა:

1. ოპერატორის სინტაქსი: typeof x.

2. ფუნქციის სინტაქსი: typeof(x).

სხვა სიტყვებით რომ ვთქვათ, ის მუშაობს ფრჩხილებით ან მის გარეშე. შედეგი ორივე შემთხვევაში ერთი და იგივეა.

typeof x-ზე მიმართვის შედეგად აბრუნებს სტრიქონს ტიპის სახელით:

```
typeof undefined // "undefined"
typeof 0 // "number"
typeof 10n // "bigint"
typeof true // "boolean"
typeof "foo" // "string"
typeof Symbol("id") // "symbol"
typeof Math // "object"  (1)
typeof null // "object"  (2)
typeof alert // "function"  (3)
```

ბოლო სამი სტრიქონი ახსნას საჭიროებს:

1. Math (მათემატიკა)არის ჩაშენებული ობიექტი, რომელიც უზრუნველყოფს მათემატიკურ ოპერაციებს და მუდმივებს;

2. null ტიპის გამოძახების შედეგი არის „object“. ეს არის typeof-ში ოფიციალურად აღიარებული შეცდომა, რომელიც

JavaScript-ის შექმნის დროიდან არსებობს. რა თქმა უნდა, null არ არის ობიექტი. ეს არის ცალკე ტიპის სპეციალური მნიშვნელობა;

3. typeof alert გამოძახება შედეგად აბრუნებს „function“-ს, რადგან alert არის ფუნქცია. ჩვენ ფუნქციებს შემდეგ თავებში განვიხილავთ, სადაც ასევე დავინახავთ, რომ JavaScript-ში არ არის სპეციალური ტიპი „ფუნქცია“. ფუნქციები მიეკუთვნება ობიექტის ტიპს, მაგრამ typeof მათ განსაკუთრებულად ამუშავებს და შედეგად აბრუნებს „function“-ს. ესეც JavaScript-ის შექმნიდან არსებობს. ფორმალურად, მართალია ეს არასწორია, მაგრამ პრაქტიკაში მოსახერხებელია.

მონაცემების შეტანა-გამოტანა

ვინაიდან ჩვენ გამოვიყენებთ ბრაუზერს, როგორც დემო გარემოს, უნდა გავეცნოთ მისი ინტერფეისის რამდენიმე მახასიათებელს, კერძოდ, alert, prompt და confirm.

alert

ჩვენ უკვე ვიცნობთ ამ ფუნქციას. ის გვიჩვენებს შეტყობინებას და ელოდება მომხმარებლის მიერ OK ღილაკზე მაუსის დაწკაპუნებას. მაგალითად:

```
alert ("Hello");
```

შეტყობინება შეიძლება იყოს ნებისმიერი ტიპის მონაცემი: სიმბოლოების მიმდევრობა მოთავსებული ორმაგ ან ერთმაგ ბრჭყალებში, რიცხვი, ცვლადი, ან გამოსახულება.

ამ პატარა შეტყობინების ფანჯარას ეწოდება მოდალური ფანჯარა. მოდალური კონცეფცია ნიშნავს, რომ მომხმარებელს არ შეუძლია ურთიერთქმედება გვერდის დანარჩენ ინტერფეისთან, დააწკაპუნოს სხვა ღილაკებზე და ა.შ. რადგანაც ის ურთიერთქმედებს მოდალურ ფანჯარასთან, ვიდრე OK ღილაკზე მაუსით არ დააწკაპუნებს.

prompt

prompt ფუნქციას აქვს ორი არგუმენტი:

```
res = prompt(title, [default]);
```

ამ კოდის შესრულების შედეგად ეკრანზე გამოჩნდება მოდალური ფანჯარა ტექსტით, ტექსტის შეყვანის ველით და OK/Cancel ღილაკებით. მისი პარამეტრებია:

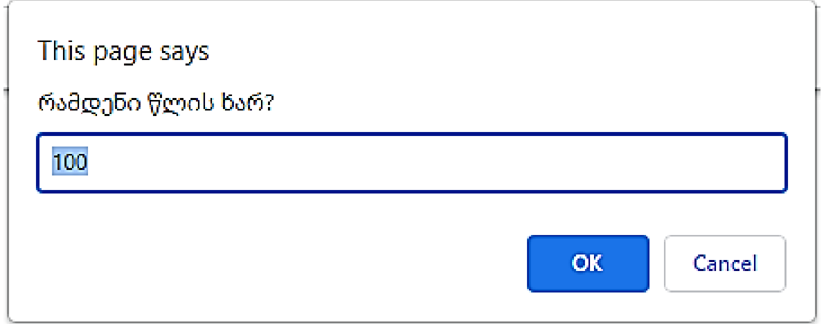
- title - ტექსტი, რომელიც გამოჩნდება ფანჯარაში;
- default - არასავალდებულო მეორე პარამეტრი (კვადრატული ფრჩხილები default პარამეტრის გარშემო გვიჩვენებს, რომ მოცემული პარამეტრი არასავალდე-ბულოა), რომელიც ფანჯარაში გვიჩვენებს ტექსტური ველის საწყის მნიშვნელობას.

მომხმარებელს შეუძლია რაღაც აკრიფოს ტექსტის შეყვანის ველში და დააჭიროს OK ღილაკს. შეყვანილი ტექსტი მიენიჭება res ცვლადს. მომხმარებელს ასევე შეუძლია გააუქმოს ჩანაწერი Cancel ღილაკზე ან <Esc> კლავიშზე ხელის დაჭერით. ამ შემთხვევაში res-ის მნიშვნელობა იქნება null.

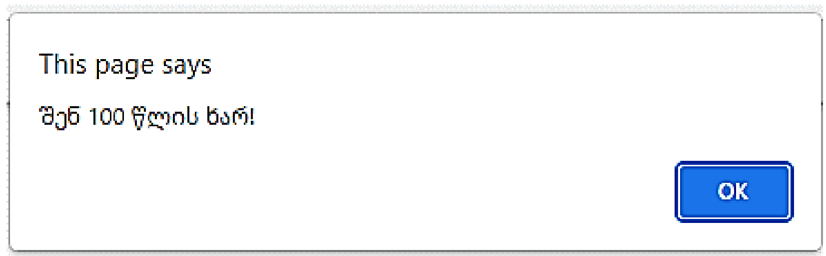
მაგალითად:

```
let age = prompt('რამდენი წლის ხარ?', 100);
alert(‘შენ ${age} წლის ხარ!’);
```

ამ კოდის შესრულების შედეგი იქნება:

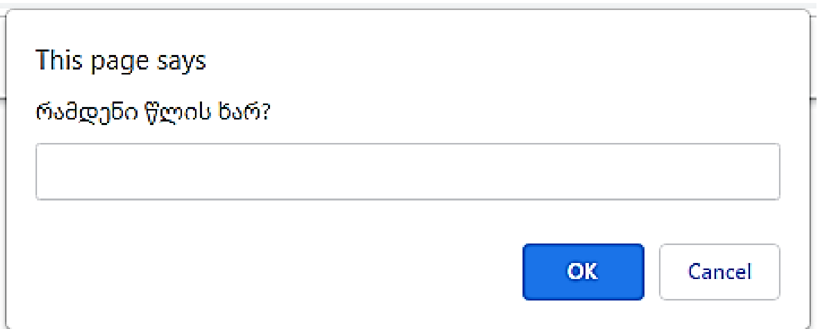


OK ღილაკზე ხელის დაჭერით მივიღებთ შემდეგ შედეგს:



ხოლო Cancel ღილაკზე ხელის დაჭერით მივიღებთ შედეგს:

თუ prompt ფუნქციაში არ არის მითითებული მეორე პარამეტრი, მაშინ მოდალურ ფანჯარას ექნება შემდეგი სახე:



confirm

მისი სინტაქსია:

```
res = confirm(question);
```

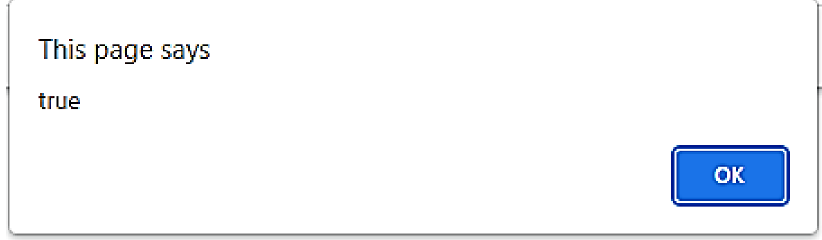
confirm ფუნქციის შესრულების შედეგად ეკრანზე გაიხსნება მოდალური ფანჯარა question შეკითხვის ტექსტით და OK/Cancel ღილაკებით. შედეგად მიიღება ისევ მოდალური ფანჯარა, რომელშიც OK ღილაკზე მაუსის დაწკაპუნების შედეგად მიიღება true, ხოლო Cancel ღილაკზე მაუსის დაწკაპუნების შედეგად კი false.

მაგალითად:

```
let isBoss = confirm("You are the Boss?");
alert( isBoss );
```

ამ კოდის შესრულების შედეგი იქნება:

OK ღილაკზე ხელის დაჭერით მივიღებთ შემდეგ შედეგს:



Cancel ღილაკზე ხელის დაჭერით მივიღებთ შედეგს:

This page says
false
OK

მონაცემთა გარდაქმნები

ხშირად, ოპერატორები და ფუნქციები მათზე გადაცემულ მნიშვნელობებს ავტომატურად ანიჭებენ სასურველ ტიპს.

მაგალითად, alert ფუნქცია ავტომატურად გადააქცევს ნებისმიერ მნიშვნელობას სტრიქონად. მათემატიკური ოპერატორები გარდაქმნის მნიშვნელობებს რიცხვებად. ასევე არის შემთხვევები, როდესაც ჩვენ გვჭირდება მნიშვნელობები გარდავქმნათ სასურველ ტიპად.

სტრიქონის გარდაქმნები

სტრიქონის გარდაქმნა ხდება მაშინ, როდესაც საჭიროა რაიმე მონაცემის სტრიქონის სახით წარმოდგენა.

მაგალითად, alert(value) მნიშვნელობას სტრიქონად გარდაქმნის.

ჩვენ ასევე გარდაქმნისათვის შეგვიძლია გამოვიყენოთ String(value) ფუნქცია:

```
let value = true;
alert(typeof value); // boolean
value = String(value); // ახლა value "true" სტრიქონია
alert(typeof value); // string
```

გარდაქმნა აშკარად ხდება. false ხდება "false", null ხდება "null" და ა.შ.

რიცხვითი გარდაქმნები

რიცხვითი გარდაქმნები ხდება მათემატიკურ ფუნქციებსა და გამოსახულებებში.

მაგალითად, როდესაც გაყოფის ოპერაცია „/“ გამოიყენება რიცხვის გარდა სხვა რამეზე:

```
alert( "6" / "2" ); // 3, სტრიქონი გარდაიქმნება რიცხვად
```

ჩვენ შეგვიძლია გამოვიყენოთ Number(value) ფუნქცია მნიშვნელობის რიცხვად გარდაქმნისთვის:

```
let str = "123";
alert(typeof str); // string
let num = Number(str); // გარდაიქმნა რიცხვად 123
alert(typeof num); // number
```

პირდაპირი გარდაქმნა ხშირად გამოიყენება, როდესაც ველოდებით სტრიქონიდან რიცხვის მიღებას, როგორიცაა ფორმის ტექსტური ველიდან.

თუ სტრიქონი ცალსახად ვერ გადაიქცევა რიცხვად, მაშინ გარდაქმნის შედეგი არის NaN. მაგალითად:

```
let age = Number("ნებისმიერი სტრიქონი");

alert(age); // NaN, გარდაქმნა შეუძლებელია
```

რიცხვითი გარდაქმნების წესი:

მნიშვნელობა	გარდაქმნის შემდეგ მნიშვნელობა
undefined	NaN
null	0
true / false	1 / 0

string	ჰარი თავში და ბოლოში უქმდება. თუ ცარიელი სტრიქონი რჩება, მაშინ ენიჭება მნიშვნელობა 0, არაცარიელი სტრიქონისაგან მიიღება რიცხვი. შეცდომის შემთხვევაში მიიღება NaN.

მაგალითი:

```
alert( Number("   523   ") ); // 523
alert( Number("523z") );      // NaN (რიცხვის წაკითხვის შეცდომა, რიცხვის ადგილზე დგას "z")
alert( Number(true) );        // 1
alert( Number(false) );       // 0
```

ლოგიკური გარდაქმნები

ლოგიკური გარდაქმნები ყველაზე მარტივია. გარდაქმნები ლოგიკური ოპერაციების დროს ხდება, მაგრამ ასევე შეიძლება შესრულდეს Boolean(value) ფუნქციის საშუალებით.

გარდაქმნის წესი ასეთია:

• მნიშვნელობები, რომლებიც ინტუიციურად „ცარიელია", როგორიცაა რიცხვი 0, ცარიელი სტრიქონი, null, undefined და NaN, არის false;

• ყველა სხვა მნიშვნელობისათვის არის true.

მაგალითად:

```
alert( Boolean(1) ); // true
alert( Boolean(0) ); // false
alert( Boolean("Hello!") ); // true
alert( Boolean("") ); // false
```

გაითვალისწინეთ, რომ სტრიქონი მნიშვნელობით "0" არის true.

ზოგიერთი ენა (მაგ. PHP) განიხილავს სტრიქონს მნიშვნელობით "0", როგორც false. მაგრამ JavaScript-ში, თუ სტრიქონი ცარიელი არ არის, მაშინ ის ყოველთვის არის true.

```
alert( Boolean("0") ); // true
alert( Boolean(" ") ); // ჰარიც არის true (ნებისმიერი არაცარიელი სტრიქონი არის true)
```

მათემატიკური საბაზო ოპერატორები

ბევრი მათემატიკური ოპერატორი ჩვენთვის სკოლის კურსიდანაა ცნობილი: შეკრება +, გამოკლება -, გამრავლება *, გაყოფა / და ა.შ.

პირველ რიგში დავაზუსტოთ ზოგიერთი ტერმინი:

- ოპერანდი - ეს არის რომლის მიმართაც გამოიყენება ოპერატორი. მაგალითად, ნამრავლში 7 * 3 არის ორი ოპერანდი, მარცხნივ მდგომი ოპერანდი 7-ის ტოლია, ხოლო მარჯვნივ მდგომი კი 3-ის. ზოგჯერ ოპერანდს არგუმენტს უწოდებენ;
- უნარული ოპერატორი ეწოდება ისეთ ოპერატორს, რომელიც მხოლოდ ერთი ოპერანდის მიმართ გამოიყენება. მაგალითად, უნარული მინუს „ – “ ოპერატორი ცვლის რიცხვის ნიშანს საწინააღმდეგო ნიშნით:

```
let x = 1;
x = -x;
```

```
alert( x ); // -1, უნარული მინუს ოპერატორის გამოყენებით
```

- ბინარული ეწოდება ოპერატორს, რომელიც ორი ოპერანდის მიმართ გამოიყენება. იგივე მინუსი ბინარული ფორმისაც არსებობს:

```
let x = 1, y = 3;
alert( y - x ); // 2, ბინარული მინუსი იძლევა სხვაობას
```

ფორმალურად, ბოლო მაგალითებში ჩვენ ორი სხვადასხვა ოპერატორი ვნახეთ, რომელშიც ერთი და იგივე სიმბოლო გამოიყენება: უარყოფის ოპერატორი და სხვაობის (გამოკლების) ოპერატორი.

JavaScript-ი მხარს უჭერს შემდეგ მათემატიკურ ოპერატორებს:

- შეკრება +;
- გამოკლება -;
- გამრავლება *;
- გაყოფა /;
- მოდულით გაყოფა %;
- ხარისხში აყვანა **.

პირველი ოთხი ოპერატორი ყველასთვის ცნობილია.

ოპერატორი მოდულით გაყოფის - % შედეგი იქნება პირველი რიცხვის მეორეზე გაყოფის ნაშთი და არავითარი კავშირი არა აქვს პროცენტთან. მაგალითად:

```
alert( 5 % 2 ); // 1, 5-ის 2-ზე გაყოფის ნაშთი
alert( 8 % 3 ); // 2, 8-ის 3-ზე გაყოფის ნაშთი
```

ხარისხში აყვანის ოპერატორი - **. გამოსახულებაa a ** b, a რიცხვი მრავლდება თავის თავზე b-ჯერ. მაგალითი:

```
alert( 2 ** 2 ); // 4  (2 მრავლდება თავის თავზე 2-ჯერ)
alert( 2 ** 3 ); // 8  (2 * 2 * 2, 3-ჯერ)
alert( 2 ** 4 ); // 16 (2 * 2 * 2 * 2, 4-ჯერ)
```

ეს ოპერატორი ასევე მუშაობს წილადი რიცხვებისათვისაც. მაგალითად, კვადრატული ფესვი წარმოადგენს 1/2 ხარისხში აყვანას:

```
alert( 4 ** (1/2) ); // 2 (1/2 კვადრატული ფესვის ეკვივალენტურია)
alert( 8 ** (1/3) ); // 2 (1/3 კუბური ფესვის ეკვივალენტურია)
```

სტრიქონების შეკრება ბინარული +-ის საშუალებით

ახლა განვიხილოთ JavaScript ოპერატორების სპეციალური შესაძლებლობები, რომლებიც სკოლის არითმეტიკის ფარგლებს სცილდება. ჩვეულებრივ, „+“-ის დახმარებით ხდება რიცხვების შეკრება, მაგრამ თუ ბინარული ოპერატორი „+“ გამოიყენება სტრიქონებზე, მაშინ ის მათ აერთიანებს, მაგალითად:

```
let s = "Hello" + "World!";
alert(s); // HelloWorld!
```

გაითვალისწინეთ, რომ თუ ერთი ოპერანდი მაინც არის სტრიქონი, მაშინ მეორეც სტრიქონად გარდაიქმნება.

მაგალითად:

```
alert( '1' + 2 ); // შედეგი იქნება "12"
alert( 2 + '1' ); // შედეგი იქნება "21"
```

როგორც ხედავთ, არ აქვს მნიშვნელობა პირველი თუ მეორე ოპერანდი არის სტრიქონი. ახლა განვიხილოთ უფრო რთული მაგალითი:

```
alert(2 + 2 + '1' ); // მივიღებთ შედეგს "41", და არა "221"
```

აქ ოპერატორები სათითაოდ მუშაობენ. პირველი პლუსი შეკრებს ორ რიცხვს და აბრუნებს შედეგს 4-ს, შემდეგ შემდეგი პლუსი აერთიანებს შედეგს სტრიქონთან, შესრულდება 4 + '1' ოპერაცია და მიიღება შედეგი 41.

სტრიქონების შეკრება და ტრანსფორმაცია ბინარული პლუსის თვისებაა. სხვა არითმეტიკული ოპერატორები მხოლოდ რიცხვებზე მოქმედებენ და ოპერანდებს ყოველთვის რიცხვებად გარდაქმნიან.

მაგალითად, გამოკლება და გაყოფა:

```
alert( 6 - '2' ); // 4, '2' გარდაქმნის რიცხვად
alert( '6' / '2' ); // 3, ორივე ოპერანდს გარდაქმნის რიცხვად
```

ბინარული +

პლუსი „+“ არსებობს ორი ფორმით: ბინარული, რომელიც ზემოთ გამოვიყენეთ, და უნარული. უნარული, ანუ ერთ მნიშვნელობასთან გამოიყენება, პლუსი „+“ რიცხვებთან არაფერს არ აკეთებს. მაგრამ, თუ ოპერანდი არ არის რიცხვი, უნარული პლუსი მას რიცხვად გარდაქმნის.

მაგალითად:

```
// რიცხვებზე გავლენას არ ახდენს
let x = 1;
alert( +x ); // 1

let y = -2;
alert( +y ); // -2
```

```
// ოპერანდს გარდაქმნის რიცხვად
alert( +true ); // 1
alert( +"" );   // 0
```

ის რეალურად იგივეა, რაც ფუნქცია Number (...), მხოლოდ მოკლე ჩანაწერი.

სტრიქონების რიცხვებად გადაქცევის აუცილებლობა ძალიან ხშირად ჩნდება. მაგალითად, როგორც წესი, HTML-ფორმის ველების მნიშვნელობები არის სტრიქონები. მაგრამ რა მოხდება, თუ საჭირო გახდება, მაგალითად, მათი შეკრება?

ბინარული პლუსი გააერთიანებს მათ როგორც სტრიქონებს:

```
let apples = "2";
let oranges = "3";

alert( apples + oranges ); // შედეგი იქნება "23"
```

ხოლო უნარული პლუსი გარდაქმნის მათ რიცხვებად:

```
let apples = "2";
let oranges = "3";

// ორივე ოპერანდი ჯერ გარდაიქმნება რიცხვებად
alert( +apples + +oranges ); // 5

// სრულ ვარიანტს ექნება შემდეგი სახე
// alert( Number(apples) + Number(oranges) ); // 5
```

მათემატიკის თვალსაზრისით, პლიუსების ასეთი სიმრავლე უცნაურად გამოიყურება. მაგრამ პროგრამისტის

გადმოსახედიდან აქ განსაკუთრებული არაფერია: ჯერ შესრულდება უნარული პლუსები, რომლებიც სტრიქონებს რიცხვებად გარდაქმნის, შემდეგ კი ბინარული „+“ შეკრებს მათ.

მინიჭების ოპერატორი

პრიორიტეტების სიაში ყველაზე დაბალი პრიორიტეტი აქვს მინიჭების ოპერატორს „=“. ამიტომ, როდესაც ცვლადს რაიმეს ვანიჭებთ ჯერ ხდება გამოსახულებაში სხვადასხვა ოპერაციების შესრულება და ამის შემდეგ მიღებული შედეგი მიენიჭება ოპერატორის მარცხნივ მდგომ ცვლადს. მაგალითად:

```
let x = 2 * 2 + 1;
alert( x ); // შედეგი ტოლია 5-ის
```

მინიჭების ოპერატორი „=“ აბრუნებს მნიშვნელობას.

JavaScript-ში ოპერატორების უმეტესობა აბრუნებს მნიშვნელობას. ზოგიერთისათვის ეს ცხადია, მაგალითად, შეკრება „+“ ან ნამრავლი „* “. მინიჭების ოპერატორიც არ წარმოადგენს გამონაკლისს.

გამოსახულება x = value მნიშვნელობა value-ს ჩაწერს x-ში და ისე დააბრუნებს.

ეს საშუალებას გვაძლევს მინიჭების ოპერატორი გამოვიყენოთ უფრო რთულ გამოსახულებებში, მაგალითად:

```
let a = 1;
let b = 2;
let c = 3 - (a = b + 1);
alert( a ); // 3
alert( c ); // 0
```

ზემოთ მოყვანილ მაგალითში (a = b + 1) გამოსახულების მიენიჭება a ცვლადს (ანუ 3). შემდეგ იგი გამოიყენება შემდგომი გამოთვლებისთვის. მსგავსი გამოსახულება შეიძლება შეგხვდეთ JavaScript-ბიბლიოთეკებში, ამიტომ ჩვენ უნდა ვიცოდეთ, თუ როგორ მუშაობს ის.

თუმცა კოდის ამ სტილში წერა არ არის რეკომენდებული, რადგან ასეთი ჩანაწერები არ გახდის თქვენს კოდს უფრო გასაგებს ან უფრო წაკითხვადს.

განვიხილოთ მინიჭების ოპერატორის კიდევ ერთი საინტერესო შესაძლებლობა: ჯაჭვური მინიჭება.

```
let a, b, c;
a = b = c = 2 + 2;
alert( a ); // 4
alert( b ); // 4
alert( c ); // 4
```

ეს მინიჭება სრულდება მარჯვნიდან მარცხნივ. ჯერ გამოითვლება ყველაზე მარჯვენა გამოსახულება 2 + 2, შემდეგ კი შედეგი ენიჭება მარცხნივ ცვლადებს: c, b და a. დასასრულს, ყველა ცვლადს ექნება ერთი და იგივე მნიშვნელობა.

ისევ კოდის ადვილად წასაკითხად, უმჯობესია ასეთი კონსტრუქციები ჩავწეროთ რამდენიმე სტრიქონად:

```
let a, b, c;
c = 2 + 2;
b = c;
a = c;
```

მინიჭების დამატებითი ოპერატორები

ჩვეულებრივი მინიჭების ოპერატორის გარდა კიდევ ხუთი დამატებითი ოპერატორი არსებობს, რომლებიც თავისთავში მოიცავს მინიჭებისა და არითმეტიკული ოპერატორების ქმედებებს.

ოპერატორი	მაგალითი	ეკვივალენტური გამოსახულება
+=	X+=Y	X=X+Y
-=	X-=Y	X=X-Y
*=	X*=Y	X=X*Y
/=	X/=Y	X=X/Y
%=	X%=Y	X=X%Y

ეს ოპერატორები საშუალებას იძლევა ეკონომიურად ჩავწეროთ მსგავსი გამოსახულებანი, მაგალითად:

```
let n = 2;
n = n + 5;
n = n * 2;
```

ეს ჩანაწერი შეიძლება შევცვალოთ მინიჭების დამატებითი ოპერატორებით:

```
let n = 2;
n += 5; // ახლა n = 7 (სრულდება როგორც n = n + 5)
n *= 2; // ახლა n = 14 (სრულდება როგორც n = n * 2)
alert( n ); // 14
```

მინიჭების დამატებით ოპერატორს აქვს იგივე პრიორიტეტი, როგორიც ჩვეულებრივ მინიჭების ოპერატორს, ანუ სრულდება უმეტესობა ოპერაციების შემდეგ, მაგალითად:

```
let n = 2;
n *= 3 + 5; // ჯერ სრულდება გამოსახულების მარჯვენა ნაწილი,
            //  გამოსახულება იდენტურია n *= 8-ის
alert( n ); // 16
```

ინკრემენტი და დეკრემენტი

ერთ-ერთი ყველაზე გავრცელებული რიცხვითი ოპერაციაა ცვლადის მნიშვნელობის ერთით გაზრდა ან შემცირება. ამისათვის სპეციალური ოპერატორებიც კი არსებობს:

- ინკრემენტი ++ ზრდის ცვლადს 1-ით:

```
let c = 2;
c++;          // მუშაობს ისე, როგორც c = c + 1, უბრალოდ შემოკლებული ჩანაწერია
alert( c ); // 3
```

- დეკრემენტი -- ზრდის ცვლადს 1-ით:

```
let c = 2;
c--;          // მუშაობს ისე, როგორც c = c - 1, უბრალოდ შემოკლებული ჩანაწერია
alert( c ); // 1
```

უნდა ვიცოდეთ, რომ ინკრემენტისა და დეკრემენტის გამოყენება მხოლოდ ცვლადების მიმართ შეიძლება.

++ და -- ოპერატორები შეიძლება განთავსებული იყოს არამარტო ცვლადის შემდეგ, არამედ მის წინაც.

- როდესაც ოპერატორი განთავსებულია ცვლადის შემდეგ - ეს არის პოსტფიქსული ფორმა: a++;
- თუ ოპერატორი განთავსებულია ცვლადის წინ - ეს არის პრეფიქსული ფორმა: ++a.

ორივე ეს ფორმა მართალია ცვლადის მნიშვნელობას ზრდის ან ამცირებს ცვლადის მნიშვნელობას ერთით, მაგრამ მათ შორისაც არის სხვაობა, რომელიც ქვემოთ მოყვანილ ცხრილშია მოცემული:

მაგალითი	დასახელება	მოქმედება
++a	პრეფიქსული ინკრემენტი	a-ს მნიშვნელობას ერთით ზრდის და აბრუნებს a-ს მნიშვნელობას.
a++	პოსტფიქსული ინკრემენტი	აბრუნებს a-ს მნიშვნელობას, ხოლო შემდეგ a-ს მნიშვნელობას ერთით ზრდის.
--a	პრეფიქსული დეკრემენტი	a-ს მნიშვნელობას ერთით ამცირებს და აბრუნებს a-ს მნიშვნელობას.
a--	პოსტფიქსული დეკრემენტი	აბრუნებს a-ს მნიშვნელობას, ხოლო შემდეგ a-ს მნიშვნელობას ერთით ამცირებს.

მაგალითები:

```
let c = 1;
let a = ++c; /* პრეფიქსული ფორმა ++c ზრდის c-ს მნიშვნელობას
                      და აბრუნებს ახალ მნიშვნელობას */
alert(a); // 2
```

```
let c = 1;
let a = c++; /* პოსტფიქსული ფორმა c++ ზრდის c-ს
                      მნიშვნელობას მაგრამ აბრუნებს ძველ
                      მნიშვნელობას */
alert(a); // 1
```

პრაქტიკაში ინკრემენტისა და დეკრემენტის ოპერაციები ძალიან ხშირად გვხვდება. მაგალითად, ისინი პრაქტიკულად ნებისმიერ ციკლის ოპერატორში გამოიყენება.

ბულის ტიპის მონაცემები ინკრემენტირებასა და დეკრემენტირებას არ ექვემდებარება.

ბიტური ოპერატორები

ბიტური ოპერატორები 32-ბიტიან მთელ რიცხვებთან მუშაობენ (აუცილებლობის შემთხვევაში მოხდება მასზე დაყვანა), მათი შიგა ორობითი წარმოდგენის დონეზე.

ეს ოპერატორები JavaScript-სთვის არ არის სპეციფიკური, ისინი პროგრამირების უმეტეს ენების მიერ არის მხარდაჭერილი.

შემდეგი ბიტური ოპერატორები JavaScript-ის მიერ არის მხარდაჭერილი:

- AND(ბიტური „და“) (&)

- OR(ბიტური „ან“) (|)
- XOR(„ან“-ის უარყოფა) (^)
- NOT(უარყოფა) (~)
- LEFT SHIFT(მარცხენა წანაცვლება) (<<)
- RIGHT SHIFT(მარჯვენა წანაცვლება) (>>)
- ZERO-FILL RIGHT SHIFT(მარჯვენა წანაცვლება ნულებით შევსება) (>>>)

ისინი იშვიათად გამოიყენება, როდესაც აუცილებელია ციფრებზე მუშაობა ძალიან დაბალ (ბიტურ) დონეზე, რადგან ვებ დეველოპერები მათ იშვიათად იყენებენ, თუმცა ზოგიერთ სფეროში (მაგალითად, კრიპტოგრაფიაში) სასარგებლოა.

ოპერატორი „მძიმე“

ოპერატორი „მძიმე“ (,) იშვიათად გამოიყენება და ერთ-ერთი ყველაზე უჩვეულო ოპერატორია. ზოგჯერ ის გამოიყენება უფრო მოკლე კოდის დასაწერად, ამიტომ უნდა ვიცოდეთ, რომ გავიგოთ რა ხდება.

ოპერატორი „მძიმე“ შესაძლებლობას გვაძლევს მათი მძიმით გამოყოფით გამოვთვალოთ რამდენიმე გამოსახულების მნიშვნელობა. თითოეული გამოსახულება შესრულდება, მაგრამ მხოლოდ ბოლო შედეგის დაბრუნება ხდება.

მაგალითად:

```
let a = (1 + 2, 3 + 4);
alert( a ); // 7 (3 + 4 გამოსახულების გამოთვლის შედეგი)
```

პირველად 1 + 2 გამოსახულება შესრულდება და შედეგი უგულებელყოფილი იქნება. შემდეგ მოდის გამოსახულება 3 + 4, შესრულდება და შედეგი დაბრუნდება.

ოპერატორ „მძიმე“-ს აქვს ძალიან დაბალი პრიორიტეტი, მინიჭების ოპერატორზე უფრო დაბალი, ამიტომ ფრჩხილების გამოყენება მნიშვნელოვანია.

მათ გარეშე, a = 1 + 2, 3 + 4, ჯერ შესრულდება რიცხვების შეკრება a = 3, 7, შემდეგ მინიჭების ოპერატორი = და მიანიჭებს a = 3-ს და რაც მოჰყვება მოხდება მისი იგნორირება. ეს იგივეა, რაც (a = 1 + 2), 3 + 4.

რატომ გვჭირდება ოპერატორი, რომელიც უგულებელყოფს ყველაფერს, გარდა ბოლო გამოსახულებისა?

ზოგჯერ იგი გამოიყენება, როგორც უფრო რთული კონსტრუქციების ნაწილი, რათა მოხდეს რამდენიმე მოქმედების შესრულება ერთ სტრიქონში.

მაგალითად:

```
for (a = 1, b = 3, c = a * b; a < 10; a++) {
 ...
}
```

ასეთი ხრიკები გამოიყენება JavaScript-ის ბევრ ფრეიმვორკებში. მაგრამ, როგორც წესი, ისინი არ აუმჯობესებენ კოდის წაკითხვადობას, ამიტომ მათ გამოყენებამდე კარგად უნდა დავფიქრდეთ.

შედარების ოპერატორები

შედარების ოპერატორების უმეტესობა ჩვენთვის სკოლის კურსიდანაა ცნობილი.

JavaScript-ში ისინი ასე იწერება:

- მეტი და ნაკლებია: a > b, a < b;

- მეტი ან ტოლი და ნაკლები ან ტოლი: a >= b, a <= b;
- ტოლია: a == b. გაითვალისწინეთ, რომ შედარებისთვის გამოიყენება ორმაგი ტოლობის ნიშანი ==. ერთი ტოლობის ნიშანი a = b ნიშნავს მინიჭებას;
- არ უდრის. მათემატიკაში იგი აღინიშნება სიმბოლოთი ≠, მაგრამ JavaScript-ში იწერება როგორც != b.

შედარების ოპერაციის შესრულების შედეგი ლოგიკური მნიშვნელობაა:

- true - ნიშნავს „ჭეშმარიტია“;
- false - ნიშნავს „მცდარია“.

მაგალითად:

```
alert( 2 > 1 );  // true
alert( 2 == 1 ); // false
alert( 2 != 1 ); // true
```

შედარების შედეგი, როგორც ნებისმიერი მნიშვნელობა, შეიძლება მიენიჭოს ცვლადს:

```
let rez = 5 > 4;
alert( rez ); // true
```

სტრიქონების შედარება

იმის დასადგენად, არის თუ არა ერთი სტრიქონი მეორეზე დიდი, JavaScript იყენებს ანბანურ ან „ლექსიკოგრაფიულ“ თანმიმდევრობას.

სხვა სიტყვებით რომ ვთქვათ, სტრიქონები ერთმანეთს სიმბოლოებით დარდება.

მაგალითად:

```
alert( 'W' > 'A' ); // true
alert( 'World' > 'Work' ); // true
alert( 'Normally' > 'Normal' ); // true
```

ორი სტრიქონის შედარების ალგორითმი საკმაოდ მარტივია:

1. პირველ რიგში ხდება სტრიქონების პირველი სიმბოლოების შედარება;

2. თუ პირველი სტრიქონის პირველი სიმბოლო მეტია (ნაკლები) ვიდრე მეორის სტრიქონის პირველი სიმბოლო, მაშინ პირველი სტრიქონი მეტია (ნაკლები) ვიდრე მეორე. შედარება დასრულდა;

3. თუ პირველი სიმბოლოები ტოლია, მაშინ სტრიქონების მეორე სიმბოლოები ანალოგიურად შედარდება;

4. შედარება გრძელდება მანამ, სანამ ერთ-ერთი სტრიქონი არ დასრულდება;

5. თუ ორივე სტრიქონი ერთდროულად მთავრდება, მაშინ ისინი ტოლია. წინააღმდეგ შემთხვევაში, გრძელი სტრიქონი უფრო დიდად ითვლება.

ზემოთ მოყვანილ მაგალითებში შედარება 'W' > 'A' დასრულდება პირველ საფეხურზე, ხოლო სტრიქონები 'World' > 'Work' შედარდება სიმბოლოები:

1. W უდრის W-ს;

2. o უდრის o-ს;

3. r უდრის r-ს;

4. l მეტია k-ზე აქ მთავრდება შედარება. პირველი ხაზი უფრო მეტია.

JavaScript-ში გამოიყენება Unicode კოდირება და არა ნამდვილი ანბანი.

ზემოაღნიშნული შედარების ალგორითმი მსგავსია ლექსიკონებსა და სატელეფონო წიგნებში გამოყენებული ალგორითმისა, მაგრამ მათ შორის არის განსხვავება.

მაგალითად, JavaScript-ში მნიშვნელობა ენიჭება რეგისტრს. ასომთავრული „A“ არ არის პატარა „a“-ს ტოლი. რომელი უფრო მეტია? პატარა ასო „a“. რატომ? იმიტომ, რომ პატარა ასოებს აქვს უფრო დიდი კოდი შიდა კოდირების ცხრილში, რომელსაც იყენებს JavaScript (Unicode), ვიდრე მაღალი რეგისტრის ასოებს.

სხვადასხვა ტიპის ცვლადის შედარება

JavaScript სხვადასხვა ტიპის ცვლადის შედარებისას თითოეულ მათგანს რიცხვის სახით წარმოადგენს, მაგალითად:

```
alert( '2' > 1 ); // true, '2' სტრიქონი ხდება რიცხვი 2
alert( '01' == 1 ); // true, '01' სტრიქონი ხდება რიცხვი 1
```

ლოგიკური მნიშვნელობა true-ს შეესაბამება 1, ხოლო false-ს – 0. მაგალითად:

```
alert( true == 1 ); // true
alert( false == 0 ); // true
```

შესაძლებელია შემდეგი სიტუაცია:

- ორი მნიშვნელობა ტოლია.
- ერთი ლოგიკური true, მეორე - false.

მაგალითად:

```
let a = 0;
alert( Boolean(a) ); // false
let b = "0";
```

```
alert( Boolean(b) ); // true
alert(a == b); // true!
```

JavaScript-ის თვალსაზრისით, შედეგი მოსალოდნელია. ტოლობა მნიშვნელობებს გარდაქმნის რიცხვითი გარდაქმნების გამოყენებით, ამიტომ, სტრიქონული "0" ხდება 0, მაშინ როდესაც Boolean-ით გარდაქმნა განსხვავებულ წესებს იყენებს.

მკაცრი შედარება

ნორმალური შედარების „==“ გამოყენებამ შეიძლება გამოიწვიოს პრობლემები. მაგალითად, ის ერთმანეთისგან არ განასხვავებს 0-ს და false-ს:

```
alert( 0 == false ); // true
```

იგივე პრობლემაა ცარიელი სტრიქონის შემთხვევაშიც:

```
alert( '' == false ); // true
```

ეს გამოწვეულია იმით, რომ სხვადასხვა ტიპის ოპერანდები == ოპერატორის მიერ გარდაიქმნება რიცხვად. შედეგად, ორივე ცარიელი სტრიქონი და false ხდება ნულის ტოლი.

მაშინ როგორ განვასხვავოთ 0 false-საგან?

მკაცრი ტოლობის ოპერატორი === ტოლობას ამოწმებს ტიპების გარდაქმნის გარეშე. სხვა სიტყვებით რომ ვთქვათ, თუ a და b სხვადასხვა ტიპისაა, მაშინ შედარება a === b შედეგად დაუყოვნებლივ აბრუნებს false-ს, მათი გარდაქმნის მცდელობის გარეშე.

მოდით შევამოწმოთ:

```
alert( 0 === false ); // false, ვინაიდან დარდება სხვადასხვა ტიპის მონაცემები
```

ასევე არსებობს მკაცრი უტოლობის ოპერატორი !==, რომელიც ჩვეულებრივი უტოლობის ოპერატორი != მსგავსია.

მკაცრი ტოლობის ოპერატორს უფრო მეტი დრო სჭირდება ჩაწერისთვის, მაგრამ ეს კოდს უფრო ცხადს ხდის და იმისი ნაკლები ალბათობაა რომ შეცდომას დავუშვებთ.

null-ისა და undefined-ის შედარებები

null-ისა და undefined-ის „ქცევები“ სხვა მნიშვნელობებთან მათი შედარების დროს განსაკუთრებულია:

მკაცრი ტოლობის === დროს ეს მნიშვნელობები სხვადასხვაა, ვინაიდან ისინი სხვადასხვა ტიპისაა, ამიტომაც

```
alert( null === undefined ); // false
```

ჩვეულებრივი ტოლობის == შემთხვევაში ეს მნიშვნელობები ერთმანეთის ტოლია, მაგრამ არ უდრის არც ერთ სხვა მნიშვნელობას. ეს ენის სპეციალური წესია.

```
alert( null == undefined ); // true
```

მათემატიკური ოპერატორებისა და შედარების სხვა ოპერატორების <, >, <=, >=, გამოყენების დროს null-ისა და undefined-ის მნიშვნელობების გარდაქმნა ხდება: null არის 0, ხოლო undefined – NaN.

ვნახოთ, რა უცნაურ შემთხვევასთან გვაქვს საქმე ამ წესების გამოყენების დროს და რაც მთავარია, როგორ ავიცილოთ თავიდან შეცდომები.

უცნაური შედეგი null-ისა და 0-ის შედარებისას:

```
alert( null > 0 );  //  false
alert( null == 0 ); //  false
alert( null >= 0 ); //  true
```

მათემატიკის თვალსაზრისით ეს ძალზე უცნაურია, მაგრამ უნდა ვიცოდეთ, რომ მკაცრი და არამკაცრი უტოლობები JavaScript-ში სხვადასხვანაირად მუშაობს. მკაცრი უტოლობების და არამკაცრი ტოლობის შემთხვევაში null-ი არ გარდაიქმნება, ხოლო არამკაცრი უტოლობების შემთხვევაში null-ი გარდაიქმნება 0-ად.

მნიშვნელობა undefined არ არის არც ერთი სხვა მნიშვნელობის ტოლი:

```
alert( undefined > 0 ); // false
alert( undefined < 0 ); // false
alert( undefined == 0 ); // false
```

ამიტომ შეცდომებიდან თავის დაღწევის მიზნით ასეთ შედარებებს უნდა მოვერიდოთ.

ლოგიკური ოპერატორი

ლოგიკური მონაცემები, რომელიც ელემენტარული შედარების ოპერატორის დახმარებით მიიღება, შეიძლება გაერთიანდეს უფრო რთული გამოსახულების სახით. ამისათვის გამოიყენება ლოგიკური (ბულის) ოპერატორები. JavaScript-ში გვაქვს სამი ლოგიკური ოპერატორი: || („ან“), && („და“) და ! (უარყოფა).

ოპერატორები „&&“ და „||“ ხშირად მოიხსენიება შესაბამისად როგორც ლოგიკური ნამრავლი და ლოგიკური ჯამი, ხოლო მათემატიკაში კონიუნქცია და დიზიუნქცია.

ბოლოს დასკვნის სახით შეიძლება ვთქვათ, რომ ერთმანეთს შეიძლება შევადაროთ რიცხვითი, სტრიქონული და ლოგიკური მნიშვნელობები. რიცხვების შედარება ხდება ჩვეულებრივად, არითმეტიკული წესების გამოყენებით, ხოლო სტრიქონების, სიმბოლოთა ASCII კოდების შედარების გზით, დაწყებული სტრიქონის მარცხენა მხრიდან. ლოგიკური მნიშვნელობების შედარება ხდება ისევე როგორც რიცხვების 1-ის და 0-ის.

შედარების ოპერატორები შეიძლება გამოყენებულ იქნეს სხვადასხვა ტიპის მონაცემთან. თუ რიცხვი უნდა შევადაროთ სტრიქონს ან ლოგიკურ მონაცემს, მაშინ ეს უკანასკნელი ინტერპრეტატორს გადაჰყავს რიცხვითში. თუ ერთმანეთს დარდება ლოგიკური მონაცემი და სტრიქონი, აქ საქმე ცოტა რთულადაა. იმ შემთხვევაში თუ სტრიქონი შეიცავს მხოლოდ რიცხვებს, მხოლოდ ჰარებს ან არის ცარიელი სტრიქონი, მაშინ ოპერანდები დაიყვანება რიცხვით ტიპზე. ამასთან, ცარიელი სტრიქონი ("") ან სტრიქონი შემდგარი მხოლოდ ჰარებისაგან გარდაიქმნება ნულად (0). ყველა სხვა შემთხვევაში, ყველა შედარების ოპერატორის (გარდა „!=“ ოპერატორისა) შედეგი არის false, ხოლო „**!=**“ ოპერატორის შემთხვევაში კი true.

|| („ან“)

ტრადიციულად პროგრამირებაში „ან“ განკუთვნილია მხოლოდ ბულის მნიშვნელობების მანიპულირებისთვის: თუ რომელიმე არგუმენტის მნიშვნელობა არის true, ის შედეგად დააბრუნებს true-ს, წინააღმდეგ შემთხვევაში ის დააბრუნებს false-ს.

JavaScript-ში, როგორც მოგვიანებით ვნახავთ, ეს ოპერატორი ოდნავ განსხვავებულად მუშაობს. მაგრამ ჯერ ვნახოთ, რა ემართება ლოგიკურ მნიშვნელობებს.

არსებობს მხოლოდ ოთხი შესაძლო ლოგიკური კომბინაცია:

```
alert( true || true );   // true
alert( false || true );  // true
alert( true || false );  // true
alert( false || false ); // false
```

როგორც ვხედავთ, ოპერაციების შედეგი ყოველთვის არის true, გარდა იმ შემთხვევისა, როდესაც ორივე არგუმენტი false-ია. თუ მნიშვნელობა არ არის ლოგიკური ტიპის, მაშინ იგი მასზე დაიყვანება გამოთვლების მიზნით.

