ევროპის უნივერსიტეტი
EUROPEAN UNIVERSITY

თეიმურაზ სტურუა
თეა თოდუა
ქეთევან ნანობაშვილი
ბესიკ ტაბატაძე

# JavaScript-ის საფუძვლები

## ნაწილი II

თბილისი
2023

ევროპის უნივერსიტეტი
EUROPEAN UNIVERSITY

თეიმურაზ სტურუა
თეა თოდუა
ქეთევან ნანობაშვილი
ბესიკ ტაბატაძე

# JavaScript-ის საფუძვლები

## ნაწილი II

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.



ISBN 978-9941-8-5257-2

# სარჩევი

# ხდომილების ცნება

მომხმარებლის ყოველი მოქმედება (კლავიშზე ხელის დაჭერა, მაუსით დაწკაპუნება და სხვა) ახდენს რაიმე ხდომილების ფორმირებას, ანუ შეტყობინებას რაიმე მოქმედებაზე. ოპერაციული სისტემა აანალიზებს ამ შეტყობინებას, რათა გაიგოს საიდან მოვიდა და რა გააკეთოს.

## ხდომილებათა თვისებები

ხდომილებაზე შეტყობინების ფორმირება ხდება ობიექტის, ანუ ინფორმაციის შესანახი კონტეინერის სახით. როგორც კი ხდომილების ობიექტი შეიქმნება, ბრაუზერი მის თვისებას მიანიჭებს მნიშვნელობას. მაგალითად, მაუსის დაწკაპუნების შესაბამისი ობიექტი შეიცავს მაუსის მაჩვენებლის კოორდინატებს და აგრეთვე ცნობას იმის შესახებ, თუ რომელი კლავიში დავაწკაპუნეთ. გარდა ამისა, ხდომილების ობიექტი ერთ-ერთ თავის თვისებაში შეიცავს მითითებას ელემენტზე, რომელთანაც არის დაკავშირებული მოცემული ხდომილება (მაგალითად, ღილაკი, გამოსახულება შეტანის ველი და სხვა).

ხდომილების ობიექტი მეხსიერებაში ინახება იმდენ ხანს, რამდენიც ესაჭიროება სცენარს მის დასამუშავებლად. ვიდრე სრულდება ხდომილების დამმუშავებელი მანამდე ბრაუზერის მეხსიერებაში არის განთავსებული ხდომილების ობიექტი. როგორც კი ის მუშაობას დაამთავრებს, ობიექტი ცარიელდება (უბრუნდება საწყის მდგომარეობას). დროის ნებისმიერ მომენტში არ არსებობს ერთზე მეტი ხდომილების ობიექტი. ყველა ინიცირებული ხდომილება ოპერაციული სისტემის მიერ

ჩაიწერება ბუფერში და იმ მიმდევრობით სრულდება, რა მიმდევრობითაც მათი იქ ჩაწერა მოხდა.

ობიექტურ მოდელში გვაქვს ობიექტი event, რომელიც არის ობიექტ window-ს ქვეობიექტი. ის შეიცავს ინფორმაციას თუ რომელი ხდომილება მოხდა, რომელმა ელემენტმა უნდა მოახდინოს მასზე რეაგირება და სხვა.

ხშირად გამოიყენება თვისებები button და srcElement. თვისება button შედეგად აბრუნებს მთელ რიცხვით მნიშვნელობას, რომელიც მიუთითებს მაუსის რომელი კლავიში ან კლავიშები დააწკაპუნა მომხმარებელმა (0 – კლავიშები არ არის დაწკაპუნებული, 1 – დაწკაპუნებულია მარცხენა კლავიში, 2 – დაწკაპუნებულია მარჯვენა კლავიში, 3 - ერთდროულად დაწკაპუნებულია ორივე კლავიში).

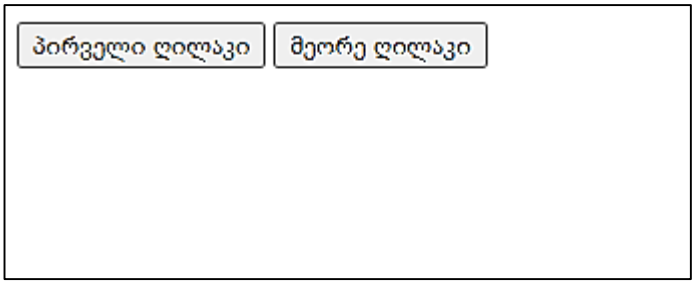
თვისება srcElement შედეგად ხდომილების მაინიცირებელი HTML-დოკუმენტის ელემენტის ობიექტზე მითითებას აბრუნებს. ასეთი მითითებით შეიძლება გავიგოთ ან შევცვალოთ ამ ობიექტის თვისების მნიშვნელობა და მის მიმართ გამოვიყენოთ მისი ნებისმიერი მეთოდი. თვისების მნიშვნელობის შეცვლა innerText თვისებით ხორციელდება.