მაგალითად, რიცხვი 1 განიხილება როგორც true, ხოლო 0 როგორც false. მაგალითად:

```
if (1 || 0) { // მუშაობს როგორც if( true || false )
  alert( 'ჭეშმარიტია!' );
}
```

ჩვეულებრივ || („ან“) ოპერატორი if ოპერატორში რომელიმე მოცემული პირობის ჭეშმარიტების შესამოწმებლად გამოიყენება.

მაგალითად:

```
let hour = 9;
if (hour < 10 || hour > 18) {
  alert( 'ოფისი დაკეტილია.' );
}
```

if ოპერატორში ორზე მეტი პირობა შეიძლება იყოს ჩაწერილი, მაგალითად:

```
let hour = 12;
let Week = true;
if (hour < 10 || hour > 18 || Week) {
  alert( 'ოფისი დაკეტილია.' );
}
```

ზემოთ აღწერილი ლოგიკა ტრადიციულ სიტუაციას შეესაბამება. ახლა მოდით ვიმუშაოთ JavaScript-ის „დამატებით“ ფუნქციებთან.

კომბინირებული ალგორითმი შემდეგნაირად მუშაობს:

|| („ან“) ოპერატორის შესრულებისას მრავალი ოპერანდით შემდეგნაირად სრულდება:

- აფასებს ოპერანდებს მარცხნიდან მარჯვნივ;
- თითოეული ოპერანდი გარდაიქმნება ლოგიკურ მნიშვნელობად. თუ შედეგი ჭეშმარიტია, გაჩერდება და შედეგად დააბრუნებს ამ ოპერანდის საწყის მნიშვნელობას;

- თუ ყველა ოპერანდი არის მცდარი, შედეგად დააბრუნებს უკანასკნელის მნიშვნელობას.

შედეგი ბრუნდება თავდაპირველი სახით, გარდაქმნის გარეშე.

სხვა სიტყვებით რომ ვთქვათ, || („ან“) ოპერატორი შედეგს აბრუნებს პირველ ნამდვილ მნიშვნელობას, ან უკანასკნელს, თუ სხვა მნიშვნელობა არ არის ნაპოვნი.

მაგალითად:

```
alert( 1 || 0 ); // 1
alert( true || 'no matter what' ); // true

alert( null || 1 ); // 1 (პირველი ჭეშმარიტი მნიშვნელობა)
alert( null || 0 || 1 ); // 1 (პირველი ჭეშმარიტი მნიშვნელობა)
alert( undefined || null || 0 ); // 0 (ვინაიდან ყველა მნიშვნელობა
                                     მცდარია, შედეგად ბრუნდება
                                     ბოლო მნიშვნელობა)
```

&& („და“)

ტრადიციულ პროგრამირებაში && („და“) ოპერატორი შედეგად აბრუნებს true-ს, თუ ორივე არგუმენტი ჭეშმარიტია, ხოლო წინააღმდეგ შემთხვევაში - false-ს:

```
alert( true && true );   // true
alert( false && true );  // false
alert( true && false );  // false
alert( false && false ); // false
```

მაგალითი if ოპერატორთან ერთად:

```
let hour = 12;
```

```
let minute = 30;
if (hour == 12 && minute == 30) {
  alert( 'The time is 12:30' );
}
```

კომბინირებული ალგორითმის დროს როგორც || („ან“) ოპერატორის შემთხვევაში && („და“) ოპერატორიც ანალოგიურად მუშაობს:

&& („და“) ოპერატორის შესრულებისას მრავალი ოპერანდით შემდეგნაირად სრულდება:

- აფასებს ოპერანდებს მარცხნიდან მარჯვნივ;
- თითოეული ოპერანდი გარდაიქმნება ლოგიკურ მნიშვნელობად. თუ შედეგი მცდარია, გაჩერდება და შედეგად დააბრუნებს ამ ოპერანდის საწყისი მნიშვნელობას;
- თუ ყველა ოპერანდი ჭეშმარიტია, შედეგად დააბრუნებს უკანასკნელის მნიშვნელობას.

შედეგი ბრუნდება თავდაპირველი სახით, გარდაქმნის გარეშე.

სხვა სიტყვებით რომ ვთქვათ, && („და“) ოპერატორი შედეგს აბრუნებს პირველ ნამდვილ მნიშვნელობას, ან უკანასკნელს, თუ სხვა მნიშვნელობა არ არის ნაპოვნი.

მაგალითად:

```
// თუ პირველი ოპერანდი ჭეშმარიტია, მაშინ აბრუნებს მეორეს:
alert( 1 && 0 ); // 0
alert( 1 && 5 ); // 5
// თუ პირველი ოპერანდი მცდარია, მაშინ აბრუნებს მას.
// ამ დროს მეორე ოპერანდის იგნორირება ხდება
alert( null && 5 ); // null
```

```
alert( 0 && "no matter what" ); // 0
```

შეგიძლიათ ზედიზედ რამდენიმე მნიშვნელობის გადაცემა. ამ შემთხვევაში, პირველი false მნიშვნელობის დროს გამოთვლები შეჩერდება, შედეგად ეს მნიშვნელობა დაბრუნდება. როდესაც ყველა მნიშვნელობა ჭეშმარიტია, შედეგად ბოლო მნიშვნელობა დაბრუნდება, მაგალითად:

```
alert( 1 && 2 && null && 3 ); // null
alert( 1 && 2 && 3 ); // 3
```

&& („და“) ოპერატორის პრიორიტეტი უფრო მეტია ვიდრე || („ან“) ოპერატორის, ამიტომ ჯერ იგი შესრულდება. ამგვარად, კოდი a && b || c && d არსებითად იგივეა, რაც (a && b) || (c && d).

! (უარყოფა)

უარყოფის ოპერატორი ”!” მხოლოდ ერთ ოპერანდთან გამოიყენება და ცვლის ამ ოპერანდის მნიშვნელობას საწინააღმდეგოთი: თუ X-ს ჰქონდა მნიშვნელობა true, მაშინ !X იქნება false და პირიქით.

უარყოფის ოპერატორის შესრულების დროს შემდეგი მოქმედებები სრულდება:

1. ჯერ არგუმენტი დაყავს ლოგიკურ მნიშვნელობაზე - true ან false;

2. შემდეგ შედეგად აბრუნებს საპირისპირო მნიშვნელობას.

მაგალითად:

```
alert( !true ); // false
alert( !0 ); // true
```

ორმაგი უარყოფის ოპერატორი მნიშვნელობის ლოგიკურ ტიპად გარდაქმნისათვის გამოიყენება:

```
alert( !!"non-empty string" ); // true
alert( !!null ); // false
```

ანუ, პირველი უარყოფის ოპერატორი მნიშვნელობას ლოგიკურად გარდაქმნის და აბრუნებს საპირისპიროს, ხოლო მეორე - გარდაქმნის მას ხელახლა. საბოლოოდ, ჩვენ გვაქვს მნიშვნელობის ლოგიკურად მარტივი გარდაქმნის ოპერაცია.

არსებობს უფრო ზუსტი გზა ამის გასაკეთებლად - ეს არის ფუნქცია Boolean:

```
alert( Boolean("non-empty string") ); // true
alert( Boolean(null) ); // false
```

უარყოფის ოპერატორის პრიორიტეტი ლოგიკურ ოპერატორებს შორის არის ყველაზე მაღალი, ამიტომაც ის პირველი სრულდება.

null-თან გაერთიანების ოპერატორი ??

ეს ფუნქცია ენას ახლახანს დაემატა. ძველ ბრაუზერებში იგივე ოპერაციის შესასრულებლად შეიძლება დაგჭირდეთ რამდენიმე ოპერატორის გამოყენება.

აქ ჩვენ ვიტყვით, რომ გამოსახულების მნიშვნელობა "განსაზღვრულია", თუ ის null-ისგან ან undefined-სგან განსხვავდება.

null-თან გაერთიანების ოპერატორი არის ორი კითხვის ნიშანი ??.

გამოთქმის შედეგი a ?? b იქნება:

- a, თუ a-ს მნიშვნელობა განსაზღვრულია;

- b თუ a-ს მნიშვნელობა არ არის განსაზღვრული.

```
alert ( null ?? 'default string'); //"default string"
alert ( 0 ?? 42) // 0;
```

ანუ ოპერატორი აბრუნებს პირველ არგუმენტს, თუ ის არ არის null ან undefined, წინააღმდეგ შემთხვევაში აბრუნებს მეორე არგუმენტს.

null-თან გაერთიანების ოპერატორი ახალი არაფერია. ეს არის მხოლოდ მოსახერხებელი სინტაქსი, თუ როგორ უნდა ამოვარჩიოთ ორი მნიშვნელობიდან ერთი "განსაზღვრული" მნიშვნელობა.

ამასთან, შეგიძლიათ დაწეროთ ?? ოპერატორების თანმიმდევრობა, რომ მიიღოთ სიაში პირველი მნიშვნელობა, რომელიც არ არის null ან undefined.

ოპერატორი || („ან“) JavaScript-ის გამოსვლის დროიდან არსებობს, ამიტომ ადრე დეველოპერები მსგავსი პრობლემების გადასაჭრელად ამ ოპერატორს იყენებდნენ. მაგრამ ზოგჯერ || („ან“) ოპერატორის საშუალებით ზოგიერთი პრობლემის გადაწყვეტა ვერ ხერხდებოდა. ვნახოთ მათ შორის არსებული მნიშვნელოვანი განსხვავება:

- || აბრუნებს პირველ ჭეშმარიტ მნიშვნელობას;
- ?? აბრუნებს პირველ „განსაზღვრულ“ მნიშვნელობას.

სხვა სიტყვებით რომ ვთქვათ, ოპერატორი || ერთმანეთისაგან არ განასხვავებს false-ს, 0-ს, ცარიელ სტრიქონს "" და null/undefined-ს. მისთვის ყველა ერთნაირია, ე.ი. false მნიშვნელობაა. თუ || ოპერატორის პირველი არგუმენტი იქნება ჩამოთვლილი მნიშვნელობებიდან რომელიმე, შედეგად მივიღებთ მეორე არგუმენტს.

მაგრამ პრაქტიკაში ხშირად საჭიროა ნაგულისხმევი მნიშვნელობის გამოყენება მხოლოდ მაშინ, როდესაც ცვლადი არის null/undefined.

განიხილეთ შემდეგი მაგალითი:

```
let height = 0;
alert(height || 100); // 100
alert(height ?? 100); // 0
```

- height || 100 ამოწმებს, არის თუ არა height-ის მნიშვნელობა false, რაც ლოგიკური გამოსახულებისათვის ასეა, ამიტომ შედეგი იქნება 100-ის ტოლი;
- height ?? 100 ამოწმებს, height შეიცავს თუ არა null-ს ან undefined-ს და რადგან ეს ასე არ არის, ამიტომ შედეგი იქნება 0-ის ტოლი.

ოპერატორების პრიორიტეტები

იმ შემთხვევაში, თუ გამოსახულებაში რამდენიმე ოპერატორია, მათი შესრულების თანმიმდევრობა პრიორიტეტით განისაზღვრება, ან სხვა სიტყვებით რომ ვთქვათ, არსებობს გარკვეული თანმიმდევრობა, რომლითაც ოპერატორები უნდა შესრულდეს.

სკოლიდან ვიცით, რომ გამოსახულებაში 1 + 2 * 2 გამრავლება შესრულდება შეკრებაზე ადრე. სწორედ ეს არის "პრიორიტეტი". ამბობენ, რომ გამრავლებას უფრო მაღალი პრიორიტეტი აქვს ვიდრე შეკრებას.

ფრჩხილები უფრო მნიშვნელოვანია, ვიდრე პრიორიტეტი, ასე რომ, თუ არ გვაკმაყოფილებს მოქმედებათა თანმიმდევრობა,

ჩვენ შეგვიძლია შევცვალოთ მათი პრიორიტეტი. მაგალითად, ჩაწერეთ (1 + 2) * 2.

JavaScript-ში მრავალი ოპერატორია. თითოეულ ოპერატორს აქვს შესაბამისი პრიორიტეტული ნომერი. უფრო მეტი პრიორიტეტის მქონე ოპერატორი უფრო ადრე შეასრულდება. თუ პრიორიტეტი ერთნაირი აქვთ, მაშინ შესრულების მიმდევრობა მოხდება მარცხნიდან მარჯვნივ.

პრიორი-ტეტი	**ოპერატორი**	**შესრულების მიმდევრობა**
17	ლოგიკური უარყოფა !, ბიტური უარყოფა ~, უნარული +, უნარული -	მარჯვნიდან მარცხნივ
16	** ახარისხება	მარჯვნიდან მარცხნივ
15	(პოსტფიქსული)++ (პოსტფიქსული)--	
14	++(პრეფიქსული) --(პრეფიქსული)	მარჯვნიდან მარცხნივ
13	* ნამრავლი, / გაყოფა, % მოდულით გაყოფა	მარცხნიდან მარჯვნივ
12	+ შეკრება, – გამოკლება	მარცხნიდან მარჯვნივ
11	<<, >>, >>> წანაცვლება	მარცხნიდან მარჯვნივ
10	<, <=, >, >= შედარება	მარცხნიდან მარჯვნივ

<table>
<tr><td>9</td><td>=== მკაცრი ტოლია, !== მკაცრი უტოლობა</td><td>მარცხნიდან მარჯვნივ</td></tr>
<tr><td>8</td><td>== ტოლია, != არ უდრის</td><td>მარცხნიდან მარჯვნივ</td></tr>
<tr><td>7</td><td>& ბიტური „და“</td><td>მარცხნიდან მარჯვნივ</td></tr>
<tr><td>6</td><td>^ ბიტური „ან“-ის უარყოფა</td><td>მარცხნიდან მარჯვნივ</td></tr>
<tr><td>5</td><td>| ბიტური „ან“</td><td>მარცხნიდან მარჯვნივ</td></tr>
<tr><td>4</td><td>&& „და“</td><td>მარცხნიდან მარჯვნივ</td></tr>
<tr><td>3</td><td>|| „ან“, ??</td><td>მარცხნიდან მარჯვნივ</td></tr>
<tr><td>2</td><td>=, +=, -=, *=, /=, %=, >>=, <<=, >>>=, &=, ^=, |=</td><td>მარჯვნიდან მარცხნივ</td></tr>
<tr><td>1</td><td>, (მძიმე)</td><td></td></tr>
</table>

პირობითი გადასვლის ოპერატორები

პროგრამაში იმისდა მიხედვით სრულდება თუ არა რაიმე წინასწარ მოცემული პირობა, გამოთვლითი პროცესი შეიძლება განხორციელდეს ამა თუ იმ მიმართულებით. ამ მიზანს ემსახურება ინსტრუქცია if, პირობითი გადასვლის ოპერატორი „?“ (კითხვის ნიშანი) და კონსტრუქცია switch.

ინსტრუქცია if

if(...) ინსტრუქცია აფასებს მდგომარეობას ფრჩხილებში და, თუ შედეგი ჭეშმარიტია, მაშინ სრულდება ბლოკის კოდი.

if ინსტრუქციის სინტაქსი შემდეგია:

if (პირობა)
{კოდი, რომელიც სრულდება, თუ პირობა ჭეშმარიტია}

მაგალითი: თუ ბლოკში გამოყენებულია მხოლოდ ერთი გამოსახულება, მაშინ ფიგურული ფრჩხილები შეიძლება არ გამოვიყენოთ.

```
let year = prompt('როდის მოხდა დიდგორის ბრძოლა?', '');
if (year == 1121) alert( 'თქვენ მართალი ხართ!' );
```

თუ გვსურს ერთზე მეტი გამოსახულების შესრულება, მაშინ ეს გამოსახულებები ფიგურულ ფრჩხილებში უნდა ჩავსვათ.

if (...) ინსტრუქცია აფასებს ფრჩხილებში მოთავსებულ გამოსახულებას და შედეგს ლოგიკურად გარდაქმნის:

- რიცხვი 0, ცარიელი სტრიქონი "", null, undefined და NaN ხდება false;
- დანარჩენი მნიშვნელობები ხდება ჭეშმარიტი.

```
let year = prompt('როდის მოხდა დიდგორის ბრძოლა?', '');

if (year == 1121) {
  alert( "თქვენ მართალი ხართ!" );
  alert( "თქვენ ძალიან ჭკვიანი ხართ!" );
}
```

ამრიგად, კოდი იმ პირობით თუ შედეგი არის false, არასოდეს შესრულდება, ხოლო თუ პირობა არის - true, შესრულდება.

ბლოკი else

ინსტრუქცია if საშუალებას იძლევა მოვახდინოთ პირობითი გამოსახულების "თუ ..., მაშინ ..., თუ არა და ..." სტრუქტურის რეალიზაცია. ინსტრუქცია if შეიძლება შეიცავდეს არასავალდებულო else ბლოკს. იგი შესრულდება, როდესაც პირობა მცდარია.

if (პირობა)

{კოდი, რომელიც სრულდება, თუ პირობა ჭეშმარიტია}

else

{კოდი, რომელიც სრულდება იმ შემთხვევაში, თუ პირობა მცდარია}

მაგალითი:

```
let year = prompt('როდის მოხდა დიდგორის ბრძოლა?', '');
```

```
if (year == 1121) {
  alert( 'თქვენ მართალი ხართ!' );
} else {
alert( 'თქვენ არ ხართ მართალი!' );
}
```

ზოგჯერ რამდენიმე პირობის შემოწმებაა საჭირო. ამისათვის else if ბლოკი გამოიყენება პირობითი ოპერატორის კონსტრუქცია ერთმანეთში ჩადგმული პირობითი ოპერატორების არსებობის საშუალებას იძლევა. ამ დროს მას აქვს უფრო რთული სტრუქტურა:

```
if (პირობა 1) {
   კოდი, რომელიც სრულდება, თუ პირობა 1 შესრულდა
} else if (პირობა 2) {
   კოდი, რომელიც სრულდება, თუ პირობა 2 შესრულდა
   } else {
   კოდი, რომელიც სრულდება, თუ პირობა 2 არ შესრულდა
   }
}
```

მაგალითი:

```
let year = prompt('როდის მოხდა დიდგორის ბრძოლა?', '');

if (year < 1121) {
  alert( 'ეს ძალიან ადრეა...' );
   } else if (year > 1121) {
        alert( 'ეს ძალიან გვიანია...' );
   } else {
  alert( 'თქვენ მართალი ხართ!' );
```

```
}
```

ფიგურული ფრჩხილების აბზაცით ჩაწერა გემოვნების საკითხია და არ არის აუცილებელი, მაგრამ საჭიროა პროგრამა ისე დაიწეროს, რომ იყოს თვალსაჩინო და მკაფიო სტრუქტურის, რომლის დროსაც მარტივად შემოწმდება ფრჩხილების განლაგების სისწორე.

ტერნარული „?“ ოპერატორი

ზოგჯერ ჩვენ, მდგომარეობიდან გამომდინარე, გვჭირდება ცვლადის განსაზღვრა როგორც ეგრეთ წოდებული „პირობითი“ ოპერატორი. ამის საშუალებას უფრო მოკლედ და მარტივად გვაძლევს „კითხვის ნიშანი“ (ტერნარული ოპერატორი).

ტერნარული ოპერატორი წარმოდგენილია კითხვის ნიშნით ?. მას ასევე უწოდებენ "სამეულს", რადგან ამ ოპერატორს, თავის მხრივ, აქვს სამი არგუმენტი.

მისი სინტაქსია:

```
let result = პირობა ? მნიშვნელობა1 : მნიშვნელობა2;
```

ზემოთ განხილული მაგალითი ასეთ სახეს მიიღებს:

```
let year = prompt ('როდის მოხდა დიდგორის ბრძოლა?', '');
result=(year == 1121) ? 'თქვენ მართალი ხართ!' : 'თქვენ არ ხართ მართალი!';
alert(result);
```

იგივე კოდი შეიძლება ასეც ჩაიწეროს:

```
let year = prompt ('როდის მოხდა დიდგორის ბრძოლა?', '');
 (year == 1121)?
            alert ('თქვენ მართალი ხართ!'):
                          alert ('თქვენ არ ხართ მართალი!');
```

რამდენიმე ტერნარული „?“ ოპერატორი

ტერნარული „?“ ოპერატორების თანმიმდევრობა საშუალებას გვაძლევს დააბრუნოს მნიშვნელობა, რომელიც ერთზე მეტ პირობაზეა დამოკიდებული.

მაგალითად:

```
let year = prompt ('როდის მოხდა დიდგორის ბრძოლა?', '');
result=(year <1121)? 'ეს ძალიან ადრეა...'  :
     (year > 1121) ? 'ეს ძალიან გვიანია...' :
   'თქვენ მართალი ხართ!';
  alert (result );
```

მიუხედავად იმისა, რომ ასეთი აღნიშვნა უფრო მოკლეა, ვიდრე მისი ეკვივალენტური if ინსტრუქცია, ის ნაკლებად კითხვადია.

კითხვისას თვალები ვერტიკალურად ათვალიერებს კოდს. კოდის ბლოკები, რომლებიც მრავალ სტრიქონს მოიცავს, ბევრად უფრო ადვილია წასაკითხია, ვიდრე გრძელი, ჰორიზონტალური ინსტრუქციების ნაკრები.

რას ნიშნავს ტერნარული ოპერატორი „?“ - პირობიდან გამომდინარე დააბრუნოს ერთი ან მეორე მნიშვნელობა. გთხოვთ იგი მხოლოდ ამისთვის გამოიყენოთ. როდესაც გჭირდებათ კოდის სხვადასხვა განშტოების შესრულება, ამ შემთხვევაში გამოიყენეთ ინსტრუქცია if.

კონსტრუქცია switch

switch კონსტრუქციას შეუძლია ერთდროულად რამდენიმე if ინსტრუქცია შეცვალოს.

რამდენიმე პირობის არსებობის შემთხვევაში უფრო მოსახერხებელი და თვალსაჩინო საშუალებაა switch კონსტრუქცია, რომელიც საშუალებას გვაძლევს ერთდროულად რამდენიმე ვარიანტიდან ამოვარჩიოთ ერთი. switch კონსტრუქციას აქვს ერთი ან რამდენიმე case ბლოკი და არა აუცილებელი default ბლოკი.

იგი განსაკუთრებით მოსახერხებელია მაშინ, როდესაც შესამოწმებელი პირობები არ არის ურთიერთგამომრიცხავი.

switch ოპერატორის სინტაქსი შემდეგია:

```
switch (გამოსახულება) {
   case ვარიანტი 1; // if (გამოსახულება === 'ვარიანტი 1')
      კოდი
      [break]
   case ვარიანტი 2; // if (გამოსახულება === 'ვარიანტი 2')
      კოდი
      [break]
   . . .
     [default:
  კოდი]
}
```

switch კონსტრუქციის პარამეტრმა „გამოსახულება“ შეიძლება მიიღოს სტრიქონული, რიცხვითი და ლოგიკური მნიშვნელობა. ოპერატორები break და default არ არის

სავალდებულო, რისთვისაც ისინი მოთავსებულია კვადრატულ ფრჩხილებში. თუ მითითებულია ოპერატორი break, მაშინ მომდევნო ოპერატორების შესრულება არ მოხდება, ხოლო თუ მითითებულია ოპერატორი default, მაშინ მის შემდეგ მდგომი სრულდება მხოლოდ იმ შემთხვევაში, როდესაც პარამეტრის „გამოსახულება“ მნიშვნელობა არც ერთ ვარიანტს არ შეესაბამება. თუ switch კონსტრუქციაში გათვალისწინებულია ყველა შესაძლო ვარიანტი, მაშინ ოპერატორი default შეიძლება არ გამოვიყენოთ.

switch კონსტრუქცია მუშაობს შემდეგნაირად: ჯერ გამოითვლება მრგვალ ფრჩხილებში მოთავსებული პარამეტრის „გამოსახულება“ მნიშვნელობა. მიღებული მნიშვნელობა შედარდება „ვარიანტი 1“ მნიშვნელობას. თუ ისინი ერთმანეთს არ ემთხვევა, მაშინ მოხდება გადასვლა შემდეგ ვარიანტზე, ხოლო თუ ისინი ერთმანეთს დაემთხვევა, მაშინ მიმდევრობით იწყება იმ კოდების შესრულება, რომლებიც შეესაბამება მოცემულ ვარიანტს. ამასთან, თუ არ არის მითითებული ოპერატორი break, მაშინ შესრულდება მომდევნო ვარიანტების კოდებიც, ვიდრე არ შეხვდება ოპერატორი break.

მაგალითები:

1. switch კონსტრუქცია break ოპერატორის გარეშე:

```
x=2
switch (x) {
case 1:
  alert (1)
case 2:
  alert (2)
```

```
case 3:
  alert (3)
}
```

მოცემულ მაგალითში მეორე და მესამე ვარიანტიც იმუშავებს.

2. switch კონსტრუქცია break ოპერატორით:

```
x=2
switch (x) {
case 1:
   alert (1)
     break
case 2:
   alert (2)
      break
case 3:
   alert (3)
      break
}
```

მოცემულ მაგალითში მხოლოდ მეორე ვარიანტი იმუშავებს.

3. დავუშვათ, ცვლადი lang შეიცავს ენის დასახელებას, რომელიც მომხმარებელმა აირჩია და შეიტანა ფორმის ველში.

```
switch (lang) {
          case "ინგლისური"
                window.open ("english.htm")
                break
```

```
            case "ფრანგული"
                  window.open ("french.htm")
                  break
            case "გერმანული"
                  window.open ("german.htm")
                  break
            default:
                  alert ("მოცემულ ენაზე დოკუმენტაცია არ
მოიძებნა")
}
```

აქ გამოსახულება window.open() ხსნის ბრაუზერის ახალ ფანჯარას და მასში ჩატვირთავს ფრჩხილებში მითითებულ HTML-დოკუმენტს.

4. იგივე ამოცანა ჩავწეროთ if ოპერატორის გამოყენებით:

```
if (lang == "ინგლისური")
                  window.open ("english.htm")
else {
if (lang ==  "ფრანგული")
                  window.open ("french.htm")
else {
if (lang ==   "გერმანული")
                  window.open ("german.htm")
          else
                  alert ("მოცემულ ენაზე დოკუმენტაცია არ
მოიძებნა")
}
}
```

აღნიშნული კოდი შესაძლებელია შემდეგნაირადაც ჩაიწეროს:

```
if (lang == "ინგლისური") window.open ("english.htm")
else { if (lang ==  "ფრანგული") window.open ("french.htm")
else { if (lang ==   "გერმანული") window.open ("german.htm")
else alert ("მოცემულ ენაზე დოკუმენტაცია არ მოიძებნა")
}
}
```

ციკლის ოპერატორები

სკრიპტების წერისას, ხშირად ერთი და იგივე ტიპის მოქმედების მრავალჯერ შესრულების აუცილებლობა ჩნდება. მაგალითად, სიიდან პროდუქტების სათითაოდ ამორჩევა ან უბრალოდ სხვადასხვა რიცხვებისათვის ერთი და იგივე ოპერაციის ათჯერ შესრულება და სხვა.

ერთი და იგივე კოდის მრავალჯერ გამეორებისთვის გამოიყენება ციკლი. JavaScript-ში გამოიყენება ციკლის სამი ოპერატორი: for, while და do-while.

ციკლის ოპერატორი while

while ოპერატორის სინტაქსი შემდეგია:

```
while ( პირობა )
{
        კოდი (ციკლის ტანი)
}
```

კოდი ციკლის ტანიდან სრულდება მანამ, სანამ „პირობა“ ჭეშმარიტია. მაგალითად, ქვემოთ მოყვანილ ციკლს გამოაქვს i-ს მნიშვნელობა, ვიდრე i < 3:

```
let i = 0;
while (i < 3) { // ჯერ გამოიტანს 0-ს, შემდეგ 1-ს, შემდეგ 2-ს
  alert( i );
  i++;
}
```

ციკლის ერთ შესრულებას იტერაციას უწოდებენ. ზემოთ მოცემულ მაგალითში ციკლი სამ იტერაციას ასრულებს. თუ ამ მაგალითში i++ სტრიქონი არ იქნებოდა, მაშინ ციკლი მუდმივად (თეორიულად) განმეორდებოდა. პრაქტიკაში, რა თქმა უნდა, ბრაუზერი ამას არ დაუშვებს, ის მომხმარებელს მისცემს შესაძლებლობას შეაჩეროს „დაკიდული“ სკრიპტი.

ციკლის პირობა შეიძლება იყოს ნებისმიერი არა მხოლოდ შედარება, არამედ გამოსახულება ან ცვლადი: ციკლის პირობა გამოითვლება და ლოგიკურ მნიშვნელობად გარდაიქმნება.

მაგალითად, while (i) არის while (i != 0) გამოსახულების მოკლე ჩანაწერი:

```
let i = 3;
while (i) { // როდესაც i გახდება 0-ის ტოლი, პირობა გახდება
                        მცდარი და ციკლი დამთავრდება
  alert( i );
  i--;
}
```

თუ ციკლის ტანი მხოლოდ ერთი სტრიქონისაგან შედგება, მაშინ ფიგურული ფრჩხილების გამოყენება აუცილებელი არ არის. მაგალითი:

```
let i = 3;
while (i) alert (i--);
```

მაგალითები:

1. გამოვთვალოთ x ხარისხად y.

```
let z = x;
i = 2;
```

```
while ( i <= y ) {
        z = z * x;
        i++;
}
```

2. გამოვთვალოთ n-ის ფაქტორიალი.

```
let z = 1;
if  ( n > 1 )  {
i = 2;
while ( i <= n  ) {
                z = z * i;
i ++;
}
}
```

ზემოთ მოყვანილ მაგალითებში გამოყენებულია ციკლის მთვლელი. მათი ინიცირება ხდებოდა ციკლის ოპერატორამდე, ხოლო მათი განახლება ხდება ციკლის ტანში.

გამოთვლითი პროცესის მართვისათვის ციკლის ოპერატორ while-შიც გამოიყენება ოპერატორები break და continue.

ციკლის ოპერატორი do...while

do...while ოპერატორი არის ორი ოპერატორის კონსტრუქცია, რომელიც ერთად გამოიყენება. „პირობის“ შემოწმება ციკლის ტანის შემდეგაა განთავსებული. მისი სინტაქსი შემდეგია:

```
do {
                    კოდი
}
while (პირობა);
```

while ოპერატორისაგან განსხვავებით do…while ოპერატორში პირობისაგან დამოუკიდებლად კოდი ერთხელ მაინც შესრულდება. „პირობა“ მოწმდება კოდის შესრულების შემდეგ. თუ იგი ჭეშმარიტია, მაშინ კვლავ სრულდება do ოპერატორის შემდეგ მდგომი კოდი.

```
let i = 0;
do {
  alert ( i );
  i++;
} while (i < 3);
```

განვიხილოთ მაგალითები, რომელიც ჩვენ უკვე ამოვხსენით ციკლის while ოპერატორების დახმარებით.

მაგალითები:

1. გამოვთვალოთ x ხარისხად y.

```
let z = x
i = 2
do {
z = z * x
i++
}
while ( i <= y )
```

2. გამოვთვალოთ n-ის ფაქტორიალი.

```
let z = 1
if ( n > 1 ) {
i = 2
do {
                    z = z * i
i ++
}
while ( i <= n  )
}
```

ციკლის ოპერატორი for

ყველაზე გავრცელებული ციკლის ოპერატორია for ოპერატორი. მისი სინტაქსია:

```
for (საწყისი გამოსახულება; პირობა; ბიჯი)
 {
          კოდი (ციკლის ტანი)
}
```

ციკლის ოპერატორის სათაურში „საწყისი გამოსახულება“ სრულდება მხოლოდ ერთხელ, ოპერატორის შესრულების დაწყების დროს. მეორე პარამეტრი არის ციკლის ოპერატორის შესრულების პირობა. იგი პირობითი გადასვლის if ოპერატორის პირობის ანალოგიურია. მესამე პარამეტრი შეიცავს გამოსახულებას, რომელიც სრულდება ფიგურულ ფრჩხილებში მოთავსებული კოდის ყველა გამოსახულების შესრულების შემდეგ. მაგალითი:

```
for (let i = 0; i < 3; i++)
```

```
{
alert(i); // ჯერ გამოიტანს 0-ს, შემდეგ 1-ს, შემდეგ 2-ს
}
```

ციკლის ოპერატორი მუშაობს შემდეგნაირად: პირველად სრულდება „საწყისი გამოსახულება“, შემდეგ მოწმდება „პირობა“. თუ ეს პირობა ჭეშმარიტია, მაშინ მთავრდება ციკლის ოპერატორის შესრულება, ხოლო წინააღმდეგ შემთხვევაში სრულდება ფიგურულ ფრჩხილებში მოთავსებული ციკლის ტანი. ამის შემდეგ სრულდება „ბიჯი“ და ამით მთავრდება ციკლის პირველი იტერაცია. შემდეგ ისევ მოწმდება „პირობა“ და ყველაფერი იწყება თავიდან.

უმეტეს შემთხვევაში „საწყისი გამოსახულების“ სახით ცვლადზე მინიჭების ოპერატორი გამოიყენება. ცვლადის სახელი და მინიჭებული მნიშვნელობა შეიძლება იყოს ნებისმიერი. ამ ცვლადს უწოდებენ ციკლის მთვლელს. ციკლის მთვლელი შეიძლება იყოს როგორც ზრდადი, ასევე კლებადი.

ცვლადის ჩაშენებული გამოცხადება

საწყისი ცვლადი i ქვემოთ მოცემულ მაგალითში პირდაპირ ციკლშია გამოცხადებული. ეს არის ეგრეთ წოდებული ცვლადის „ჩაშენებული“ გამოცხადება. ასეთი ცვლადები მხოლოდ ციკლის შიგნით არსებობენ.

```
for (let i = 0; i < 3; i++) {
  alert(i); // შედეგად გამოიტანს 0, 1, 2
}
alert(i); // შეცდომაა, ასეთი ცვლადი არ არსებობს
```

ახალი ცვლადის გამოცხადების მაგივრად შეიძლება გამოვიყენოთ უკვე არსებული ცვლადი:

```
let i = 0;
for (i = 0; i < 3; i++) { // ვიყენებთ არსებულ ცვლადს
  alert(i); // შედეგად გამოიტანს 0, 1, 2
}
alert(i); // შედეგად გამოიტანს 3, ვინაიდან ცვლადი ციკლის
              გარეთ იყო გამოცხადებული
```

for ციკლის ნაწილების გამოტოვება

for ციკლის ნებისმიერი ნაწილი შეიძლება იყოს გამოტოვებული.

მაგალითისათვის, შეგვიძლია გამოვტოვოთ საწყისი გამოსახულება, თუ ციკლის დაწყების წინ არაფრის გაკეთება არ გვჭირდება:

```
let i = 0; // აქ გვაქვს უკვე გამოცხადებული ცვლადი i
მინიჭებული მნიშვნელობით
for (; i < 3; i++) { // საწყისი გამოსახულება აღარ არის საჭირო
  alert( i ); // შედეგად გამოიტანს 0, 1, 2
}
```

შეიძლება არ ჩავწეროთ ბიჯიც:

```
let i = 0;
for (; i < 3;) {
  alert( i++ ); // შედეგად გამოიტანს 0, 1, 2
}
```

ეს მოცემულ ციკლს გახდის while (i < 3) ციკლის ანალოგიურს.

შეიძლება არც ერთი პარამეტრი არ ჩავწეროთ, მაშინ მივიღებთ უსასრულო ციკლს:

```
for (;;) {
  // ციკლი მუდმივად შესრულდება
}
```

ამასთან, უნდა გვახსოვდეს, რომ წერტილ-მძიმე (;) აუცილებლად არ უნდა გამოვტოვოთ, რადგან ეს სინტაქსურ შეცდომას გამოიწვევს.

მაგალითები:

1. გამოვთვალოთ მთელი დადებითი რიცხვების ჯამი 1-დან 25-მდე:

```
let s = 0
for ( i = 1;  i <= 25;  i ++ ) {
            s = s + i
}
```

2. გამოვთვალოთ x ხარისხად y, სადაც x, y – მთელი დადებითი რიცხვებია. ამ ამოცანის ალგორითმი მარტივია, უნდა გამოითვალოს x*x*x* ... *x, სადაც x თანამამრავლად შეგვხვდება y-ჯერ.

```
let z = x
for ( i = 2;  i <= y;  i ++ ) {
            z = z * x
}
```

3. გამოვთვალოთ n-ის ფაქტორიალი. ფაქტორიალი აღინიშნება n!-ით. როცა n ტოლია 0 ან 1-ის, მაშინ n! უდრის 1-ს, დანარჩენ შემთხვევაში n!=2*3*4* ... * n.

```
let z = 1
if  ( n > 1 )  {
for ( i = 2;  i <= n;  i ++ ) {
                        z = z * i
}
}
```

ოპერატორი break

ჩვეულებრივ ციკლის შესრულება მთავრდება მაშინ, როდესაც პირობის შედეგი არის false, თუმცა შესაძლებელია ციკლიდან დროზე ადრე გამოსვლაც. ამისთვის გამოიყენება ოპერატორი break. თუ გამოთვლით პროცესს შეხვდება ეს ოპერატორი, იგი მაშინვე შეწყვეტს ყოველგვარ გამოთვლებს ციკლის ტანში. ჩვეულებრივ break ოპერატორი გამოიყენება რაიმე დამატებითი პირობის შემოწმების დროს, რომლის შესრულებასაც უნდა მოჰყვეს ციკლის დამთავრება.

მაგალითად, ქვემოთ მოცემული კოდი შეყვანილი რიცხვების ჯამს გამოთვლის მანამ, სანამ მომხმარებელი არ შეწყვეტს მონაცემების შეტანას, ხოლო შემდეგ გამოიტანს ჯამს:

```
let sum = 0;
while (true) {
  let A = +prompt("შეიტანეთ რიცხვი", ' ');
  if (!A) break; // (*)
```

```
  sum += A;
}
alert ( 'ჯამი: ' + sum );
```

ამ კოდში break ოპერატორი მთლიანად წყვეტს ციკლის შესრულებას და მართვა გადაეცემა ციკლის ტანის შემდეგ სტრიქონს, ანუ alert ოპერატორს.

ოპერატორი continue

გამოთვლების სამართავად ციკლის ოპერატორში აგრეთვე გამოიყენება ოპერატორი continue. როგორც ოპერატორი break, ოპერატორი continue გამოიყენება ციკლის ტანში, ოღონდ მისგან განსხვავებით იგი არ წყვეტს ციკლის მუშაობას, არამედ გადადის მომდევნო იტერაციის შესრულებაზე და გამოთვლით პროცესს აბრუნებს ციკლის ოპერატორის დასაწყისში, სადაც მოწმდება პირობა.

მაგალითად, მომდევნო კოდში ოპერატორი continue გამოიყენება რათა კენტი რიცხვები გამოვიტანოთ:

```
for (let i = 0; i < 10; i++) {
  if (i % 2 == 0) continue;
  alert(i); // შედეგად გამოიტანს 1, 3, 5, 7, 9
}
```

უნდა გვახსოვდეს, რომ ოპერატორები break და continue არ შეიძლება გამოვიყენოთ „?“ (კითხვის ნიშანი) ოპერატორის მარჯვენა მხარეს.

ჭდეები break და continue ოპერატორებისათვის

ზოგჯერ საჭიროა რამდენიმე ციკლიდან ერთდროულად გამოსვლა.

მაგალითად, ქვემოთ მოყვანილ კოდში ციკლი ორი i და j პარამეტრით მიმდინარეობს, prompt ოპერატორის საშუალებით შეგვყავს (i, j) კოორდინატები (0,0)-დან (2,2)-მდე:

```
for (let i = 0; i < 3; i++) {
  for (let j = 0; j < 3; j++) {
    let input = prompt ('კოორდინატების მნიშვნელობა (${i},${j})', '');
  }
}
alert ('შედეგი!');
```

როდესაც მომხმარებელი შეწყვეტს მონაცემების შეტანას ჩვენ ციკლის შეწყვეტის საშუალება უნდა მოვძებნოთ. ჩვეულებრივ, input ოპერატორის შემდეგ break ოპერატორი მხოლოდ შიდა ციკლის მუშაობას შეწყვეტს, მაგრამ კოდის მუშაობის სრულად დამთავრებისათვის ეს საკმარისი არ არის. სასურველი შედეგის მიღწევა ჭდის გამოყენებითაა შესაძლებელი. ჭდეს აქვს იდენტიფიკატორის სახე, რომელიც ციკლის ოპერატორისაგან ორი წერტილითაა გამოყოფილი.

ზემოთ მოყვანილ მაგალითს ჭდის გამოყენებით ექნება შემდეგი სახე:

```
label: for (let i = 0; i < 3; i++) {
  for (let j = 0; j < 3; j++) {
    let input = prompt ('კოორდინატების მნიშვნელობა (${i},${j})', '');
```

```
    if (!input) break  label;
  }
}
alert ('შედეგი!');
```

ციკლში break label ასეთი ჭდით უახლოეს გარე ციკლს ეძებს და მის ბოლოში გადადის. ჭდე შეიძლება ცალკე სტრიქონზე განთავსდეს, მაგალითად:

```
label:
for (let i = 0; i < 3; i++) { ... }
```

ოპერატორი continue ასევე შეიძლება გამოყენებული იყოს ჭდესთან ერთად. ამ შემთხვევაში მართვა გადაეცემა ჭდიანი ციკლის შემდეგ იტერაციას.

ჭდე საშუალებას არ იძლევა, სადაც გვინდა იქ გადავიდეთ. ჭდე არ იძლევა საშუალებას მართვა გადაეცეს კოდის ნებისმიერ ოპერატორს. ოპერატორების break და continue გამოყენება შესაძლებელია მხოლოდ ციკლის შიგნით და ჭდეც უნდა იყოს ამ ოპერატორების ზემოთ.

ფუნქციები

ხშირად პროგრამის სხვადასხვა ნაწილში ერთი და იგივე მოქმედების გამეორება გვჭირდება. იმისთვის, რომ ბევრგან არ განმეორდეს იგივე კოდი, ფუნქციები გამოიყენება. ფუნქციები პროგრამის ძირითადი „სამშენებლო ბლოკებია“.

JavaScript-ში ფუნქციები ორ ჯგუფად იყოფა - სტანდარტული და მომხმარებლის მიერ განსაზღვრული ფუნქციები. სტანდარტული ფუქციები არის ენის ნაწილი და შეიძლება გამოყენებულ იქნეს წინასწარი აღწერის გარეშე, მაგრამ შეუძლებელია მისი კოდის რედაქტირება ან დათვალიერება. რაც მათ შესახებ შეიძლება გავიგოთ – ეს არის მათი მოქმედების, პარამეტრებისა და დასაბრუნებელი მნიშვნელობების აღწერა. JavaScript-ში სტანდარტული ფუნქციების ფართო სპექტრი გამოიყენება.

ზოგიერთ სტანდარტულ ფუნქციას ჩვენ უკვე ვიცნობთ - ესენია alert(message), prompt(message, default) და confirm(question).

ასევე შესაძლებელია შევქმნათ საკუთარი ფუნქციაც.

მომხმარებლის ფუნქციები

ფუნქციების შესაქმნელად, შეგვიძლია გამოვიყენოთ ფუნქციის გამოცხადება.

ფუნქციის გამოცხადების მაგალითი:

```
function Message() {
  alert( 'Hello!' );
}
```

პირველად იწერება საკვანძო სიტყვა function, რასაც მოჰყვება ფუნქციის სახელი, შემდეგ პარამეტრთა სია მრგვალ ფრჩხილებში, ერთმანეთისაგან გამოყოფილი მძიმეებით (ზემოთ მოყვანილ მაგალითში ის ცარიელია) და ბოლოს ფიგურულ ფრჩხილებში მოთავსებული ფუნქციის კოდი, რომელსაც ასევე უწოდებენ „ფუნქციის ტანს“.

```
function ფუნქციის სახელი (პარამეტრები)
{
  ... ფუნქციის ტანი ...
}
```

ჩვენი ახალი ფუნქცია შეიძლება გამოძახებული იყოს თავისი სახელით პროგრამის ნებისმიერი ადგილიდან რამდენჯერაც გვჭირდება. მაგალითი:

```
function Message() {
  alert( 'Hello!' );
}

Message();
Message();
```

Message() გამოძახება ფუნქციის კოდის შესრულებას გამოიწვევს. აქ შეტყობინება ორჯერ გამოჩნდება. ეს მაგალითი ცხადად გვიჩვენებს ფუნქციის ერთ-ერთ ძირითად დანიშნულებას - კოდის დუბლირების თავიდან აცილებას.

ფუნქციის სახელის შერჩევა ისევე ხდება, როგორც ცვლადის სახელის შერჩევა. ფუნქციის სახელს აუცილებლად მოსდევს მრგვალი ფრჩხილები. თუ ფუნქციას არ ესაჭიროება

პარამეტრები, მაშინ მრგვალ ფრჩხილებში არ მიეთითება არაფერი.