ქვემოთ მოყვანილია HTML-კოდის მაგალითი, რომლის საშუალებითაც ხდება დოკუმენტის ფორმირება. ეს დოკუმენტი ორ ღილაკს შეიცავს. მაუსით შეიძლება დაწკაპუნება ერთ-ერთ ღილაკზე ან თავისუფალ ადგილზე.

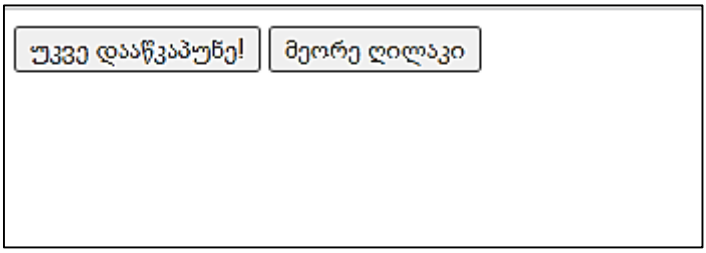
```
<!DOCTYPE html>
<html>
<body onclick = "changetext ( )">
<button> პირველი ღილაკი </button>
<button> მეორე ღილაკი </button>
```

```
</body>
<script>
function changetext ( ) {
x = window . event . srcElement
x . innerText = "უკვე დააწკაპუნე!"
}
</script>
</html>
```

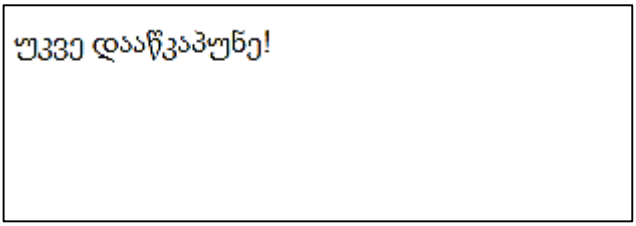
პროგრამის გაშვების შედეგად მიიღება:




პირველ შემთხვევაში ის ღილაკი, რომელზეც მოხდა დაწკაპუნება შეიცვლება სხვა, ”უკვე დააწკაპუნე!” ღილაკით (იხილეთ სურათი):




თუ ფანჯრის ნებისმიერ სხვა ადგილზე დავაწკაპუნებთ, მაშინ მივიღებთ სურათზე გამოსახულ შედეგს:




იმისათვის, რომ ზემოთ მოყვანილმა კოდმა რეაგირება მოახდინოს მხოლოდ ღილაკზე მაუსით დაწკაპუნებაზე, საჭიროა ამ კოდის მოდიფიცირება:

```
<!DOCTYPE html>
<html>
<body>
<button onclick = "changetext ( )"> პირველი ღილაკი </button>
<button onclick = "changetext ( )"> მეორე ღილაკი </button>
</body>
<script>
function changetext ( ) {
x = window . event . srcElement
x . innerText = "უკვე დააწკაპუნე!"
}
</script>
</html>
```

# ბრაუზერისა და დოკუმენტის ობიექტური მოდელი

## ობიექტი Window

HTML-დოკუმენტში <iframe> ტეგის საშუალებით მოცემული ობიექტი window შეიცავს ყველა frames ჩარჩოს კოლექციას. ობიექტ window-ს აქვს თვისებები, მეთოდები, ხდომილებები და აგრეთვე, შვილობილი ობიექტები. ქვემოთ მოყვანილია მათი სრული ჩამონათვალი და განხილულია მხოლოდ მათ შორის პრაქტიკული თვალსაზრისით ყველაზე უფრო საჭირო.

### თვისებები window

- closed – აბრუნებს ლოგიკურ მნიშვნელობას, თუ ფანჯარა დახურულია;
- console - აბრუნებს ფანჯრის console ობიექტს;
- defaultStatus – მოძველებულია;
- document – აბრუნებს ფანჯრის document ობიექტს;
- frameElement - აბრუნებს ჩარჩოს, რომელშიც ფანჯარა გაიხსნა;
- frames - აბრუნებს ფანჯარაში გაშვებულ ყველა ფანჯრის ობიექტს;
- history – აბრუნებს ფანჯრის history ობიექტს;
- innerHeight - აბრუნებს ფანჯრის შიგთავსის არეალის სიმაღლეს გადახვევის ზოლების ჩათვლით;
- innerWidth - აბრუნებს ფანჯრის შიგთავსის არეალის სიგანეს გადახვევის ზოლების ჩათვლით;

- length - აბრუნებს <iframe> ელემენტების რაოდენობას მიმდინარე ფანჯარაში;
- localStorage - საშუალებას გაძლევთ შეინახოთ გასაღები/მნიშვნელობის წყვილები ვებბრაუზერში. ინახავს მონაცემებს ვადის გასვლის თარიღის გარეშე;
- location – აბრუნებს ფანჯრის location ობიექტს;
- parent – გვაბრუნებს მიმდინარე ფანჯრიდან მშობლიურ ფანჯარაში;
- name – ადგენს ან აბრუნებს ფანჯრის სახელს;
- navigator – აბრუნებს ფანჯრის navigator ობიექტს;
- opener – აბრუნებს მითითებას ფანჯარაზე, რომელმაც შექმნა მოცემული ფანჯარა;
- outerHeight - აბრუნებს ბრაუზერის ფანჯრის სიმაღლეს, ხელსაწყოთა ზოლების/გადახვევის ზოლების ჩათვლით;
- outerWidth - აბრუნებს ბრაუზერის ფანჯრის სიგანეს, ხელსაწყოთა ზოლების/გადახვევის ზოლების ჩათვლით;
- pageXOffset - აბრუნებს პიქსელებს, რომლებზეც მიმდინარე დოკუმენტი ფანჯრის ზედა მარცხენა კუთხიდან გადავიდა (ჰორიზონტალურად);
- pageYOffset - აბრუნებს პიქსელებს, რომლებშიც მიმდინარე დოკუმენტი ფანჯრის ზედა მარცხენა კუთხიდან გადავიდა (ვერტიკალურად);
- screenLeft - აბრუნებს ფანჯრის ჰორიზონტალურ კოორდინატს ეკრანთან მიმართებით;
- screenTop - აბრუნებს ფანჯრის ვერტიკალურ კოორდინატს ეკრანთან მიმართებით;

- screenX - აბრუნებს ფანჯრის ჰორიზონტალურ კოორდინატს ეკრანთან მიმართებით;
- screenY - აბრუნებს ფანჯრის ვერტიკალურ კოორდინატს ეკრანთან მიმართებით;
- screen – აბრუნებს ფანჯრის screen ობიექტს;
- scrollX - pageXOffset-ის შემოკლებული დასახელება;
- scrollY - pageYOffset-ის შემოკლებული დასახელება;
- self - აბრუნებს მიმდინარე ფანჯარას;
- sessionStorage - საშუალებას გაძლევთ შეინახოთ გასაღები/მნიშვნელობის წყვილები ვებ-ბრაუზერში. ინახავს მონაცემებს ერთი სესიისთვის;
- status – მოძველებულია. მოერიდეთ მის გამოყენებას;
- top – გვაბრუნებს ბრაუზერის წინა ფანჯარაში.

## მეთოდები window

- addEventListener() – ფანჯარას მიამაგრებს დამმუშავებელ ხდომილებას;
- alert() – აჩვენებს გაფრთხილების ფანჯარას შეტყობინებითა და OK ღილაკით;
- atob() - მოხდება კოდირებული სტრიქონის Base-64-ით დეკოდირება;
- blur() – ხსნის ფოკუსს მიმდინარე ფანჯრიდან; მისი შესაბამისი ხდომილებაა – onblur;
- btoa() - სტრიქონის კოდირება Base-64-ში;
- clearInterval() – ანულებს setInterval() მეთოდით ჩართულ ტაიმერს;
- clearTimeout() – ანულებს setTimeout() მეთოდით ჩართულ ტაიმერს;

- close() – დახურავს ბრაუზერის მიმდინარე ფანჯარას;
- confirm() - აჩვენებს დიალოგურ ფანჯარას შეტყობინებით და OK და Cancel ღილაკით;
- focus ( ) – ფოკუსს დააყენებს მიმდინარე ფანჯარაზე; მისი შესაბამისი ხდომილებაა – onfocus;
- getComputedStyle() - იღებს ელემენტის მიმდინარე გამოთვლილ CSS სტილებს;
- getSelection() - აბრუნებს შერჩეულ ობიექტს, რომელიც წარმოადგენს მომხმარებლის მიერ არჩეული ტექსტის დიაპაზონს;
- matchMedia() - აბრუნებს MediaQueryList ობიექტს, რომელიც წარმოადგენს მითითებულ CSS მედია მოთხოვნის სტრიქონს;
- moveBy() - ფანჯრის გადაადგილება მის არსებულ პოზიციასთან მიმართებაში;
- moveTo() - ფანჯრის გადაადგილება მითითებულ პოზიციაზე;
- open() – ბრაუზერის ახალი ფანჯრის გახსნა;
- print() - მიმდინარე ფანჯრის შიგთავსის დაბეჭდვა;
- prompt() – აჩვენებს მოწვევის ფანჯარას შეტყობინებით, ტექსტური ველითა და OK და Cancel ღილაკებით;
- removeEventListener() - ფანჯრიდან შლის ხდომილების დამმუშავებელს;
- requestAnimationFrame() - თხოვს ბრაუზერს, გამოიძახოს ფუნქცია, რათა განაახლოს ანიმაცია მომდევნო გადაღებამდე;
- resizeBy() - მითითებული პიქსელებით ცვლის ფანჯრის ზომას;