თუ გსურთ შეცვალოთ შეტყობინება ან მისი გამოტანის საშუალება, საკმარისია შეცვალოთ იგი ერთ ადგილზე: ფუნქციაში, რომელიც მას გამოიტანს.

ლოკალური ცვლადები

ცვლადები, რომელთა მოდიფიკატორებია var ან let და რომლებიც გამოცხადებულია ფუნქციის შიგნით, გამოიყენება მხოლოდ ამ ფუნქციის შიგნით და მათ ლოკალური ცვლადები ეწოდება.

მაგალითად:

```
function Message() {
  let mes = "Hello, I am JavaScript!"; // mes - ლოკალური ცვლადია
  alert( mes );
}
Message();    // შედეგად გამოიტანს - Hello, I am JavaScript!
alert( mes ); // გამოიტანს შეცდომას, ვინაიდან mes ცვლადი
                            გამოცხადებულია ფუნქციის შიგნით
```

გლობალური ცვლადები

ფუნქციებს წვდომა აქვთ ფუნქციის გარეთ არსებულ ცვლადებზე, მაგალითად:

```
let Name = 'Jon';
```

```
function Message() {
  let message = 'Hello, ' + Name+'!';
  alert(message);
}

Message(); // შედეგად გამოიტანს - Hello, Jon!
```

ფუნქციებს სრული წვდომა გააჩნიათ ფუნქციის გარეთ არსებულ ცვლადებზე და მათი მნიშვნელობების შეცვლაც შეუძლიათ, მაგალითად:

```
let Name = 'Jon';

function Message() {
    Name = "Tom"; // ვცვლით გარე ცვლადის მნიშვნელობას
let message = 'Hello, ' + Name+'!';
  alert(message);
}
alert( Name ); // შედეგად გამოიტანს - Jon
Message();
alert( Name ); // შედეგად გამოიტანს - Hello, Tom!
```

ფუნქციის გარეთ გამოყენებული ცვლადი ფუნქციაში იმ შემთხვევაში გამოიყენება, თუ ფუნქციაშიც არ გვაქვს ასეთივე ლოკალური ცვლადი გამოყენებული. თუ იმავე სახელით ფუნქციის შიგნით გვაქვს გამოცხადებული ცვლადი, მაშინ ფუნქციის შიგნით უპირატესობა ლოკალურ ცვლადს ენიჭება. მაგალითი:

```
let Name = 'Jon';
```

```
function Message() {
   let Name = "Tom"; // ვცვლით გარე ცვლადის მნიშვნელობას
let message = 'Hello, ' + Name+'!';
  alert(message);
}
Message();      // შედეგად გამოიტანს - Hello, Tom!
alert( Name ); // შედეგად გამოიტანს - Jon
```

ცვლადებს, რომლებიც გამოცხადებულია ყველა ფუნქციის გარეთ, გლობალური ცვლადები ეწოდებათ. ასევე გლობალური ცვლადი შეიძლება განსაზღვრული იყოს ფუნქციის შიგნით, გლობალური ცვლადის განსაზღვრისთვის ცვლადის სახელის წინ არ ვწერთ ხილვადობის მოდიფიკატორს var, let ან const-ს.

გლობალური ცვლადები ხილულია ნებისმიერი ფუნქციისთვის (თუ ისინი გადაფარული არ არის იმავე სახელის ლოკალური ცვლადებით).

სასურველია გლობალური ცვლადების გამოყენება მინიმუმამდე დაიყვანოთ. თანამედროვე კოდს ჩვეულებრივ აქვს რამდენიმე გლობალური ცვლადი ან საერთოდ არ აქვს. მიუხედავად იმისა, რომ ისინი ზოგჯერ სასარგებლოა საერთო მონაცემების შესანახად.

პარამეტრები

ჩვენ შეგვიძლია ფუნქციის შიგნით გადავიტანოთ ნებისმიერი ინფორმაცია პარამეტრების გამოყენებით, რომლებსაც ასევე ფუნქციის არგუმენტებსაც უწოდებენ.

ქვემოთ მოცემულ მაგალითში ფუნქციას გადაეცემა ორი პარამეტრი: from და text.

```
function Message(from, text) { // არგუმენტებია: from, text
  alert(from + '  ' +text);
}

Message('Anna', 'Hello!'); // Anna Hello!
Message('Anna', 'How are you?'; // Anna How are you?
```

როდესაც მოხდება ფუნქციის გამოძახება, გადაცემული მნიშვნელობების კოპირება ხდება text და from ლოკალურ ცვლადებში. ამის შემდეგ ისინი გამოიყენება ფუნქციის ტანში.

აი კიდევ ერთი მაგალითი: ჩვენ გვაქვს from ცვლადი და გადავცემთ მას ფუნქციას. ყურადღება მიაქციეთ, რომ ფუნქცია ცვლის from-ის მნიშვნელობას, მაგრამ ეს ცვლილება გარედან არ ჩანს. ფუნქცია ყოველთვის იღებს მხოლოდ მნიშვნელობის ასლს:

```
function Message(from, text) { // არგუმენტებია: from, text
  from = '*' + from + '*'; // ცოტათი შევუცვალოთ ფორმა "from" ცვლადს
  alert(text + '  ' +from+'!');
}
let from = " Anna";

Message('Anna', 'Hello!'); // *Anna* Hello!
// from ცვლადის მნიშვნელობა არ შეცვლილა
alert( from ); // Anna
```

ნაგულისხმევი პარამეტრები

თუ პარამეტრი არ არის მითითებული, მაშინ მისი მნიშვნელობა ხდება undefined.

მაგალითად, ზემოთ მოყვანილი Message(from,text) ფუნქციის გამოძახება ერთი არგუმენტითაც შეიძლება:

```
Message('Anna');
```

ეს არ გამოიწვევს შეცდომას. ასეთი გამოძახების შედეგად მივიღებთ "Anna undefined". ფუნქციაზე მიმართვა არ აკონკრეტებს text პარამეტრს, ამიტომ text === undefined.

თუ გვინდა, რომ text პარამეტრს მივანიჭოთ ნაგულისხმევი მნიშვნელობა, მაშინ ის უნდა მივუთითოთ =-ის შემდეგ:

```
function Message(from, text='ტექსტი არ არის დამატებული') {
  alert(from + '  ' +text);
}


Message('Anna'); // Anna ტექსტი არ არის დამატებული
```

თუ პარამეტრი text არ არის მითითებული, მისი მნიშვნელობა იქნება "ტექსტი არ არის დამატებული".

ამ შემთხვევაში, "ტექსტი არ არის დამატებული" არის სტრიქონი, მაგრამ ეს შეიძლება ყოფილიყო უფრო რთული გამოსახულება, რომელიც გამოითვლებოდა და მიენიჭებოდა პარამეტრის არარსებობის შემთხვევაში. მაგალითად:

```
function Message(from, text = anotherFunction()) {
  // anotherFunction() ფუნქცია მხოლოდ მაშინ შესრულდება თუ
პარამეტრი text არ არის მითითებული
```

```
  // გამოთვლის შედეგი იქნება text პარამეტრის მნიშვნელობა
}
```

JavaScript-ში ნაგულისხმევი პარამეტრები გამოითვლება ყოველ ჯერზე, როცა ფუნქციას მივმართავთ შესაბამისი პარამეტრის გარეშე. ზემოთ მოყვანილ მაგალითში anotherFunction() გამოიძახება ყოველთვის, როცა Message() გამოიძახება text პარამეტრის გარეშე.

JavaScript-ის ადრეულ ვერსიებს არ ჰქონდათ ნაგულისხმევი პარამეტრების მხარდაჭერა. აქედან გამომდინარე, არსებობს ალტერნატიული გზები, რომლებიც შეიძლება მოიძებნოს ძველ სკრიპტებში.

მაგალითად, undefined-ზე აშკარა შემოწმება ხდება შემდეგნაირად:

```
function Message(from, text) {
  if (text === undefined) {
    text = 'ტექსტი არ არის დამატებული';
  }
  alert( from + "  " + text );
}
```

მნიშვნელობის დაბრუნება

ფუნქციას შეუძლია დააბრუნოს შედეგი, რომელიც გადაეცემა კოდს, საიდანაც მოხდა მისი გამოძახება. ამის უმარტივესი მაგალითია ორი რიცხვის ჯამის ფუნქცია:

```
function sum(a, b) {
  return a + b;
}
```

```
let res = sum(7, 8);
alert( res ); // 15
```

დირექტივა return შეიძლება გამოჩნდეს ფუნქციის ტანში ნებისმიერ ადგილას. როგორც კი ფუნქციის შესრულება მიაღწევს ამ წერტილს, ფუნქცია ჩერდება და მნიშვნელობა უბრუნდება კოდს, რომლისგანაც მოხდა მისი გამოძახება (მიენიჭა res ცვლადს).

დირექტივა return შეიძლება იყოს რამდენიმე, მაგალითად:

```
function checkAge(age) {
  if (age > 18) {
    return true;
  } else {
    return confirm('თქვენ არ ხართ სრულწლოვანი');
  }
}

let age = prompt('რამდენი წლის ხართ?', 18);

if ( checkAge(age) ) {
  alert( 'თქვენ შეგიძლიათ არჩევნებში მონაწილეობის მიღება!' );
} else {
  alert( 'თქვენ არ შეგიძლიათ არჩევნებში მონაწილეობის მიღება!' );
}
```

return დირექტივის გამოყენება შესაძლებელია მნიშვნელობის გარეშეც. ეს გამოიწვევს ფუნქციიდან დაუყოვნებლივ გამოსვლას.

არასოდეს არ ჩაწეროთ return დირექტივა და მისი მნიშვნელობა სხვადასხვა სტრიქონში. return-ში დიდი გამოსახულებისათვის ზოგჯერ საჭიროა ცალკეულ სტრიქონზე განთავსება, მაშინ return დირექტივის შემდეგ გავხსნათ მრგვალი ფრჩხილი და გამოსახულების ბოლოს დავხუროთ. კოდი არ იმუშავებს, რადგან JavaScript თარჯიმანი return-ის შემდეგ დასვამს მძიმეს და მოხდება ფუნქციიდან გამოსვლა.

ფუნქციის სახელის შერჩევა

ფუნქცია არის მოქმედება. ამიტომ, ფუნქციის სახელად ჩვეულებრივ ზმნას ირჩევენ. ის უნდა იყოს მარტივი, ზუსტი და აღწეროს რას აკეთებს ფუნქცია, რათა პროგრამისტს, რომელიც კოდს წაიკითხავს, სწორი წარმოდგენა ჰქონდეს იმის შესახებ, თუ რას აკეთებს ფუნქცია.

როგორც წესი, გამოიყენება სიტყვიერი პრეფიქსები, რომლებიც აღნიშნავენ მოქმედების ზოგად ხასიათს, რასაც მოჰყვება განმარტება. ამ პრეფიქსების მნიშვნელობებთან დაკავშირებით არსებობს ზოგიერთი შეთანხმება.

მაგალითად, ფუნქციები, რომლებიც იწყება სიტყვით „show“, ჩვეულებრივ აჩვენებს რაღაცას.

ფუნქციები დაწყებული სიტყვებით:

- „get...“ - აბრუნებს რაიმე მნიშვნელობას;
- „calc…“ – გამოითვლის რაღაცას;
- „create…“ – ქმნის რაღაცას;

- „check...“ - რაღაც ამოწმებს და აბრუნებს ლოგიკური მნიშვნელობას და ა.შ.

ასეთი სახელების მაგალითებია:

```
showMessage(..)     // აჩვენებს შეტყობინებას
getAge(..)          // შედეგად აბრუნებს ასაკს
calcSum(..)         // გამოითვლის ჯამს და აბრუნებს შედეგს
createForm(..)      // ქმნის ფორმას და ჩვეულებრივ აბრუნებს მას
checkPermission(..) // ამოწმებს წვდომის არსებობას და შედეგად აბრუნებს true/false
```

პრეფიქსების წყალობით, ფუნქციის სახელისთვის ერთი შეხედვით, ცხადი ხდება, რას აკეთებს მისი კოდი და რა მნიშვნელობის დაბრუნება შეუძლია მას.

ფუნქციამ უნდა გააკეთოს მხოლოდ ის, რასაც მისი სახელი პირდაპირ გულისხმობს. და ეს უნდა იყოს ერთი მოქმედება.

ფუნქცია - კომენტარი

ფუნქციები უნდა იყოს მოკლე და მხოლოდ ერთ რამეს აკეთებდეს. თუ ფუნქცია დიდია, მაშინ აზრი აქვს ფუნქციის რამდენიმე მცირე ფუნქციებად დაყოფას. ზოგჯერ ამ წესის დაცვა ადვილი არ არის, მაგრამ ეს ნამდვილად კარგი წესია.

მცირე ფუნქციები არა მხოლოდ აადვილებს ტესტირებას და გამართვას - ასეთი ფუნქციების არსებობა კარგი კომენტარის როლსაც ასრულებს.

მაგალითად, შევადაროთ ქვემოთ მოცემული ორი showPrimes(n) ფუნქცია. თითოეული მათგანი გამოიტანს 1-დან n-მდე მარტივ რიცხვებს.

პირველი ვარიანტი იყენებს nextPrime ჭდეს:

```
function showPrimes(n) {
  nextPrime: for (let i = 2; i < n; i++) {
    for (let j = 2; j < i; j++) {
      if (i % j == 0) continue nextPrime;
    }
    alert( i ); // მარტივი რიცხვი
  }
}
```

მეორე ვარიანტი იყენებს დამატებით isPrime(n) ფუნქციას მარტივი რიცხვის შესამოწმებლად:

```
function showPrimes(n) {
  for (let i = 2; i < n; i++) {
    if (!isPrime(i)) continue;
    alert(i);  // მარტივი რიცხვი
  }
}
function isPrime(n) {
  for (let i = 2; i < n; i++) {
    if ( n % i == 0) return false;
  }
  return true;
}
```

მეორე ვარიანტი უფრო ადვილი გასაგებია. კოდის ნაცვლად, ჩვენ ვხედავთ მოქმედების სახელს (isPrime). ზოგჯერ დეველოპერები მოიხსენიებენ ასეთ კოდს, როგორც თვითდოკუმენტირებად კოდს. ასე რომ, მისაღებია ფუნქციების შექმნა მაშინაც კი, თუ არ ვგეგმავთ მათ განმეორებით

გამოყენებას. ასეთი ფუნქციები აყალიბებს კოდს და უფრო გასაგებს ხდის მას.

პროგრამა შეიძლება შეიცავდეს როგორც ფუნქციის აღწერას, ასევე მისი გამოძახების გამოსახულებას. ამასთან, პროგრამაში მათი ჩაწერის მიმდევრობას არა აქვს მნიშვნელობა. ფუნქცია შეიძლება ჩაწერილი იყოს როგორც პროგრამის დასაწყისში, ასევე პროგრამის ბოლოს და .js გაფართოების მქონე ცალკე ფაილის სახითაც.

ფუნქციის გამოძახების ბრძანება შეიძლება გამოყენებულ იქნეს ნებისმიერ გამოსახულებაში როგორც ოპერანდი.

მაგალითები:

1. ვთქვათ საჭიროა მართკუთხედის ფართობის გამოსათვლელი ფუნქციის განსაზღვრა. ამ ფუნქციას დავარქვათ Srectangle.

```
function Srectangle (width, height) {
S = width * height
return S
}
```

იგივე ფუნქცია შეიძლება ჩაიწეროს უფრო მოკლედ:

```
function Srectangle (width, height) {
return width * height
}
```

განვიხილოთ ფუნქციაზე მიმართვისა და ცვლადების მოქმედების არესთან დაკავშირებული სიტუაციები:

ა. ცვლადი S განსაზღვრულია ფუნქციის ტანში:

```
function Srectangle (width, height) {
```

```
S = width * height
return S
}
z = Srectangle (2, 3)     /* z-სა და S-ის მნიშვნელობა ტოლია 6-ის */
```

ბ. ცვლადი S განსაზღვრულია გარე პროგრამაში:

```
function Srectangle (width, height) {
let S = width * height
return S
}
let S = 2
z = Srectangle (3, 4+2)    /* z-ის მნიშვნელობა ტოლია 18-ის, ხოლო S-ის დარჩება 2-ის ტოლი */
```

გ. ლოკალური ცვლადისათვის:

```
function Srectangle (width, height) {
let S = width * height
let x = 12
return S
}
w = 4
h = 5
z = Srectangle (w, h)     /* z-ის მნიშვნელობა ტოლია 20-ის, ხოლო S და x გარე პროგრამაში განსაზღვრული არ არის */
```

დ. ფუნქციის პარამეტრების სახელი ემთხვევა გარე პროგრამისა და ფუნქციის გამოძახების პარამეტრებს:

```
function Srectangle (width, height) {
let S = width * height
width = width + 10
return S
}
width = 3
height = 4
z = Srectangle (width, height)     /* z-ის მნიშვნელობა ტოლია 12-ის, ხოლო width-ის 3-ის */
```

ე. ფუნქციაზე მიმართვის დროს პარამეტრების რაოდენობა არ ემთხვევა ერთმანეთს:

```
function Srectangle (width, height) {
S = width * height
return S
}
z = Srectangle (2)     /* მეორე პარამეტრის უქონლობის გამო z-ისა და S-ის მნიშვნელობა ტოლია 0-ის */
```

2. n! ფაქტორიალის გამოთვლის ფუნქცია ციკლის ოპერატორის გამოყენებით:

```
function factorial (n) {
if ( n <= 1 ) return 1
R = 2
for ( i = 3; i <= n; i++ ) {
            R = R * i
}
return R
```

```
}
let m = 10
x = factorial (m)     /* x-ის მნიშვნელობა ტოლია 3628800 */
```

3. n! ფაქტორიალის გამოთვლის ფუნქცია რეკურსიის გამოყენებით (ფუნქციის განსაზღვრა შეიძლება შეიცავდეს ამავე ფუნქციის გამოძახებას – ე. წ. ფუნქციის რეკურსიული განსაზღვრა):

```
function factorial (n) {
if ( n <= 1 ) {
return 1
}
return n * factorial (n-1)
}
```

იგივე შეიძლება ჩაიწეროს მოკლედ:

```
function factorial (n) {
if ( n <= 1 ) return 1
return n * factorial (n-1)
}
```

4. ორ რიცხვს შორის მინიმუმის არჩევის ფუნქცია, რომელიც დააბრუნებს a და b რიცხვებს შორის უმცირესს. ამოცანის გადასაწყვეტად დავწეროთ ფუნქცია if ოპერატორის გამოყენებით:

```
function min(a, b) {
  if (a < b) {
    return a;
```

```
  } else {
    return b;
  }
}
```

მეორე ვარიანტი დავწეროთ „?" ოპერატორის გამოყენებით:

```
function min(a, b) {
  return a < b ? a : b;
}
```

a == b ტოლობის შემთხვევაში არა აქვს მნიშვნელობა რომელს გადავაგზავნით.

5. დავწეროთ ფუნქცია pow(x,n), რომელიც დააბრუნებს x-ს ხარისხად n-ს. სხვა სიტყვებით რომ ვთქვათ, x თავისთავად ამრავლებს n-ჯერ და აბრუნებს შედეგს.

შექმენით გვერდი, რომელიც მოითხოვს x და n მნიშვნელობების შეტანას და შემდეგ დაბეჭდავს შედეგს pow(x,n).

ამ ამოცანის გადაწყვეტის დროს ფუნქციამ უნდა გაითვალისწინოს მხოლოდ n-ის ნატურალური მნიშვნელობები, ე.ი. მთელი რიცხვები 1-დან და მეტი.

```
function pow(x, n) {
  let result = x;
  for (let i = 1; i < n; i++) {
    result *= x;
  }
  return result;
}
```

```
let x = prompt("x - ?", ' ');
let n = prompt("n - ?", ' ');
if (n < 1) {
  alert(‘მოცემული ფუნქცია ${n} ხარისხს მხარს არ უჭერს, გამოიყენეთ მხოლოდ ნატურალური რიცხვები’);
} else {
  alert( pow(x, n) );
}
```

Function Expression (ფუნქციური გამოსახულება)

JavaScript-ში ფუნქცია არ არის ენის სტრუქტურა, არამედ მნიშვნელობის განსაკუთრებული ტიპია. სინტაქსს, რომელსაც აქამდე ვიყენებდით, ეწოდება Function Declaration (ფუნქციის გამოცხადება):

```
function sayHi() {
  alert( "Hello!" );
}
```

არსებობს კიდევ ერთი სინტაქსი ფუნქციების შესაქმნელად, სახელწოდებით Function Expression (ფუნქციური გამოსახულება). ის ასე გამოიყურება:

```
let sayHi = function() {
  alert( "Hello!" );
};
```

ზემოთ მოცემულ კოდში, ფუნქცია იქმნება და აშკარად ენიჭება ცვლადს, როგორც ნებისმიერი სხვა მნიშვნელობა. ძირითადად, არ აქვს მნიშვნელობა როგორ განვსაზღვრეთ ფუნქცია, ეს არის მხოლოდ sayHi ცვლადში შენახული მნიშვნელობა.

ორივე კოდის მაგალითის მნიშვნელობა იგივეა: „შექმენი ფუნქცია და მისი მნიშვნელობა მიანიჭე sayHi ცვლადს“.

ჩვენ შეგვიძლია გამოვიტანოთ ეს მნიშვნელობა alert-ის საშუალებით:

```
function sayHi() {
  alert( "Hello!" );
}

alert( sayHi ); // გამოიტანს ფუნქციის კოდს
```

გაითვალისწინეთ, რომ ბოლო სტრიქონი არ გამოიძახებს sayHi ფუნქციას, მისი სახელის შემდეგ არ არის მრგვალი ფრჩხილები. არის პროგრამირების ენები, რომლებშიც ფუნქციის სახელის ნებისმიერი ხსენება მის გამოძახებას გამოიწვევს. JavaScript კი არ არის ერთ-ერთი მათგანი.

JavaScript-ში ფუნქციები არის მნიშვნელობები, რის გამოც ჩვენ მათ ვიყენებთ როგორც მნიშვნელობებს. ზემოთ მოცემული კოდი გამოიტანს ფუნქციის სტრიქონულ წარმოდგენას, რომელიც არის მისი საწყისი კოდი.

რა თქმა უნდა, ფუნქცია არ არის ჩვეულებრივი მნიშვნელობა, იმ გაგებით, რომ იგი შეგვიძლია გამოვიძახოთ ფრჩხილების დახმარებით: sayHi(). მაგრამ ის მაინც მნიშვნელობაა. ამიტომ, ჩვენ შეგვიძლია იგივე გავაკეთოთ,

რასაც ნებისმიერი სხვა მნიშვნელობის შემთხვევაში გავაკეთებდით.

ჩვენ შეგვიძლია დავაკოპიროთ ფუნქცია სხვა ცვლადში:

```
function sayHi() {   // ვქმნით ფუნქციას
  alert( "Hello!" );
}

let func = sayHi;    // ვაკოპირებთ ფუნქციას

func();   // Hello!    // ვიძახებთ ფუნქციის ასლს. მუშაობს!
sayHi(); // Hello!   //     ძველიც მუშაობს
```

მოდით, უფრო დაწვრილებით განვიხილოთ თუ რა მოხდა აქ:

1. ფუნქციის გამოცხადებამ შექმნა ფუნქცია და მისი მნიშვნელობა მიანიჭა ცვლადს სახელად sayHi;

2. მესამე სტრიქონში ჩვენ დავაკოპირეთ მისი მნიშვნელობა func ცვლადში. გაითვალისწინეთ (კიდევ ერთხელ), რომ sayHi-ს შემდეგ არ არის ფრჩხილები. ისინი რომ ყოფილიყვნენ, მაშინ გამოსახულება func = sayHi() func ცვლადში ჩაწერდა sayHi() ფუნქციის გამოძახების შედეგს და არა თავად sayHi ფუნქციას;

3. ფუნქციის გამოძახება შესაძლებელია ორივე sayHi() და func() ცვლადის გამოყენებით.

გაითვალისწინეთ, რომ ჩვენ ასევე შეგვიძლია გამოგვეყენებინა Function Expression, რათა შეგვექმნა sayHi პირველ სტრიქონში:

```
let sayHi = function() {
  alert( "Hello!" );
```

```
};

let func = sayHi;
// ...
```

შედეგი იქნებოდა იგივე.

ფუნქცია „call back“

განვიხილოთ ფუნქციის სხვა მაგალითი, ფუნქციის მნიშვნელობად გადაცემა.

მოდით დავწეროთ ფუნქცია ask(question, yes, no) სამი პარამეტრით:

Question - შეკითხვის ტექსტი;

yes - ფუნქცია, რომელიც უნდა გამოიძახოთ, თუ პასუხი არის „დიახ“;

no - ფუნქცია, რომელიც უნდა გამოიძახოთ, თუ პასუხი არის „არა“.

ჩვენმა ფუნქციამ უნდა დასვას შეკითხვა და, იმისდა მიხედვით, თუ როგორია მომხმარებლის პასუხი, გამოიძახოს yes() ან no() ფუნქცია:

```
function ask(question, yes, no) {
  if (confirm(question)) yes()
  else no();
}

function showOk() {
  alert( "თქვენ დაეთანხმეთ." );
}
```

```
function showCancel() {
  alert( "თქვენ შეწყვიტეთ მოქმედება." );
}

// ფუნქციები showOk და showCancel გადაეცემა ask ფუნქციას არგუმენტების სახით
ask("თქვენ თანახმა ხართ?", showOk, showCancel);
```

პრაქტიკაში, ასეთი ფუნქციები ძალიან სასარგებლოა. „რეალურ" ask ფუნქციასა და ზემოთ მოცემულ მაგალითს შორის მთავარი განსხვავება ისაა, რომ ის იყენებს მომხმარებელთან ურთიერთობის უფრო რთულ გზას, ვიდრე confirm-ის მარტივი გამოძახება დასადასტურებლად. ბრაუზერებში, ასეთი ფუნქციები ჩვეულებრივ ლამაზ დიალოგურ ფანჯარას გამოიტანს.

Ask ფუნქციის არგუმენტებს ასევე „call back" ფუნქციებს ან უბრალოდ „call back"-ს უწოდებენ.

მთავარი იდეა იმაში მდგომარეობს, რომ ჩვენ პარამეტრების სახით გადავცემთ ფუნქციას და ველით, რომ ის მოგვიანებით, საჭიროების შემთხვევაში, უკუგამოძახებით გამოიძახებს (ინგლ. „call back" - უკუგამოძახება). ჩვენს შემთხვევაში, showOk „yes" პასუხისათვის showOk ხდება „call back", ხოლო „no" პასუხისთვის showCancel ხდება „call back".

ჩვენ შეგვიძლია ეს მაგალითი Function Expression-ის გამოყენებით ბევრად უფრო მოკლედ გადავწეროთ:

```
function ask(question, yes, no) {
  if (confirm(question)) yes()
```

```
  else no();
}

ask(
  "თქვენ თანახმა ხართ?",
  function() { alert("თქვენ დაეთანხმეთ."); },
  function() { alert("თქვენ შეწყვიტეთ მოქმედება."); }
);
```

აქ ფუნქციები გამოცხადებულია უშუალოდ ask(...) გამოძახების შიგნით. მათ სახელები არ აქვთ, ამიტომ მათ ანონიმურებს უწოდებენ. ასეთი ფუნქციები ask-ის გარეთ ხელმისაწვდომი არ არის, მაგრამ ეს არის ზუსტად ის, რაც ჩვენ გვჭირდება.

ფუნქცია არის მნიშვნელობა, რომელიც წარმოადგენს „მოქმედებას“.

ჩვეულებრივი მნიშვნელობები, როგორიცაა სტრიქონები ან რიცხვები, წარმოადგენს მონაცემებს.

მეორეს მხრივ, ფუნქციები შეიძლება ჩაითვალოს როგორც „მოქმედება“.

ჩვენ შეგვიძლია გადავცეთ ისინი ცვლადიდან ცვლადს და შესრულებაზე გავუშვათ საჭიროების შემთხვევაში.

Named Function Expression

Named Function Expression ან NFE ეს არის ტერმინი Function Expression-ისთვის, რომელსაც აქვს სახელი.

მაგალითად, ფუნქცია გამოვაცხადოთ Function Expression-ის საშუალებით:

```
let sayHi = function(who) {
  alert(‘გამარჯობა, ${who}!’);
};
```

და მივანიჭოთ მას სახელი:

```
let sayHi = function func(who) {
  alert(‘გამარჯობა, ${who}!’);
};
```

შენიშვნა: ფუნქციაში alert ოპერატორში გამოყენებულია დახრილი ბრჭყალი (‘) და არა ჩვეულებრივი ბრჭყალი (').

რას გვაძლევსეს სახელი აქ? რა არის ამ ფუნქციის დამატებითი სახელის მიზანი?

პირველ რიგში, გაითვალისწინეთ, რომ ფუნქცია კვლავ განსაზღვრულია, როგორც Function Expression-ი. function-ის შემდეგ „func“-ის დამატება მას არ გადააქცევს Function Declaration-ში გამოცხადებად, რადგან ის მაინც მინიჭების გამოსახულების ნაწილს წარმოადგენს.

ასეთი სახელის დამატება არაფერს არ ცვლის.

ფუნქცია კვლავ ხელმისაწვდომია, როგორც sayHi():

```
let sayHi = function func(who) {
  alert(‘გამარჯობა, ${who}!’);
};


sayHi("გიორგი"); // გამარჯობა, გიორგი!
```

არსებობს ფუნქციის func სახელის ორი მნიშვნელოვანი მახასიათებელი, რომლისთვისაც ეს სახელია მოცემული:

1. ის საშუალებას აძლევს ფუნქციას მიმართოს საკუთარ თავს;

2. ის არ არის ხელმისაწვდომი ფუნქციის გარეთ.

მაგალითად, ქვემოთ მოცემულ მაგალითში sayHi ფუნქცია საკუთარ თავს გამოიძახებს „სტუმარის“ სახელით, თუ არ არის მოცემული who პარამეტრი:

```
let sayHi = function func(who) {
  if (who) {
  alert(‘გამარჯობა, ${who}!’);
  } else {
    func("სტუმარო"); // გამოიყენება func სახელი, რათა
      მიმართოს საკუთარ თავს
  }
};

sayHi(); // გამარჯობა, სტუმარო!

// ხოლო ასე არ მუშაობს:
func(); // შეცდომაა, func განსაზღვრული არ არის (არ არის
              ხელმისაწვდომი ფუნქციის გარეთ)
```

რატომ ვიყენებთ ფუნქციის func სახელს? რატომ არ ვიყენებთ sayHi-ის ჩასმული გამოძახებისთვის?

ზოგადად, ჩვენ ჩვეულებრივ შეგვიძლია ეს გავაკეთოთ:

```
let sayHi = function func(who) {
  if (who) {
  alert(‘გამარჯობა, ${who}!’);
  } else {
```

```
    sayHi("სტუმარო");
  }
};
```

თუმცა, ამ კოდს აქვს პრობლემა, რომელიც იმაში მდგომარეობს, რომ sayHi-ის მნიშვნელობა შეიძლება შეიცვალოს. ფუნქცია შეიძლება სხვა ცვლადს მიენიჭოს და შემდეგ კოდი დაიწყებს შეცდომების გამოტანას:

```
let sayHi = function func(who) {
  if (who) {
  alert(‘გამარჯობა, ${who}!’);
  } else {
    sayHi("სტუმარო");
  }
};

let welcome = sayHi;
sayHi = null;

welcome(); // შეცდომაა, sayHi-ის ჩასმული გამოძახება უკვე
      აღარ მუშაობს!
```

ეს ხდება იმიტომ, რომ არ არსებობს ლოკალური ცვლადი sayHi, გამოიყენება გარე ცვლადი. გამოძახების დროს, ეს გარე ცვლადი sayHi არის null.

არააუცილებელი სახელი, რომელიც შეიძლება ჩასმული იყოს Function Expression-ში, ზუსტად ამ სახის პრობლემის გადასაჭრელად გამოიყენება.

ჩავწეროთ გასწორებული კოდი:

```
let sayHi = function func(who) {
  if (who) {
  alert(‘გამარჯობა, ${who}!’);
  } else {
    func("სტუმარო");
  }
};

let welcome = sayHi;
sayHi = null;

welcome(); // გამარჯობა, სტუმარო!
```

ეს Function Declaration-თან არ მუშაობს.

სინტაქსი new Function

არსებობს ფუნქციის გამოცხადების კიდევ ერთი ვარიანტი. იგი გამოიყენება ძალიან იშვიათად, მაგრამ ზოგჯერ სხვა გამოსავალი არ არის. ფუნქციის გამოცხადების სინტაქსია:

```
let func = new Function([arg1, arg2, ...argN], functionBody);
```

ფუნქცია იქმნება მოცემული არგუმენტებით arg1...argN და ტანით functionBody.

ამის გაგება უფრო ადვილი იქნება კონკრეტული მაგალითით. ქვემოთ მაგალითში გამოცხადებულია ფუნქცია ორი არგუმენტით:

```
let sum = new Function('a', 'b', 'return a + b');
```

```
alert( sum(1, 2) ); // 3
```

ქვემოთ მოცემულია ფუნქცია არგუმენტების გარეშე, ამ შემთხვევაში საკმარისია მხოლოდ ფუნქციის ტანის მითითება:

```
let sayHi = new Function('alert("Hello")');

sayHi(); // Hello
```

მთავარი განსხვავება ფუნქციის გამოცხადების სხვა გზებისგან, რომლებიც ადრე იყო განხილული, არის ის, რომ ფუნქცია იქმნება სტრიქონიდან მთლიანად გაშვების დროს.

ყველა ადრე განხილული ფუნქციის გამოცხადების წესები, პროგრამისტებისაგან მოითხოვდა სკრიპტში ფუნქციის გამოცხადების ჩაწერას.

new Function საშუალებას გაძლევთ ნებისმიერი სტრიქონი ფუნქციად გადააქციოთ. მაგალითად, შეგიძლიათ მიიღოთ ახალი ფუნქცია სერვერიდან და შემდეგ შეასრულოთ ის:

```
let str = ... კოდი დინამიურად მიღებული სერვერიდან...

let func = new Function(str);
func();
```

ეს ძალიან სპეციფიკური შემთხვევების დროს გამოიყენება, მაგალითად, როდესაც ვიღებთ კოდს სერვერიდან, რთულ ვებ-აპლიკაციებში შაბლონიდან ფუნქციის დინამიური კომპილაციისათვის.

ფუნქცია-ისარი

ფუნქციების შესაქმნელად არსებობს კიდევ უფრო მარტივი და ლაკონური სინტაქსი, რომელიც ხშირად უკეთესია ვიდრე Function Expression სინტაქსი.

მას „ფუნქცია-ისარს“ ან „ისრიან ფუნქციას“ (arrow functions) უწოდებენ, რადგან იგი შემდეგნაირად გამოიყურება:

```
let func = (arg1, arg2, ...argN) => expression
```

ასეთი კოდი ქმნის func ფუნქციას არგუმენტებით arg1,..,argN და გამოთვლის მის მარჯვენა მხარეს მდგომ გამოსახულებას expression და აბრუნებს შედეგს.

სხვა სიტყვებით რომ ვთქვათ, ეს არის შემდეგი ჩანაწერის მოკლე ვერსია:

```
let func = function(arg1, arg2, ...argN) {
  return expression;
};
```

განვიხილოთ კონკრეტული მაგალითი:

```
let sum = (a, b) => a + b;
/* ამ მაგალითის უფრო მოკლე ფორმა:
let sum = function(a, b) {
  return a + b;
};
*/
alert( sum(1, 2) ); // 3
```

ანუ, (a, b) => a + b განსაზღვრავს ფუნქციას ორი a და b არგუმენტით, რომელიც გაშვებისას გამოთვლის ისრის მარჯვნივ მდგომ a + b გამოსახულებას და აბრუნებს მის შედეგს.

თუ გვაქვს მხოლოდ ერთი არგუმენტი, მაშინ პარამეტრების ირგვლივ ფრჩხილები შეიძლება გამოტოვოთ, რაც აღნიშვნას კიდევ უფრო მოკლეს გახდის:

```
let double = n => n * 2;  // იგივეა, რაც let double = function(n) {
return n * 2 }

alert( double(3) ); // 6
```

თუ არგუმენტი საერთოდ არ გვაქვს, მაშინ უნდა ჩავწეროთ მხოლოდ მრგვალი ფრჩხილები:

```
let sayHi = () => alert("Hello!");

sayHi();
```

ფუნქცია-ისარი შეიძლება გამოყენებულ იქნას ისევე, როგორც Function Expression.

მაგალითად, ფუნქციის დინამიურად შესაქმნელად:

```
let age = prompt("How old are you?", 18);

let welcome = (age < 18) ?
  () => alert('Hi!') :
  () => alert("Hello!");

welcome();
```

ფუნქცია-ისარი თავიდან შეიძლება უცნაურად და ძნელად წასაკითხად მოგეჩვენოთ, მაგრამ ეს სწრაფად გაივლის, როგორც კი თვალები მიეჩვევა ამ კონსტრუქციებს. ისინი ძალიან მოსახერხებელია უბრალო ერთსტრიქონიანი მოქმედებე-ბისთვის.

ზემოთ მოყვანილ მაგალითებში არგუმენტები განთავსებული იყო =>-ის მარცხნივ, ხოლო მარჯვნივ გამოითვლებოდა გამოსახულების მნიშვნელობა.

ზოგჯერ ჩვენ გვჭირდება რაღაც უფრო რთული, მაგალითად, რამდენიმე ინსტრუქციის შესრულება. ეს ასევე შესაძლებელია, უბრალოდ ინსტრუქციები ფიგურულ ფრჩხილებში უნდა ჩასვათ და აუცილებლად უნდა გამოიყენოთ return ოპერატორი მათ შიგნით, როგორც ჩვეულებრივ ფუნქციაში.

მაგალითი:

```
let sum = (a, b) => { // ფიგურული ფრჩხილი ხსნის მრავალსტრიქონიანი ფუნქციის ტანს
 let result = a + b;
 return result; // ბოლოს აუცილებლად უნდა მივუთითოთ return ოპერატორი
};

alert( sum(1, 2) ); // 3
```

სტანდარტული ფუნქციები.

JavaScript-ში გამოიყენება ქვემოთ ჩამოთვლილი სტანდარტული ფუნქციები:

parseInt (სტრიქონი, ათვლის სისტემის ფუძე) – მითითებულ სტრიქონს გარდაქმნის მთელ რიცხვად, მითითებულ ათვლის სისტემაში (8, 10 ან 16); თუ არ არის მითითებული ფუძე, მაშინ იგულისხმება ათვლის ათობითი სისტემა.

მაგალითები:

parseInt ("2.5")	შედეგი = 2
parseInt ("-16.254")	შედეგი = -16
parseInt ("15 ლარი")	შედეგი = 15
parseInt ("ფასი 150 ლარი")	შედეგი = NaN

parseFloat (სტრიქონი, ათვლის სისტემის ფუძე) – მითითებულ სტრიქონს გარდაქმნის მცურავმძიმიან რიცხვად, მითითებულ ათვლის სისტემაში (8, 10 ან 16); თუ არ არის მითითებული ფუძე, მაშინ იგულისხმება ათვლის ათობითი სისტემა.

მაგალითები:

parseFloat ("2.5")	შედეგი = 2.5
parseFloat ("-16.254"”)	შედეგი = -16.254
parseFloat ("15.3 ლარი")	შედეგი = 15.3
parseFloat ("ფასი 150 ლარი")	შედეგი = NaN

isNaN (მნიშვნელობა) – შედეგი არის true (ჭეშმარიტი), თუ პარამეტრად მითითებული მნიშვნელობა არ არის რიცხვი, წინააღმდეგ შემთხვევაში – false (მცდარი).

მაგალითები:

isNaN (123)	შედეგი false
isNaN ("123")	შედეგი false

```
isNaN ("tel. 123456)                    შედეგი true
isNaN (true)                            შედეგი false
isNaN (false)                           შედეგი false
isNaN ("მიხო")                          შედეგი true
```

eval (სტრიქონი) – გამოითვლება სტრიქონში მითითებული გამოსახულების მნიშვნელობა; გამოსახულება ჩაწერილი უნდა იყოს JavaScript ენაზე (არ უნდა შეიცავდეს HTML-ის ტეგებს).

მაგალითი:

```
let y = 5
let x = "if ( y < 10 )  { y = y + 2 }"
eval (x)                                შედეგი x ტოლია 7-ის
```

escape (სტრიქონი) – შედეგი არის %XX სახის სტრიქონი, სადაც XX – მითითებული სიმბოლოს ASCII-კოდია; ასეთ სტრიქონს კიდევ escape-მიმდევრობასაც უწოდებენ. ASCII-კოდი არის მოცემული სიმბოლოს კოდი თექვსმეტობით სისტემაში, რომლის წინ მითითებულია სიმბოლო "%". მაგალითად, ჰარი (ინტერვალი) escape-მიმდევრობაში წარმოდგება როგორც %20.

unescape (სტრიქონი) – ახდენს პირუკუ გარდაქმნას.

typeOf (ობიექტი) – სიმბოლური სტრიქონის სახით აბრუნებს მითითებული ობიექტის ტიპს. მაგალითად, „boolean“, „function“ და სხვა.

რეკურსიული ფუნქციები

რეკურსია არის პროგრამირების ტექნიკა, რომელიც სასარგებლოა იმ სიტუაციებში, როდესაც დავალება შეიძლება

ბუნებრივად დაიყოს რამდენიმე მსგავს, მაგრამ უფრო მარტივ ამოცანად, ან როდესაც დავალება შეიძლება დაყვანილ იყოს მარტივ ქმედებებამდე, ან ვიმუშაოთ მონაცემთა გარკვეულ სტრუქტურებთან.

ფუნქციის ტანში დავალების შესრულებისას შეიძლება მოხდეს სხვა ფუნქციების გამოძახება ქვედავალებების შესასრულებლად. განსაკუთრებული შემთხვევაა, როდესაც ფუნქცია თავის თავს იძახებს. ამას რეკურსია ეწოდება.

განვიხილოთ პირველი მაგალითი და დავწეროთ sub(x,n) ფუნქცია, რომელიც x რიცხვს აიყვანს n ნატურალურ ხარისხში. სხვა სიტყვებით რომ ვთქვათ x გავამრავლოთ თავის თავზე n-ჯერ.

sub(2, 2) = 4

sub(2, 3) = 8

sub(2, 4) = 16

ამ ამოცანის გადაწყვეტის ორი მეთოდი განვიხილოთ.

1. იტერაციული მეთოდი - for ციკლის გამოყენებით:

```
function sub(x, n) {
  let res = 1;

  // res მნიშვნელობას ვამრავლებთ x-ზე n-ჯერ ციკლში
  for (let i = 0; i < n; i++) {
    res *= x;
  }

  return res;
}
```

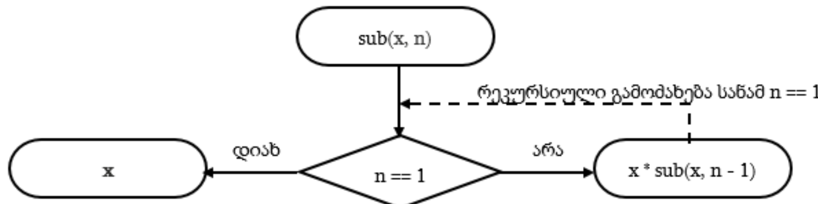
```
alert( sub(2, 3) ); // 8
```

2. რეკურსიული მეთოდი - ამოცანის გამარტივება და ფუნქცია იძახებს თავის თავს:

```
function sub(x, n) {
  if (n == 1) {
    return x;
  } else {
    return x * sub(x, n - 1);
  }
}

alert( sub(2, 3) ); // 8
```

ყურადღება მიაქციეთ იმას, რომ ფუნქციის რეკურსიული ვერსია ფუნდამენტურად განსხვავებულია. როდესაც sub(x,n) ფუნქციის გამოძახება მოხდება, შესრულება ორ ტოტად იყოფა:



1. თუ n == 1, მაშინ ყველაფერი მარტივია. ამ განშტოებას რეკურსიის ბაზას უწოდებენ, რადგან ის მაშინვე ცხად შედეგს იწვევს: sub(x, 1) უდრის x.

2. ჩვენ შეგვიძლია წარმოვადგინოთ sub(x, n) როგორც: x * sub(x, n - 1). რომელიც მათემატიკაში იწერება როგორც: $x^n = x * x^{n-1}$. ეს განშტოება რეკურსიის საფეხურია: პრობლემას ვამცირებთ

უფრო მარტივ მოქმედებამდე (გამრავლება x-ზე) და უფრო მარტივ ანალოგიურ ამოცანამდე (sub უფრო მცირე n-ით). შემდეგი ნაბიჯები აადვილებს დავალებას, სანამ n არ მიაღწევს 1-ს.

ამბობენ, რომ sub ფუნქცია თავის თავს რეკურსიულად უწოდებს n == 1-მდე.

მაგალითად, sub(2,4) გამოთვლის რეკურსიული ვერსია შემდეგი ნაბიჯებისგან შედგება:

```
sub(2, 4) = 2 * sub(2, 3)
sub 2, 3) = 2 * sub(2, 2)
sub(2, 2) = 2 * sub(2, 1)
sub(2, 1) = 2
```

ამგვარად, რეკურსია გამოიყენება, როდესაც ფუნქციის გამოთვლები შეიძლება შემცირდეს მის უფრო მარტივ გამოძახებამდე და შეიძლება შემცირდეს კიდევ უფრო მარტივზე და ასე შემდეგ, სანამ მნიშვნელობა არ გახდება ცხადი.

პრობლემის რეკურსიული გადაწყვეტა ჩვეულებრივ უფრო მოკლეა, ვიდრე განმეორებითი.

თუ გამოვიყენებთ პირობით ოპერატორს ?-ს if-ის ნაცვლად, ჩვენ შეგვიძლია sub(x, n), მაშინ ფუნქციის კოდი შეგვიძლია უფრო ლაკონური, მაგრამ მაინც ადვილად წასაკითხი გავხადოთ:

```
function sub(x, n) {
  return (n == 1) ? x : (x * sub(x, n - 1));
}
```

ჩადგმული გამოძახებების საერთო რაოდენობას (მათ შორის პირველის) რეკურსიის სიღრმე ეწოდება. ჩვენს შემთხვევაში, ეს იქნება n-ის ტოლი.

მაქსიმალური რეკურსიის სიღრმე JavaScript ძრავით არის შეზღუდული. ჩვენ შეგვიძლია დავითვალოთ 10000 ჩადგმული გამოძახება, ზოგიერთი ინტერპრეტატორი უფრო მეტს საშუალებას იძლევა, მაგრამ მათი უმეტესობისთვის 100000 გამოძახებით შემოისაზღვრება.

ეს ზღუდავს რეკურსიის გამოყენებას, მაგრამ ის მაინც ფართოდ გამოიყენება: დიდი რაოდენობის პრობლემების გადასაჭრელად, გადაჭრის რეკურსიული გზას უფრო მარტივ კოდამდე მივყავართ, რომლის მხარდაჭერაც უფრო ადვილია.

ნებისმიერი რეკურსია შეიძლება ციკლად გარდაიქმნას. როგორც წესი, ციკლის ვარიანტი უფრო ეფექტურია. რეკურსიის ციკლად გადაქცევა შეიძლება მარტივი არ იყოს, განსაკუთრებით მაშინ, როდესაც ფუნქცია იყენებს სხვადასხვა რეკურსიულ გამოძახებას, რომელთა შედეგები კომბინირებულია, ან როდესაც განშტოება უფრო რთულია.

ხშირად რეკურსიის გამოყენებით კოდი უფრო მოკლეა, უფრო ადვილი გასაგებია. ოპტიმიზაცია ყველგან არ არის საჭირო.

რეკურსია არის დაპროგრამების ტერმინი, რომელიც ნიშნავს ფუნქციის მიერ თავისი თავის გამოძახებას. რეკურსიული ფუნქციები შეიძლება გამოყენებულ იქნას გარკვეული ამოცანების ოპტიმალურად გადასაჭრელად.

როდესაც ფუნქცია თავის თავს იძახებს, ამას რეკურსიის ბიჯი ეწოდება. რეკურსიის ბაზა ეს არის ფუნქციის ისეთი არგუმენტები, რომლებიც ამოცანას იმდენად მარტივს ხდის,

რომ მათი გადაწყვეტა შემდგომ ჩადგმულ გამოძახებებს აღარ საჭიროებს.

რეკურსიულად განსაზღვრული მონაცემთა სტრუქტურა არის მონაცემთა სტრუქტურა, რომელიც შეიძლება განისაზღვროს საკუთარი თავის გამოყენებით.

ნებისმიერი რეკურსიული ფუნქცია შეიძლება გადაიწეროს ციკლის გამოყენებით.

რეკურსიული ფუნქციის გამოყენების მაგალითები

1. გამოვთვალოთ რიცხვთა ჯამი 1-დან n-მდე (1 + 2 + ... + n):

```
sum(1) = 1
sum(2) = 1 + 2 = 3
sum(3) = 1 + 2 + 3 = 6
sum(4) = 1 + 2 + 3 + 4 = 10
...
sum(100) = 1 + 2 + ... + 99 + 100 = 5050
```

ამ ამოცანის გადაწყვეტა სამი ვარიანტით შეიძლება:

1. ციკლის გამოყენებით;
2. რეკურსიული ფუნქციის გამოყენებით - sum(n) = n + sum(n-1) როცა n > 1;
3. არითმეტიკული პროგრესიის ფორმულის გამოყენებით.

ამ ამოცანის ციკლის გამოყენებით გადაწყვეტას შემდეგი სახე ექნება:

```
function sum(n) {
  let s = 0;
```

```
  for (let i = 1; i <= n; i++) {
    s += i;
  }
  return s;
}

alert( sum(100) );  // 5050
```

რეკურსიული ფუნქციის გამოყენებით:

```
function sum(n) {
  if (n == 1) return 1;
  return n + sum(n - 1);
}

alert( sum(100) );  // 5050
```

არითმეტიკული პროგრესიის ფორმულის sum(n) = n*(n+1)/2 გამოყენებით:

```
function sumTo(n) {
  return n * (n + 1) / 2;
}

alert( sumTo(100) );  // 5050
```

უნდა აღინიშნოს, რომ პროგრამა არითმეტიკული პროგრესიის ფორმულის გამოყენებით მუშაობს ყველაზე სწრაფად, რადგან იგი ნებისმიერი n-სთვის იყენებს მხოლოდ სამ ოპერაციას, ხოლო ციკლი და რეკურსია მოითხოვს მინიმუმ n დამატებით ოპერაციას.

ციკლის ვარიანტი მეორეა სისწრაფით. ის უფრო სწრაფია, ვიდრე რეკურსი, რადგან გამოიყენება იგივე რაოდენობის დამატების ოპერაციები, მაგრამ არ არის დამატებითი მიმართვა თავის თავზე. ამიტომ, რეკურსია ამ შემთხვევაში მუშაობს ყველაზე ნელა.

2. ფაქტორიალის გამოთვლა:

ნატურალური რიცხვის ფაქტორიალი ეს არის 1-დან n-მდე რიცხვების ნამრავლი. ფაქტორიალის განმარტება ასე შეიძლება ჩაიწეროს:

```
n! = n * (n - 1) * (n - 2) * ...*1
```

სხვადასხვა n-სათვის ფაქტორიალის მნიშვნელობებია:

```
1! = 1
2! = 2 * 1 = 2
3! = 3 * 2 * 1 = 6
4! = 4 * 3 * 2 * 1 = 24
5! = 5 * 4 * 3 * 2 * 1 = 120
```

ფაქტორიალის განმარტების თანახმად n! შეიძლება ჩაიწეროს როგორც n*(n-1)!.

სხვა სიტყვებით რომ ვთქვათ, fact(n) შეიძლება მივიღოთ როგორც n გამრავლებული fact(n-1)-ის შედეგზე. შედეგი n-1-ისთვის, თავის მხრივ, შეიძლება გამოითვალოს რეკურსიულად და ასე შემდეგ 1-მდე.

```
function fact(n) {
  return (n != 1) ? n * fact(n - 1) : 1;
}
```

```
alert( fact(5) ); // 120
```

ამ რეკურსიული ფუნქციის საბაზისო მნიშვნელობა არის 1. საბაზისო მნიშვნელობად შესაძლებელი იყო ყოფილიყო 0, მაგრამ ეს რეკურსიას კიდევ ერთ ბიჯს დაუმატებდა:

```
function fact(n) {
  return n ? n * fact(n - 1) : 1;
}


alert( fact(5) ); // 120
```

3. ფიბონაჩის რიცხვის გამოთვლა:

ფიბონაჩის რიცხვების მიმდევრობა განისაზღვრება ფორმულით $F_n = F_{n-1} + F_{n-2}$. ანუ შემდეგი რიცხვი მიიღება წინა ორი რიცხვის ჯამით.

პირველი ორი რიცხვი არის 1, შემდეგი 2(1+1), შემდეგი 3(1+2), 5(2+3) და ასე შემდეგ: 1, 1, 2, 3, 5, 8, 13, 21... .

ფიბონაჩის რიცხვები ოქროს კვეთასთან[3] და ჩვენს გარშემო არსებულ ბევრ ბუნებრივ მოვლენასთანაა მჭიდრო კავშირში.

---

[3] ოქროს კვეთა (ოქროს პროპორცია, ოქროს შუალედი) — ჰარმონიული გაყოფა მთელისა ისეთ ორ არატოლ ნაწილად, როდესაც მცირე ნაწილი ისე შეეფარდება დიდს, როგორც დიდი მთელს და პირიქით, მთელი ისე შეეფარდება დიდს, როგორც დიდი მცირეს. ალგებრულად ოქროს კვეთა დაიყვანება შემდეგი განტოლების ამოხსნამდე:

$$\frac{a+b}{a} = \frac{a}{b} = \varphi,$$

სადაც φ ოქროს კვეთის რიცხვია.

დაწერეთ ფუნქცია fib(n), რომელიც აბრუნებს მე-n-ე ფიბონაჩის რიცხვს.

n-ის სხვადასხვა მნიშვნელობისათვის ფიბონაჩის რიცხვის მნიშვნელობებია:

```
alert(fib(3)); // 2
alert(fib(4)); // 3
alert(fib(5)); // 5
alert(fib(6)); // 8
alert(fib(7)); // 13
  . . .
alert(fib(50)); // 12586269025
```

ზემოთ მოყვანილი მაგალითიდან ფუნქციის გაშვებამ სწრაფად უნდა იმუშაოს. fib(77) გამოთვლას უნდა დასჭირდეს არაუმეტეს რამდენიმე წამი.

ჯერ რეკურსიული ფუნქციის მეშვეობით გამოვთვალოთ ფიბონაჩის რიცხვები:

```
function fib(n) {
  return n <= 1 ? n : fib(n - 1) + fib(n - 2);
}


alert( fib(3) ); // 2
alert( fib(7) ); // 13
alert( fib(10) ); // 13
fib(50); // ითვლის ძალიან დიდხანს
```

n-ის დიდი მნიშვნელობებისთვის, ეს გამოთვლები იმუშავებს ძალიან დიდი ხნის განმავლობაში. მაგალითად,

fib(50)-ს შეუძლია ბრაუზერის გათიშვა გარკვეული ხნით, რაც იკავებს პროცესორის მთელ რესურსს.

ეს არის იმის გამო, რომ ფუნქცია ქმნის ჩადგმული გამოძახებების ფართო ხეს. ამ შემთხვევაში, ცალკეული მნიშვნელობები მრავალჯერ გამოითვლება.

მაგალითად, შევხედოთ გამოთვლის ამონაწერს fib(5):

```
...
fib (5) = fib (4) + fib (3)
fib (4) = fib (3) + fib (2)
...
```

აქ ხედავთ, რომ fib(3) მნიშვნელობა საჭიროა ერთდროულად fib(5)-ის და fib(4)-ის გამოთვლებისთვისაც. კოდში იგი შესრულდება ორჯერ, სრულიად დამოუკიდებლად.

თქვენ ხედავთ, რომ fib(3)-ის მნიშვნელობა ითვლება ორჯერ, ხოლო fib(2)-ის სამჯერ. გამოთვლების საერთო რაოდენობა n-ზე ბევრად სწრაფად იზრდება, რაც მას დიდს ხდის n=50-თვისაც კი.

ამის ოპტიმიზაცია შეგვიძლია უკვე გამოთვლილი მნიშვნელობების დამახსოვრების გზით: თუ მნიშვნელობა, ვთქვათ, fib(3) გამოითვლება ერთხელ, მაშინ ჩვენ უბრალოდ ხელახლა ვიყენებთ ამ მნიშვნელობას შემდგომი გამოთვლებისთვის.

კიდევ ერთი ვარიანტი იქნება რეკურსიაზე უარის თქმა და სრულიად განსხვავებული ციკლზე დაფუძნებული ალგორითმის გამოყენება.

იმის ნაცვლად, რომ დაიწყოთ n-დან და გამოთვალოთ თქვენთვის საჭირო წინა მნიშვნელობები, შეგიძლიათ დაწეროთ

ციკლი, რომელიც იწყება 1-ით და 2-ით, შემდეგ მიიღებს fib(3)-ს, როგორც მათ ჯამს, შემდეგ fib(4)-ს, როგორც წინა მნიშვნელობების ჯამს. შემდეგ fib(5) და ასე შემდეგ, საბოლოო შედეგამდე. თითოეულ ნაბიჯზე, ჩვენ მხოლოდ თანმიმდევრობით უნდა გვახსოვდეს წინა ორი რიცხვის მნიშვნელობა.

```
function fib(n) {
  let a = 1;
  let b = 1;
  for (let i = 3; i <= n; i++) {
    let c = a + b;
    a = b;
    b = c;
  }
  return b;
}

alert( fib(3) ); // 2
alert( fib(7) ); // 13
alert( fib(10) ); // 55
alert(fib(50) );
```

n=50-სთვის მიიღება:

This page says

12586269025

OK

ციკლი იწყება i=3-დან, რადგან მიმდევრობის პირველი და მეორე მნიშვნელობები მოცემულია a=1, b=1.

ამ მეთოდს ეწოდება დინამიური პროგრამირება ქვემოდან ზევით.

რეკურსიული გამოთვლების დროსაც რეალურად იგივე ხდება: გაივლის მთელ სიას, იმახსოვრებს ელემენტებს ერთმანეთში ჩადგმული გამოძახებების ჯაჭვში (შესრულების კონტექსტში) და შემდეგ გამოიტანს შედეგებს.

setTimeout და setInterval

ჩვენ შეგვიძლია გამოვიძახოთ ფუნქცია არა მოცემულ მომენტში, არამედ მოგვიანებით, განსაზღვრული დროის ინტერვალის შემდეგ. ამას ეწოდება „გამოძახების დაგეგმვა“.

ამისთვის ორი მეთოდი არსებობს:

- setTimeout საშუალებას გაძლევთ ერთხელ გამოიძახოთ ფუნქცია გარკვეული დროის ინტერვალის შემდეგ;
- setInterval საშუალებას გაძლევთ რეგულარულად გამოიძახოთ ფუნქცია, გამოძახება გაიმეოროთ გარკვეული დროის ინტერვალით.

ეს მეთოდები არ არის JavaScript-ის სპეციფიკაციის ნაწილი. მაგრამ JS-კოდის გაშვების დროის უმეტესობას აქვს შიდა დამგეგმავი და უზრუნველყოფს ამ მეთოდებზე წვდომას. კერძოდ, ისინი მხარდაჭერილია ყველა ბრაუზერის და Node.js-ის მიერ.

setTimeout

setTimeout-ის სინტაქსია:

```
let timerId = setTimeout(func|code, [delay], [arg1], [arg2], ...);
```

სადაც func|code შესასრულებელი კოდის ფუნქცია ან სტრიქონია. როგორც წესი, ეს ფუნქციაა. ისტორიული მიზეზების გამო, თქვენ ასევე შეგიძლიათ გადასცეთ კოდის სტრიქონი, მაგრამ ეს რეკომენდებული არ არის;

delay გაშვების წინ დაყოვნება მილიწამებში (1000 მიკროწმ = 1 წმ). ნაგულისხმევი მნიშვნელობა არის 0;

arg1, arg2… ფუნქციისთვის გადასაცემი არგუმენტები

მაგალითად, მოცემული კოდი იძახებს sayHi() ხუთი წამის შემდეგ:

```
 function sayHi() {
  alert('Hello!');
}

setTimeout(sayHi, 5000);
```

არგუმენტებით:

```
function sayHi(phrase, who) {
  alert( phrase + ' ' + who );
}

setTimeout(sayHi, 1000, "Hello!", "Georgi");
```

This page says

Hello! Georgi

OK

თუ პირველი არგუმენტი არის სტრიქონი, მაშინ JavaScript შექმნის მისგან ფუნქციას.

ეს ასევე იმუშავებს:

```
setTimeout("alert('Hello!')", 1000);
```

სტრიქონის გამოყენება არ არის რეკომენდებული. ამის ნაცვლად, გამოიყენეთ ანონიმური ფუნქციები. მაგალითად, ასე:

```
setTimeout(() => alert('Hello!'), 1000);
ან
setTimeout(function(){ alert('Hello!')}, 1000);
```

დამწყები დეველოპერები ზოგჯერ ცდებიან, ფუნქციის შემდეგ ფრჩხილებს () უმატებენ:

```
setTimeout(sayHi(), 1000);
```

ეს არ მუშაობს, რადგან setTimeout მოელის ფუნქციის მითითებას. აქ sayHi() იწყებს ფუნქციის შესრულებას და შესრულების შედეგი იგზავნება setTimeout-ში. ჩვენს შემთხვევაში, sayHi()-ის შესრულების შედეგი არის undefined (რადგან ფუნქცია არაფერს აბრუნებს), ამიტომ დაგეგმილი არაფერია.

clearTimeout

setTimeout-ის გამოძახება აბრუნებს „ტაიმერის იდენტიფიკატორს“ timerId-ს, რომელიც შეიძლება გამოყენებულ იქნას შემდგომი შესრულების გასაუქმებლად.

სინტაქსი გაუქმებისთვის:

```
let timerId = setTimeout(...);
clearTimeout(timerId);
```

ქვემოთ მოცემულ კოდში ჩვენ ვგეგმავთ ფუნქციის გამოძახებას და შემდეგ ვაუქმებთ მას (უბრალოდ გადავიფიქრეთ). შედეგად, არაფერი ხდება:

```
let timerId = setTimeout(() => alert("არაფერი ხდება"), 1000);
alert (timerId); // ტაიმერის იდენტიფიკატორი
```

```
clearTimeout(timerId);
alert (timerId); // იგივე იდენტიფიკატორი (გაუქმების შემდეგ
    ღებულობს მნიშვნელობას null)
```

როგორც alert გამოტანის ოპერატორიდან ვხედავთ, ბრაუზერში ტაიმერის იდენტიფიკატორი არის რიცხვი. სხვა გარემოში, ეს შეიძლება იყოს რაღაც სხვა. მაგალითად, Node.js აბრუნებს ტაიმერის ობიექტს დამატებითი მეთოდებით.

ე. წ. ტაიმერის იდენტიფიკატორები განსხვავებულია ზემოთ მოცემული ორივე ფუნქციის გამოყენების დროს და ზოგჯერ პროგრამულ კოდში ხდება გაუგებრობა იდენტიფიკატორების გაუქმებასთან დაკავშირებით, ამიტომ აღნიშნული ფუნქციების გამოყენებისას პროგრამული კოდის ყურადღებით შემოწმებაა საჭირო.

setInterval

setInterval მეთოდს აქვს იგივე სინტაქსი, როგორც setTimeout:

```
let timerId = setInterval(func|, [delay], [arg1], [arg2], ...);
```

ყველა არგუმენტს იგივე მნიშვნელობები აქვს. მაგრამ განსხვავება ამ მეთოდსა და setTimeout-ს შორის არის ის, რომ ფუნქციის გაშვება ხდება არა ერთხელ, არამედ პერიოდულად განსაზღვრული დროის ინტერვალით.

ფუნქციის შემდგომი შესრულების შესაჩერებლად, უნდა გამოიძახოთ clearInterval(timerId) ფუნქცია.

შემდეგი მაგალითი ბეჭდავს შეტყობინებას ყოველ 2 წამში. 10 წამის შემდეგ გამოტანა წყდება:

```
// გაიმეორე 2 წამის ინტერვალით
let timerId = setInterval(() => alert('tick'), 2000);

// გამოტანა შეაჩერე 10 წამის შემდეგ
setTimeout(() => {clearInterval(timerId); alert('stop'); }, 10000);
```

ბრაუზერების უმეტესობაში, მათ შორის Chrome-სა და Firefox-ში, შიდა მრიცხველი გამოტანის დროს alert/confirm/prompt მუდმივად მიდის.

ასე რომ, თუ ზემოთ მოცემულ კოდს გაუშვებთ და დაელოდებით რამდენიმე წამს, სანამ alert-ს დახურავთ, შემდეგი alert გამოჩნდება, როგორც კი წინას დახურავთ. alert შეტყობინებებს შორის დროის ინტერვალი 2 წამზე ნაკლები იქნება.

ობიექტები

როგორც ზემოთ ავღნიშნეთ JavaScript-ში 8 ძირითადი მონაცემთა ტიპია. შვიდ მათგანს უწოდებენ „პრიმიტიულს“, რადგან ისინი შეიცავს მხოლოდ ერთ მნიშვნელობას (იქნება ეს სტრიქონი, რიცხვი თუ სხვა რამ).

ობიექტები გამოიყენება სხვადასხვა მნიშვნელობებისა და უფრო რთული ერთეულების კოლექციების შესანახად. ობიექტები ხშირად გამოიყენება JavaScript-ში, ისინი ენის ერთ-ერთი საფუძველია.

ობიექტის შექმნა შესაძლებელია ფიგურული ფრჩხილების გამოყენებით {…} არააუცილებელი თვისებების სიით. თვისება არის „გასაღები:მნიშვნელობა“ წყვილი, სადაც გასაღები არის სტრიქონი (ასევე უწოდებენ „თვისების სახელს“) და მნიშვნელობა, რომელიც შეიძლება იყოს ნებისმიერი.

ჩვენ შეგვიძლია წარმოვადგინოთ ობიექტი, როგორც ყუთი საქაღალდეებით. თითოეული მონაცემთა ელემენტი ინახება საკუთარ საქაღალდეში, რომელზეც დაწერილია საკვანძო სიტყვა. გასაღებით, საქაღალდის პოვნა ადვილია, ამოიღოთ ან ჩაამატოთ მასში რაიმე.

ცარიელი ობიექტი ("ცარიელი ყუთი") შეიძლება შეიქმნას ორიდან ერთ-ერთი სინტაქსის გამოყენებით:

```
let user = new Object();
let user = {};
```

უფრო ხშირად ობიექტის შესაქმნელად ფუგურული ფრჩხილები {…} გამოიყენება. ობიექტის ასეთ გამოცხადებას

ობიექტის ლიტერალი (Literals ლიტერალები მარტივად რომ ვთქვათ არის „ღირებულება“) ან ლიტერალური ნოტაცია ჰქვია.

ლიტერალი და თვისება

ლიტერალური სინტაქსის {...} გამოყენების დროს ჩვენ ობიექტში შეგვიძლია დაუყოვნებლივ ჩავდოთ რამდენიმე თვისება „გასაღები:მნიშვნელობა“ წყვილის სახით. მაგალითად:

```
let user = {     // ობიექტი
  name: "გიორგი",   // name გასაღებით ინახება მნიშვნელობა „გიორგი“
  age: 30        // age გასაღებით ინახება მნიშვნელობა 30
};
```

თითოეულ თვისებას აქვს გასაღები (ასევე ეწოდება „სახელი“ ან „იდენტიფიკატორი“). თვისების სახელს მოსდევს ორწერტილი „:“, რასაც მოჰყვება თვისების მნიშვნელობა. თუ ობიექტს აქვს რამდენიმე თვისება, ისინი ერთმანეთისაგან მძიმეებით გამოიყოფა.

user ობიექტს ახლა ორი თვისება აქვს:

1. პირველი თვისება სახელწოდებით „name“ და მნიშვნელობით „გიორგი“;

2. მეორე თვისება სახელწოდებით „age“ და მნიშვნელობით 30.

შეგვიძლია ვთქვათ, რომ ჩვენი მომხმარებლის ობიექტი არის ყუთი, რომელშიც ორი საქაღალდეა წარწერით „name“ და „age“.

ჩვენ შეგვიძლია ნებისმიერ დროს დავამატოთ მას ახალი საქაღალდეები, წავშალოთ ან წავიკითხოთ ნებისმიერი საქაღალდის შინაარსი.

ობიექტის თვისებებზე მიმართვისთვის გამოიყენება წერტილი:

```
// ობიექტის თვისებაზე მიმართვა:
alert( user.name ); // გიორგი
alert( user.age ); // 30
```

მნიშვნელობა შეიძლება იყოს ნებისმიერი ტიპის. მოდით დავამატოთ თვისება ლოგიკური მნიშვნელობით:

```
user.isAdmin = true;
```

ობიექტის თვისების წასაშლელად გამოიყენება ოპერატორი delete:

```
delete user.age;
```

თვისების სახელი შეიძლება რამდენიმე სიტყვისაგან შედგებოდეს, ამ შემთხვევაში სახელი ბრჭყალებში უნდა მოთავსებული:

```
let user = {
  name: "გიორგი",
  age: 30,
  "likes birds": true // რამდენიმე სიტყვისაგან შემდგარი სახელი
                        ბრჭყალებშია მოთავსებული
};
```

ობიექტის ბოლო თვისება შეიძლება მძიმით დამთავრდეს.

```
let user = {
  name: "გიორგი",
  age: 30,
}
```

ამას ჰქვია „დაკიდებული მძიმე“. ეს მიდგომა ამარტივებს თვისებების დამატებას, ამოღებას და გადატანას, რადგან ობიექტის ყველა სტრიქონი ერთნაირი ხდება.

const-ით გამოცხადებული ობიექტი შეიძლება შეიცვალოს. მაგალითი:

```
const user = {
  name: "გიორგი"
};
user.name = "ნიკოლოზი";
alert(user.name); // ნიკოლოზი
```

შეიძლება იფიქროთ, რომ user.name = "Pete"; სტრიქონმა უნდა გამოიწვიოს შეცდომა, მაგრამ არა, აქ ყველაფერი რიგზეა. ფაქტია, რომ const დეკლარაცია იცავს მხოლოდ მომხმარებლის ცვლადს ცვლილებებისგან და არა მის თვისებას.

const-ის დეკლარაცია შეცდომას მხოლოდ მაშინ გამოიწვევს, თუ ცვლადს სხვა მნიშვნელობას მივანიჭებთ: user=....

კვადრატული ფრჩხილები

თვისებებისთვის, რომელთა სახელები რამდენიმე სიტყვისგან შედგება, მათ მნიშვნელობაზე წვდომა „წერტილის საშუალებით“ არ მუშაობს:

```
// ეს გამოიწვევს სინტაქსურ შეცდომას
user.likes birds = true
```

JavaScript ხედავს, რომ ჩვენ შევდივართ user.likes თვისებაზე და შემდეგ მოდის გაუგებარი სიტყვა birds. შედეგი არის შეცდომა.

წერტილი მოითხოვს, რომ გასაღები იყოს დასახელებული ცვლადების დასახელების წესების მიხედვით. ანუ მას არ უნდა ჰქონდეს ჰარები, არ იწყებოდეს რიცხვით და არ შეიცავდა სპეციალურ სიმბოლოებს გარდა $ და _ სიმბოლოებისა.

ასეთი შემთხვევებისთვის არსებობს თვისებებზე წვდომის ალტერნატიული გზა კვადრატული ფრჩხილები. ეს იმუშავებს თვისების ნებისმიერი სახელთან:

```
let user = {};

// თვისების მნიშვნელობის მინიჭება
user["likes birds"] = true;

// თვისების მნიშვნელობის მიღება
alert(user["likes birds"]); // true

// თვისების წაშლა
delete user["likes birds"];
```

ახლა ყველაფერი რიგზეა. გაითვალისწინეთ, რომ კვადრატულ ფრჩხილებში ჩასმული სტრიქონი ჩასმულია ბრჭყალებში (აქ ნებისმიერი ტიპის ბრჭყალების გამოყენებაა შესაძლებელი).

კვადრატული ფრჩხილები ასევე საშუალებას გაძლევთ მიმართოთ თვისებას, რომლის სახელი შეიძლება იყოს გამოთვლის შედეგი. მაგალითად, თვისების სახელი შეიძლება შენახული იყოს ცვლადში:

```
let key = "likes birds";

// ეს იგივეა, რაც user["likes birds"] = true;
user[key] = true;
```

აქ შეიძლება key ცვლადის გამოთვლა მოხდეს გაშვების დროს ან დამოკიდებული იყოს მომხმარებლის მიერ ინფორმაციის შეყვანაზე. ამის შემდეგ ჩვენ ვიყენებთ მას თვისებაზე წვდომისათვის. ეს ჩვენ კოდის დიდ მოქნილობას აძლევს.

მაგალითი:

```
let user = {
  name: "გიორგი",
  age: 30
};

let key = prompt("ღის გაგება გსურთ მომხმარებლის შესახებ?", "name");

// ამ შემთხვევაში თვისებაზე წვდომა ხორციელდება ცვლადის გამოყენებით
alert( user[key] ); // გიორგი (თუ მომხმარებელმა შეიყვანა სიტყვა „name“)
```

```
alert( user[key] ); // 30 (თუ მომხმარებელმა შეიყვანა სიტყვა „age“)
```

უნდა გვახსოვდეს, რომ ამ შემთხვევაში აუცილებლად უნდა გამოვიყენოთ კვადრატული ფრჩხილები.

გამოთვლადი თვისებები

ჩვენ შეგვიძლია გამოვიყენოთ კვადრატული ფრჩხილები ლიტერალურ ნოტაციაში გამოთვლადი თვისებების შესაქმნელად.

მაგალითი:

```
let fruit = prompt("What fruit to buy?", "apple");

let bag = {
  [fruit]: 5, // თვისების სახელის აღება მოხდება ცვლადიდან fruit
};

alert( bag.apple ); // 5, თუ fruit="apple"
```

გამოთვლადი თვისების მნიშვნელობა მარტივია: აღნიშვნა [fruit] ნიშნავს, რომ თვისების სახელი უნდა იყოს აღებული fruit ცვლადიდან.

და თუ ვიზიტორი შეიყვანს სიტყვას "apple", მაშინ bag ობიექტი ახლა შეიცავს {apple: 5} თვისებას.

კვადრატული ფრჩხილები გაძლევთ ბევრად მეტ შესაძლებლობას, ვიდრე წერტილი. ისინი საშუალებას გაძლევთ გამოიყენოთ თვისებების ნებისმიერი სახელები და ცვლადები, თუმცა მათ უფრო რთული კონსტრუქციების კოდი სჭირდებათ.

უმეტეს შემთხვევაში, სადაც თვისებების სახელები ცნობილია და მარტივია, წერტილით აღნიშვნა გამოიყენება. თუ რაიმე უფრო რთული გვჭირდება, მაშინ ვიყენებთ კვადრატულ ფრჩხილებს.

თვისება ცვლადისაგან

რეალურ კოდში, ჩვენ ხშირად გვჭირდება არსებული ცვლადების გამოყენება, როგორც იმავე სახელწოდების თვისებების მნიშვნელობები.

მაგალითად:

```
function makeUser(name, age) {
  return {
    name: name,
    age: age
    // ...სხვა თვისებები
  };
}

let user = makeUser("გიორგი", 30);
alert(user.name); // გიორგი
```

ზემოთ მოცემულ მაგალითში, თვისებები name და age იგივეა, რაც იმ ცვლადების სახელები, რომლებსაც ჩვენ ვანაცვლებთ ამ თვისებების მნიშვნელობების მაგიერ. ეს მიდგომა იმდენად გავრცელებულია, რომ ამ აღნიშვნის გასამარტივებლად არსებობს სპეციალური მოკლე ჩანაწერები.

name:name-ის ნაცვლად შეგვიძლია უბრალოდ დავწეროთ name:

```
function makeUser(name, age) {
  return {
    name, // იგივეა, რაც name: name
    age   // იგივეა, რაც age: age
    // ...
  };
}
```

ერთსა და იმავე ობიექტში ჩვენ შეგვიძლია გამოვიყენოთ როგორც ჩვეულებრივი, ასევე მოკლე თვისებები:

```
let user = {
  name,  // იგივეა, რაც name:name
  age: 30
};
```

როგორც უკვე ვიცით, ცვლადის სახელი არ შეიძლება იყოს იგივე, რაც დარეზერვირებული სიტყვები, როგორიცაა „for“, „let“, „return“ და ა.შ.

მაგრამ ობიექტის თვისებებისთვის ასეთი შეზღუდვა არ არსებობს.

სხვა სიტყვებით რომ ვთქვათ, არ არსებობს შეზღუდვები თვისებების დასახელებაზე. ისინი შეიძლება იყოს ნებისმიერი სტრიქონი ან სიმბოლო.

თვისების არსებობის შემოწმება

ბევრი სხვა ენისგან განსხვავებით, JavaScript-ის ობიექტების თავისებურება ის არის, რომ თქვენ შეგიძლიათ წვდომა მიიღოთ ნებისმიერ თვისებაზე. თვისება რომ არც არსებობდეს, შეცდომა არ იქნება!

არარსებულ თვისებაზე წვდომისას, შედეგად undefined ბრუნდება. ეს საშუალებას გაძლევთ უბრალოდ თვისების არსებობა შეამოწმოთ:

```
let user = {};

alert( user.noSuchProperty === undefined ); // true ნიშნავს, რომ თვისება არ არსებობს
```

ასევე არსებობს სპეციალური ოპერატორი in ობიექტში თვისების არსებობის შესამოწმებლად.

ამ ოპერატორის სინტაქსია:

```
"key" in object
```

მაგალითი:

```
let user = { name: "გიორგი", age: 30 };

alert( "age" in user ); // true, user.age არსებობს
alert( "blabla" in user ); // false, user.blabla არ არსებობს
```

გაითვალისწინეთ, რომ თვისების სახელი უნდა იყოს in ოპერატორის მარცხნივ. ჩვეულებრივ ეს სტრიქონი უნდა იყოს მოთავსებული ბრჭყალებში.

თუ ბრჭყალებს გამოვტოვებთ, ეს ნიშნავს, რომ ჩვენ მივუთითებთ ცვლადზე, რომელიც შეიცავს თვისების სახელს. მაგალითად:

```
let user = { age: 30 };

let key = "age";
```

```
alert( key in user ); // true, თვისების სახელი აღებული იყო key ცვლადიდან
```

რისთვის გამოიყენება in ოპერატორი? არ არის საკმარისი undefined-თან შედარება?

უმეტეს შემთხვევაში, undefined-თან შედარება კარგად იმუშავებს. მაგრამ არის განსაკუთრებული შემთხვევა, როცა ის სწორ შედეგს არ გვაძლევს და in ოპერატორი უნდა გამოვიყენოთ.

ეს არის მაშინ, როდესაც თვისება არსებობს, მაგრამ შეიცავს მნიშვნელობას undefined:

```
let obj = {
  test: undefined
};

alert( obj.test ); //  გამოიტანს undefined
alert( "test" in obj ); // true, თვისება არსებობს
```

ზემოთ მოცემულ მაგალითში, obj.test თვისება ობიექტში ტექნიკურად არსებობს. ოპერატორმა სწორად იმუშავა.

მსგავსი სიტუაციები ძალიან იშვიათია, რადგან undefined მნიშვნელობა, როგორც წესი, აშკარად არ მიენიჭება. უცნობი ან ცარიელი თვისებებისთვის ვიყენებთ null მნიშვნელობას. ამრიგად, in ოპერატორი კოდში იშვიათად გამოიყენება.

ციკლი for … in

for..in ციკლი გამოიყენება ობიექტის ყველა თვისების ნახვისათვის. ეს ციკლი განსხვავდება for ციკლისგან, რომელიც ადრე ვისწავლეთ.

მისი სინტაქსია:

```
for (key in object) {
//   ციკლის   ტანი   სრულდება   ობიექტის   თითოეული
თვისებისთვის
}
```

მაგალითად, მოდით ვაჩვენოთ მომხმარებლის ობიექტის ყველა თვისება:

```
let user = {
  name: "გიორგი",
  age: 30,
  isAdmin: true
};

for (let key in user) {
  // გასაღები
  alert( key );  // name, age, isAdmin
  // გასაღების მნიშვნელობა
  alert( user[key] ); // გიორგი, 30, true
}
```

გაითვალისწინეთ, რომ ყველა "for" კონსტრუქცია საშუალებას გვაძლევს გამოვაცხადოთ ცვლადი ციკლის შიგნით, მაგალითად როგორც აქ არის let key.

ასევე, ჩვენ შეგვიძლია გამოვიყენოთ ცვლადის სხვა სახელი. მაგალითად, ხშირად გამოიყენება „for (let prop in obj)“.

ობიექტთა თვისებების მოწესრიგება

მოწესრიგებულია თუ არა ობიექტის თვისებები? სხვა სიტყვებით რომ ვთქვათ, თუ ობიექტის ყველა თვისებას გადავხედავთ, მივიღებთ თუ არა მათ იმავე თანმიმდევრობით, როგორც ვამატებდით?

პასუხი ესეთია: თვისებები დალაგებულია სპეციალური წესით: მთელრიცხვიანი თვისებები დალაგებულია ზრდადობით, დანარჩენები შექმნის თანმიმდევრობით. მოდით უფრო დაწვრილებით განვიხილოთ.

მაგალითად, განვიხილოთ ობიექტი სატელეფონო კოდებით:

```
let codes = {
  "49": "Germany",
  "41": "Switzerland",
  "44": "United Kingdom",
  "1": "USA"
};

for (let code in codes) {
  alert(code); // 1, 41, 44, 49
}
```

თუ ჩვენ ვაკეთებთ საიტს გერმანელი აუდიტორიისთვის, მაშინ ალბათ გვინდა, რომ კოდი 49 იყოს პირველი.

მაგრამ თუ კოდს გავუშვებთ, ჩვენ დავინახავთ სრულიად განსხვავებულ სურათს:

აშშ (1) პირველ ადგილზეა, შემდეგ შვეიცარია (41) და ა.შ.

აკრეფილი კოდები არის ზრდადობით დალაგებული, რადგან ისინი მთელი რიცხვებია: 1, 41, 44, 49.

ტერმინი მთელრიცხვიანი თვისება ნიშნავს სტრიქონს, რომელიც შეიძლება გარდაიქმნას მთელ რიცხვად და უკან გარდაქმნის შემთხვევაში უცვლელი დარჩება.

ანუ "49" არის მთელი რიცხვი თვისების სახელი, რადგან თუ ის გადაკეთდება მთელ რიცხვად და შემდეგ ისევ სტრიქონში, ის არ შეიცვლება. მაგრამ თვისებები „+49“ ან „1.2“ ასეთი არ არის:

```
// Math.trunc - სტანდარტული ფუნქცია, რომელიც შლის წილად ნაწილს
alert( String(Math.trunc(Number("49"))) ); // "49", იგივე ⇒ თვისება მთელრიცხვიანია
alert( String(Math.trunc(Number("+49"))) ); // "49", იგივე არ არის, რაც „+49“ ⇒ თვისება არ არის მთელრიცხვიანი
alert( String(Math.trunc(Number("1.2"))) ); // "1", იგივე არ არის, რაც „1.2“ ⇒ თვისება არ არის მთელრიცხვიანი
```

მეორეს მხრივ, თუ თვისებები არ არის მთელი რიცხვი, მაშინ ისინი დალაგდება შექმნის თანმიმდევრობით, მაგალითად:

```
let user = {
  name: "გიორგი",
  surname: "ზერიძე"
};
user.age = 25; // დავამატოთ კიდევ ერთი თვისება
```

```
// არა მთელრიცხვიანი თვისებები დალაგებული იქნება შექმნის რიგის მიხედვით
for (let prop in user) {
  alert( prop ); // name, surname, age
}
```

ასე რომ, სატელეფონო კოდების პრობლემის გადასაჭრელად, შეგვიძლია ვიხმაროთ ხერხი და კოდების არამთელრიცხვიან თვისებებად გადასაქცევად საკმარისი იქნება თითოეული კოდის წინ „+“ ნიშნის დამატება.

მაგალითი:

```
let codes = {
  "+49": "Germany",
  "+41": "Switzerland",
  "+44": "United Kingdom",
  // ..,
  "+1": "USA"
};

for (let code in codes) {
  alert( +code ); // 49, 41, 44, 1
}
```

კოდი უკვე ისე იმუშავებს, როგორც ჩავიფიქრეთ.

ობიექტისა და ბმულის კოპირება

ერთ-ერთი ფუნდამენტური განსხვავება ობიექტებსა და პრიმიტიულ მონაცემებს შორის არის ის, რომ ისინი ინახება და მათი კოპირება ხდება ბმულით.

პრიმიტიული მონაცემები: სტრიქონები, რიცხვები, ლოგიკური მნიშვნელობები - ენიჭება და მათი კოპირება ხდება მნიშვნელობის მიხედვით.

მაგალითად:

```
let message = "Hello!";
let phrase = message;
```

შედეგად გვაქვს ორი დამოუკიდებელი ცვლადი, თითოეულ მათგანში ინახება სტრიქონული მნიშვნელობა „Hello!“.

ობიექტებში ეს ყველაფერი სხვაგვარად ხდება. ცვლადი თავისთავში ინახავს არა თვით ობიექტს, არამედ მის მისამართს მეხსიერებაში, სხვა სიტყვებით რომ ვთქვათ, მასზე მიმართვის ბმულს. მაგალითი:

```
let user = {
  name: "გიორგი"
};
```

თვითონ ობიექტი ინახება სადღაც მეხსიერებაში. ხოლო user ცვლადში ინახება, მეხსიერების ამ ნაწილზე წვდომის ბმული.

როდესაც ობიექტის ცვლადის კოპირება ხდება, კოპირდება ბმული, მაგრამ თავად ობიექტი არ არის დუბლირებული.

თუ ჩვენ წარმოვიდგენთ ობიექტს უჯრის სახით, მაშინ ცვლადი არის მისი გასაღები. ცვლადის კოპირებით მიიღება გასაღების დუბლიკატი, მაგრამ არა თავად უჯრა.

მაგალითად:

```
let user = { name: "გიორგი" };

let admin = user; // ხდება ბმულის კოპირება
```

ახლა გვაქვს ორი ცვლადი, თითოეული შეიცავს ერთი და იმავე ობიექტზე მიმართვის ბმულს.

ჩვენ შეგვიძლია გამოვიყენოთ ნებისმიერი ცვლადი „უჯრის“ გასახსნელად და მისი შინაარსის შესაცვლელად:

```
let user = { name: 'გიორგი' };

let admin = user;

admin.name = 'მიხეილი'; // შეიცვალა ბმულით „admin“ ცვლადის საშუალებით
```

ზემოთ მოყვანილი მაგალითი აჩვენებს, რომ არსებობს მხოლოდ ერთი ობიექტი. თითქოს ჩვენ გვქონდა ერთი უჯრა ორი გასაღებით და ვიყენებდით ერთს (admin) რომ შევიტანოთ და შევცვალოთ რაღაც, შემდეგ კი უჯრის სხვა გასაღებით (user) გახსნის შემთხვევაში დავინახავდით ამ ცვლილებებს.

ბმულით შედარება

შედარების == და მკაცრი შედარების ოპერატორები === ობიექტებისთვის ერთნაირად მუშაობენ.

ორი ობიექტი ტოლია მხოლოდ იმ შემთხვევაში, თუ ისინი ერთი და იგივე ობიექტია.

ქვემოთ მოცემულ მაგალითში ორი ცვლადი ერთსა და იმავე ობიექტს მიმართავს, ამიტომ ისინი ერთმანეთის ტოლია:

```
let a = {};
let b = a; // ბმულით კოპირება

alert( a == b ); // true, ვინაიდან ორივე ცვლადი მიმართავს ერთი და იმავე ობიექტს
alert( a === b ); // true
```

შემდეგ მაგალითში ორი სხვადასხვა ობიექტი არ არის ტოლი, თუმცა ორივე ობიექტი ცარიელია:

```
let a = {};
let b = {}; // ორი დამოუკიდებელი ობიექტი

alert( a == b ); // false
```

obj1 > obj2 ტიპების შედარებისთვის, ან პრიმიტიული ცვლადების obj == 5 ობიექტთან შედარებისთვის, ობიექტები პრიმიტიულ ცვლადებად გარდაიქმნება. მსგავსი შედარება იშვიათად არის საჭირო და, როგორც წესი, პროგრამისტის შეცდომის შედეგია.

JSON ფორმატი, toJSON მეთოდი

ვთქვათ, გვაქვს რთული ობიექტი და გვსურს მისი გარდაქმნა სტრიქონად, რათა გადავაგზავნოთ ქსელში ან უბრალოდ გამოვიტანოთ სახელისთვის.

ბუნებრივია, ასეთი სტრიქონი უნდა შეიცავდეს ყველა მნიშვნელოვან თვისებას.

შეგვიძლია ეს გარდაქმნა შემდეგნაირად განვახორციელოთ:

```
let user = {
  name: "გიორგი ბერიძე",
  age: 22,

  toString() {
    return `{name: "${this.name}", age: ${this.age}}`;
  }
};

alert(user); // {name: "გიორგი ბერიძე", age: 22}
```

This page says

{name: "გიორგი ბერიძე", age: 22}

OK

მაგრამ დამუშავების პროცესში, ახალი თვისებები ემატება, ძველი თვისებები სახელშეცვლილია და წაშლილია. ასეთი

toString-ის განახლება ყოველ ჯერზე შეიძლება პრობლემა გახდეს. ჩვენ შეგვიძლია ვცადოთ მასში არსებული თვისებების გამეორება, მაგრამ თუ ობიექტი რთულია და მის თვისებებში შეიძლება იყოს ჩადგმული ობიექტები? ჩვენ მათი გარდაქმნაც მოგვიწევს.

საბედნიეროდ, ამის მოსაგვარებლად საჭირო არ არის კოდის დაწერა. ამ პრობლემას მარტივი გამოსავალი აქვს.

JSON.stringify

JSON (JavaScript Object Notation) - არის საერთო ფორმატი მნიშვნელობებისა და ობიექტების წარმოსადგენად. ის თავდაპირველად JavaScript-სთვის შეიქმნა, მაგრამ ბევრ სხვა ენას ასევე აქვს ბიბლიოთეკები, რომლებსაც შეუძლიათ მასთან მუშაობა. ამრიგად, JSON მარტივი გამოსაყენებელია მონაცემთა გაცვლისთვის, როდესაც მომხმარებელი იყენებს JavaScript-ს, ხოლო სერვერი Ruby/PHP/Java ან სხვა ენაზეა დაწერილი.

JavaScript გთავაზობთ მეთოდებს:

- JSON.stringify ობიექტების JSON სტრიქონად გარდასაქმნელად;
- JSON.parse JSON სტრიქონის ისევ ობიექტად გარდასაქმნელად.

მაგალითად, აქ ჩვენ გარდავქმნით სტუდენტის მონაცემებს JSON.stringify-ის საშუალებით:

```
let student = {
  name: 'გიორგი ბერიძე',
  age: 22,
  isAdmin: 'არა',
  courses: ['html', 'css', 'js'],
```

```
  wife: 'უცოლო'
};

let json = JSON.stringify(student);

alert(typeof json); // მივიღეთ String

alert(json);
/* ობიექტს გამოიტანს JSON ფორმატში:
  "name": "გიორგი ბერიძე", "age": 22, "isAdmin": "არა",
  "courses": ["html", "css", "js"], "wife": "უცოლო"
*/
```

This page says

{"name":"გიორგი ბერიძე","age":22,"isAdmin":"არა","courses":
["html","css","js"],"wife":"უცოლო"}

OK

JSON.stringify(student) მეთოდი იღებს ობიექტს და გარდაქმნის მას სტრიქონად. მიღებულ json სტრიქონს ეწოდება JSON ფორმატირებული ან სერიული ობიექტი. ჩვენ შეგვიძლია გავაგზავნოთ იგი ქსელის საშუალებით ან მოვათავსოთ ჩვეულებრივ მონაცემთა საცავში.

გასათვალისწინებელია, რომ JSON ობიექტს აქვს რამდენიმე მნიშვნელოვანი განსხვავება ჩვეულებრივი ობიექტისაგან:

სტრიქონებში გამოიყენება ორმაგი ბრჭყალები. JSON-ში არც ერთმაგი ან საწინააღმდეგოდ მიმართული ბრჭყალი არ არის. ასე რომ, 'გიორგი ბერიძე' ხდება "გიორგი ბერიძე".

ობიექტების თვისებების სახელები ასევე ორმაგ ბრჭყალებშია მოთავსებული. ეს აუცილებელია. ასე რომ, age: 22 ხდება "age": 22.

JSON.stringify შეიძლება პრიმიტივებზეც იქნას გამოყენებული.

JSON მხარს უჭერს მონაცემთა შემდეგ ტიპებს:

- ობიექტები { ... };
- მასივები [...];
- პრიმიტივები:
  - სტრიქონი;
  - რიცხვები;
  - ლოგიკური მნიშვნელობები true/false;
  - null.

მაგალითად:

```
// რიცხვი JSON-ში რიცხვად დარჩება
alert( JSON.stringify(1) ) // 1

// სტრიქონი JSON-ში ისევ სტრიქონად დარჩება, ოღონდ
    ორმაგ ბრჭყალებში მოთავსებული
alert( JSON.stringify('test') ) // "test"

alert( JSON.stringify(true) ); // true

alert( JSON.stringify([1, 2, 3]) ); // [1,2,3]
```

JSON მონაცემებისათვის არის ენიდან დამოუკიდებელი სპეციფიკაცია (მეთოდი), ამიტომ JSON.stringify ტოვებს JavaScript ობიექტების ზოგიერთ სპეციფიკურ თვისებას.

კერძოდ:

- ფუნქციების თვისებებს (მეთოდებს);
- სიმბოლურ თვისებებს;
- თვისებებს, რომლებიც შეიცავს undefined.

```
let user = {
  sayHi() { // მოხდება მისი გამოტოვება
    alert("Hello");
  },
  [Symbol("id")]: 123, // ასევე მოხდება მისი გამოტოვება
  something: undefined // ესეც გამოიტოვება
};

alert( JSON.stringify(user) ); // {} (ცარიელი ობიექტი)
```

როგორც წესი, ეს ნორმალურია. მთავარი ის არის, რომ ჩადგმული ობიექტები მხარდაჭერილია და ავტომატურად გარდაიქმნება.

მაგალითად:

```
let meetup = {
  title: "Conference",
  room: {
    number: 225,
    participants: ["გიორგი ბერიძე", "მარიამ მაისურაძე"]
  }
};
```

```
alert( JSON.stringify(meetup) );
/* вся структура преобразована в строку:
  "title":"Conference",
  "room":{"number":225,"participants":["გიორგი ბერიძე","მარიამ
     მაისურაძე"]},
*/
```

This page says

{"title":"Conference","room":{"number":225,"participants":["გიორგი ბერიძე","მარიამ მაისურაძე"]}}

OK

მნიშვნელოვანი შეზღუდვა: არ უნდა იყოს ციკლური მიმართვები.

აქ გარდაქმნა ვერ ხერხდება ციკლური მიმართვების გამო: room.occupiedBy მიმართავს meetup-ს და meetup.place მიმართავს room-ს.

JSON.stringify-ის სრული სინტაქსია:

```
let json = JSON.stringify(value[, replacer, space]);
```

სადაც

Value - არის მნიშვნელობა კოდირებისათვის;

Replacer - თვისებების მასივი კოდირებისთვის, ან შესაბამისობის ფუნქცია function(key, value);

Space - დამატებითი სივრცე, რომელიც გამოიყენება დაფორმატებისთვის.

უმეტეს შემთხვევაში, JSON.stringify-ის მიერ მხოლოდ პირველი არგუმენტი გამოიყენება. მაგრამ თუ ჩვენ გვჭირდება ჩანაცვლების პროცესის მოწყობა, როგორიცაა ციკლური მიმართვების გაფილტვრა, მაშინ JSON.stringify-ის მეორე არგუმენტი შეიძლება იქნას გამოყენებული. თუ მას გადავცემთ თვისებების მასივს, მაშინ მხოლოდ ეს თვისებები იქნება კოდირებული. მაგალითად:

```
let room = {
  number: 225
};

let meetup = {
  title: "Conference",
    participants: ["გიორგი ბერიძე", "მარიამ მაისურაძე"],
  place: room // meetup მიმართავს room-ს
};

room.occupiedBy = meetup; // room მიმართავს meetup-ს

alert( JSON.stringify(meetup, ['title', 'participants']) );
```

```
This page says
{"title":"Conference","participants":["გიორგი ბერიძე","მარიამ მაისურაძე"]}
OK
```

თვისებების სია ობიექტის მთელ სტრუქტურაზე ვრცელდება. ასე რომ, participants-ის შიგნით ცარიელი ობიექტებია, რადგან name არ არის სიაში.

სიაში შევიყვანოთ ყველა თვისება, გარდა room.occupiedBy-ისა, რომლის გამოც ციკლური მითითებები წარმოიშვება:

```
let room = {
  number: 225
};

let meetup = {
  title: "Conference",
    participants: [{name:"გიორგი ბერიძე"},{name:"მარიამ
      მაისურაძე"}],
  place: room // meetup მიმართავს room-ს
};

room.occupiedBy = meetup; // room მიმართავს meetup-ს

alert( JSON.stringify(meetup, ['title', 'participants', 'place', 'name',
      'number']) );
/*
  "title":"Conference",
  "participants":[{"name":"გიორგი ბერიძე"},{"name":"მარიამ
      მაისურაძე"}],
  "place":{"number":225}
*/
```

This page says

{"title":"Conference","participants":[{"name":"გიორგი ბერიძე"},
{"name":"მარიამ მაისურაძე"}],"place":{"number":225}}

OK

ახლა ყველაფერი occupiedBy-ის გარდა სერიულია. მაგრამ თვისებების სია საკმაოდ გრძელია.

საბედნიეროდ, ჩვენ შეგვიძლია replacer-ის ნაცვლად გამოვიყენოთ ფუნქცია და არა მასივი.

ფუნქციის გამოძახება მოხდება თითოეული (key, value) წყვილისთვის და მან უნდა დააბრუნოს შეცვლილი მნიშვნე-ლობა, რომელიც გამოყენებული იქნება საწყისი მნიშვნელობის ნაცვლად, ან undefined, რათა მოხდეს მნიშვნელობის გამოტოვება.

ჩვენს შემთხვევაში, შეგვიძლია ყველაფრისთვის დავაბრუნოთ value „როგორც არის“, გარდა occupiedBy-სა. occupiedBy-ის იგნორირებისთვის, ქვემოთ მოცემული კოდი აბრუნებს undefined-ს:

```
let room = {
  number: 225
};

let meetup = {
  title: "Conference",
```

```
  participants: [{name:"გიორგი ბერიძე"},{name:"მარიამ
    მაისურაძე"}],
  place: room // meetup მიმართავს room-ს
};

room.occupiedBy = meetup; // room მიმართავს meetup-ს

alert( JSON.stringify(meetup, function replacer(key, value) {
  alert(`${key}: ${value}`);
  return (key == 'occupiedBy') ? undefined : value;
}));

/* (key, value) წყვილი: მნიშვნელობები, რომლებიც მოდის
    replacer-ში:
:           [object Object]
title:       Conference
participants: [object Object],[object Object]
0:          [object Object]
name:        გიორგი ბერიძე
1:          [object Object]
name:        მარიამ მაისურაძე
place:       [object Object]
number:      225
occupiedBy: [object Object]
*/
```

გამოიტანს (key, value) წყვილებს და ბოლოს:

This page says

{"title":"Conference","participants":[{"name":"გიორგი ბერიძე"},
{"name":"მარიამ მაისურაძე"}],"place":{"number":225}}

OK

ყურადღება მიაქციეთ იმას, რომ replacer ფუნქცია იღებს key/value-ს ყველა წყვილს, მათ შორის ჩადგმულ ობიექტებს და მასივის ელემენტებს და იგი გამოიყენება რეკურსიულად. replacer შიგნით this მნიშვნელობა არის ობიექტი, რომელიც მიმდინარე თვისებას შეიცავს.

პირველი გამოწვევა განსაკუთრებულია. მას გადაეცემა სპეციალური „ობიექტი-ჩარჩო“: {"": meetup}. სხვა სიტყვებით რომ ვთქვათ, პირველ (key, value) წყვილს აქვს ცარიელი გასაღები, ხოლო მნიშვნელობა არის ზოგადად მიზანობრივი ობიექტი. ამიტომ, ზემოთ მოყვანილი მაგალითიდან პირველი სტრიქონი იქნება ":[object Object]".

იდეა მდგომარეობს იმაში, რომ replacer-ს მივცეთ რაც შეიძლება მეტი შესაძლებლობა - მას აქვს შესაძლებლობა გააანალიზოს და საჭიროების შემთხვევაში ჩაანაცვლოს/გამოტოვოს თუნდაც მთელი ობიექტი,.

JSON.stringify(value, replacer, space)-ში მესამე არგუმენტი არის ჰარების რაოდენობა, რომლის გამოყენება მოსახერხებელია დაფორმატირებისთვის.

ადრე, არც ერთ JSON-ფორმატირებულ ობიექტს არ ჰქონდა შეწევა ან დამატებითი ჰარები. ეს კარგია, თუ გვინდა ობიექტის

გაგზავნა ქსელში. space არგუმენტი გამოიყენება მხოლოდ გამოტანილი ინფორმაციის მოხერხებულად წაკითხვის მიზნით.

ქვემოთ space = 2 JavaScript-ს უჩვენებს, რომ ჩადგმული ობიექტები გამოტანილ იქნას რამდენიმე სტრიქონად, ობიექტის შიგნით 2 ჰარით შეწეული:

```
let user = {
  name: 'გიორგი ზერიძე',
  age: 22,
  roles: {
    isAdmin: 'არა',
    isEditor: 'დიახ'
  }
};

alert(JSON.stringify(user, null, 2));
```

შეწეულია 2 ჰარით:

This page says

```
{
  "name": "გიორგი ბერიძე",
  "age": 22,
  "roles": {
    "isAdmin": "არა",
    "isEditor": "დიახ"
  }
}
```

OK

JSON.stringify(user, null, 8)-თვის შედეგი იქნება უფრო მეტი ინტერვალით:

This page says

```
{
        "name": "გიორგი ბერიძე",
        "age": 22,
        "roles": {
                "isAdmin": "არა",
                "isEditor": "დიახ"
        }
}
```

OK

space პარამეტრი გამოიყენება ინფორმაციის მოხერხებულად და ლამაზად გამოსატანად.

JSON.parse

იმისათვის, რომ მოვახდინოთ JSON-სტრიქონის დეკოდირება, ამისათვის დაგვჭირდება სხვა მეთოდი, რომლის სახელია JSON.parse. მისი სინტაქსია:

```
let value = JSON.parse(str, [reviver]);
```

სადაც str – JSON-ის ობიექტად გადასაყვანად;

reviver - არასავალდებულო ფუნქცია, რომლის გამომახებაც მოხდება თითოეული (key, value) წყვილისთვის და შეუძლია გარდაქმნას მნიშვნელობა.

მაგალითად:

```
// სტრიქონული მასივი
let numbers = "[0, 1, 2, 3]";

numbers = JSON.parse(numbers);

alert( numbers[1] ); // 1
```

ან ჩადგმული ობიექტისათვის:

```
let user = '{ "name": "გიორგი ბერიძე", "age": 22, "isAdmin":"არა",
    "friends": [0,1,2,3] }';

user = JSON.parse(user);

alert( user.friends[1] ); // 1
```

JSON შეიძლება იყოს რთული, როგორსაც აუცილებლობა წარმოადგენს, ობიექტები და მასივები შეიძლება სხვა ობიექტებს

და მასივებს შეიცავდეს. მაგრამ ისინი იგივე JSON ფორმატში უნდა იყოს.

ქვემოთ მოცემულია JSON-ში ტიპიური შეცდომები (ზოგჯერ საჭიროა ის დაიწეროს პროგრამის გამართვისთვის):

```
let json = '{
  name: "გიორგი",      // შეცდომაა: თვისების სახელი
     ბრჭყალების გარეშეა
  "surname": 'ზერიძე', // შეცდომაა: მნიშვნელობა ერთმაგ
                          ბრჭყალებშია მოთავსებული (უნდა
                          იყოს ორმაგ ბრჭყალებში)
  'isAdmin': false        // შეცდომაა: საკვანძო სიტყვა  ერთმაგ
                            ბრჭყალებშია მოთავსებული (უნდა
                            იყოს ორმაგ ბრჭყალებში)
  "birthday": new Date(2000, 2, 3), // შეცდომაა: კონსტრუქტორ
                              "new"-ს გამოყენება დაშვებული არ
                              არის, მხოლოდ მნიშვნელობა.
  "friends": [0,1,2,3]       // აქ ყველაფერი სწორად არის
}';
```

JSON მხარს არ უჭერს კომენტარებს. JSON-ზე კომენტარის დამატება მის გაბათილებას იწვევს.

წარმოიდგინეთ, რომ ჩვენ მივიღეთ ობიექტი meetup სერვერიდან, მონაცემთა სტრიქონის სახით.

```
// title: (meetup title), date: (meetup date)
let str = '{"title":"Conference","date":"2017-11-30T12:00:00.000Z"}';
```

ახლა საჭიროა მისი დესერიალიზაცია, ე.ი. დავაბრუნოთ ის JavaScript-ის ობიექტად. გავაკეთოთ ეს JSON.parse-ის გამოძახებით:

```
let str = '{"title":"Conference","date":"2017-11-30T12:00:00.000Z"}';

let meetup = JSON.parse(str);

alert( meetup.date.getDate() ); // Error!
```

meetup.date-ის მნიშვნელობა არის სტრიქონი და არა Date ობიექტი. როგორ შეეძლო JSON.parse-ს სცოდნოდა, რომ უნდა გადაექცია ეს სტრიქონი Date-ად?

გადავიტანოთ JSON.parse აღდგენის ფუნქცია მეორე არგუმენტად, რომელიც აბრუნებს ყველა მნიშვნელობას "როგორც არის", მაგრამ date ხდება Date:

```
let str = '{"title":"Conference","date":"2017-11-30T12:00:00.000Z"}';

let meetup = JSON.parse(str, function(key, value) {
  if (key == 'date') return new Date(value);
  return value;
});

alert( meetup.date.getDate() ); // 30
```

სხვათა შორის, ეს ასევე მუშაობს ჩადგმული ობიექტებისთვისაც:

```
let schedule = ‘{
  "meetups": [
```

```
    {"title":"Conference","date":"2017-11-30T12:00:00.000Z"},
    {"title":"Birthday","date":"2017-04-18T12:00:00.000Z"}
  ]
}';

schedule = JSON.parse(schedule, function(key, value) {
  if (key == 'date') return new Date(value);
  return value;
});

alert( schedule.meetups[1].date.getDate() ); // 18
```

გლობალური ობიექტი

გლობალური ობიექტი წარმოადგენს ცვლადებს და ფუნქციებს, რომლებიც ხელმისაწვდომია პროგრამის ნებისმიერ ადგილას. ნაგულისხმევად, ეს არის ის, რაც ჩაშენებულია ენაში ან შესრულების გარემოში.

ბრაუზერში მას window ჰქვია, Node.js-ში - გლობალური, სხვა გარემოში შეიძლება სხვანაირად ეწოდოს.

ცოტა ხნის წინ, globalThis დაემატა ენას, როგორც გლობალური ობიექტისათვის სტანდარტული სახელი, რომელიც მხარდაჭერილი უნდა იყოს ნებისმიერ გარემოში. ის მხარდაჭერილია ყველა ძირთად ბრაუზერში.

შემდეგში, ჩვენ window-ს გამოვიყენებთ, როდესაც ჩვენი გარემო იქნება ბრაუზერი. თუ სკრიპტის გაშვება შესაძლებელია სხვა გარემოში, მაშინ globalThis-ის გამოყენება იქნება უკეთესი.

გლობალური ობიექტის ყველა თვისებაზე წვდომა პირდაპირ არის შესაძლებელი:

```
alert("Hello!");
// იგივეა რაც
window.alert("Hello!");
```

ბრაუზერში var-ით გამოცხადებული გლობალური ფუნქციები და ცვლადები (არა let/const!) გლობალური ობიექტის თვისებების მატარებელი ხდება:

```
var gVar = 5;


alert(window.gVar); // 5
```

იგივე ეხება ფუნქციებს, რომლებიც Function Declaration სინტაქსის გამოყენებითაა გამოცხადებული (გამოსახულებები function საკვანძო სიტყვით ძირითადი კოდის ნაკადში და არა Function Expression-ით).

ეს ქცევა თავსებადობისთვისაა მხარდაჭერილი. ეს არ ხდება თანამედროვე პროექტებში, რომლებიც JavaScript-მოდულებს იყენებენ.

ჩვენ რომ გამოგვეცხადებინა ცვლადი let-ით, მაშინ ეს არ მოხდებოდა:

```
let gLet = 5;

alert(window.gLet); // undefined
```

თუ თვისება იმდენად მნიშვნელოვანია, რომ გსურთ ის ხელმისაწვდომი გახადოთ მთელი პროგრამისთვის, ჩაწერეთ იგი პირდაპირ გლობალურ ობიექტში:

```
window.User = {
  name: "John"
};

// კოდის ნებისმიერ ადგილზე
alert(User.name); // John

// ან თუ ჩვენ გვაქვს ლოკალური ცვლადი სახელითმას
     მივიღებთ პირდაპირ "User",
// მას მივიღებთ პირდაპირ window-ით (უსაფრთხოა!)
alert(window.User.name); // John
```

ზოგადად არ არის რეკომენდებული გლობალური ცვლადების გამოყენება. ისინი რაც შეიძლება იშვიათად უნდა იქნას გამოყენებული. კოდის დიზაინი, რომელის დროსაც ფუნქცია იღებს შემავალ პარამეტრებს და გასცემს გარკვეულ შედეგს, უფრო საიმედო და მოსახერხებელია ტესტირებისთვის, ვიდრე როდესაც გამოიყენებს გარე და მით უმეტეს გლობალურ ცვლადებს.

გლობალური ობიექტი შეიძლება გამოყენებულ იქნას თანამედროვე ენის მახასიათებლების მხარდაჭერის შესამოწმებლად.

მაგალითად, ჩაშენებული Promise ობიექტის არსებობის შესამოწმებლად (ასეთი მხარდაჭერა არ არის ძალიან ძველ ბრაუზერებში):

```
if (!window.Promise) {
  alert("თქვენი ბრაუზერი ძალიან ძველია!");
}
```

თუ არა და (ვთქვათ, გამოყენებულია ძველი ბრაუზერი), შეგვიძლია შევქმნათ პოლიფილი[4]: დავამატოთ ფუნქციები, რომლებსაც არსებული გარემო მხარს არ უჭერს, მაგრამ თანამედროვე სტანდარტში არსებობს.

```
if (!window.Promise) {
  window.Promise = ... // ენის თანამედროვე შესაძლებლობების
     საკუთარი რეალიზაცია
```

[4] პოლიფილი არის კოდი, რომელიც ახორციელებს რაიმე სახის ფუნქციონირებას, რომელიც არ არის მხარდაჭერილი ზოგიერთი ბრაუზერის მიერ. საკუთარი პოლიფილის რეალიზება ერთნაირი შედეგების მიღებას უზრუნველყოფს სხვადასხვა ბრაუზერში.

```
}
```

ფუნქციის ობიექტი, NFE

როგორც უკვე ვიცით, JavaScript-ში ფუნქცია არის მნიშვნელობა.

JavaScript-ში ყველა მნიშვნელობას აქვს ტიპი. რა ტიპისაა თვით ფუნქცია?

JavaScript-ში ფუნქციები არის ობიექტები.

ფუნქცია შეიძლება წარმოადგინოთ, როგორც „ობიექტი, რომელსაც რაღაცის გაკეთება შეუძლია". ფუნქციების არა მხოლოდ გამოძახება შეიძლება, არამედ ჩვეულებრივი ობიექტების მსგავსად მათი გამოყენებაც: თვისებების დამატება/წაშლა, მათ ბმულით გადაცემა და ა.შ.

თვისება name

ობიექტი ფუნქცია რამდენიმე სასარგებლო თვისებას შეიცავს.

მაგალითად, ფუნქციის სახელი ხელმისაწვდომია, როგორც „name" თვისება:

```
function Func() {
  alert("Hello!");
}


alert(Func.name);
```

This page says

Func

OK

Name ანიჭებს სწორ სახელს მაშინაც კი, თუ ფუნქცია სახელის გარეშეა შექმნილი და მას დაუყოვნებლივ ენიჭება სახელი ასე:

```
let Func = function () {
  alert("Hello!");
}

alert(Func.name);   //  Func (არის სახელი)
```

ეს მუშაობს მაშინაც კი, თუ ნაგულისხმევი მნიშვნელობა ენიჭება:

```
function f(Func = function() {}) {
  alert(Func.name);   //  Func
}

f();
```

სპეციფიკაში, ამას ეწოდება „კონტექსტური სახელი“: თუ ფუნქციას არ აქვს სახელი (name), მაშინ JavaScript მის განსაზღვრას კონტექსტიდან ცდილობს.

ობიექტის მეთოდებს ასევე აქვთ სახელები:

```
let user = {
```

```
  sayHi() {
   //  ...
  },

  sayBye: function() {
   // ...
  }
}

alert(user.sayHi.name); // sayHi
alert(user.sayBye.name); // sayBye
```

თვისება length

კიდევ ერთი ჩაშენებული თვისება „length" შეიცავს მის ფუნქციის გამოცხადებაში ფუნქციის პარამეტრების რაოდენობას. მაგალითად:

```
function f1(a) {}
function f2(a, b) {}
function many(a, b, ...more) {}

alert(f1.length); // 1
alert(f2.length); // 2
alert(many.length); // 2
```

როგორც ვხედავთ, სამი წერტილი, რომელიც აღნიშნავს „ნარჩენ პარამეტრებს", აქ, როგორც ჩანს, „არ ითვლება".

მომხმარებლის თვისებები

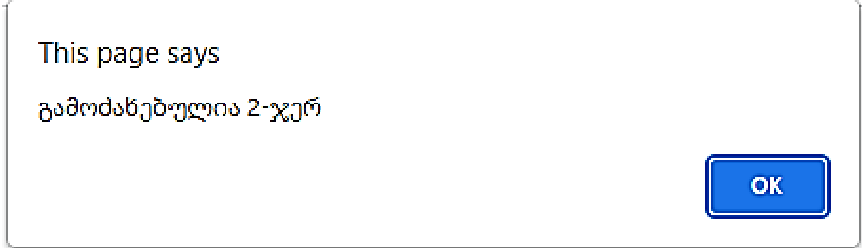
ჩვენ შეგვიძლია ასევე დავამატოთ საკუთარი თვისებები.

დავუმატოთ თვისება counter იმისათვის, რომ თვალყური ვადევნოთ საერთო გამოძახებების რაოდენობას:

```
function Func() {
  alert("Hello!");
// დავთვალოთ რამდენჯერ გამოვიძახეთ ეს ფუნქცია
Func.count++
}
Func.count = 0;  // საწყისი მნიშვნელობა

Func();   //  Hello!
Func();   //  Hello!

alert( ‘გამოძახებულია ${Func.count}-ჯერ’ );
```



ფუნქციის თვისება მინიჭებული როგორც Func.count = 0 არ აცხადებს count ლოკალურ ცვლადს მის შიგნით. სხვა სიტყვებით რომ ვთქვათ, count თვისება და let count ცვლადი არის ორი დამოუკიდებელი ელემენტი.

ჩვენ შეგვიძლია გამოვიყენოთ ფუნქცია, როგორც ობიექტი, შევინახოთ მასში თვისებები, მაგრამ ისინი არანაირად არ იმოქმედებენ მის შესრულებაზე. ცვლადები არ არიან ფუნქციის თვისებები და პირიქით. ეს ორი პარალელური სამყაროა.

მონაცემთა ტიპი Symbol

სპეციფიკაციის მიხედვით, ობიექტის თვისებების გასაღებად შეიძლება გამოყენებულ იქნას მხოლოდ სტრიქონები ან სიმბოლოები. არც რიცხვები და არც ლოგიკური მნიშვნელობები არ გამოიყენება, მხოლოდ ამ ორი ტიპის მონაცემებია დაშვებული.

სიმბოლოები

სიმბოლო წარმოადგენს უნიკალურ იდენტიფიკატორს. ახალი სიმბოლოები იქმნება Symbol() ფუნქციის დახმარებით:

```
// ვქმნით ახალ სიმბოლოს - id
let id = Symbol();
```

სიმბოლოს შექმნისას შეგიძლიათ მისცეთ მას აღწერა (მას ასევე სახელს უწოდებენ), რომელიც ძირითადად კოდის გამართვისთვის გამოიყენება:

```
// ვქმნით სიმბოლოს id აღწერით (სახელით) „id“
let id = Symbol("id");
```

სიმბოლოები გარანტირებულად უნიკალურებია. მაშინაც კი, თუ ჩვენ შევქმნით ბევრ სიმბოლოს ერთი და იგივე აღწერით, ისინი მაინც განსხვავებული სიმბოლოები იქნებიან. აღწერა მხოლოდ ჭდეა, რომელიც არაფერზე არ მოქმედებს.

მაგალითად, აქ არის ორი სიმბოლო ერთი და იგივე აღწერით - მაგრამ ისინი ტოლები არ არიან:

```
let id1 = Symbol("id");
let id2 = Symbol("id");

alert(id1 == id2); // false
```

სიმბოლოები ავტომატურად არ გარდაიქმნება სტრიქონებად.

JavaScript-ში მონაცემთა ტიპების უმეტესობა შეიძლება გარდაიქმნას სტრიქონად. მაგალითად, alert ფუნქცია იღებს თითქმის ნებისმიერ მნიშვნელობას, ავტომატურად გარდაქმნის მას სტრიქონად და შემდეგ ბეჭდავს ამ მნიშვნელობას შეცდომის შეტყობინების გარეშე. სიმბოლოები განსაკუთრებულია და ავტომატურად არ გარდაიქმნება.

მაგალითად, ქვემოთ მოცემული alert ფუნქცია გამოიწვევს შეცდომას:

```
let id = Symbol("id");
alert(id); // TypeError: Cannot convert a Symbol value to a string
```

ეს არის ენის „დაცვა“ არეულობისგან, რადგან სტრიქონები და სიმბოლოები ფუნდამენტურად განსხვავებული მონაცემთა ტიპებია და არ უნდა იყოს უკონტროლოდ გადაყვანილი ერთმანეთში.

თუ ჩვენ ნამდვილად გვსურს სიმბოლოს გამოტანა alert ფუნქციის გამოყენებით, მაშინ ჩვენ გვჭირდება მისი ცალსახად გარდაქმნა .toString() მეთოდის გამოყენებით, ასე:

```
let id = Symbol("id");
```

```
alert(id.toString()); // Symbol(id), ამ შემთხვევაში იმუშავებს
```

ან შეგვიძლია მივმართოთ symbol.description თვისებას მხოლოდ აღწერილობის საჩვენებლად:

```
let id = Symbol("id");
alert(id.description); // id
```

„დაფარული“ თვისებები

სიმბოლოები საშუალებას გაძლევთ შექმნათ ობიექტების "დაფარული" თვისებები, რომლებზეც შემთხვევითი წვდომა და გადაწერა შეუძლებელია პროგრამის სხვა ნაწილების მიერ.

მაგალითად, ჩვენ ვმუშაობთ user ობიექტებთან, რომლებიც ეკუთვნის მესამე მხარის კოდს. ჩვენ გვსურს დავამატოთ მათ იდენტიფიკატორები.

ამისთვის ვიყენებთ სიმბოლურ გასაღებს:

```
let user = {
  name: "გიორგი"
};
let id = Symbol("id");
user[id] = 1;
alert( user[id] ); // ჩვენ შეგვიძლია მონაცემებზე წვდომა გასაღები-სიმბოლოს საშუალებით
```

რატომ ჯობია გამოვიყენოთ Symbol("id") ვიდრე სტრიქონი "id"?

ვინაიდან user ობიექტი ეკუთვნის მესამე მხარის კოდს და ეს კოდიც მუშაობს მასთან, ჩვენ არ უნდა დავამატოთ მას რაიმე ველი. ეს არ არის უსაფრთხო. მაგრამ სიმბოლოს ძნელია

უნებურად მიმართო, ნაკლებად სავარაუდოა, რომ მესამე მხარის კოდმა დაინახოს იგი და დიდი ალბათობით, ველის დამატება ობიექტზე არავითარ პრობლემას არ გამოიწვევს.

ასევე, დავუშვათ, რომ სხვა სკრიპტს სურს ჩაწეროს საკუთარი იდენტიფიკატორი user ობიექტში გარკვეული მიზნით. ეს სკრიპტი შეიძლება იყოს JavaScript-ის ერთგვარი ბიბლიოთეკა, რომელიც სრულიად არ არის დაკავშირებული ჩვენს სკრიპტთან.

ამისათვის, მესამე მხარის კოდს შეუძლია შექმნას საკუთარი სიმბოლო Symbol("id").

მათსა და ჩვენს იდენტიფიკატორს შორის კონფლიქტი არ იქნება, რადგან სიმბოლოები ყოველთვის უნიკალურია, თუნდაც მათი სახელები ერთი და იგივე იყოს.

მაგრამ თუ ჩვენ გამოვიყენებდით სტრიქონს "id" სიმბოლოს ნაცვლად, მაშინ იქნებოდა კონფლიქტი:

```
let user = { name: "გიორგი" };
// ჩვენ სკრიპტში გამოვაცხადებთ „id“ თვისებას
user.id = "ჩვენი იდენტიფიკატორი";

// ...სხვა სკრიპტს უნდა სხვა იდენტიფიკატორი ქონდეს...

user.id = "სხვისი იდენტიფიკატორი";
// თვისება გადაწერილია მესამე მხარის ბიბლიოთეკის მიერ!
```

სიმბოლოები ლიტერარულ ობიექტებში

თუ გვსურს გამოვიყენოთ სიმბოლო ლიტერარული ობიექტის გამოცხადებაში {...}, ის უნდა იყოს ჩასმული კვადრატულ ფრჩხილებში.

მაგალითად,

```
let id = Symbol("id");

let user = {
  name: "გიორგი",
  [id]: 123 // უბრალოდ "id: 123" არ იმუშავებს
};
```

ეს იმიტომ ხდება, რომ გასაღებად უნდა გამოვიყენოთ id ცვლადის მნიშვნელობა და არა „id“ სტრიქონი.

სიმბოლოები და for…in ციკლი

თვისებები, რომელთა გასაღებით-სიმბოლო for...in ციკლის მიერ არის იგნორირებული.

მაგალითი:

```
let id = Symbol("id");
let user = {
  name: "გიორგი",
  age: 30,
  [id]: 123
};

for (let key in user) alert(key); // name, age (თვისება გასაღებით-სიმბოლო არ არის ჩამონათვალში)

// თუმცა სიმბოლოთი პირდაპირი წვდომა მუშაობს
alert( "პირდაპიი წვდომა: " + user[id] );
```

ეს არის „სიმბოლური თვისებების დამალვის“ ზოგადი პრინციპის ნაწილი. თუ სხვა ბიბლიოთეკა ან სკრიპტი მუშაობს ჩვენს ობიექტთან, მაშინ, გადარჩევის დროს, ისინი უნებურად არ მიიღებენ ჩვენს სიმბოლურ თვისებას. Object.keys(user) ასევე უგულებელყოფს სიმბოლოებს.

მაგრამ Object.assign, for...in ციკლისგან განსხვავებით, აკოპირებს როგორც სტრიქონებს, ასევე სიმბოლურ თვისებებსაც:

```
let id = Symbol("id");
let user = {
  [id]: 123
};

let clone = Object.assign({}, user);

alert( clone[id] ); // 123
```

აქ არანაირი წინააღმდეგობა არ არის. იდეა იმაში მდგომარეობს, რომ როდესაც ვახდენთ ობიექტების კლონირებას ან შერწყმას, ჩვეულებრივ გვინდა ყველა თვისების კოპირება (მათ შორის თვისებები გასაღები-სიმბოლო, როგორიცაა ზემოთ მოცემულ მაგალითში id თვისება).

გლობალური სიმბოლოები

როგორც ვნახეთ, ჩვეულებრივ, ყველა სიმბოლო უნიკალურია, თუნდაც მათი სახელები ერთი და იგივე იყოს. მაგრამ ზოგჯერ, პირიქით გვინდა, რომ ერთნაირი სახელის სიმბოლოები ერთი და იგივე იყოს. მაგალითად, ჩვენი

დანართის სხვადასხვა ნაწილს სურს "id" სიმბოლოზე წვდომა, რაც ზუსტად ერთი და იგივე თვისებას გულისხმობს.

ამისათვის არსებობს გლობალური სიმბოლოების რეესტრი. ჩვენ შეგვიძლია შევქმნათ მასში სიმბოლოები და მივმართოთ მათ მოგვიანებით, და ყოველ ჯერზე, როდესაც ჩვენ მივმართავთ მას, გარანტირებული გვაქვს, რომ ერთი და იგივე სიმბოლო დაგვიბრუნდება.

Symbol.for(key) ბრძანება გამოიყენება რეესტრიდან სიმბოლოს წასაკითხად (ან, თუ არ არსებობს, შესაქმნელად).

ის ამოწმებს გლობალურ რეესტრს და, თუ შეიცავს სიმბოლოს სახელად key, აბრუნებს მას, წინააღმდეგ შემთხვევაში, იქმნება ახალი სიმბოლო Symbol(key) და ჩაიწერება რეესტრში key გასაღების სახელით.

მაგალითად:

```
// ვკითხულობთ სიმბოლოს გლობალური რეესტრიდან და ვწერთ მას ცვლადში
let id = Symbol.for("id"); // თუ სიმბოლო არ არსებობს, მაშინ ის შეიქმნება

// ვკითხულობთ მას ხელახლა და ვწერთ სხვა ცვლადში (შესაძლოა კოდის სხვა ადგილიდან)
let idAgain = Symbol.for("id");

// // ვამოწმებთ იგივე სიმბოლოა თუ არა
alert( id === idAgain ); // true
```

რეესტრში შემავალ სიმბოლოებს გლობალურ სიმბოლოებს უწოდებენ. თუ გჭირდებათ სიმბოლო, რომელიც

ხელმისაწვდომი იქნება ყველგან კოდში, გამოიყენეთ გლობალური სიმბოლოები.

Symbol.keyFor

გლობალური სიმბოლოებისთვის, Symbol.for(key)-ს გარდა, რომელიც ეძებს სიმბოლოს სახელის მიხედვით, არსებობს საწინააღმდეგო მეთოდი: Symbol.keyFor(sym), რომელიც, პირიქით, იღებს გლობალურ სიმბოლოს და აბრუნებს მის სახელს.

მაგალითად:

```
// ვღებულობთ სიმბოლოს სახელით
let sym = Symbol.for("name");
let sym2 = Symbol.for("id");

// ვღებულობთ სახელს სიმბოლოთი
alert( Symbol.keyFor(sym) ); // name
alert( Symbol.keyFor(sym2) ); // id
```

Symbol.keyFor მეთოდის შიგნით, სიმბოლოს სახელის საპოვნელად გამოიყენება გლობალური სიმბოლოების რეესტრი. ასე რომ, ეს მეთოდი არაგლობალურ სიმბოლოებისათვის არ იმუშავებს. თუ სიმბოლო არაგლობალურია, მეთოდი ვერ პოულობს მას და შედეგად დააბრუნებს undefined.

თუმცა, ნებისმიერი სიმბოლოსთვის ხელმისაწვდომია თვისება description.

მაგალითად:

```
let globalSymbol = Symbol.for("name");
let localSymbol = Symbol("name");
```

```
alert( Symbol.keyFor(globalSymbol) ); // name, გლობალური სიმბოლო
alert( Symbol.keyFor(localSymbol) ); // undefined არაგლობალური სიმბოლოსათვის

alert( localSymbol.description ); // name
```

სისტემური სიმბოლოები

არსებობს მრავალი „სისტემური“ სიმბოლო, რომელიც JavaScript-ში გამოიყენება და ჩვენ შეგვიძლია გამოვიყენოთ ისინი ობიექტების სხვადასხვა ასპექტების მოსამართად.

ეს სიმბოლოები ქვემოთ, ცხრილის სპეციფიკაციაშია ჩამოთვლილი:

სპეციფიკაციის სახელი	[[აღწერა]]	მნიშვნელობა და დანიშნულება
@@asyncIterator	"Symbol.asyncIterator"	მეთოდი, რომელიც ობიექტისათვის გულისხმობის პრინციპით აბრუნებს AsyncIterator-ს. მისი გამოძახება for-await-of

		ოპერატორის სემანტიკით ხდება.
@@hasInstance	"Symbol.hasInstance"	მეთოდი, რომელიც განსაზღვრავს, აღიქვამს თუ არა კონსტრუქტორის ობიექტი ობიექტს, როგორც კონსტრუქტორის ერთ-ერთ ეგზემპლარს. მისი გამოძახება instanceof ოპერატორის სემანტიკით ხდება.
@@isConcatSpreadable	"Symbol.isConcatSpreadable"	თვისება ლოგიკური მნიშვნელობით, რომელიც, თუ მისი მნიშვნელობა true-ს ტოლია მიუთითებს, რომ

		ობიექტი მასივის ელემენტებამდე უნდა იყოს დაყვანილი Array.prototype.concat მეთოდის დახმარებით.
@@iterator	"Symbol.iterator"	მეთოდი, რომელიც ობიექტისათვის გულისხმობის პრინციპით აბრუნებს Iterator-ს. მისი გამოძახება for-of ოპერატორის სემანტიკით ხდება.
@@match	"Symbol.match"	რეგულარული გამოხატვის მეთოდი, რომელიც რეგულარულ გამოსახულებას შეუსაბამებს სტრიქონს. მისი

		გამოძახება String.prototype.match მეთოდით ხდება.
@@matchAll	"Symbol.matchAll"	რეგულარული გამოხატვის მეთოდი, რომელიც აბრუნებს Iterator-ს და რომელიც რეგულარულ გამოსახულებას შეუსაბამებს სტრიქონს. მისი გამოძახება String.prototype.matchAll მეთოდით ხდება.
@@replace	"Symbol.replace"	რეგულარული გამოხატვის მეთოდი, რომელიც სტრიქონის შესატყვის ქვესტრიქონებს

		ცვლის. სტრიქონს. მისი გამოძახება String.prototype.replace მეთოდით ხდება.
@@search	"Symbol.search"	რეგულარული გამოხატვის მეთოდი, რომელიც აბრუნებს ინდექსს სტრიქონში, რომელიც შეესაბამება რეგულარულ გამოსახულებას.
@@species	"Symbol.species"	ფუნქციის მიერ შეფასებული თვისება, რომელიც არის ფუნქცია-კონსტრუქტორი, რომელიც წარმოებული ობიექტების შესაქმნელად გამოიყენება.

@@split	"Symbol.split"	მეთოდი, რომელიც სტრიქონს ყოფს რეგულარული გამოხატვის შესაბამის ინდექსებზე. მისი გამოძახება String.prototype.split მეთოდით ხდება.
@@toPrimitive	"Symbol.toPrimitive"	მეთოდი, რომელიც ობიექტს შესაბამის პრიმიტიულ მნიშვნელობად გარდაქმნის. მისი გამოძახება ToPrimitive აბსტრაქტული ოპერაციით ხდება.
@@toStringTag	"Symbol.toStringTag"	String თვისება, რომელიც გამოიყენება ობიექტის სტრიქონული

		აღწერის ნაგულისხმევად შესაქმნელად. მასზე წვდომა Object.prototype.toString ჩაშენებული მეთოდით ხდება.
@@unscopables	"Symbol.unscopables"	ობიექტის შეფასების თვისება, რომლის თვისებების საკუთარი და მემკვიდრეობითი სახელები წარმოადგენენ თვისებების სახელებს, რომლებიც არ უკავშირდება ასოცირებული ობიექტის გარემოს.

კერძოდ, Symbol.toPrimitive საშუალებას გაძლევთ აღწეროთ ობიექტის ის წესები, რომლის მიხედვითაც იგი პრიმიტივად გარდაიქმნება.

ობიექტების პრიმიტივად გარდაქმნა

რა მოხდება, თუ ორ ობიექტს obj1 + obj2 შეკრებთ, ერთს მეორეს obj1 - obj2 გამოაკლებთ, ან გამოიტანთ მათ alert(obj) ბრძანების გამოყენებით?

ამ შემთხვევაში, ობიექტები ჯერ ავტომატურად პრიმიტივებად გარდაიქმნება და შემდეგ შესრულდება ოპერაცია.

ტიპების გარდაქმნის დროს ჩვენ ვნახეთ რიცხვითი, სტრიქონი და ლოგიკური მონაცემების გარდაქმნის წესები, მაგრამ ობიექტების გარდაქმნა არ შეგვისწავლია. ახლა, რადგან ჩვენ უკვე ვიცით ობიექტების მეთოდებისა და სიმბოლოების შესახებ, შეგვიძლია ეს გამოვასწოროთ.

1. ლოგიკური კონტექსტში ყველა ობიექტი ჭეშმარიტია. არსებობს მხოლოდ მათი რიცხვითი და სტრიქონული გარდაქმნები;
2. რიცხვითი გარდაქმნები ხდება მაშინ, როდესაც ვაკლებთ ობიექტებს ან ვასრულებთ მათემატიკურ მოქმედებებს. მაგალითად, თარიღის ობიექტები შეიძლება გამოაკლდეს date1 - date2 და მიიღება დროის ინტერვალს ორ თარიღს შორის სხვაობა;
3. რაც შეეხება სტრიქონების გარდაქმნებს, ისინი ჩვეულებრივ ხდება მაშინ, როდესაც ჩვენ გამოვიტანთ alert(obj) ობიექტს და ასევე სხვა შემთხვევებში, როდესაც ობიექტი გამოიყენება როგორც სტრიქონი.

ჩვენ შეგვიძლია დავაზუსტოთ სტრიქონები და რიცხვითი გარდაქმნები ობიექტის სპეციალური მეთოდების გამოყენებით.

გარდაქმნებისთვის არსებობს სამი ვარიანტი ("სამი მინიშნება"), რომელიც აღწერილია სპეციფიკაციაში: string.

ობიექტის სტრიქონად გადაქმნა, როდესაც ოპერაცია ელოდება სტრიქონის მიწოდებას, როგორიცაა alert:

```
// გამოტანა
alert(obj);

// ობიექტს ვიყენებთ თვისების სახელად
anotherObj[obj] = 123;
```

ობიექტის რიცხვად გარდაქმნა მათემატიკური მოქმედებების შემთხვევაში:

```
// აშკარა გარდაქმნა
let num = Number(obj);

// მათემატიკური გარდაქმნა (ბინარული „+“-ის გამოკლებით)
let n = +obj; // უნარული პლუსი
let delta = date1 - date2;

// შედარება მეტია/ნაკლებია
let greater = user1 > user2;
```

იშვიათად ხდება, როდესაც მიიღება default ოპერატორი. მაგალითად, ბინარული პლუსი „+“ შეიძლება მუშაობდეს ორივე ტიპის დროს: სტრიქონებზე (გააერთიანეთ ისინი) და რიცხვებზე (შეკრიბეთ ისინი). ამრიგად, ორივე შემთხვევაში შესრულდება ოპერაცია. ან როდესაც ობიექტების არამკაცრი ტოლობით ხდება შედარება == სტრიქონთან, რიცხვთან ან სიმბოლოსთან და გაუგებარია, რომელი გარდაქმნა უნდა განხორციელდეს.

```
// ბინარული პლუსი
```

```
let total = car1 + car2;

// obj == string/number/symbol
if (user == 1) { ... };
```

ოპერატორი მეტია/ნაკლებია <> ასევე მუშაობს როგორც სტრიქონებთან, ასევე რიცხვებთან. თუმცა, ისტორიული მიზეზების გამო, ის იყენებს „number“ მინიშნებას და არა „default“.

პრაქტიკაში, ყველა ჩაშენებული ობიექტი, Date-ის გარდა, „default“ გარდაქმნებს ახორციელებს ისევე, როგორც „number“. და ჩვენც ასევე უნდა მოვიქცეთ.

გთხოვთ გაითვალისწინოთ, რომ მინიშნებებისთვის არსებობს მხოლოდ სამი ვარიანტი. ყველაფერი მარტივია. არ არსებობს „boolean“ მინიშნება (ლოგიკურ კონტექსტში ყველა ობიექტი არის true) ან რაიმე სხვა. და თუ „default“ და „number“ იგივეა, როგორც ჩაშენებული ობიექტების უმეტესობა, მაშინ გარდაქმნების მხოლოდ ორი ვარიანტი დარჩება.

გარდაქმნის პროცესში, JavaScript-ის ძრავა ცდილობს ობიექტის შემდეგი სამი მეთოდი მოიძიოს და გამოიძახოს obj[Symbol.toPrimitive](hint) მეთოდი Symbol.toPrimitive (სისტემური სიმბოლო) სიმბოლური გასაღებით, თუ ასეთი მეთოდი არსებობს და გადასცეს მას მინიშნება.

იმ შემთხვევაში, თუ მინიშნება უდრის „string“-ს, ცდილობს გამოიძახოს obj.toString(), და თუ ის არ არსებობს, მაშინ obj.valueOf() თუ ის არსებობს ; თუ მინიშნება არის „default“ ან „number“ ცდილობს გამოიძახოს obj.valueOf(), და თუ ის არ არის, მაშინ obj.toString() - თუ ის არის.

დავიწყოთ უნივერსალური მიდგომით - Symbol.toPrimitive სიმბოლო: ამ სახელწოდების მეთოდი (თუ არის) ყველა გარდაქმნისთვის გამოიყენება:

```
obj[Symbol.toPrimitive] = function(hint) {
  // უნდა დააბრუნოს პრიმიტიული მნიშვნელობა
  // hint უდრის ერთ-ერთ მნიშვნელობას: „string“, „number“ ან „default“
};
```

მაგალითისათვის გამოვიყენოთ იგი user ობიექტის რეალიზების დროს:

```
let user = {
  name: "გიორგი",
  money: 1000,

  [Symbol.toPrimitive](hint) {
    alert(‘hint: ${hint}’);
    return hint == "string" ? ‘{name: "${this.name}"}’ : this.money;
  }
};

// გარდაქმნების შედეგების დემონსტრირება:
alert(user); // hint: string -> {name: "გიორგი"}
alert(+user); // hint: number -> 1000
alert(user + 500); // hint: default -> 1500
```

როგორც კოდიდან ჩანს, user ან ინფორმაციულ წასაკითხ სტრიქონად, ან ფინანსურ ანგარიშად გარდაიქმნება, hint-ის

მნიშვნელობიდან გამომდინარე. ერთადერთმა user[Symbol.toPrimitive] მეთოდმა შეძლო ყველა გარდაქმნის დამუშავება.

toString და valueOf მეთოდები დიდი ხანია გამოიყენება. ეს არ არის სიმბოლოები, რადგან სიმბოლოები ჯერ კიდევ არ არსებობდა იმ დროს, მაგრამ უბრალოდ ობიექტების ჩვეულებრივი მეთოდები სტრიქონული სახელებით. ისინი ობიექტების გარდაქმნის ძველი გზის განხორციელებას უზრუნველყოფენ.

თუ არ არსებობს Symbol.toPrimitive მეთოდი, JavaScript ძრავა ცდილობს მოძებნოს ეს მეთოდები და შემდეგნაირად გამოიძახოს ისინი:

- toString -> valueOf – „string“ მნიშვნელობისათვის;
- valueOf -> toString - სხვა შემთხვევაში.

მაგალითად, ჩვენ მათ ვიყენებთ იმავე user ობიექტის რეალიზებაში. მოვახდინოთ მისი განხორციელება toString და valueOf მეთოდების კომბინაციით:

```
let user = {
  name: "გიორგი",
  money: 1000,

  // hint უდრის „string“-ს
  toString() {
    return ‘{name: "${this.name}"}’;
  },

  // hint უდრის „number“ ან „default“
  valueOf() {
```

```
    return this.money;
  }

};

alert(user); // toString -> {name: "გიორგი"}
alert(+user); // valueOf -> 1000
alert(user + 500); // valueOf -> 1500
```

როგორც ხედავთ, შედეგი იგივეა, რაც წინა მაგალითში Symbol.toPrimitive-ით იყო.

ხშირად ჩვენ გვინდა ობიექტის ყველა სიტუაციისთვის პრიმიტივად გარდაქმნა ერთი „უნივერსალურით" აღვწეროთ. ამისათვის, საკმარისია შექმნათ ერთი toString:

```
let user = {
  name: "გიორგი",

  toString() {
    return this.name;
  }
};

alert(user); // toString -> გიორგი
alert(user + 1000); // toString -> გიორგი1000
```

Symbol.toPrimitive-ისა და valueOf-ის არარსებობის შემთხვევაში, toString პრიმიტივებად ყველა გარდაქმნას გაუმკლავდება.

მნიშვნელოვანია გვესმოდეს, რომ ობიექტების გარდაქმნის ყველა აღწერილი მეთოდი არ არის საჭირო მოთხოვნილი პრიმიტიული ტიპის ზუსტად დასაბრუნებლად.

აუცილებელი მოთხოვნა არ არის, რომ toString()-მა დააბრუნოს ზუსტად სტრიქონი, ან რომ Symbol.toPrimitive მეთოდმა დააბრუნოს ზუსტად რიცხვი „number" მინიშნებისთვის.

ერთადერთი სავალდებულო მოთხოვნაა, რომ მეთოდებმა უნდა დააბრუნონ პრიმიტივი და არა ობიექტი.

ისტორიული მიზეზების გამო, თუ toString ან valueOf დააბრუნებს ობიექტს, მაშინ შეცდომა არ იქნება, მაგრამ ასეთი მნიშვნელობის იგნორირება მოხდება (თითქოს მეთოდი საერთოდ არ არსებობდეს).

მეორეს მხრივ, Symbol.toPrimitive მეთოდმა უნდა დააბრუნოს პრიმიტივი, წინააღმდეგ შემთხვევაში იქნება შეცდომა.

ოპერაცია, რომელმაც დაიწყო გარდაქმნები, იღებს პრიმიტივს და შემდეგ აგრძელებს მუშაობას, საჭიროების შემთხვევაში ასრულებს შემდგომ გარდაქმნებს.

მაგალითად:

- მათემატიკური ოპერაციები, ბინარული პლუსის გამოკლებით, პრიმიტივს გარდაქმნის რიცხვად:

```
let obj = {
  // toString სხვა მეთოდების არ არსებობის შემთხვევაში ყველა
გარდაქმნებს ამუშავებს
  toString() {
    return "2";
  }
```

```
};

alert(obj * 2); // 4, მოხდა ობიექტის გარდაქმნა პრიმიტივად „2“, შემდეგ გამრავლებამ გადააქცია იგი რიცხვად
```

- ანალოგიურ სიტუაციაში ბინარული პლუსი მოახდენს სტრიქონების გაერთიანებას:

```
let obj = {
  toString() {
    return "2";
  }
};

alert(obj + 2); // 22 (პრივიტივად გარდაქმნამ დააბრუნა სტრიქონი => კონკატენაცია)
```

პრიმიტივების მეთოდები

JavaScript საშუალებას გვაძლევს პრიმიტიული ტიპის მონაცემებთან ვიმუშაოთ - სტრიქონები, რიცხვები და ა.შ. - თითქოს ისინი იყვნენ ობიექტები. მათაც აქვთ მეთოდები. ჯერ ვნახოთ, როგორ მუშაობს ეს ყველაფერი, რადგან, რა თქმა უნდა, პრიმიტივები არ არის ობიექტები.

ვნახოთ პრიმიტივებსა და ობიექტებს შორის ძირითადი განსხვავება.

პრიმიტივი - ეს არის „პრიმიტიული“ ტიპის მნიშვნელობები.

არსებობს 7 ძირითადი პრიმიტიული ტიპი: string, number, boolean, symbol, null, undefined და bigint.

ობიექტს, როგორც მისი თვისება, მრავალი მნიშვნელობის შენახვა შეუძლია.

მისი გამოცხადება ფიგურული ფრჩხილების {} გამოყენებით ხდება, მაგალითად: {სახელი: "გიორგი", ასაკი: 30}. JavaScript-ში არის სხვა სახის ობიექტები: მაგალითად, ფუნქციები ასევე ობიექტებია.

ობიექტების ერთ-ერთი საუკეთესო თავისებურება არის ის, რომ ჩვენ შეგვიძლია შევინახოთ ფუნქცია, როგორც ობიექტის ერთ-ერთი თვისება.

```
let Georgi = {
  name: "Georgi",
  Func: function() {
    alert("Hello my friend!");
  }
};

Georgi.Func(); // Hello my friend!
```

ზემოთ განხილულ მაგალითში შევქმენით Georgi ობიექტი Func მეთოდით.

არსებობს ბევრი ჩაშენებული ობიექტი. მაგალითად, ისინი, რომლებიც მუშაობენ თარიღებთან, შეცდომებთან, HTML ელემენტებთან და ა.შ. მათ აქვთ განსხვავებული თვისებები და მეთოდები.

თუმცა, ამ შესაძლებლობებს უარყოფითი მხარეც აქვს.

ობიექტები „უფრო მძიმეა“, ვიდრე პრიმიტივები. შიდა სტრუქტურის შესანარჩუნებლად მათ დამატებითი რესურსები სჭირდებათ.

თვისებები „გეტერები“ და „სეტერები“

არსებობს ობიექტის ორი თვისება.

პირველი არის მონაცემთა თვისებები (data properties). ჩვენ უკვე ვიცით, როგორ ვიმუშაოთ მათთან. ყველა თვისება, რომელსაც ჩვენ ამ მომენტამდე ვიყენებდით, იყო მონაცემთა თვისებები.

მეორე ტიპის თვისება ეს არის თვისება-აქსესუარი (accessor properties). ეს არის ფუნქციები, რომლებიც გამოიყენება მნიშვნელობის მისანიჭებლად და მისაღებად, მაგრამ გარე კოდში ისინი ჩვეულებრივი ობიექტის თვისებების მსგავსად გამოიყენება.

თვისება-აქსესუარი წარმოდგენილია მეთოდებით: „გეტერები“ - წასაკითხად და „სეტერები“ - ჩაწერისთვის. როდესაც ობიექტი პირდაპირ არის გამოცხადებული, ისინი აღინიშნება get და set-ით:

```
let obj = {
  get propName() {
    // გეტერი, მუშაობს წაკითხვის დროს obj.propName
  },

  set propName(value) {
    // სეტერი, მუშაობს ჩაწერის დროს obj.propName = value
  }
};
```

გეტერი მაშინ მუშაობს, როდესაც obj.propName-ის წაკითხვა ხდება, ხოლო სეტერი - როდესაც მნიშვნელობის

მინიჭება ხდება. მაგალითად, ჩვენ გვაქვს user ობიექტი name და surname თვისებებით

```
let user = {
  name: "გიორგი",
  surname: "მაღლაკელიძე"
};
```

ახლა მოდით სრულ სახელისათვის დავამატოთ ობიექტის თვისება fullName, რომელიც ჩვენს შემთხვევაში არის „გიორგი მაღლაკელიძე“. რა თქმა უნდა, ჩვენ არ გვსურს იმ ინფორმაციის დუბლირება, რაც უკვე გვაქვს, ამიტომ ამას განვახორციელებთ აქსესუარის დახმარებით:

```
let user = {
  name: "გიორგი",
  surname: "მაღლაკელიძე",

  get fullName() {
    return ‘${this.name} ${this.surname}’;
  }
};

alert(user.fullName); // გიორგი მაღლაკელიძე
```

გარედან, აქსესუარი-თვისება ჩვეულებრივ თვისებას ჰგავს. ზუსტად ამაში მდგომარეობს აქსესუარი-თვისების არსი. ჩვენ არ ვიძახებთ user.fullName-ს, როგორც ფუნქციას, მაგრამ ვკითხულობთ როგორც ჩვეულებრივი თვისებას: გეტერი ყველა სამუშაოს შეასრულებს კულისებში.

ამ დროისთვის, fullName-ს აქვს მხოლოდ გეტერი. თუ ჩვენ ვცდილობთ მივანიჭოთ user.fullName=, მოხდება შეცდომა:

```
let user = {
  get fullName() {
    return '...';
  }
};

user.fullName = "Тест"; // შეცდომაა (თვისებას აქვს მხოლოდ
    გეტერი)
```

გავასწოროთ ეს შეცდომა და user.fullName-ს დავუმატოთ სეტერი:

```
let user = {
  name: "გიორგი",
  surname: "მაღლაკელიძე",
  get fullName() {
    return '${this.name} ${this.surname}';
  },

  set fullName(value) {
    [this.name, this.surname] = value.split(" ");
  }
};

// set fullName გაიშვება მოცემული მნიშვნელობით
user.fullName = "ნიკოლოზ ბერიძე";
```

```
alert(user.name); // ნიკოლოზ
alert(user.surname); // ბერიძე
```

შედეგად ჩვენ მივიღეთ fullName „ვირტუალური თვისება. ის შეიძლება წავიკითხოთ და შევცვალოთ.

წვდომა თვისების დესკრიპტორები

აქსესუარი-თვისების დესკრიპტორები განსხვავდება ჩვეულებრივი მონაცემთა თვისებებისგან.

აქსესუარი-თვისებებს არ აქვს value ან writable, მაგრამ ამის ნაცვლად გთავაზობთ get და set ფუნქციებს. ანუ, აქსესუარის დესკრიპტორებს შეიძლება ჰქონდეს:

- get - ფუნქცია არგუმენტების გარეშე, რომელიც იმუშავებს თვისების წაკითხვის დროს;
- set - ფუნქცია, რომელიც იღებს ერთ არგუმენტს, და რომლის გამოძახება ხდება, თვისების მინიჭების დროს;
- numerable - იგივეა რაც მონაცემთა თვისებებისთვის;
- configurable - იგივეა, რაც მონაცემთა თვისებებისთვის.

მაგალითად, defineProperty-ის გამოყენებით fullName აქსესუარის შესაქმნელად, ჩვენ შეგვიძლია გადავცეთ დესკრიპტორი get და set-ის გამოყენებით:

```
let user = {
  name: "გიორგი",
  surname: "მაღლაკელიძე",
};

Object.defineProperty(user, 'fullName', {
  get() {
```

```
    return `${this.name} ${this.surname}`;
  },

  set(value) {
    [this.name, this.surname] = value.split(" ");
  }
});

alert(user.fullName); // გიორგი მაღლაკელიძე

for(let key in user) alert(key); // name, surname
```

ობიექტის თვისება შეიძლება იყოს ან აქსესუარი-თვისება (get/set მეთოდებით) ან მონაცემთა თვისება (value მნიშვნელობით).

თუ თქვენ ცდილობთ მიუთითოთ get-იც და value-ც ერთ დესკრიპტორში, მაშინ ეს შეცდომას გამოიწვევს.

ნარჩენი პარამეტრები და გაფართოების ოპერატორი

ბევრი ჩაშენებული JavaScript ფუნქცია მხარს უჭერს არგუმენტების ნებისმიერ რაოდენობას.

მაგალითად:

- Math.max(arg1, arg2, ..., argN) – მოცემულ არგუმენტებს შორის გამოითვლის მაქსიმალურ მნიშვნელობას;
- Object.assign(dest, src1, ..., srcN) - საწყისი ობიექტებიდან src1...srcN dest მიზნის ობიექტში აკოპირებს თვისებებს;

- …და ასე შემდეგ.

აქ ჩვენ შევისწავლით თუ როგორ გავაკეთოთ იგივე მომხმარებლის ფუნქციებში და როგორ გადავცეთ ასეთ ფუნქციებს პარამეტრები მასივის სახით.

ნარჩენი პარამეტრები

თქვენ შეგიძლიათ გამოიძახოთ ფუნქცია ნებისმიერი რაოდენობის არგუმენტებით, მიუხედავად იმისა, თუ როგორ იყო იგი განსაზღვრული.

მაგალითად:

```
function sum(a, b) {
  return a + b;
}

alert( sum(1, 2, 3, 4, 5) );
```

დამატებითი არგუმენტები არ გამოიწვევს შეცდომას. მაგრამ, რა თქმა უნდა, მხედველობაში მიიღება მხოლოდ პირველი ორი.

ნარჩენი პარამეტრები შეიძლება აღვნიშნოთ სამი წერტილით ... სიტყვასიტყვით ეს ნიშნავს: „შეაგროვეთ დარჩენილი პარამეტრები და შეინახე ისინი მასივში“.

მაგალითად, შევაგროვოთ ყველა არგუმენტი args მასივში:

```
function sum(...args) { // args - მასივის სახელი
  let s = 0;

  for (let arg of args) s += arg;

```

```
  return s;
}


alert( sum(1) ); // 1
alert( sum(1, 2) ); // 3
alert( sum(1, 2, 3) ); // 6
```

ჩვენ შეგვიძლია პირველი რამდენიმე პარამეტრი ჩავსვათ როგორც ცვლადი, ხოლო დანარჩენი გავაერთიანოთ მასივად. მაგალითად:

```
function sum(a1, a2, ...args) {
ფუნქციის ტანი
}
```

ამ შემთხვევაში, პირველი ორი არგუმენტი იქნება აუცილებელი, ხოლო დანარჩენი არგუმენტების რაოდენობა არ იქნება შეზღუდული.

ნარჩენი პარამეტრები უნდა განთავსდეს პარამეტრების ბოლოს, მათ შემდეგ რაიმე პარამეტრის მითითება შეცდომას გამოიწვევს.

გაფართოების ოპერატორი

ჩვენ ვისწავლეთ როგორ მივიღოთ მასივი პარამეტრების სიიდან. მაგრამ ზოგჯერ ზუსტად საპირისპიროს გაკეთებაა საჭირო.

მაგალითად, არის ჩაშენებული Math.max ფუნქცია. ის სიიდან ამოარჩევს და შედეგად აბრუნებს ყველაზე დიდ რიცხვს:

```
alert( Math.max(3, 5, 1) ); // 5
```

დავუშვათ, გვაქვს რიცხვების მასივი [3, 5, 1]. ამისთვის როგორ გამოვიძახოთ ფუნქცია Math.max?

ჩვენ არ შეგვიძლია უბრალოდ ჩასვათ ისინი Math.max-ში, იგი მოელის, რომ მიიღებს რიცხვების სიას და არა ერთი მასივის.

```
let arr = [3, 5, 1];

alert( Math.max(arr) ); // NaN
```

რა თქმა უნდა, შეგვიძლია რიცხვების ხელით შეყვანა: Math.max(arr[0], arr[1], arr[2]). მაგრამ, ჯერ ერთი, ცუდად გამოიყურება და მეორეც, ყოველთვის არ ვიცით რამდენი არგუმენტი იქნება. შეიძლება ბევრი იყოს, ან საერთოდ არ იყოს. ასეთ შემთხვევაში, დაგვეხმარება გაფართოების ოპერატორი. ის ნარჩენი პარამეტრების მსგავსია. ის ასევე იყენებს ..., მაგრამ აკეთებს ზუსტად საპირისპიროს.

როდესაც ...arr გამოიყენება ფუნქციის გამოძახების დროს, ის „აფართოებს“ arr გადასარჩევ ობიექტს არგუმენტების სიაში.

Math.max-ისთვის:

```
let arr = [3, 5, 1];

alert( Math.max(...arr) ); // 5 (ეს ოპერატორი მასივს
                                 წარმოადგენს არგუმენტების სიის
                                 სახით)
```

ამავე წესით შეგვიძლია რამდენიმე იტერაციული ობიექტის გადაგზავნა:

```
let arr1 = [1, -2, 3, 4];
let arr2 = [8, 3, -8, 1];
```

```
alert( Math.max(...arr1, ...arr2) ); // 8
```

ჩვენ ასევე შეგვიძლია გაფართოების ოპერატორისა და ჩვეულებრივი მნიშვნელობების კომბინირებული გამოყენება:

```
let arr1 = [1, -2, 3, 4];
let arr2 = [8, 3, -8, 1];

alert( Math.max(1, ...arr1, 2, ...arr2, 25) ); // 25
```

გაფართოების ოპერატორი შეიძლება მასივების შერწყმისთვისაც გამოვიყენოთ:

```
let arr = [3, 5, 1];
let arr2 = [8, 9, 15];

let merged = [0, ...arr, 2, ...arr2];

alert(merged); // 0,3,5,1,2,8,9,15 (0, შემდეგ arr, შემდეგ 2, ხოლო
    ბოლოს arr2)
```

ზემოთ მოყვანილ მაგალითებში გაფართოების ოპერატორის თვისებების დემონსტრირებისთვის მასივი გამოვიყენეთ, მაგრამ ის ნებისმიერ გამეორებად ობიექტთანაც მუშაობს.

მაგალითად, გაფართოების ოპერატორი შეიძლება გამოვიყენოთ სტრიქონის სიმბოლოების მასივად გადაქცევისთვის:

```
let str = "Hello!";
```

```
alert( [...str] ); // H,e,l,l,o,!
```

ვნახოთ რა მოხდება. გაფართოების ოპერატორი ელემენტების გადარჩევისთვის იყენებს იტერატორებს[5]. ისევე როგორც ამას აკეთებს for..of.

for..of ციკლის ოპერატორი სტრიქონს გადააქცევს სიმბოლოების მიმდევრობად, ამიტომ ...str-დან მიიღება „H“, „e“, „l“, „l“, „o“, „!“. შედეგად მიღებული სიმბოლოები გროვდება მასივში სტანდარტული მასივის გამოცხადების გამოყენებით: [...str].

ამ ამოცანისთვის ჩვენ ასევე შეგვიძლია გამოვიყენოთ Array.from. ისიც გამეორებად ობიექტს (როგორიცაა სტრიქონი) მასივად გარდაქმნის:

```
let str = "Hello!";

alert(Array.from(str) ); // H,e,l,l,o,!
```

[5] იტერატორი (ინგლ. iterator-დან - აღმრიცხველი) არის ინტერფეისი, რომელიც კოლექციის ელემენტებზე (მასივი ან კონტეინერი) წვდომას და მათში ნავიგაციას უზრუნველყოფს. იტერატორებს სხვადასხვა სისტემაში შეიძლება განსხვავებული სახელები ჰქონდეს. მონაცემთა ბაზის მართვის სისტემაში, იტერატორებს უწოდებენ კურსორებს. უმარტივეს შემთხვევაში, დაბალი დონის ენებში იტერატორი არის მაჩვენებელი. იტერატორების გამოყენება ზოგად დაპროგრამებაში საშუალებას გაძლევთ კონტეინერებთან მუშაობის უნივერსალური ალგორითმების რეალიზება განახორციელოთ. იტერატორის მთავარი მიზანია მომხმარებელს მისცეს წვდომა კონტეინერის ნებისმიერ ელემენტზე, ხოლო კონტეინერის შიდა სტრუქტურა მომხმარებლისგან დამალული იყოს.

შედეგი იქნება ანალოგიური, მაგრამ Array.from(obj)-სა და [...obj]-ს შორის არის სხვაობა:

• Array.from მუშაობს როგორც ფსევდო მასივებით, ასევე იტერაციული ობიექტებით;

• გაფართოების ოპერატორი მუშაობს მხოლოდ იტერაციული ობიექტებით.

გამოდის, რომ თუ რაიმესგან მასივის გაკეთება გჭირდებათ, მაშინ Array.from უფრო უნივერსალური მეთოდია.

მონაცემთა ტიპები

რიცხვები

თანამედროვე JavaScript-ში ორი ტიპის რიცხვი გამოიყენება:

1. JavaScript-ში ჩვეულებრივი რიცხვები 64-ბიტიან IEEE-754 ფორმატში ინახება, რომელსაც ასევე უწოდებენ "ორმაგი სიზუსტის მცურავმძიმიან რიცხვებს" (double precision floating point numbers). ეს ის რიცხვებია, რომლებსაც ყველაზე ხშირად ვიყენებთ.

2. BigInt რიცხვები ნებისმიერი სიგრძის მთელ რიცხვებთან მუშაობის შესაძლებლობას იძლევა. ისინი იშვიათად არის საჭირო და იმ შემთხვევებში გამოიყენება, როდესაც 2^{53}-ზე მეტი ან -2^{53}-ზე ნაკლები რიცხვებთან მუშაობაა საჭირო.

ქვემოთ განვიხილავთ ჩვეულებრივი ტიპის რიცხვებს: number ტიპის რიცხვებს. მოდით უფრო დაწვრილებით შევისწავლოთ, თუ როგორ უნდა JavaScript-ში მათთან მუშაობა.

რიცხვების ჩაწერის წესები

წარმოიდგინეთ, რომ ჩვენ უნდა ჩავწეროთ რიცხვი 1 მილიარდი. მისი ჩაწერის ყველაზე ჩვეულებრივი გზაა:

```
let bil = 1000000000;
```

რეალურ ცხოვრებაში, ჩვენ ჩვეულებრივ გამოვტოვებთ მრავალი ნულის ჩაწერას, რადგან შეცდომის დაშვება ადვილია. შემცირებული ჩანაწერი შეიძლება ასე გამოიყურებოდეს „1 მლრ.“ ან „7,3 მლრ.“ 7 მილიარდ 300 მილიონისთვის. ეს

პრინციპი შეიძლება ყველა დიდი რიცხვებისათვის გამოვიყენოთ.

JavaScript-ში შეგიძლიათ გამოიყენოთ ასო „e" რიცხვის ჩანაწერის შესამცირებლად. იგი ემატება რიცხვს და ცვლის ნულების მითითებულ რაოდენობას:

```
let bil = 1e9;  // 1 მილიარდი, დეტალურად: 1 და 9 ნული
alert( 7.3e9 );  // 7.3 მილიარდი (7,300,000,000)
```

სხვა სიტყვებით, რომ ვთქვათ „e" 10-ის შესაბამის ხარისხზე გამრავლების ოპერაციას ასრულებს.

```
1e3 = 1 * 10^3 = 1 * 1000
1.23e6 = 1.23 * 10^6 = 1.23 * 1000000
```

ახლა ჩავწეროთ რაღაც ძალიან პატარა. მაგალითად, 1 მიკროწამი (წამის მემილიონედი):

```
let ms = 0.000001;
```

მიკროწამის შემოკლებულ ჩაწერისათვის ისევ „e" დაგვეხმარება:

```
let ms = 1e-6; // 0,000001 = 1 / 1000000 = 1 * 10^-6
```

თექვსმეტობითი, ორობითი და რვაობითი რიცხვები

თექვსმეტობითი რიცხვები ფართოდ გამოიყენება JavaScript-ში ფერების, სიმბოლოების კოდირებისა და სხვათა გამოსასახად. ჩვეულებრივ, არსებობს მოკლე ჩანაწერი: 0x, რასაც მოჰყვება რიცხვი.

მაგალითად:

```
alert( 0xff ); // 255
```

```
alert( 0xFF ); // 255 (იგივეა, რადგან რეგისტრ ამ შემთხვევაში მნიშვნელობა არა აქვს)
```

ორობითი და რვაობითი რიცხვები არც ისე ხშირად გამოიყენება, მაგრამ ისინიც მხარდაჭერილია 0b ორობითი რიცხვებისთვის და 0o რვაობითი რიცხვებისთვის:

```
let a = 0b11111111; // 255 რიცხვის ჩაწერის ბინარული ფორმა
let b = 0o377; // 255 რიცხვის ჩაწერის რვაობითი ფორმა

alert( a == b ); // true, რადგან ორივე მხარეს 255 რიცხვია
```

ასეთი მხარდაჭერით მხოლოდ ათვლის 3 სისტემა სარგებლობს. სხვა რიცხვითი სისტემებისთვის, გირჩევთ parseInt ფუნქცია გამოიყენოთ.

toString(base)

num.toString(base) მეთოდს num რიცხვი ათვლის ათობითი სისტემიდან გაჰყავს base ათვლის სიტემაში. მაგალითად:

```
let num = 255;

alert( num.toString(16) );  // ff
alert( num.toString(2) );   // 11111111
```

base ცვლადის მნიშვნელობა შეიძლება იცვლებოდეს 2-დან 36-მდე (გულისხმობით იგი 10-ის ტოლია).

ხშირად გამოყენებული მნიშვნელობებია:

- base=16 - ფერის, სიმბოლოების თექვსმეტობითი კოდირებისთვის და ა.შ., მასში გამოყენებული ციფრები შეიძლება იყოს 0..9 ან A..F.

- base=2 - ჩვეულებრივ გამოიყენება ბიტური ოპერაციებისთვის, ციფრი 0 ან 1.
- base=36 - ციფრების მაქსიმალური რაოდენობა, ციფრები შეიძლება იყოს 0..9 ან A..Z. ანუ რიცხვების წარმოსადგენად გამოიყენება მთელი ლათინური ანბანი. მაგრამ შეგიძლიათ გამოიყენოთ 36-ბიტიანი რიცხვების სისტემა დიდი რიცხვითი იდენტიფიკატორების მოკლედ წარმოსადგენად. მაგალითად, მოკლე ბმულის შესაქმნელად. ამისათვის, უბრალოდ წარმოვადგინოთ იგი 36-ობით ათვლის სისტემაში:

```
alert( 123456..toString(36) ); // 2n9c
```

ორი წერტილი 123456..toString(36) არ არის შეცდომა. თუ ჩვენ გვჭირდება მეთოდის გამოძახება პირდაპირ რიცხვზე, როგორიცაა toString-ი ზემოთ მოცემულ მაგალითში, მაშინ რიცხვის შემდეგ უნდა დავწეროთ ორი წერტილი .. .

თუ დავსვამთ ერთ წერტილს: 123456.toString(36) მაშინ ეს შეცდომას გამოიწვევს, რადგან JavaScript-ის სინტაქსი ვარაუდობს, რომ პირველი წერტილის შემდეგ წილადი ნაწილი იწყება. თუ თქვენ დაწერთ ორ წერტილს, მაშინ JavaScript-ს ესმის, რომ წილადი ნაწილი აკლია და მეთოდი იწყება. ის ასევე შეიძლება ჩაიწეროს როგორც (123456).toString(36).

დამრგვალება

რიცხვებთან მუშაობისას ერთ-ერთი ყველაზე ხშირად გამოყენებული ოპერაცია არის დამრგვალება.

JavaScript-ს აქვს დამრგვალების რამდენიმე ჩაშენებული ფუნქცია:

Math.floor

დამრგვალება ნაკლებობით: 5,1 ხდება 5 და -2,1 ხდება -3.

Math.ceil

დამრგვალება მეტობით: 5,1 ხდება 6 და -2,1 ხდება -2.

Math.round

დამრგვალება უახლოეს მთელ რიცხვამდე: 5,1 ხდება 5, 5,7 ხდება 6 და -2,1 ხდება -2.

Math.trunc (მხარდაჭერილი არ არის Internet Explorer-ის მიერ)

წაშლის რიცხვის წილად ნაწილს დამრგვალების გარეშე: 5,1 ხდება 5, ხოლო -2,1 ხდება -2.

ქვემოთ მოცემულია ცხრილი, რომელშიც ასახულია განსხვავებები დამრგვალების ფუნქციებს შორის:

	Math.floor	Math.ceil	Math.round	Math.trunc
5,1	5	6	5	5
5,6	5	6	6	5
-2,1	-3	-2	-2	-2
-2,6	-3	-2	-3	-2

ეს ფუნქციები მოიცავს წილადი ნაწილის დამუშავების ყველა შესაძლო გზას. რა მოხდება, თუ დაგვჭირდება რიცხვის დამრგვალება წილადი ნაწილის მე-n-ე რიცხვამდე?

მაგალითად, გვაქვს 1,2345 და გვინდა დავამრგვალოთ რიცხვი და შევინარჩუნოთ 2 თანრიგი მძიმის შემდეგ და დარჩეს მხოლოდ 1,23.

ამ ამოცანის გადაწყვეტის ორი გზა არსებობს:

1. გამრავლება და გაყოფა.

მაგალითად, რიცხვში, რომ შევინარჩუნოთ ორი თანრიგი მძიმის შემდეგ, შეგვიძლია რიცხვი გავამრავლოთ 100-ზე, გამოვიძახოთ დამრგვალების ფუნქცია და ისევ გავყოთ 100-ზე.

```
let num = 1.23456;

alert( Math.floor(num * 100) / 100 ); // 1.23456 -> 123.456 -> 123 -> 1.23
```

2. toFixed(n) მეთოდი მძიმის შემდეგ n თანრიგს ინარჩუნებს და შედეგად აბრუნებს სტრიქონს.

```
let num = 12.34;
alert( num.toFixed(1) ); // "12.3"
```

მნიშვნელობას ამრგვალებს უახლოეს რიცხვამდე, მეტობით და ნაკლებობით, Math.round მეთოდის მსგავსად:

```
let num = 12.36;
alert( num.toFixed(1) ); // "12.4"
```

გაითვალისწინეთ, რომ toFixed-ის შედეგი არის სტრიქონი. თუ წილადი ნაწილი საჭიროზე მოკლეა, სტრიქონის ბოლოს ნულები დაემატება:

```
let num = 12.34;
alert( num.toFixed(5) ); // "12.34000", მძიმის შემდეგ 5 თანრიგის მისაღებად ბოლოში დაემატება ნულები
```

ჩვენ შეგვიძლია მიღებული მნიშვნელობა რიცხვად გარდავქნათ უნარული + ოპერატორის ან Number()-ს გამოყენებით, მაგალითად, უნარული ოპერატორით: +num.toFixed(5).

არაზუსტი გამოთვლები

JavaScript-ში რიცხვი წარმოდგენილია როგორც 64-ბიტიანი IEEE-754 ფორმატი. რიცხვის შესანახად გამოიყენება 64 ბიტი: მათგან 52 გამოიყენება ციფრების შესანახად, 11 გამოიყენება ათობითი წერტილის პოზიციის შესანახად (თუ რიცხვი მთელი რიცხვია, მაშინ ინახება 0), ხოლო ერთი ბიტი რიცხვის ნიშნისთვისაა დარეზერვებული.

თუ რიცხვი ძალიან დიდია, ის გადაავსებს 64-ბიტიან მეხსიერებას და JavaScript-ი დააბრუნებს უსასრულობას:

```
alert( 1e500 ); // Infinity
```

JavaScript-ში ციფრებთან მუშაობისას ყველაზე გავრცელე-ბული შეცდომა არის სიზუსტის დაკარგვა.

შევხედოთ ქვემოთ მოყვანილ (არასწორ) შედარებას:

```
alert( 0.1 + 0.2 == 0.3 ); // false
```

დიახ, 0,1-ის და 0,2-ის ჯამი არ უდრის 0,3-ს. ახლა ვნახოთ რას უდრის ეს ჯამი:

```
alert( 0.1 + 0.2 ); // 0.30000000000000004
```

რატომ ხდება ეს?

რიცხვები მეხსიერებაში ინახება ორობითი ფორმით, ბიტების თანმიმდევრობით - ერთიანები და ნულები. მაგრამ წილადები, როგორიცაა 0.1, 0.2, რომლებიც საკმაოდ მარტივად გამოიყურება ათვლის ათობითი სისტემაში, ორობით სისტემაში არის უსასრულო წილადი.

JavaScript-ში არ არსებობს გზა, რომ 0.1-ის ან 0.2-ის ორობით სისტემაში ზუსტი მნიშვნელობები შეინახოს, ისევე,

როგორც არ არსებობს ათობით სისტემაში ერთი მესამედის შესანახად.

IEEE-754 რიცხვითი ფორმატი ამ პრობლემას უახლოეს შესაძლო რიცხვამდე დამრგვალებით წყვეტს. დამრგვალების წესები ჩვეულებრივ საშუალებას არ გვაძლევს „სიზუსტის ეს მცირე დანაკარგი“ დავინახოთ, მაგრამ ის არსებობს.

მაგალითი:

```
alert( 0.1.toFixed(20) ); // 0.10000000000000000555
```

როდესაც ხდება ორი რიცხვის შეკრება, მათთან ერთად ეს „უზუსტობებიც“ იკრიბება.

სამართლიანობისთვის უნდა აღინიშნოს, რომ მცურავმძიმიანი რიცხვების გამოთვლების სიზუსტის შეცდომა ნებისმიერ სხვა ენაშიც გვხდება, რომელიც იყენებს IEEE 754 ფორმატს, მათ შორის PHP, Java, C, Perl, Ruby.

შესაძლებელია თუ არა პრობლემის თავიდან აცილება? რა თქმა უნდა, ყველაზე საიმედო გზაა შედეგის დამრგვალება toFixed(n) მეთოდის გამოყენებით:

```
let sum = 0.1 + 0.2;
alert( sum.toFixed(2) ); // 0.30
```

გახსოვდეთ, რომ toFixed მეთოდი ყოველთვის აბრუნებს სტრიქონს. ეს გარანტიას იძლევა, რომ შედეგი იქნება წილად ნაწილში ციფრების მოცემული რაოდენობა. სტრიქონის რიცხვად გადაქცევისთვის შეგიძლიათ გამოიყენოთ უნარული + ოპერატორი:

```
 let sum = 0.1 + 0.2;
alert( +sum.toFixed(2) ); // 0.3
```

თქვენ ასევე შეგიძლიათ დროებით გაამრავლოთ რიცხვი 100-ზე (ან მეტზე), რათა მიიყვანოთ იგი მთელ რიცხვამდე, შეასრულოთ მათემატიკური მოქმედება და შემდეგ ისევ გაყოთ იმავე რიცხვზე. მთელი რიცხვების შეჯამებით ჩვენ შეცდომას ვამცირებთ, მაგრამ საბოლოო გაყოფის დროს ის მაინც ჩნდება:

```
alert( (0.1 * 10 + 0.2 * 10) / 10 ); // 0.3
alert( (0.28 * 100 + 0.14 * 100) / 100); // 0.4200000000000001
```

ამგვარად, გამრავლება/გაყოფის მეთოდი შეცდომას ამცირებს, მაგრამ ამ პრობლემას ბოლომდე არ წყვეტს. უბრალოდ გამოიყენეთ დამრგვალების მეთოდი.

isFinite და isNaN შემოწმება

ჩვენ ზემოთ განვიხილეთ სპეციალური რიცხვითი მნიშვნელობები:

- Infinity (-Infinity) არის სპეციალური რიცხვითი მნიშვნელობა, რომელიც იქცევა ზუსტად ისე, როგორც მათემატიკური უსასრულობა ∞.
- NaN წარმოადგენს შეცდომას.

ეს რიცხვითი მნიშვნელობები არის number ტიპის, მაგრამ ისინი არ არიან „ჩვეულებრივი" რიცხვები, ამიტომ მათ შესამოწმებლად არსებობს ფუნქციები:

isNaN(value) გარდაქმნის მნიშვნელობას რიცხვად და ამოწმებს არის თუ არა ის NaN:

```
alert( isNaN(NaN) ); // true
alert( isNaN("str") ); // true
```

გვჭირდება თუ არა ეს ფუნქცია? არ შეგვიძლია უბრალოდ შევადაროთ === NaN-ს? სამწუხაროდ არა. NaN-ის მნიშვნელობა

უნიკალურია იმით, რომ ის არაფრის ტოლი არ არის, არც საკუთარი თავის:

```
alert( NaN === NaN ); // false
```

isFinite(value) არგუმენტს გარდაქმნის რიცხვად და შედეგად აბრუნებს true, თუ არგუმენტი არის ჩვეულებრივი რიცხვი და არა NaN/Infinity/-Infinity:

```
alert( isFinite("15") ); // true
alert( isFinite("str") ); // false, ვინაიდან სპეციალური მნიშვნელობაა: NaN
alert( isFinite(Infinity) ); // false, ვინაიდან სპეციალური მნიშვნელობაა: Infinity
```

ზოგჯერ isFinite გამოიყენება იმის შესამოწმებლად, შეიცავს თუ არა სტრიქონი რიცხვს:

```
let num = +prompt("Enter a number", '');

// ყოველთვის დააბრუნებს შედეგად true-ს, გარდა იმ შემთხვევისა, როდესაც არგუმენტი არის Infinity/-Infinity ან არაა რიცხვი
alert( isFinite(num) );
```

გახსოვდეთ, რომ ცარიელი სტრიქონი ყველა რიცხვით ფუნქციაში ინტერპრეტირებულია როგორც 0, მათ შორის არის isFinite.

Object.is შედარება

არსებობს სპეციალური Object.is მეთოდი, რომელიც ადარებს მნიშვნელობებს, მაგალითად ===, მაგრამ უფრო საიმედოა ორ განსაკუთრებულ სიტუაციაში:

1. მუშაობს NaN-თან: Object.is(NaN, NaN) === true, აქ კარგია.

2. 0 და -0 მნიშვნელობები განსხვავებულია: Object.is(0, -0) === false, ეს იშვიათად გამოიყენება, მაგრამ ტექნიკურად ეს მნიშვნელობები განსხვავებულია.

ყველა სხვა შემთხვევაში, Object.is(a, b) იდენტურია a === b-ის.

შედარების ეს გზა ხშირად გამოიყენება JavaScript-ის სპეციფიკაციაში. როდესაც შიგა ალგორითმს სჭირდება 2 მნიშვნელობის ზუსტი შედარებისთვის, ის იყენებს Object.is.

parseInt და parseFloat

რიცხვად გადაყვანის დროს, შეგიძლიათ გამოიყენოთ უნარული + ან Number(). თუ სტრიქონი რიცხვი არ არის, მაშინ შედეგი იქნება NaN:

```
alert( +"100px" ); // NaN
```

ერთადერთი გამონაკლისი არის ჰარები სტრიქონის წინ და ბოლოს, ისინი იგნორირებული იქნებიან.

რეალურ ცხოვრებაში ხშირად ვხვდებით მნიშვნელობებს, რომლებსაც აქვთ საზომი ერთეული, როგორიცაა „100px“ ან „12pt“ CSS-ში. ასევე, ბევრ ქვეყანაში ვალუტის სიმბოლო იწერება „25 ლ“ ნომინალის შემდეგ. როგორ მივიღოთ რიცხვითი მნიშვნელობა ასეთი სტრიქონებიდან? ამისათვის გამოიყენება parseInt და parseFloat ფუნქციები. ისინი „კითხულობენ“ რიცხვს

სტრიქონიდან. თუ წაკითხვისას მოხდა შეცდომა, აბრუნებენ შეცდომამდე მიღებულ რიცხვს. parseInt ფუნქცია აბრუნებს მთელ, ხოლო parseFloat აბრუნებს მცურავმძიმიან რიცხვს:

```
alert( parseInt('100px') ); // 100
alert( parseFloat('12.5em') ); // 12.5

alert( parseInt('12.3') ); // 12, შედეგად დააბრუნებს მხოლოდ მთელ ნაწილს
alert( parseFloat('12.3.4') ); // 12.3, რიცხვის წაკითხვა მეორე წერტილზე შეჩერდება
```

parseInt/parseFloat ფუნქციები დააბრუნებს NaN-ს, თუ ისინი ვერ წაიკითხავენ ციფრებს:

```
alert( parseInt('a123') ); // NaN, წაკითხვა შეჩერდება პირველივე სიმბოლოზე
```

parseInt(str, radix) ფუნქციას შეიძლება არააუცილებელი მეორე არგუმენტიც ჰქონდეს. მეორე არგუმენტი განსაზღვრავს ათვლის სისტემას. ასე, რომ parseInt ფუნქციას შეუძლია წაიკითხოს სტრიქონები ორობითი, თექვსმეტობითი და ა. შ. რიცხვებით:

```
alert( parseInt('0xff', 16) ); // 255
alert( parseInt('ff', 16) ); // 255, 0x-ის გარეშეც მუშაობს

alert( parseInt('2n9c', 36) ); // 123456
```

სხვა მათემატიკური ფუნქციები

JavaScript-ს აქვს Math ჩაშენებული ობიექტი, რომელიც შეიცავს სხვადასხვა მათემატიკურ ფუნქციას და მუდმივებს თავის თვისებებში და მეთოდებში. Math ობიექტი არ არის ფუნქციური ობიექტი.

Math არ მუშაობს BigInt რიცხვებთან.

სხვა გლობალური ობიექტებისგან განსხვავებით, Math ობიექტი არ არის კონსტრუქტორი. Math ობიექტის ყველა თვისება და მეთოდი სტატიკურია. თქვენ მიმართავთ π მუდმივას Math.PI-ის მეშვეობით და მიმართავთ სინუს ფუნქციას Math.sin(x) მეშვეობით, სადაც x არის მეთოდის არგუმენტი. JavaScript-ში მუდმივები განისაზღვრება ნამდვილი რიცხვების სრული სიზუსტით.

Math.E - ეილერის ან ნეპერის რიცხვი, ნატურალური ლოგარითმი, დაახლოებით უდრის 2,718-ს;

Math.LN2 - 2-ის ნატურალური ლოგარითმი, დაახლოებით 0,693;

Math.LN10 - 10-ის ნატურალური ლოგარითმი, დაახლოებით 2.303;

Math.LOG2E - E-ს ლოგარითმი 2-ის ფუძით, დაახლოებით 1,443;

Math.LOG10E - E-ს ლოგარითმი 10-ის ფუძით, დაახლოებით 0,434;

Math.PI - წრეწირის შეფარდება მის დიამეტრთან, დაახლოებით 3,14159;

Math.SQRT1_2 - კვადრატული ფესვი ½-დან; ან ეს ეკვივალენტურია, 1 გაყოფილი კვადრატულ ფესვი 2-დან, დაახლოებით 0,707;

Math.SQRT2 - კვადრატული ფესვი 2-დან, დაახლოებით 1,414.

შენიშვნა: გთხოვთ გაითვალისწინოთ, რომ ტრიგონომეტრიული ფუნქციები (sin(), cos(), tan(), asin(), acos(), atan() და atan2()) პარამეტრებად იყენებენ ან აბრუნებენ კუთხეებს რადიანებში. რადიანების გრადუსებში გადასაყვანად, მნიშვნელობა გაყავით (Math.PI / 180) სიდიდეზე; პირიქით გადასაყვანად, გრადუსი გაამრავლეთ იმავე სიდიდეზე.

Math.abs(x) - აბრუნებს რიცხვის აბსოლუტურ მნიშვნელობას;

Math.cbrt(x) - აბრუნებს რიცხვის კუბურ ფესვს;

Math.ceil(x) - აბრუნებს რიცხვის მნიშვნელობას, რომელიც დამრგვალებულია მეტობით უახლოეს მთელ რიცხვამდე;

Math.cos(x) - აბრუნებს რიცხვის კოსინუსს;

Math.exp(x) - აბრუნებს e^x, სადაც x არის არგუმენტი და e არის ეილერის რიცხვი (2.718…), ნატურალური ლოგარითმის ფუძე;

Math.floor(x) - აბრუნებს რიცხვის მნიშვნელობას, რომელიც დამრგვალებულია ნაკლებობით უახლოეს მთელ რიცხვამდე;

Math.hypot([x[, y[, …]]]) - აბრუნებს თავისი არგუმენტების კვადრატების ჯამიდან კვადრატულ ფესვს;

Math.log(x) - აბრუნებს რიცხვის ნატურალურ ლოგარითმს (ეს ცნობილია როგორც ln);

Math.log10(x) - აბრუნებს რიცხვის ლოგარითმს ფუძით 10;

Math.log2(x) - აბრუნებს რიცხვის ორობით ლოგარითმს;

Math.max([x[, y[, …]]]) - აბრუნებს არგუმენტებს შორის უდიდეს რიცხვს;

Math.min([x[, y[, …]]]) - აბრუნებს არგუმენტებს შორის უმცირეს რიცხვს;

Math.pow(n, power) - შედეგად აბრუნებს n რიცხვს აყვანილი power ხარისხში;

Math.random() - აბრუნებს ფსევდო შემთხვევით რიცხვს 0-დან (0-ის ჩათვლით) 1-მდე დიაპაზონში;

Math.round(x) - აბრუნებს რიცხვის მნიშვნელობას, რომელიც დამრგვალებულია უახლოეს მთელ რიცხვამდე;

Math.sign(x) - აბრუნებს რიცხვის ნიშანს, რომელიც მიუთითებს, არის თუ არა რიცხვი დადებითი, უარყოფითი ან ნულის ტოლი;

Math.sin(x) - აბრუნებს რიცხვის სინუსს;

Math.sqrt(x) - აბრუნებს რიცხვის დადებით კვადრატულ ფესვს;

Math.tan(x) - აბრუნებს რიცხვის ტანგენტს;

Math.trunc(x) - აბრუნებს რიცხვის მთელ ნაწილს, წილად ნაწილს ჩამოაცლის.

BigInt ფუნქცია

BigInt არის სპეციალური რიცხვითი ტიპი, რომელიც ნებისმიერი სიგრძის მთელ რიცხვებთან მუშაობის შესაძლებლობას იძლევა.

BigInt ტიპის მნიშვნელობის შესაქმნელად, დაამატეთ რიცხვითი ლიტერალის ბოლოს სიმბოლო n ან გამოიძახეთ BigInt ფუნქცია, რომელიც მიღებული არგუმენტიდან BigInt ტიპის რიცხვს შექმნის. არგუმენტი შეიძლება იყოს რიცხვი, სტრიქონი და ა.შ.

მაგალითად:

```
const sameBig =
BigInt("1234567890123456789012345678901234567890");
            // შედეგად მიიღება
1234567890123456789012345678901234567890n
const bigintFN = BigInt(10); // ეს იგივეა რაც 10n
```

მათემატიკური ოპერატორები

BigInt ტიპის რიცხვები შეიძლება გამოყენებულ იქნეს, როგორც ჩვეულებრივი რიცხვები, მაგალითად:

```
alert(1n + 2n); // 3

alert(5n / 2n); // 2
```

გაითვალისწინეთ, რომ გაყოფის ოპერაცია 5/2 შედეგად აბრუნებს დამრგვალებულ რიცხვს, წილადი ნაწილის გარეშე. bigint ტიპის რიცხვებზე ყველა ოპერაცია შედეგად აბრუნებს bigint ტიპის რიცხვს.

მათემატიკური ოპერაციების დროს, ჩვენ არ შეგვიძლია შევურიოთ bigint ტიპის რიცხვი და ჩვეულებრივი რიცხვები:

```
alert (1n + 2); // შეცდომა: BigInt და სხვა ტიპების შერევა შეუძლებელია
```

ჩვენ უნდა მოვახდინოთ მათი კონვერტაცია BigInt() ან Number() ფუნქციების გამოყენებით, მაგალითად:

```
let bigint = 1n;
let number = 2;
```

```
// ვახდენთ number ტიპის რიცხვის bigint ტიპად კონვერტირებას
alert(bigint + BigInt(number)); // 3

// ვახდენთ bigint ტიპის რიცხვის number ტიპად კონვერტირებას
alert(Number(bigint) + number); // 3
```

bigint ტიპის რიცხვის კონვერტაცია ყოველთვის უშეცდომოდ ხორციელდება, მაგრამ თუ bigint ტიპის რიცხვის მნიშვნელობა ძალიან დიდია და არ შეესაბამება ჩვეულებრივ რიცხვის ტიპს, დამატებითი ბიტები გაუქმდება, ამიტომ ფრთხილად უნდა იყოთ ასეთ კონვერტაციებთან დაკავშირებით.

უნარული + ოპერატორი BigInt ტიპის რიცხვებთან არ გამოიყენება.

როგორც ვიცით, +value უნარული ოპერატორი არის ცნობილი გზა ნებისმიერი value მნიშვნელობის რიცხვად გადაქცევისთვის.

ეს ოპერატორი არ არის მხარდაჭერილი BigInt ტიპის რიცხვებთან მუშაობისას.

```
let bigint = 1n;

alert(+bigint); // სინტაქსური შეცდომა: განუსაზღვრელი იდენტიფიკატორი
```

შედარების ოპერაცია

შედარების ოპერაციები, როგორიცაა < და >, bigint ტიპის და ჩვეულებრივი ტიპის რიცხვებთან მუშაობს ჩვეულებრივად:

```
alert( 2n > 1n ); // true

alert( 2n > 1 ); // true
```

გთხოვთ მიაქციოთ ყურადღება იმას, რომ ჩვეულებრივი და bigint ტიპის რიცხვები სხვადასხვა ტიპს მიეკუთვნება. ისინი ტოლი შეიძლება იყოს მხოლოდ არამკაცრი == შედარების დროს:

```
alert( 1 == 1n ); // true

alert( 1 === 1n ); // false
```

ლოგიკური ოპერაციები

if ან ნებისმიერ სხვა ლოგიკურ ოპერატორში bigint ტიპის რიცხვები ისევე იქცევიან, როგორც ჩვეულებრივი ტიპის რიცხვები. მაგალითად, if bigint 0n შედეგად აბრუნებს false მნიშვნელობას, ხოლო ნებისმიერი სხვა - true-ს:

```
if (0n) {
  // არასდროს არ შესრულდება
}
```

ლოგიკური ოპერატორები ||, && და სხვა მუშაობენ bigint ტიპის რიცხვებთან, როგორც ჩვეულებრივი ტიპის რიცხვებთან:

```
alert( 1n || 2 ); // 1

alert( 0n || 2 ); // 2
```

სტრიქონები

JavaScript-ში ნებისმიერი ტექსტური მონაცემი არის სტრიქონი. არ არსებობს ცალკე სიმბოლოს ტიპი, რომელიც არაერთ სხვა ენაში გამოიყენება.

სტრიქონების შიდა ფორმატი, გვერდის კოდირების მიუხედავად, ყოველთვის არის UTF-16.

ბრჭყალები

JavaScript-ში სხვადასხვა ტიპის ბრჭყალი გამოიყენება.

სტრიქონი შეიძლება შეიქმნას ერთმაგი, ორმაგი ან საწინააღმდეგო ერთმაგი ბრჭყალი.

```
let single = 'ერთმაგი ბრჭყალი';
let double = "ორმაგი ბრჭყალი";

let backticks = ‘საწინააღმდეგო ბრჭყალი’;
```

ერთმაგი და ორმაგი ბრჭყალები არსებითად ერთნაირად მუშაობს და თუ იყენებთ საწინააღმდეგო ბრჭყალებს, მაშინ ჩვენ შეგვიძლია ნებისმიერი გამონათქვამები ჩავსვათ ${…}-ში:

```
function sum(a, b) {
  return a + b;
}

alert(‘1 + 2 = ${sum(1, 2)}.’); // 1 + 2 = 3.
```

საწინააღმდეგო ბრჭყალების კიდევ ერთი უპირატესობა ის არის, რომ მათ შორის მოთავსებული სტრიქონი შეიძლება ერთზე მეტ სტრიქონში იყოს განთავსებული, მაგალითად:

```
let guestList = ‘Guests:
 * გიორგი
 * ნიკოლოზი
 * მარიამი
‘;

alert(guestList); // მრავალ სტრიქონიანი სტუმრების სია
```

თუ ამ მაგალითში გამოვიყენებთ ერთმაგ ან ორმაგ ბრჭყალებს, მაშინ შეცდომა წარმოიშვება.

ერთმაგი და ორმაგი ბრჭყალები ენაში დიდ ხანია რაც გამოიყენება: მაშინ არ იყო გათვალისწინებული მრავალ სტრიქონიანობის საჭიროება. რაც შეეხება საწინააღმდეგო ბრჭყალებს, ის ენაში გაცილებით გვიან გამოჩნდა და შესაბამისად, უფრო მოქნილია.

სპეციალური სიმბოლოები

მრავალ სტრიქონიანობა ასევე შეიძლება შეიქმნას ერთმაგი და ორმაგი ბრჭყალების გამოყენებითაც ეგრეთ წოდებული „ახალ სტრიქონზე გადასვლის სიმბოლოს“ გამოყენებით, რომელიც ასე იწერება \n:

```
let guestList = Guests:\n * გიორგი\n * ნიკოლოზი\n * მარიამი";

alert(guestList); // მრავალ სტრიქონიანი სტუმრების სია
```

კერძოდ, ეს ორი სტრიქონი ეკვივალენტურია, უბრალოდ სხვადასხვანაირადაა ჩაწერილი:

```
// ახალი სტრიქონი დაემატა ახალ სტრიქონზე გადასვლის სიმბოლოს საშუალებით
let str1 = "Hello\nWorld";

// მრავალ სტრიქონიანობა შექმნილია საწინააღმდეგო ბრჭყალების გამოყენებით
let str2 = ‘Hello
World’;

alert(str1 == str2); // true
```

არსებობს სხვა სპეციალური სიმბოლოებიც, რომლებიც შედარებით იშვიათად გამოიყენება. ეს სიმბოლოებია:

სიმბოლო	**აღწერა**
\n	ახალ სტრიქონზე გადასვლა
\r	Windows ტექსტურ ფაილებში \r\n სიმბოლოების კომბინაცია ახალ სტრიქონზე გადასასვლელად გამოიყენება, ხოლო სხვა ოპერაციულ სისტემებში მხოლოდ \n. Windows-ის ქვეშ მომუშავე პროგრამული უზრუნველყოფები როგორც წესი, უბრალოდ \n-ს იყენებს.
\', \"	ბრჭყალები
\\	საწინააღმდეგოდ დახრილი ხაზი
\t	ტაბულაციის ნიშანი

\xXX	სიმბოლოს თექვსმეტობითი უნიკოდის კოდი XX, მაგალითად, '\x7A' იგივეა რაც 'z'.
\uXXXX	UTF-16 კოდირებული სიმბოლოს XXXX თექვსმეტობითი კოდი, მაგალითად, \u00A9 არის საავტორო უფლებების სიმბოლოს კოდი უნიკოდში - ©. კოდი უნდა იყოს ზუსტად ოთხნიშნა თექვსმეტობითი რიცხვი.

ყველა სპეციალური სიმბოლო საწინააღმდეგოდ დახრილი ხაზით იწყება, \ - ე.წ. „ეკრანირების სიმბოლო“.

ის ასევე გამოიყენება, როდესაც სტრიქონის ბრჭყალებში ჩასმა გსურთ.

მაგალითად:

```
alert( 'I \'m the Nikoloz!' ); // I'm the Nikoloz!
```

ამ მაგალითში, სტრიქონში მეორე ბრჭყალის წინ უნდა დაემატოს საწინააღმდეგოდ დახრილი ხაზი - \' - წინააღმდეგ შემთხვევაში, ეს სტრიქონის დასასრულზე მიუთითებს.

რა თქმა უნდა, „ეკრანირების სიმბოლო“ გამოიყენება იმ შემთხვევაში, როდესაც სტრიქონი იმავე ტიპის ბრჭყალებშია მოთავსებული, რომლის გამოტანაც გვსურს. ჩვენ შეგვიძლია სტრიქონისთვის ორმაგი ან საწინააღმდეგო ბრჭყალების გამოყენებით, ამ ამოცანის უფრო ელეგანტური გადაწყვეტა:

```
alert( ‘I'm the Nikoloz!’ ); // I'm the Nikoloz!
```

ან

```
alert(I'm the Nikoloz!" ); // I'm the Nikoloz!
```

სტრიქონის სიგრძე

length თვისება სტრიქონის სიგრძეს გვაძლევს:

```
alert( 'My\n'.length ); // 3
```

გაითვალისწინეთ, რომ \n არის ერთი სპეციალური სიმბოლო, ამიტომ აქ ყველაფერი სწორია: სტრიქონის სიგრძეა 3.

დაპროგრამების ბევრ ენაში str.length-ის ნაცვლად წერენ str.length(). JavaScript-ში ეს არ მუშაობს, რადგან str.length არის რიცხვითი თვისება და არა ფუნქცია, რის გამოც თქვენ არ გჭირდებათ ფრჩხილების გამოყენება.

სიმბოლოებზე მიმართვა

თუ თქვენ გსურთ მიმართოთ სტრიქონის რომელიმე სიმბოლოს, მაშინ ამისათვის გამოიყენება კვადრატული ფრჩხილები, რომელშიც იწერება პოზიციის ნომერი, რომელიც უკავია ამ სიმბოლოს მოცემულ სტრიქონში. პოზიცის ათვლა ნულით იწყება. მაგალითად, თუ სიმბოლო სტრიქონში იკავებს pos პოზიციას, მაშინ ამ სიმბოლოზე მიმართვა კვადრატული ფრჩხილების გამოყენებით ასე ჩაიწერება [pos]. სიმბოლოზე მიმართვისათვის ასევე შეგიძლიათ გამოიყენოთ charAt მეთოდი: str.charAt(pos). მაგალითად:

```
let str = 'Hello';

// პირველი სიმბოლოს მიღება
alert( str[0] ); // H
alert( str.charAt(0) ); // H

// ბოლო სიმბოლოს მიღება
```

```
alert( str[str.length - 1] ); // o
```

კვადრატული ფრჩხილებისა და charAt მეთოდის გამოყენებას შორის არის მხოლოდ ერთი სხვაობაა, თუ სტრიქონში მითითებული პოზიციის სიმბოლო არ არსებობს, მაშინ [ ] შედეგად დააბრუნებს undefined, ხოლო charAt მეთოდი კი - ცარიელ სტრიქონს.

```
let str = ‘Hello’;

alert( str[1000] ); // undefined
alert( str.charAt(1000) ); // '' (ცარიელი სტრიქონი)
```

ასევე, შესაძლებელია სტრიქონი „დავშალოთ“ სიმბოლოებად for..of ციკლის გამოყენებით:

```
for (let char of "Hello") {
  alert(char); // H,e,l,l,o (char - პირველად არის „H“, შემდეგ „e“,
შემდეგ „l“ და ა.შ.)
}
```

JavaScript-ში სტრიქონის შიგთავსის შეცვლა შეუძლებელია. თქვენ არ შეგიძლიათ აიღოთ სიმბოლო სტრიქონის შუაში და შეცვალოთ იგი. როგორც კი სტრიქონი შეიქმნება, ბოლომდე ასე დარჩება.

ვცადოთ ეს და ვნახოთ, რა მოხდება:

```
let str = 'Hi';

str[0] = 'h'; // შეცდომაა
alert( str[0] ); // არ მუშაობს
```

თუ ძველი სტრიქონის შეცვლა გსურთ, შეგიძლიათ შექმნათ ახალი სტრიქონი იმავე სახელით და ჩაწეროთ იგი ძველის ნაცვლად.

მაგალითად:

```
let str = 'Hi';

str = 'h' + str[1]; // ვცვლით სტრიქონს
alert( str ); // hi
```

toLowerCase() და toUpperCase() მეთოდები სიმბოლოების რეგისტრს ცვლის:

```
alert( 'Interface'.toUpperCase() ); // INTERFACE
alert( 'Interface'.toLowerCase() ); // interface
```

თუ რომელიმე კონკრეტული სიმბოლოს რეგისტრის შეცვლა გსურთ, ეს შეიძლება შემდეგნაირად განახორციელოთ:

```
alert( 'Interface'[0].toLowerCase() ); // 'i'
```

ქვესტრიქონის ძებნა

სტრიქონში ქვესტრიქონის ძებნის რამდენიმე მეთოდი არსებობს.

str.indexOf

პირველი მეთოდი არის str.indexOf(substr, pos). ის ეძებს substr ქვესტრიქონს str სტრიქონში, pos პოზიციიდან დაწყებული, და აბრუნებს პოზიციას, საიდანაც მოცემული ქვესტრიქონი იწყება, ან -1, თუ ასეთ ქვესტრიქონს არ შეიცავს.

მაგალითად:

```
let str = 'Widget with id';

alert( str.indexOf('Widget') ); // 0, ვინაიდან 'Widget' არის დასაწყისში
alert( str.indexOf('widget') ); // -1, არ ემთხვევა, ვინაიდან ძებნა რეგისტრზე არის დამოკიდებული

alert(str.indexOf("id") ); // 1, ქვესტრიქონი "id" ნაპოვნია 1 პოზიციიდან (..idget with id)
```

არააუცილებელი მეორე არგუმენტი საშუალებას იძლევა ძებნა კონკრეტული პოზიციიდან დაიწყოთ.

მაგალითად, „id“-ის პირველ შემთხვევაში 1 პოზიციიდან იწყება. შემდეგის საპოვნელად ძებნა მე-2 პოზიციიდან დაიწყეთ:

```
let str = 'Widget with id';


alert( str.indexOf('id', 2) ); // 12
```

ასევე არსებობს მსგავსი მეთოდი str.lastIndexOf(substr, position), რომელიც ეძებს ქვესტრიქონს სტრიქონის ბოლოდან მის დასაწყისამდე. ეს მეთოდი გამოიყენება მაშინ, როდესაც გჭირდებათ ბოლო ქვესტრიქონის მიღება: სტრიქონის ბოლოდან ან გარკვეულ პოზიციამდე (მათ შორის) დასაწყისამდე.

```
let text = "Hello planet earth, you are a great planet.";


let result = text.lastIndexOf("planet");
alert (result);  // 36
```

includes, startsWith, endsWith

str.includes(substr, pos) უფრო თანამედროვე მეთოდი აბრუნებს true-ს, თუ str სტრიქონი substr ქვესტრიქონს შეიცავს და თუ არა, მაშინ false-ს.

ეს არის სწორი არჩევანი, თუ ჩვენ უნდა შევამოწმოთ სტრიქონი შეიცავს თუ არა ქვესტრიქონს, მაგრამ ჩვენ არ გვჭირდება პოზიცია:

```
let text = "Hello world, welcome to the universe.";
let result = text.includes("world");
alert (result);  // true
```

არასავალდებულო მეორე არგუმენტი str.includes-ს საშუალებას აძლევს ძებნა დაიწყოს კონკრეტული პოზიციიდან:

```
let text = "Hello world, welcome to the universe.";
let result = text.includes("world", 12);
alert (result);  // false
```

str.startsWith და str.endsWith მეთოდები ამოწმებს, შესაბამისად, იწყება თუ მთავრდება სტრიქონი კონკრეტული სტრიქონით:

```
let text = "Hello world, welcome to the universe.";
alert (text.startsWith("Hello"));// true
alert (text.startsWith("Hello", 7));// false
```

```
let text = "Hello world";
alert (text.endsWith("world"));  // true
alert (text.endsWith("World"));  // false
```

ქვესტრიქონის მიღება

JavaScript-ში ქვესტრიქონის მისაღებად სამი მეთოდია: substring, substr და slice.

str.slice(start [, end])

ეს მეთოდი აბრუნებს სტრიქონის ნაწილს start-დან end-მდე (არ მოიცავს). თუ არ არის end არგუმენტი, slice აბრუნებს სიმბოლოებს სტრიქონის ბოლომდე. თქვენ ასევე შეგიძლიათ start/end-სთვის დააყენოთ უარყოფითი მნიშვნელობები. ეს ნიშნავს, რომ პოზიცია სტრიქონში ქვესტრიქონი განისაზღვრება სტრიქონის ბოლოდან:

მაგალითად:

```
let text = "Hello world!";
alert (text.slice(3, 8));  //  lo wo, სიმბოლოები 3-დან 8-მდე
alert (text.slice(0, 1));  //  H, 0-დან 1-მდე (არ მოიცავს), მხოლოდ
ერთი სიმბოლო
alert (text.slice(3));  //  lo world!, სიმბოლოები 3-დან ბოლომდე
alert (text.slice(-6, -1));  //  world, სიმბოლოები ბოლო 6-დან 5
სიმბოლო
alert (text.slice(0));  //  Hello world!, მთლიანად სტრიქონი
alert (text.slice(-1));  //   !, სტრიქონის ბოლო სიმბოლო
```

str.substring(start [, end])

ეს მეთოდიც აბრუნებს სტრიქონის ნაწილს start-დან end-მდე (არ მოიცავს).

ეს მეთოდი იგივეა, რაც slice, მაგრამ შეგიძლიათ დააყენოთ start უფრო დიდი ვიდრე end. თუ start აღემატება end-ს, მაშინ

ქვესტრიქონის მეთოდი იმუშავებს ისე, თითქოს არგუმენტებმა ადგილები შეცვალეს.

```
let text = "Hello world!";

// substring მეთოდისათვის ეს ორი მაგალიტი ერტიდაიგივეა
alert (text. substring (3, 8)); // lo wo
alert (text. substring (8, 3)); // lo wo, იგივეა

// slice მეთოდისათვის არა
alert (text.slice(3, 8)); // lo wo, იგივეა
alert (text.slice(8, 1)); // " " ცარიელი სტრიქონი
```

substring მეთოდი, slice მეთოდისგან განსხვავებით, მხარს არ უჭერს უარყოფით მნიშვნელობებს, ისინი ინტერპრეტირებული იქნება როგორც 0.

str.substr(start [, length])

მოცემული მეთოდი start-დან დაწყებული length სიგრძის სტრიქონის ნაწილს აბრუნებს. წინა მეთოდებისგან განსხვავებით, ეს საშუალებას გაძლევთ ბოლო პოზიციის ნაცვლად მიუთითოთ სიგრძე. პირველი არგუმენტის მნიშვნელობა შეიძლება იყოს უარყოფითი, ამ შემთხვევაში, პოზიცია განისაზღვრება ბოლოდან:

```
let text = "Hello world!";
alert (text. substr(1, 4)); // ello, 1-დან 4 სიმბოლო
alert (text. substr(3)); // lo world!
alert (text. substr(0, 1)); // H, მხოლოდ ერთი სიმბოლო
alert (text. substr(-6, 5)); // world
```

სტრიქონების შედარება

როგორც ვიცით, სტრიქონების შედარების დროს ისინი ანბანის მიხედვით სიმბოლოებით დარდება ერთმანეთს.

თუმცა, არსებობს რამდენიმე ნიუანსი.

პატარა ასოები უფრო მეტია, ვიდრე ასომთავრული:

```
alert( 'a' > 'Z' ); // true
```

იმის გასაგებად, თუ რა ხდება, მოდით გადავხედოთ JavaScript-ში სიმბოლოების შიდა წარმოდგენას.

ყველა სიმბოლო დაშიფრულია UTF-16-ში. ამრიგად, ნებისმიერ სიმბოლოს აქვს შესაბამისი კოდი. არსებობს სპეციალური მეთოდები, რომლებიც საშუალებას გაძლევთ სიმბოლო მიიღოთ მისი კოდით და პირიქით.

str.codePointAt(pos) - შედეგად აბრუნებს pos პოზიციაში მდგომი სიმბოლოს კოდს (ზედა და ქვედა რეგისტრის ერთი და იგივე სიმბოლოს სხვადასხვა კოდი აქვს):

```
alert( "z".codePointAt(0) ); // 122
alert( "Z".codePointAt(0) ); // 90
```

String.fromCodePoint(code) - შედეგად აბრუნებს code-ის შესაბამის სიმბოლოს:

```
alert( String.fromCodePoint(90) ); // Z
```

ასევე კოდებით უნიკოდის სიმბოლოების დამატებაა შესაძლებელი, \u-ს გამოყენებით სიმბოლოს თექვსმეტობით კოდებში:

```
// 90 - 5a ათვლის თექვსმეტობით სისტემაში
```

```
alert( '\u005a' ); // Z
```

ucFirst(str) - შედეგად აბრუნებს str სტრიქონს, რომლის პირველი სიმბოლო ასომთავრული იქნება.

```
ucFirst("georgia") == "Georgia";
```

მასივი

ობიექტები საშუალებას გვაძლევს შევინახოთ მონაცემები სტრიქონული საკვანძო სიტყვებით. ეს ძალიან კარგია. მაგრამ საკმაოდ ხშირად ჩვენ გვჭირდება მონაცემთა მოწესრიგებული ნაკრები, რომელსაც აქვს 1-ლი, მე-2, მე-3 ელემენტები და ა.შ. მაგალითად, ის დაგვჭირდება რაღაცის სიის შესანახად: მომხმარებლები, პროდუქტები, HTML ელემენტები და ა.შ.

ამ შემთხვევაში, ობიექტის გამოყენება მოუხერხებელია, რადგან ის ელემენტების რიგის გაკონტროლების საშუალებას არ იძლევა. ჩვენ არ შეგვიძლია ჩავსვათ ახალი თვისება არსებულებს შორის. ობიექტები უბრალოდ არ არის შექმნილი ამ მიზნით.

მოწესრიგებული მონაცემების შესანახად არსებობს მონაცემთა სპეციალური სტრუქტურა, რომელსაც მასივი (Array) ეწოდება.

მასივი არის მონაცემთა მოწესრიგებული ნაკრები. იგი შეიძლება წარმოვადგინოთ ერთსვეტიანი ცხრილის სახით, რომელიც შეიცავს გარკვეული რაოდენობის სტრიქონებს. ასეთი ცხრილის უჯრედში შეიძლება იყოს ნებისმიერი ტიპის მონაცემი, მათ შორის მასივებიც. ამ ბოლო შემთხვევაში გვაქვს მრავალგანზომილებიანი მასივი. მასივში ელემენტების რაოდენობას ეწოდება მასივის სიგრძე. მასივის ელემენტებს შეიძლება პროგრამაში მივმართოთ მისი რიგითი ნომრით (ინდექსით). მასივის ელემენტების ნუმერაცია ნულით იწყება, პირველი ელემენტის ინდექსია 0, ხოლო ბოლოსი – ერთი ერთეულით ნაკლები, ვიდრე მასივის სიგრძე.

თუ გამოყენებულ მონაცემებს შორის არის ჯგუფი, რომლის დამუშავება უნდა მოხდეს ერთი და იგივე სახით, მაშინ უმჯობესია მოხდეს მათი მასივის სახით ორგანიზება.

მასივის გამოცხადება

ცარიელი მასივის შექმნისათვის ორი ვარიანტი გამოიყენება:

```
let arr = new Array();
let arr = [];
```

პრაქტიკულად, უმეტეს შემთხვევაში მეორე ვარიანტს იყენებენ. ფრჩხილებში შეიძლება ელემენტების საწყისი მნიშვნელობები მივუთითოთ:

```
let fruits = ["ვაშლი", "ფორთოხალი", "ქლიავი"];
```

მასივის ელემენტების ნუმერაცია ნულიდან იწყება.

ჩვენ ელემენტის მიღება კვადრატულ ფრჩხილებში მისი ნომრის მითითებით შეგვიძლია.

```
let fruits = ["ვაშლი", "ფორთოხალი", "ქლიავი"];

alert( fruits[0] ); // ვაშლი
alert( fruits[1] ); // ფორთოხალი
alert( fruits[2] ); // ქლიავი
```

ჩვენ შეგვიძლია შევცვალოთ ელემენტი:

```
fruits[2] = 'მსხალი'; // ახლა გვაქვს ["ვაშლი", "ფორთოხალი",
"მსხალი"]
```

ან არსებულ მასივს დავუმატოთ ახალი ელემენტი

```
fruits[3] = 'ლიმონი'; // ახლა გვაქვს ["ვაშლი", "ფორთოხალი",
"მსხალი", "ლიმონი"]
```

მასივის ელემენტების საერთო რაოდენობა მისი length თვისებით განისაზღვრება:

```
let fruits = ["ვაშლი", "ფორთოხალი", "ქლიავი"];

alert( fruits.length ); // 3
```

მასივის სრულად გამოტანა alert-ის საშუალებითაა შესაძლებელი.

```
let fruits = ["ვაშლი", "ფორთოხალი", "ქლიავი"];

alert( fruits ); // ვაშლი, ფორთოხალი, ქლიავი
```

მასივში შეიძლება ინახებოდეს ნებისმიერი ტიპის ელემენტი.

მაგალითად:

```
// სხვადასხვა ტიპის მნიშვნელობები
let arr = [ 'ვაშლი', { name: 'ნიკა' }, true, function() { alert('Hello'); }
];

// მივიღოთ ელემენტი ინდექსით 1 (ობიექტი) და შემდეგ
ვაჩვენოთ მისი თვისება
alert( arr[1].name ); // ნიკა
```

```
// მივიღოთ ელემენტი ინდექსით 3 (ფუნქცია) და
შევასრულოთ იგი
arr[3](); // Hello
```

მასივის ელემენტების სია, ისევე როგორც ობიექტის თვისებების სია, შეიძლება მძიმით დასრულდეს:

```
let fruits = [
  "ვაშლი",
  "ფორთოხალი",
  "ქლიავი",
];
```

ბოლო მძიმეები ამარტივებს ელემენტების დამატების/წაშლის პროცესს, რადგან ყველა სტრიქონი იდენტური ხდება.

pop/push, shift/unshift მეთოდები

რიგი არის მასივის გამოყენების ერთ-ერთი ყველაზე გავრცელებული ვარიანტი. კომპიუტერულ მეცნიერებაში ეს არის ელემენტების მოწესრიგებული კოლექცია, რომელიც მხარს უჭერს ორ სახის ოპერაციას:

- push ამატებს ელემენტს ბოლოში;
- shift შლის პირველ ელემენტს, შეცვლის რიგს ისე, რომ მეორე ელემენტი გახდეს პირველი.

მასივები მხარს უჭერენ ორივე ოპერაციას.

პრაქტიკაში, ამის საჭიროება ხშირად წარმოიშვება. მაგალითად, ეკრანზე გვინდა გამოვიტანოთ რაღაც შეტყობინებები გარკვეული რიგით.

არსებობს მასივების გამოყენების კიდევ ერთი ვარიანტი, მონაცემთა სტრუქტურა, რომელსაც ეწოდება სტეკი[6].

იგი მხარს უჭერს ორი სახის ოპერაციას:

- push ამატებს ელემენტს ბოლოში;
- pop შლის ბოლო ელემენტს.

ასე რომ, ახალი ელემენტი ყოველთვის ემატება ბოლოში ან იშლება ბოლო ელემენტი.

სტეკის მაგალითი, როგორც წესი, არის ბანქოს დასტა: ახალი ბანქო ემატება ზემოდან და ასევე იღებენ ზემოდან:

მასივებს JavaScript-ში შეუძლია იმუშაოს როგორც რიგის, ასევე სტეკის სახით. ჩვენ შეგვიძლია დავამატოთ/წავშალოთ ელემენტები მასივის დასაწყისშიც და ბოლოშიც.

კომპიუტერულ მეცნიერებაში მონაცემთა სტრუქტურას, რომელიც ამის შესაძლებლობას იძლევა, ორმხრივი რიგი ეწოდება.

მასივის ბოლოსთან მომუშავე მეთოდები

pop

შლის მასივის ბოლო ელემენტს და ისე აბრუნებს მასივს:

```
let fruits = ["ვაშლი", "ფორთოხალი", "მსხალი"];

alert( fruits.pop() ); // შლის ბოლო ელემენტს - მსხალი
```

[6] სტეკი (ინგლ. stack) - აბსტრაქტული მონაცემთა ტიპი, რომელიც წარმოადგენს LIFO (ინგლ. last in - first out, „ბოლოს მოვიდა - პირველი გავიდა“) პრინციპით ორგანიზებულ ელემენტთა სიას). 1946 წელს ალან ტიურინგმა შემოიტანა სტეკის ცნება, ხოლო 1957 წელს გერმანელებმა კლაუს სამელსონმა და ფრიდრიხ ლ. ბაუერმა დააპატენტეს ტურინგის იდეა.

```
alert( fruits ); // ვაშლი, ფორთოხალი
```

push

ამატებს მასივის ბოლოში ელემენტს:

```
let fruits = ["ვაშლი", "ფორთოხალი"];

fruits.push("მსხალი");

alert( fruits ); // ვაშლი, ფორთოხალი, მსხალი
```

მასივის დასაწყისთან მომუშავე მეთოდები

shift

შლის მასივის პირველ ელემენტს და ისე აბრუნებს მასივს:

```
let fruits = ["ვაშლი", "ფორთოხალი", "მსხალი"];

alert( fruits.shift() ); // შლის მასივის პირველ ელემენტს - ვაშლი

alert( fruits ); // ფორთოხალი, მსხალი
```

unshift

ამატებს მასივის დასაწყისში ელემენტს:

```
let fruits = ["ფორთოხალი", "მსხალი"];

fruits.unshift('ვაშლი');

alert( fruits ); // ვაშლი, ფორთოხალი, მსხალი
```

push და unshift მეთოდებს ერთდროულად რამდენიმე ელემენტის დამატება შეუძლიათ:

```
let fruits = ["ვაშლი"];

fruits.push("ფორთოხალი", "მსხალი");
fruits.unshift("ანანასი", "ლიმონი");

alert( fruits ); //  ანანასი, ლიმონი, ვაშლი, ფორთოხალი,
მსხალი
```

მასივის მოწყობა

მასივი არის ობიექტების სპეციალური ქვეჯგუფი. კვადრატული ფრჩხილები, რომლებიც გამოიყენება arr[0] თვისებაზე წვდომისთვის, ძირითადად არის ჩვეულებრივ გასაღებზე წვდომის სინტაქსი, როგორიცაა obj[key], სადაც obj-ის როლში გვაქვს arr, ხოლო გასაღების სახით რიცხვითი ინდექსი.

მასივები აფართოებენ ობიექტებს, რადგანაც უზრუნველყოფენ მონაცემთა მოწესრიგებულ კოლექციებთან მუშაობის სპეციალურ მეთოდებს, ასევე length თვისებით. მაგრამ ობიექტი მაინც საფუძველია.

გაითვალისწინეთ, რომ JavaScript-ში 8 ძირითადი მონაცემთა ტიპი არსებობს. მასივი არის ობიექტი და შესაბამისად იქცევა როგორც ობიექტი.

მაგალითად, კოპირებულია ბმულით:

```
let fruits = ["ბანანი"]
```

```
let arr = fruits; // კოპირებულია ბმულით (ორი ცვლადი ერთი და იმავე მასივს მიმართავენ)

alert( arr === fruits ); // true

arr.push("მსხალი"); // მასივი იცვლება ბმულით

alert( fruits ); // ბანანი, მსხალი - ახლა ორი ელემენტია
```

მაგრამ ის, რაც მასივებს განსაკუთრებულს ხდის, არის მათი შიდა წარმოდგენა. JavaScript-ის ძრავა ცდილობს შეინახოს მასივის ელემენტები მეხსიერების მომიჯნავე არეში, ერთმანეთის მიყოლებით.

მასივი უნდა ჩაითვალოს სპეციალურ სტრუქტურად, რომელიც საშუალებას გაძლევთ იმუშაოთ მოწესრიგებულ მონაცემებთან. ამისათვის, მასივები გთავაზობთ სპეციალურ მეთოდებს. მასივები საგულდაგულოდ არის მორგებული JavaScript-ის ძრავაში, რათა იმუშაონ ერთნაირი ტიპის მოწესრიგებულ მონაცემებთან, ამიტომ ასეთ შემთხვევებში შეგიძლიათ გამოიყენოთ ისინი. თუ თქვენ გჭირდებათ ნებისმიერი ტიპის გასაღები, მაგრამ შეიძლება უკეთესი იყოს ჩვეულებრივი ობიექტის {} გამოყენება.

ეფექტურობა

Push/pop მეთოდები სწრაფია, ხოლო shift/unshift მეთოდები შედარებით ნელი.

რატომ არის უფრო სწრაფი მუშაობა მასივის ბოლოსთან, ვიდრე მის დასაწყისთან? ვნახოთ რა ხდება შესრულების დროს:

```
fruits.shift(); // ამოიღეთ პირველი ელემენტი დასაწყისიდან
```

საკმარისი არ არის მხოლოდ 0 ნომრით ელემენტის წაშლა. თქვენ ასევე უნდა ხელახლა დანომროთ დანარჩენი ელემენტები.

shift ოპერაციამ უნდა შეასრულოს 3 რამ:

1. წაშალეთ ელემენტი ინდექსით 0;
2. გადაიტანეთ ყველა ელემენტი მარცხნივ, ხელახლა დანომრეთ ისინი, შეცვალეთ 1 0-ით, 2 - 1-ით და ა.შ;
3. length თვისების განახლება.

რაც უფრო მეტ ელემენტს შეიცავს მასივი, მით მეტი დრო დასჭირდება მათი ნომრების შეცვლას, მით მეტი მეხსიერების ოპერაციაა ჩასატარებელი.

იგივე ხდება unshift-თან დაკავშირებით: იმისათვის, რომ ელემენტი დავამატოთ მასივის დასაწყისში, ჯერ არსებული ელემენტები მარჯვნივ უნდა გადავიტანოთ, მათი ინდექსების გაზრდით.

რაც შეეხება push/pop მათ არაფრის გადატანა არ სჭირდებათ. მასივის ბოლოს ელემენტის მოსაშორებლად, pop მეთოდი ასუფთავებს ინდექსს და ამცირებს length-ის მნიშვნელობას.

pop ოპერაციის დროს მოქმედება:

```
fruits.pop(); // ვშლით მასივის ბოლო ელემენტს
```

pop მეთოდს არ სჭირდება გადატანა, რადგან დანარჩენი ელემენტები იმავე ინდექსებით რჩებიან. ამიტომ ის ძალიან სწრაფად მუშაობს.

push მეთოდიც ანალოგიურად მუშაობს.

მასივის ელემენტების გამოტანა

მასივის ელემენტების გამოტანის ერთ-ერთი უძველესი გზა ციფრული ინდექსებით for ციკლის გამოყენებაა:

```
let arr = ["ვაშლი", "ფორთოხალი", "მსხალი"];

for (let i = 0; i < arr.length; i++) {
  alert( arr[i] );
}
```

მასივისათვის ასევე სხვა ვარიანტის for..of ციკლის გამოყენებაა შესაძლებელი:

```
let fruits = ["ვაშლი", "ფორთოხალი", "მსხალი"];

// მასივს გაივლის მნიშვნელობების მიხედვით
for (let fruit of fruits) {
  alert( fruit );
}
```

for..of ციკლს არა აქვს წვდომა მასივის მიმდინარე ელემენტის ნომერზე, მხოლოდ მის მნიშვნელობაზე, მაგრამ უმეტეს შემთხვევაში ეს საკმარისია, ამასთან, უფრო მოკლე გზაა.

ტექნიკურად, ვინაიდან მასივი არის ობიექტი, ასევე შეგიძლიათ გამოიყენოთ ციკლის for..in ვარიანტი:

```
let arr = ["ვაშლი", "ფორთოხალი", "მსხალი"];

for (let key in arr) {
  alert( arr[key] ); // ვაშლი, ფორთოხალი, მსხალი
```

```
}
```

სინამდვილეში ეს არ არის კარგი იდეა. ამ მეთოდს გააჩნია ფარული უარყოფითი მხარეები:

1. for..in ციკლი იმეორებს ობიექტის ყველა თვისებას და არა მხოლოდ ციფრულს.

ბრაუზერში და პროგრამირების სხვა გარემოში ასევე არის ეგრეთ წოდებული „ფსევდო-მასივები“ – ობიექტები, რომლებიც მასივს ჰგავს. ანუ, მათ აქვთ length თვისება და ინდექსები, მაგრამ მათ ასევე შეიძლება ჰქონდეთ დამატებითი არარიცხობრივი თვისებები და მეთოდები, რომლებიც ჩვეულებრივ არ გვჭირდება. თუმცა, for..in ციკლი მათაც გამოიტანს. ამიტომ, თუ საქმე გვაქვს მასივის მსგავს ობიექტებთან, ასეთი „დამატებითი“ თვისებები შეიძლება პრობლემა გახდეს;

2. for..in ციკლი ნებისმიერი ობიექტისთვის არის ოპტიმიზებული და არა მხოლოდ მასივებისთვის, შესაბამისად, 10-ჯერ და 100-ჯერ უფრო ნელი. შესრულების სიჩქარის გაზრდამ შეიძლება გამოიწვიოს განსხვავება მხოლოდ მაშინ, როდესაც წარმოიქმნება შეფერხებები. მაგრამ ჩვენ მაინც უნდა წარმოვადგინოთ განსხვავება.

ზოგადად, მასივებისთვის არ უნდა გამოვიყენოთ for..in ციკლი.

length თვისება

length თვისება ავტომატურად განახლდება მასივის შეცვლისას. უფრო ზუსტად რომ ვთქვათ, ეს არ არის მასივის ელემენტების რაოდენობა, არამედ უდიდესი რიცხვითი ინდექსი პლუს ერთი.

მაგალითად, მასივის ერთადერთი ელემენტი, რომელსაც აქვს რიცხვითი დიდი ინდექსი, მასივის დიდ სიგრძეს იძლევა:

```
let fruits = [];
fruits[123] = "ვაშლი";

alert( fruits.length ); // 124
```

გაითვალისწინეთ, რომ ჩვენ ჩვეულებრივ ამ გზით მასივებს არ ვიყენებთ.

length თვისების კიდევ ერთი საინტერესო ფაქტი არის ის, რომ მისი შეცვლა შესაძლებელია.

თუ length თვისების მნიშვნელობას ხელოვნურად გავზრდით, საინტერესო არაფერი მოხდება, მაგრამ თუ შევამცირებთ, მასივი უფრო მოკლე გახდება. ეს პროცესი შეუქცევადია, როგორც ეს ქვემოთ მოყვანილი მაგალითიდან შეგვიძლია დავინახოთ:

```
let arr = [1, 2, 3, 4, 5];

arr.length = 2; // ვამცირებთ ორ ელემენტამდე
alert( arr ); // [1, 2]

arr.length = 5; // დავაბრუნოთ length თვისება საწყის
მდგომარეობაში
alert( arr[3] ); // undefined: მნიშვნელობა აღარ აღდგა
```

ამგვარად, მასივის გასუფთავების ყველაზე მარტივი საშუალებაა arr.length = 0;.

new Array()

მასივის შექმნის კიდევ ერთი ვარიანტი არსებობს:

```
let arr = new Array("ვაშლი", "ფორთოხალი", "მსხალი");
```

იგი იშვიათად გამოიყენება, რადგან კვადრატული ფრჩხილები [] უფრო მოკლეა. გარდა ამისა, მას აქვს ერთი თვისება.

თუ new Array-ის გამოძახება ერთი არგუმენტით ხდება, რომელიც რიცხვია, ის ქმნის მასივს ელემენტების გარეშე, მაგრამ მოცემული სიგრძით.

ვნახოთ, რა ხდება ამ შემთხვევაში:

```
let arr = new Array(2); //

alert( arr[0] ); // undefined! არ არის ელემენტები.

alert( arr.length ); // length 2
```

როგორც ვხედავთ, ზემოთ მოცემულ კოდში, new Array(number)-ის ყველა ელემენტი უდრის undefined-ს.

ასეთი მოულოდნელი სიტუაციების თავიდან ასაცილებლად, ჩვენ ჩვეულებრივ ვიყენებთ კვადრატულ ფრჩხილებს, თუ, რა თქმა უნდა, დანამდვილებით არ ვიცით, რომ რაიმე მიზეზით ზუსტად Array არის საჭირო.

toString

მასივები toString მეთოდის რეალიზაციას თავისებურად ახდენენ, იგი აბრუნებს მძიმით გამოყოფილ ელემენტების სიას.

მაგალითად:

```
let arr = [1, 2, 3];

alert( arr ); // 1,2,3
alert( String(arr) === '1,2,3' ); // true
```

მრავალგანზომილებიანი მასივი

ზემოთ განხილული მასივები იყო ერთგანზომილებიანი. თუ რომელიმე ერთგანზომილებიანი მასივის ელემენტად ისევ მასივს შევქმნით, მაშინ მივიღებთ ორგანზომილებიან მასივს.

ეს შეიძლება გამოყენებულ იქნას მრავალგანზომილებიანი მასივის შესაქმნელად, მაგალითად, მატრიცის შესანახად:

```
let matrix = [
  [1, 2, 3],
  [4, 5, 6],
  [7, 8, 9]
];

alert( matrix[0][0] ); // 1,
alert( matrix[0][1] ); // 2,
alert( matrix[0][2] ); // 3,
alert( matrix[1][0] ); // 4,
alert( matrix[1][1] ); // 5,
alert( matrix[1][2] ); // 6,
alert( matrix[2][0] ); // 7,
alert( matrix[2][1] ); // 8,
alert( matrix[2][2] ); // 9.
```

ზემოთ მოყვანილი მაგალითის მსგავსად შეიძლება მრავალგანზომილებიანი მასივების შექმნა.

მასივის კოპირება

ზოგჯერ საჭიროა მასივის ასლის შექმნა, რათა მოხდეს საწყისი მნიშვნელობების შენახვა და მოხდეს ამ ელემენტების შემდგომი მოდიფიცირებისაგან დაცვა. ამასთან, მასივის კოპირებისათვის საკმარისი არ არის მხოლოდ სხვა ცვლადისათვის მისი მინიჭება. ახალი ცვლადისა და მინიჭების ოპერატორის გამოყენებით ვქმნით მხოლოდ ძველ მასივზე ახალ მიმართვას, და არა ახალ მასივს.

მაგალითი:

```
a = new Array ( 5, 2, 4, 3 );
x = a;              // მიმართვა a მასივზე
a [2] = 25;         // მე-3 ელემენტის მნიშვნელობის შეცვლა
alert (x [2]);      // 25
```

ამ მაგალითში a და x მასივები ერთმანეთს ემთხვევა.

იმისათვის, რომ მასივის კოპირება მოხდეს, ანუ ახალი მასივი შეიქმნას, რომლის ელემენტები საწყისი მასივის ელემენტების ტოლია, საჭიროა ვისარგებლოთ ციკლის ოპერატორით, რომელშიც ახალი მასივის ელემენტებს მიენიჭება საწყისი მასივის ელემენტების მნიშვნელობები.

მაგალითი:

```
a = new Array ( 5, 2, 4, 3 );
x = new Array ( );
for ( i = 0; i < a . length; i ++ )
{
```

```
  x [i] =a [i];
}
```

თვისება prototype

prototype – თვისება საშუალებას იძლევა დავამატოთ ახალი თვისება და მეთოდი ყველა შექმნილ მასივს, თუ არსებული საკმარისი არ იქნება.

მაგალითი:

მასივის ელემენტების ჯამის გამოთვლა. განვსაზღვროთ aSum ( ) ფუნქცია, რომელიც აბრუნებს რიცხვითი მასივის ელემენტების ჯამს.

```
function aSum ( xarray ) {
        let s = 0;
        for ( i = 0; i <= xarray . length - 1; i ++ )
         {
                s = s + xarray [i];
        }
        return s;
   }

myarray = new Array ( 2, 3, 4 );
Array . prototype . Sum = aSum;
myarray . Sum ( myarray );
```

ეს მაგალითი არის უბრალოდ იმის ილუსტრაცია, თუ როგორ შეიძლება prototype თვისების გამოყენება. მასივის ელემენტების ჯამის გამოსათვლელად საკმარისია ჩაიწეროს შემდეგი გამოსახულება:

```
s = aSum ( myarray );
```

ობიექტ Array-ის მეთოდები

ობიექტ **Array**-ის მეთოდები განკუთვნილია მასივის სტრუქტურებში შენახული მონაცემების მართვისათვის.

concat

arr.concat მეთოდი ქმნის ახალ მასივს, რომელშიც ხდება სხვა მასივებიდან მონაცემებისა და დამატებით მნიშვნელობების კოპირება.

მისი სინტაქსია:

```
arr.concat(arg1, arg2...)
```

მას შეიძლება ქონდეს ნებისმიერი რაოდენობის არგუმენტები, რომლებიც შეიძლება იყოს მასივები ან მარტივი მნიშვნელობები.

შედეგად, ჩვენ ვიღებთ ახალ მასივს, რომელიც მოიცავს ელემენტებს arr მასივიდან, ასევე arg1, arg2 და ა.შ.

თუ რომელიმე argN არის მასივი, მაშინ მისი ყველა ელემენტი იქნება კოპირებული. წინააღმდეგ შემთხვევაში, თვით არგუმენტის კოპირება მოხდება. შედეგად მიიღება მასივი. მოცემული მეთოდი საწყის მასივებს არ ცვლის.

მაგალითად:

```
a1 = new Array ( 1, 2, "a" );
a2 = new Array ("b", "c", "d", "e" );
a3 = a1 . concat ( a2, 3, 4);
```

```
alert(a3);    // შედეგი მიიღება მასივი ელემენტებით 1, 2, "a", "b", "c", "d", "e", 3, 4
```

ჩვეულებრივ, ის უბრალოდ მასივის ელემენტების კოპირებას ახდენს. სხვა ობიექტები, თუნდაც ისინი მასივებივით გამოიყურებოდეს, შემდეგნაირად ემატება:

```
let arr = [1, 2];

let arrayLike = {
  0: "something",
  length: 1
};

alert( arr.concat(arrayLike) ); // 1,2,[object Object]
```

თუ ობიექტს აქვს სპეციალური თვისება Symbol.isConcatSpreadable, მაშინ ის concat-ით საშუალებით განიხილება როგორც მასივი: მის ნაცვლად მისი რიცხვითი თვისებები ემატება.

მათი დამუშავებისთვის ობიექტს უნდა ჰქონდეს რიცხვითი და length თვისებები:

```
let arr = [1, 2];

let arrayLike = {
  0: "something",
  1: "more",
  [Symbol.isConcatSpreadable]: true,
  length: 2
```

```
};

alert( arr.concat(arrayLike) ); // 1,2, something, more
```

split და join

განვიხილოთ რეალური ცხოვრებისეული სიტუაცია. მომხმარებელი წერს განცხადებას შეტყობინებებისთვის და შეაქვს მათი სახელები, ვისაც უნდა გადაუგზავნოს ის, სახელებს ერთმანეთისგან მძიმეებით გამოყოფს: დათო, ლუკა, ნიკოლოზი, ალექსანდრე. ჩვენთვის ბევრად უფრო მოსახერხებელია სახელების მასივთან მუშაობა, ვიდრე ერთ სტრიქონთან. როგორ მივიღოთ იგი?

str.split(delim) მეთოდი სწორედ ამას აკეთებს. ის ყოფს სტრიქონს მასივში მოცემულ delim გამყოფის მიხედვით.

ქვემოთ მოყვანილ მაგალითში, ეს გამყოფი არის სტრიქონი, რომელიც შედგება მძიმისა და ჰარისაგან:

```
let names = 'დათო, ლუკა, ნიკოლოზი, ალექსანდრე';

let arr = names.split(', ');

alert( 'შეტყობინებას მიიღებენ:  ' );
for (let i = 0; i < arr.length; i++) {
alert ( arr[i]); }                    /* შეტყობინებას მიიღებენ:
                                         დათო
                                         ლუკა
                                         ნიკოლოზი
                                         ალექსანდრე  */
```

Split მეთოდს აქვს არააუცილებელი მეორე რიცხვითი არგუმენტი - რომელიც მიუთითებს ელემენტების რაოდენობაზე მასივში. თუ რაოდენობა მითითებულზე მეტია, მაშინ მასივის დარჩენილი ნაწილი გაუქმდება. პრაქტიკაში, ეს იშვიათად გამოიყენება:

```
let arr = 'დათო, ლუკა, ნიკოლოზი, ალექსანდრე'.split(', ', 2);


alert ( arr);   // დათო, ლუკა
```

Split(s)-ის გამოძახება s ცარიელი არგუმენტით სტრიქონს სიმბოლოების მასივად დაყოფს:

```
let str = "დავითი";


alert( str.split('') ); // დ,ა,ვ,ი,თ,ი
```

arr.join(glue) გამოძახება split-ის ზუსტად საპირისპიროს აკეთებს. ის ქმნის arr ელემენტების სტრიქონს მათ შორის glue-ს ჩასმით.

მაგალითი:

```
a = new Array ( 1, 2, "a", "b", "c" )
a . join (", " )                    // შედეგი არის სტრიქონი 1, 2, a, b, c
a = new Array ( 1, 2, "a", "b", "c" )
a . join (" " )                     //შედეგი არის სტრიქონი 1 2 a b c
```

splice

როგორ ამოიღოთ ელემენტი მასივიდან?

ვინაიდან მასივები არის ობიექტები, შეგიძლიათ სცადოთ delete:

```
let arr = ["დათო","ლუკა","ნიკოლოზი","ალექსანდრე"];

delete arr[1]; // წავშალოთ "ლუკა"

alert( arr[1] ); // undefined

// ახლა arr = ["დათო","ნიკოლოზი","ალექსანდრე"];
alert( arr.length ); // 4
```

როგორც ჩანს, ელემენტი ამოღებულია, მაგრამ შემოწმების შემდეგ ირკვევა, რომ მასივი ჯერ კიდევ შედგება 4 ელემენტისაგან arr.length == 4.

ეს ნორმალურია, რადგან ყველა delete obj.key არის მნიშვნელობის წაშლა მოცემული ნომრით. ეს კარგია ობიექტებისთვის, მაგრამ მასივებისთვის, ჩვენ ჩვეულებრივ გვინდა, რომ დარჩენილი ელემენტებმა გადაინაცვლონ და დაიკავონ განთავისუფლებული არე. ჩვენ გვინდა, რომ მასივის ელემენტების რაოდენობა შევამციროთ.

ამიტომ ამისათვის სპეციალური მეთოდი უნდა იქნას გამოყენებული.

arr.splice(str) მეთოდი არის უნივერსალური მასივებთან მუშაობისთვის. ყველაფრის გაკეთება შეუძლია: ელემენტების დამატება, ამოღება და შეცვლა.

მისი სინტაქსია:

```
arr.splice(index[, deleteCount, elem1, ..., elemN])
```

ის იწყებს index პოზიციიდან, შლის deleteCount რაოდენობის ელემენტს და მათ ადგილას ჩასვამს elem1, ..., elemN. შედეგად აბრუნებს წაშლილი ელემენტებისაგან

შედგენილ მასივს. მოცემული მეთოდი ცვლის საწყის მასივს. პირველი ორი პარამეტრი აუცილებელია. პირველი პარამეტრი არის პირველი წასაშლელი ელემენტის ინდექსი, ხოლო მეორე – წასაშლელი ელემენტების რაოდენობა.

ამ მეთოდის გაგება ყველაზე მარტივია მაგალითის განხილვით:

```
a = new Array ( 1, 2, "a", "b", "c", "d" )
x = a . splice ( 1, 3 )      // x მასივის ელემენტებია – 2, "a", "b"
                             // a მასივის ელემენტებია – 1, "c", "d"

a = new Array ( 1, 2, "a", "b", "c", "d" )
x = a . splice ( 1, 3, "e", "f", "g" )
                              // x მასივის ელემენტებია – 2, "a", "b"
                              // a მასივის ელემენტებია – 1, "e", "f", "g",
                                    "c", "d"
```

slice

arr.slice მეთოდი გაცილებით მარტივია, ვიდრე მისი მსგავსი arr.splice მეთოდი.

მისი სინტაქსია:

```
arr.slice([start], [end])
```

ის აბრუნებს ახალ მასივს, რომელშიც აკოპირებს ელემენტებს start ინდექსიდან end-მდე (end-ის გარეშე). start და end ინდექსი შეიძლება იყოს უარყოფითი. ამ შემთხვევაში, დათვლა მასივის ბოლოდან დაიწყება.

მიღებული ოპერაციის შედეგი იქნება მასივი. მოცემული მეთოდი არ ცვლის საწყის მასივს. მეორე პარამეტრი

აუცილებელი არ არის. თუ მოცემულია მხოლოდ ერთი პარამეტრი, მაშინ შედეგად მიღებული მასივის ელემენტები საწყისი მასივის ელემენტები იქნება, start ინდექსიდან დაწყებული მასივის ბოლომდე. თუ მოცემულია ორივე პარამეტრი, მაშინ საწყისი მასივიდან აიღება ელემენტები start ინდექსიდან end ინდექსიმდე ამ ბოლო ელემენტის გამოკლებით. თუ არც ერთი პარამეტრი არ არის მოცემული, მაშინ საწყისი მასივის ასლი მიიღება. ეს ხშირად გამოიყენება მასივის ასლის შესაქმნელად შემდგომი გარდაქმნების მიზნით, როდესაც არ უნდა შეიცვალოს საწყისი მასივი.

ეს str.slice სტრიქონის მეთოდის მსგავსია, მაგრამ ქვესტრიქონების ნაცვლად ქვემასივებს აბრუნებს.

მაგალითად:

```
a = new Array ( 1, 2, "a", "b", "c" )
a . slise ( 1, 3 )  // შედეგი არის მასივი ელემენტებით 2, "a"
a . slise ( 3 )     // შედეგი არის მასივი ელემენტებით "b", "c"
a . slise (  )      // შედეგი არის მასივი ელემენტებით 1, 2, "a", "b",
                       "c"
```

sort

ის აბრუნებს დახარისხებულ მასივს, მაგრამ ჩვეულებრივ დაბრუნების მნიშვნელობა იგნორირებულია, რადგან თავად arr იცვლება.

arr.sort(fn) – შედარების ფუნქციის დახმარებით მასივის ელემენტების მოწესრიგება (დალაგება). მისი სინტაქსია:

arr.sort(fn)

ოპერაციის შედეგი იქნება დახარისხებული მასივი. მოცემული მეთოდი ცვლის საწყის მასივს. პარამეტრი აუცილებელი არ არის. პარამეტრად შეიძლება მითითებული იყოს ელემენტების მოწესრიგების საკუთარი ფუნქციის სახელი.

მაგალითად:

```
let arr = [ 1, 2, 15 ];

// ეს მეთოდი ახდენს arr მასივის მოწესრიგებას
arr.sort();

alert( arr );  // 1, 15, 2
```

შედეგად მივიღეთ [1, 15, 2] მასივი, რაც გამოწვეულია იმით, რომ ამ მეთოდით ხდება მასივის ელემენტების დალაგება ხდება, როგორც სტრიქონი. მასივის ელემენტების გარდაქმნა ხდება სტრიქონად.

იმისათვის, რომ გამოვიყენოთ ჩვენი საკუთარი დალაგების ფუნქცია, ჩვენ ფუნქციის სახელი უნდა გამოვიყენოთ როგორც arr.sort() მეთოდის არგუმენტი.

ფუნქცია შედეგს უნდა აბრუნებდეს წყვილი მნიშვნელობებისთვის:

```
function Num(a, b) {
  if (a > b) return 1;          // თუ პირველი მნიშვნელობა მეტია
                                მეორეზე
  if (a == b) return 0;         // თუ ტოლია
  if (a < b) return -1;         // თუ პირველი მნიშვნელობა
                                ნაკლებია მეორეზე
}
```

მაგალითად, რიცხვების მოსაწესრიგებლად:

```
function Num(a, b) {
  if (a > b) return 1;
  if (a == b) return 0;
  if (a < b) return -1;
}

let arr = [ 1, 25, 15, 7, 18 ];

arr.sort(Num);

alert(arr);  // 1,7,15,18,25
```

მასივი arr შეიძლება იყოს ნებისმიერი მასივი, ის შეიძლება შეიცავდეს რიცხვებს, სტრიქონებს, ობიექტებს ან სხვას. ჩვენ გვაქვს რაიმე ელემენტების ნაკრები. მის დასალაგებლად გვჭირდება ფუნქცია, რომელმაც იცის როგორ შეადაროს მისი ელემენტები ერთმანეთს. ნაგულისხმევად, ელემენტები დალაგდება როგორც სტრიქონული მონაცემები.

arr.sort(fn) მეთოდი დახარისხებას ახორციელებს ზოგად ალგორითმის მიხედვით. ჩვენ არ გვჭირდება იმაზე ფიქრი, თუ როგორ მუშაობს ეს მეთოდი. ის დაათვალიერებს მასივს, ადარებს მის ელემენტებს მოცემული ფუნქციის გამოყენებით და აწესრიგებს მათ. მთავარია, ჩვენ მას მივაწოდოთ fn ფუნქცია, რომელიც ამ შედარებას შეასრულებს.

```
let arr = [ 1, 25, 15, 7, 18 ];

```

```
arr.sort(function(a, b) { return a - b; });

alert(arr);  // 1,7,15,18,25
```

reverse

arr.reverse მეთოდი ცვლის მასივის ელემენტების მიმდევრობას საწინააღმდეგო მიმდევრობით.

შედეგი იქნება მასივი. მოცემული მეთოდი ცვლის საწყის მასივს.

მაგალითი:

```
a1 = new Array ( 1, 2, "a", "b" )
a . reverse ( )                  // შედეგი არის მასივი ელემენტებით
                                 "b", "a", 2, 1
```

toLocaleString, toString

toLocaleString, toString – მასივის ელემენტებს გარდაქმნის სიმბოლურ სტრიქონად. toLocaleString მეთოდით გარდაქმნის ალგორითმი დამოკიდებულია ბრაუზერის ვერსიაზე.

რიცხვითი მასივების დამუშავების ფუნქციები.

ხშირად საჭიროა რიცხვითი მონაცემების სტატისტიკური მახასიათებლების გამოთვლა: ყველა მონაცემების ჯამი, საშუალო, მაქსიმალური და მინიმალური მნიშვნელობა.

არაცარიელი მასივის ყველა ელემენტთა ჯამის გამოსათვლელი ფუნქცია:

```
function S ( aN ) {
let S = aN [0];
for ( let i = 1; i <= aN . length – 1; i ++ ) {
```

```
                        S += aN [i];
   }
return S;
          }
```

ცხადია, საშუალო არითმეტიკულის გამოსათვლელად უნდა ვისარგებლოთ გამოსახულებით S (aN) / aN . length.

მასივის ელემენტებს შორის მინიმალური ელემენტის მნიშვნელობის პოვნის ფუნქცია:

```
function Nmin ( aN ) {
let Nmin = aN [0];
for ( let i = 1; i <= aN . length – 1; i ++ ) {
                        if ( aN [i] < Nmin ) Nmin= aN [i];
   }
return Nmin;
          }
```

მასივის ელემენტებს შორის მაქსიმალური ელემენტის მნიშვნელობის პოვნის ფუნქცია:

```
function Nmax ( aN ) {
let Nmax = aN [0];
for ( let i = 1; i <= aN . length – 1; i ++ ) {
                        if ( aN [i] < Nmax ) Nmax= aN [i];
   }
return Nmax;
          }
```

შევქმნათ ერთი ფუნქცია, რომლის საშუალებითაც შეიძლება ყველა ზემოთ ჩამოთვლილი სტატისტიკური

მახასიათებლები გამოვთვალოთ და რომელიც შედეგად ამ მნიშვნელობებს მასივის სახით დააბრუნებს:

```
function statistic ( aN ) {
if ( aN == 0 | | aN == null | | aN == " " )
            return new Array ( 0, 0, 0, 0 );
let S = aN [0];
let Nmin = aN [0];
let Nmax = aN [0];
for ( let i = 1; i <= aN . length – 1; i ++ ) {
                                    S += aN [i];
                                  if ( aN [i] < Nmin ) Nmin= aN [i];
                                  if ( aN [i] < Nmax ) Nmax= aN [i];
   }
return new Array ( S, S / aN . length, Nmin, Nmax );
            }
```

თარიღი და დრო

პროგრამათა უმრავლესობაში საჭიროა თარიღისა და დროის ასახვა, დღეების რაოდენობის გამოთვლა და სხვა. ხშირად ზოგიერთი პროგრამების მართვა თარიღისა და დროის მიხედვით ხდება. ამასთან, უნდა გვახსოვდეს, რომ არსებობს დროის სარტყელი და დროის სეზონური ცვლილებები (ზაფხულისა და ზამთრის დრო).

იმისათვის, რომ მოვახდინოთ ორგანიზაციისა და ფიზიკური პირის მოქმედების კოორდინაცია დროში, ჩვენი პლანეტის სხვადასხვა წერტილში შემოღებულ იქნა დროის ათვლის სისტემა. იგი უკავშირდება მერიდიანს, რომელიც გადის დიდი ბრიტანეთის ქალაქ გრინვიჩში მდებარე ასტრონომიულ ობსერვატორიაზე. ამ დროით ზონას უწოდებენ საშუალო დროს გრინვიჩის მიხედვით (Greenwich Mean Time – GMT). ახლახან ამ აბრევიატურის გარდა გამოჩნდა კიდევ ერთი – UTC (Universal Time Coordinated – საყოველთაო კოორდინირებული დრო).

თუ ჩვენი კომპიუტერის დრო დაყენებულია სწორად, მაშინ დროის ათვლა GMT სისტემაში ხდება, მაგრამ ამოცანათა პანელზე ლოკალურ დროს აჩვენებს, რომელიც ჩვენი დროის სარტყელს შეესაბამება.

JavaScript-ზე დაწერილ პროგრამაში უბრალოდ არ შეიძლება ჩავწეროთ თარიღი (მაგ., 30.05.2023). დროისა და თარიღის შექმნა სპეციალურად Date ობიექტის გამოყენებით ხდება.

ის შეიცავს თარიღსა და დროს და უზრუნველყოფს მათი მართვის მეთოდებს.

მაგალითად, ის შეიძლება გამოყენებულ იქნას შექმნის/შეცვლის დროის შესანახად, დროის გასაზომად ან უბრალოდ მიმდინარე თარიღის საჩვენებლად.

Date ობიექტის შექმნა

ახალი Date ობიექტის შესაქმნელად, გამოიძახეთ new Date() კონსტრუქტორი ერთ-ერთი შემდეგი არგუმენტით:

new Date() არგუმენტების გარეშე - შექმენით Date ობიექტი მიმდინარე თარიღით და დროით:

```
let now = new Date();
alert( now ); // აჩვენებს მიმდინარე თარიღსა და დროს
```

შექმენით Date ობიექტი რომელიც 1970 წლის 1 იანვრიდან UTC+0 გასული მილიწამების (წამის მეათასედი) ტოლია.

```
// 0 შეესაბამება 01.01.1970 UTC+0
let Jan1 = new Date(0);
alert( Jan1 );

// ახლა დავუმატოთ 24*5 საათი და მივიღებთ 06.01.1970
                        UTC+0
let Jan = new Date((24 * 3600 * 1000)*5);
alert( Jan );
```

მთელ რიცხვს, რომელიც წარმოადგენს 1970 წლის დასაწყისიდან გასული მილიწამების რაოდენობას, timestamp-ი (დროის შტამპი) ეწოდება.

ეს არის თარიღის ციფრული წარმოდგენა. თქვენ ყოველთვის შეგიძლიათ დროის შტამპიდან მიიღოთ თარიღი new Date(timestamp) გამოყენებით და გადაიყვანოთ არსებული Date ობიექტი timestamp-ად date.getTime() მეთოდის გამოყენებით (იხ. ქვემოთ).

1970 წლის 1 იანვრამდე თარიღებს უარყოფითი timestamp ექნება, მაგალითად:

```
// 31 დეკემბერი 1969 წელი
let Dec31_1969 = new Date(-24 * 3600 * 1000);
alert( Dec31_1969 );
```

თარიღის ობიექტის new Date () გამოსახულებით შექმნის დროს შეიძლება პარამეტრის სახით მივუთითოთ ის თარიღი და დრო, რომელიც ამ ობიექტზე უნდა დავაყენოთ. ეს შეიძლება ხუთი ხერხით გაკეთდეს:

```
new Date ("Month, dd, yyyy hh:mm:ss")
new Date ("Month, dd, yyyy")
new Date ( yy, mm, dd, hh, mm, ss)
new Date ( yy, mm, dd)
new Date (milliseconds)
```

პირველ ორ შემთხვევაში პარამეტრი სტრიქონის სახით მიეთითება, რომელშიც თარიღისა და დროის კომპონენტებია მითითებული. ასოებით აღნიშნულია პარამეტრის შაბლონი. ყურადღება უნდა მიექცეს გამყოფებს – მძიმესა და ორ წერტილს. დროის მითითება აუცილებელი არ არის. ამ შემთხვევაში დრო ნულის ტოლი იქნება. თარიღის კომპონენტი აუცილებლად უნდა მიეთითოს. თვის დასახელება უნდა ჩაიწეროს

ინგლისურად (აბრევიატურა არ შეიძლება). დანარჩენი კომპონენტები მიეთითება რიცხვების სახით. თუ რიცხვი 10-ზე ნაკლებია, მაშინ შეიძლება ერთი ციფრით ჩაიწეროს. მესამე და მეოთხე შემთხვევაში თარიღისა და დროის კომპონენტები რიცხვების სახით ჩაიწერება და ერთმანეთისაგან მძიმით გამოიყოფა.

ბოლო შემთხვევის დროს უნდა ჩაიწეროს მთელი რიცხვი, რაც მიუთითებს მილიწამების რაოდენობას, რომელიც გასულია 1970 წლის 1 იანვრიდან (ანუ 00:00:00 მომენტიდან). მილიწამების რაოდენობის მიხედვით თარიღისა და დროის ყველა კომპონენტი გამოითვლება.

```
let date = new Date("2023-01-26");
alert(date);
```

დრო არ არის მითითებული, ამიტომ დაყენებულია შუაღამის GMT და იცვლება დროის ზონის მიხედვით, სადაც კოდი შესრულებულია. ასე რომ, შედეგი შეიძლება იყოს Thu Jan 26 2023 04:00:00 GMT+0400 (Georgia Standard Time) - ხუთ 26 იანვარი 2023 04:00:00 GMT+0400 (საქართველოს სტანდარტული დრო).

new Date(year, month, date, hours, minutes, seconds, ms)

შექმენით თარიღის ობიექტი მოცემული კომპონენტებით ლოკალურ საათობრივ სარტყელში. საჭიროა მხოლოდ პირველი ორი არგუმენტი.

- year უნდა იყოს ოთხნიშნა: 2023 სწორია, 23 არა;
- month იწყება 0 (იანვარი) 11 (დეკემბერი) ჩათვლით;

- date პარამეტრი აქ წარმოადგენს თვის დღეს. თუ პარამეტრი არ არის დაყენებული, მაშინ მიიღება მნიშვნელობა 1-ის ტოლი;
- თუ hours/minutes/seconds/ms პარამეტრები აკლია, მაშინ მათი მნიშვნელობები ნულის ტოლი ხდება.

მაგალითად:

```
alert (new Date(2023, 0, 1, 0, 0, 0, 0));  // 1 Jan 2023, 00:00:00
alert (new Date(2023, 0, 1)); // იგივეა, რადგან
                              hours/minutes/seconds/ms  0-ის ტოლია
//მაქსიმალური სიზუსტე – 1 მილიწამი (1/1000 წამი):
let date = new Date(2023, 0, 1, 2, 3, 4, 567);
alert( date ); // 1.01.2023, 02:03:04.567
```

Date ობიექტის მეთოდები.

თარიღის ობიექტში შენახულ თარიღზე და დროზე ინფორმაციის წასაკითხად ან შესაცვლელად ობიექტ Date-ის მეთოდები გამოიყენება. თარიღის ობიექტზე მეთოდის გამოსაყენებლად უნდა ჩაიწეროს:

```
ობიექტის სახელი . მეთოდი ([პარამეტრი]);
```

ყველა მეთოდის საკმაოდ დიდი რაოდენობა შეიძლება დაიყოს ორ კატეგორიად: მნიშვნელობის მიღების მეთოდები (მათ დასახელებას აქვს პრეფიქსი get) და ახალი მნიშვნელობის დადგენის მეთოდები (მათ დასახელებას აქვს პრეფიქსი set). თითოეულ კატეგორიაში გამოიყოფა მეთოდების ორი ჯგუფი – ლოკალური ფორმატისათვის და UTC ფორმატისათვის. ეს მეთოდები საშუალებას იძლევა თარიღისა და დროის ცალკეულ

კომპონენტებზე (წელი, თვე, რიცხვი, კვირის დღე, საათი, წუთი, წამი, მილიწამი) ვიმუშაოთ.

იგივე დასახელების ფუნქციები, ოღონდ პრეფიქსი set შესაბამისი კომპონენტის ახალ მნიშვნელობას ადგენს.

მეთოდების გამოყენების დროს უნდა გვახსოვდეს, რომ არ უნდა გამოვიყენოთ გამოსახულება, სადაც დრო სხვადასხვა ფორმატში იქნება გამოყენებული. ამასთან, უნდა გვახსოვდეს, რომ თვეების, კვირის დღეების, საათის, წუთისა და წამის ნუმერაცია 0-ით იწყება. XX საუკუნის წლების ჩაწერა შესაძლებელია ორნიშნა რიცხვით, ხოლო 1900-ზე ნაკლები და 1999-ზე მეტი წლები აუცილებლად უნდა ჩაიწეროს ოთხნიშნა რიცხვით.

Date ობიექტის მეთოდები

მეთოდი	მნიშვნელობის დიაპაზონი	აღწერა
getFullYear ( )	1970-...	წელი
getMonth ( )	0-11	თვე (იანვარი = 0)
getDate ( )	1-31	რიცხვი
getDay ( )	0-6	კვირის დღე (კვირა = 0)
getHours ( )	0-23	საათი 24 საათიან ფორმატში
getMinutes ( )	0-59	წუთები
getSeconds ( )	0-59	წამები
getTime ( )	0-...	მილიწამები დაწყებული 1.1.70 00:00:00 GMT-დან
getMilliseconds ( )	0-...	მილიწამები დაწყებული 1.1.70 00:00:00 GMT-დან

getUTCFullYear ( )	1970-...	წელი UTC
getUTCMonth ( )	0-11	თვე UTC (იანვარი = 0)
getUTCDate ( )	1-31	რიცხვი UTC
getUTCDay ( )	0-6	კვირის დღე UTC (კვირა = 0)
getUTCHours ( )	0-23	საათი UTC 24 საათიან ფორმატში
getUTCMinutes ( )	0-59	წუთები UTC
getUTCSeconds ( )	0-59	წამები UTC
getUTCTime ( )	0-...	მილიწამები UTC 1.1.70 00:00:00 GMT-დან
getUTCMilliseconds( )	0-...	მილიწამები UTC 1.1.70 00:00:00 GMT-დან
getTimezoneOffset ( )	0-...	სხვაობა წუთებში GMT-სა და UTC-ს შორის

მაგალითი:

თუ თქვენი ადგილობრივი დროის ზონა წანაცვლებულია UTC-დან, მაშინ შემდეგი კოდი სხვადასხვა საათს აჩვენებს:

```
// მიმდინარე თარიღი
let date = new Date();

// საქართველოს დროითი სარტყელის დრო
alert( date.getHours());
alert( date.getMinutes());
```

```
alert ( date.getSeconds());
alert ( date.getMilliseconds());

// UTC+0 დროითი სარტყელის დრო (ლონდონის დრო
                              ზაფხულის დროზე გადასვლის
                              გარეშე)
alert( date.getUTCHours() );
alert( date.getUTCMinutes());
alert ( date.getUTCSeconds());
alert ( date.getUTCMilliseconds());
```

getTimezoneOffset() აბრუნებს UTC-სა და ადგილობრივ დროით სარტყელს შორის წუთებში განსხვავებას:

```
// რადგან ჩვენ UTC+4 საათობრივ ზონაში ვიმყოფებით
                              გამოიტანს -240
alert( new Date().getTimezoneOffset() );
```

განვსაზღვროთ თარიღი, რომელიც ერთი კვირის შემდეგ დადგება მიმდინარე თარიღთან მიმართებაში:

```
week = 1000 * 60 * 60 * 24 * 7;   /* მილიწამების რაოდენობა
                              კვირაში */
alert (week);   // 604800000
mydate = new Date ( );
alert (mydate);        // მიმდინარე თარიღი Fri Mar 10 2023
                              20:38:39 GMT+0400 (Georgia Standard
                              Time)
mydate_ms = mydate . getTime ( );
alert (mydate_ms);   // 1678466319071
```

```
mydate_ms += week;
alert (mydate_ms);   //1679071119071
mydate . setTime (mydate_ms);
alert (mydate);      // Fri Mar 17 2023 20:38:39 GMT+0400 (Georgia
                               Standard Time)
newdate = mydate . toLocaleString ( );  /* ახალი თარიღი
                               სტრიქონის სახით */
alert (newdate);     // 3/17/2023, 8:38:39 PM
```

განვსაზღვროთ დღეების რაოდენობა ორ თარიღს შორის. მაგალითად, 2007 წლის 10 თებერვალსა და 5 მარტს შორის. ამისათვის, ჯერ უნდა შევქმნათ ორი თარიღის ობიექტი:

```
let date1 = new Date("2023-02-15");
alert (date1);         // Wed Feb 15 2023 04:00:00 GMT+0400
                               (Georgia Standard Time)
let date2 = new Date ("2023-03-05");
alert (date2);         // Sun Mar 05 2023 04:00:00 GMT+0400
                               (Georgia Standard Time)
// სხვაობა უნდა გაიყოს ერთ დღეში მილიწამების
                               რაოდენობაზე:
m1=date2.getTime();
m2=date1.getTime();
days= (m1-m2)/1000/60/60/24;
alert (days);          // 18
```

ზოგჯერ პროგრამაში საჭიროა დროის შეყოვნება წინასწარ განსაზღვრული ინტერვალით:

```
function pause (mSec) {
    clock = new Date ( );
    justMinute = clock . getTime ( );
    while (true) {
    just = new Date ( );
    if ( just . getTime ( ) - justMinute . mSec ) break;
    }
    }
```

თარიღის ავტომატური კორექტირება

ავტოკორექტირება არის თარიღის ობიექტების ძალიან სასარგებლო ფუნქცია. თქვენ შეგიძლიათ დააყენოთ თარიღის კომპონენტები მნიშვნელობების ნორმალური დიაპაზონის მიღმა და ობიექტი თავად გასწორდება.

მაგალითი:

```
let date = new Date(2023, 0, 32); // 32 Jan 2023 ?!?
alert(date); //  1st Feb 2023!
```

This page says

Wed Feb 01 2023 00:00:00 GMT+0400 (Georgia Standard Time)

OK

არასწორი თარიღის კომპონენტები ავტომატურად ნაწილდება დანარჩენებზე.

დავუშვათ, თარიღი „2020 წლის 28 თებერვალი“ ორი დღით უნდა გავზარდოთ. იმის მიხედვით, წელიწადი ნაკიანია

(29-ით არის) თუ არა, შედეგი იქნება „2 მარტი“ ან „1 მარტი“. ჩვენ არ გვჭირდება ამაზე ფიქრი. უბრალოდ დაამატეთ ორი დღე. დანარჩენზე Date ობიექტი იზრუნებს:

```
let date = new Date(2020, 1, 28);
date.setDate(date.getDate() + 2);

alert( date ); // 1 Mar 2020
```

This page says

Sun Mar 01 2020 00:00:00 GMT+0400 (Georgia Standard Time)

OK

ეს ფუნქცია ხშირად გამოიყენება თარიღის მისაღებად გარკვეული დროის გასვლის შემდეგ. მაგალითად, ჩვენ გვინდა მივიღოთ თარიღი "ამ დროიდან 120 წამის შემდეგ":

```
let date = new Date();
date.setSeconds(date.getSeconds() + 120);

alert( date ); // გამოიტანს ზუსტ თარიღს
```

This page says

Fri Mar 17 2023 19:37:34 GMT+0400 (Georgia Standard Time)

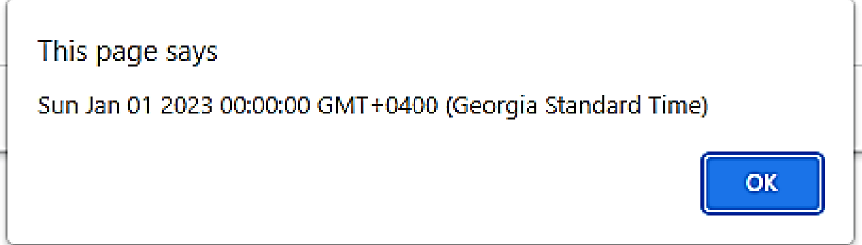
OK

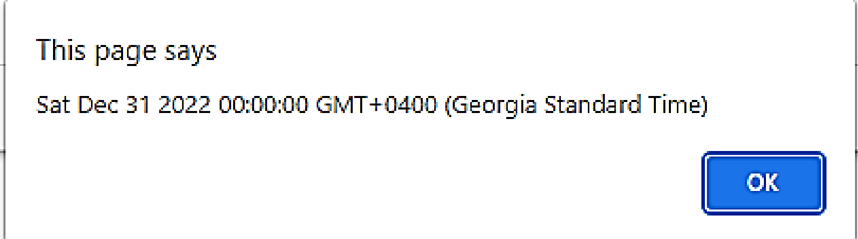
თქვენ ასევე შეგიძლიათ მიუთითოთ ნულოვანი ან თუნდაც უარყოფითი მნიშვნელობები. მაგალითად:

```
let date = new Date(2023, 0, 2); // 2 Jan 2023

date.setDate(1); // მივუთითოთ თვის პირველი რიცხვი
alert( date );

date.setDate(0); // თვის პირველი რიცხვი არის 1, ასე, რომ
                         გამოიტანს წინა თვის ბოლო რიცხვს
alert( date ); // 31 Dec 2022
```





თარიღის სხვაობის რიცხვად გადაქცევა

თუ Date ობიექტი გარდაიქმნება რიცხვად, მაშინ მივიღებთ timestamp-ს (დროის შტამპი) date.getTime() ანალოგიურად:

```
let date = new Date();
```

```
alert(+date); // რაოდენობა მილიწამში, იგივეა რაც
     date.getTime()
```

This page says

1679067555924

OK

თარიღების გამოკლება შესაძლებელია, რის შედეგადაც სხვაობას მივიღებთ მილიწამებში. ეს ტექნიკა შეიძლება გამოყენებულ იქნას დროის გასაზომად:

```
let start = new Date(); // ვიწყებთ დროის აღრიცხვას

// ვასრულებთ რაიმე მოქმედებას
for (let i = 0; i < 100000; i++) {
  let Something = i * i * i;
}

let end = new Date(); // ვამთავრებთ დროის აღრიცხვას

alert( ‘ციკლი ${end - start} მილიწამში შესრულდა ‘ );F
```

შედეგად მიიღება:

This page says

ციკლი 5 მილიწამში შესრულდა

OK

Date.now()

თუ უბრალოდ დროის გაზომვა გვინდა, არ გვჭირდება Date ობიექტი. არსებობს სპეციალური Date.now() მეთოდი, რომელიც მიმდინარე დროს აბრუნებს. ის სემანტიკურად new Date().getTime() მეთოდის ეკვივალენტურია, თუმცა მეთოდი არ ქმნის შუალედურ Date ობიექტს. ასე რომ, ეს მეთოდი უფრო სწრაფია.

ამ მეთოდის გამოიყენება მოხერხებულია ან როცა სწრაფქმედებაა მნიშვნელოვანი, მაგალითად, JavaScript-ში თამაშების დამუშავებისას ან სხვა სპეციალიზებულ აპლიკაციებში.

```
let start = Date.now(); // მილიწამების რაოდენობა 1970 წლის 1
    იანვრიდან

// ვასრულებთ რაიმე მოქმედებას
for (let i = 0; i < 100000; i++) {
  let Something = i * i * i;
}

let end = Date.now(); // ვამთავრებთ დროის აღრიცხვას
```

```
alert( ‘ციკლი ${end - start} მილიწამში შესრულდა ‘ );  // აკლდება რიცხვები და არა თარიღები
```

This page says

ციკლი 3 მილიწამში შესრულდა

OK

კვირის დღის ჩვენება

დავწეროთ ფუნქცია getWeekDay(date), რომელიც კვირის დღეების მასივს შემოკლებული ფორმატით შექმნის: 'კვ', 'ორ', 'სმ', 'ოთ', 'ხთ', 'პრ', 'შბ'.

date.getDay() მეთოდი აბრუნებს კვირის დღის ნომერს დაწყებული კვირიდან.

მაგალითი:

```
function getWeekDay(date) {
  let days = ['კვ', 'ორ', 'სმ', 'ოთ', 'ხთ', 'პრ', 'შბ'];

  return days[date.getDay()];
```

```
}

let date = new Date(2023, 3, 15); // 15 აპრილი 2023 წელი
alert( getWeekDay(date) ); // შბ
```

This page says

შბ

OK

ევროპის ქვეყნებში კვირა იწყება ორშაბათს (დღე ნომერი 1), შემდეგ მოდის სამშაბათი (ნომერი 2) და ასე გრძელდება კვირამდე (ნომერი 7). დავწეროთ getLocalDay(date) ფუნქცია, რომელიც კვირის „ევროპულ" დღეს დააბრუნებს.

```
function getLocalDay(date) {

  let day = date.getDay();

  if (day == 0) { // კვირის დღე 0 (კვირა) ევროპულ ნუმერაციაში
     იქნება 7
    day = 7;
  }

  return day;
}
let date = new Date(2023, 3, 15);  // 15 აპრილი 2023 წელი
alert( getLocalDay(date) );       // შაბათი, 6
```

This page says
6
OK

გამოყენებული ლიტერატურა

1. https://tc39.es/ecma262/#sec-ecmascript-language-types-symbol-type
2. https://learn.javascript.ru/
3. https://learn.javascript.ru/courses/jsbasic
4. https://www.w3schools.com/js/default.asp
5. Scott Duffy, JavaScript, California, U.S.A. four Edition
https://www.pdfdrive.com/how-to-do-everything-with-java-script-d162390075.html
6. Danny Goodman, JavaScript Bible, New York, U.S.A. https://everythingcomputerscience.com/books/all.pdf
7. Phil Ballard, Sams Teach Yourself JavaScript in 24 Hours, Sixth Edition, Indianapolis, Indiana, USA. https://vulms.vu.edu.pk/Courses/CS202/Downloads/JavaScript%20 Book.pdf

გადაეცა წარმოებას 20.11.2023. ხელმოწერილია დასაბეჭდად 27.11.2023.
ქაღალდის ზომა 60X84 1/16. პირობითი ნაბეჭდი თაბახი 18,9.

„ევროპის უნივერსიტეტი“,
თბილისი, დ. გურამიშვილის 76

ი.მ. „გოჩა დალაქიშვილი“,
თბილისი, ვარკეთილი 3, კორპ. 333, ბინა 38