- resizeTo() - ცვლის ფანჯრის ზომას მითითებული სიგანით და სიმაღლით;
- scroll() – მოძველებულია. ეს მეთოდი შეიცვალა scrollTo() მეთოდით;
- scrollBy() - გადაახვევს დოკუმენტს პიქსელების მითითებული რაოდენობის მიხედვით;
- scrollTo() - გადააქვს დოკუმენტი მითითებულ კოორდინატებზე;
- setInterval() – პროცედურას მიუთითებს შესრულდეს პერიოდულად წინასწარ მოცემული მილიწამების ინტერვალით;
- setTimeout() – უშვებს პროგრამას, გვერდის ჩატვირთვის შემდეგ წინასწარ მოცემული მილიწამების ინტერვალით;
- stop() - აჩერებს ფანჯრის ჩამოტვირთვას.

## window ხდომილება

- onblur – ფანჯრის გამოსვლა ფოკუსიდან;
- onfocus – ფანჯარა ხდება აქტიური;
- onhelp – მომხმარებლის მიერ <F1> კლავიშზე ხელის დაჭერა;
- onresize – მომხმარებლის მიერ ფანჯრის ზომების შეცვლა;
- onscrokk – მომხმარებლის მიერ ფანჯრის გადაფურცვლა;
- onerror – შეცდომა გადაცემის დროს;
- onbeforeunload – გვერდის მიერ ფანჯრის დატოვების შემთხვევაში, რათა მოხდეს მონაცემების შენარჩუნება;

- onload – გვერდი სრულად არის ჩატვირთული;
- onunload – უშუალოდ გვერდის მიერ ფანჯრის დატოვების შემთხვევაში.

ზემოთ ჩამოთვლილი ხდომილებიდან სამი სრულდება მომხმარებლის მოქმედების შედეგად. თუ გახსნილია ბრაუზერის რამდენიმე ფანჯარა, მომხმარებელს შეუძლია გააქტიუროს ნებისმიერი მათგანი ფოკუსის გადატანით ერთი ფანჯრიდან მეორეზე. ეს მოქმედებები ხორციელდება onblur და onfocus ხდომილებებით. იგივე მოქმედება შეიძლება გამოწვეულ იქნეს პროგრამულადაც blur და focus მეთოდების გამოყენებით. გვერდის ან მისი ელემენტის ჩატვირთვის დროს თუ ხდება შეცდომა, მაშინ ხდება onerror ხდომილების ინიცირება. ჩვენ შეიძლება გამოვიყენოთ ეს ხდომილება პროგრამაში რათა, მაგალითად, ვცადოთ კიდევ ერთხელ ჩავტვირთოთ გვერდი ან რაიმენაირად შევცვალოთ შემდგომი ქმედებები.

onload ხდომილება ხდება მაშინ, როდესაც გვერდი მთლიანად ჩატვირთულია ფანჯარაში; ხდომილება onbeforeunload – ვიდრე გვერდი დატოვებს ფანჯარას; ხდომილება onunload – როდესაც გვერდმა დატოვა ფანჯარა, ახალი გვერდის ჩატვირთვის ან ბრაუზერის დახურვის წინ.

ობიექტ window-ს აქვს რამდენიმე შვილობილი ობიექტი, რომელთა გამოყენება შესაძლებელია მისი საშუალებით: document, history, navigator, location, event და screen.

## ობიექტი history

ობიექტი history შეიცავს ინფორმაციას გვერდების მისამართების შესახებ, რომელიც ბრაუზერმა მოინახულა

მიმდინარე სეანსის დროს. ჩვენ შეგვიძლია ამ სიაში გადავინაცვლოთ სცენარის მიხედვით და ჩავტვირთოთ შესაბამისი გვერდები. ობიექტ history-ს აქვს მხოლოდ ერთი თვისება და სამი მეთოდი.

## თვისება history

length – მონახულებული გვერდების სიაში ელემენტების რაოდენობა.

ეს თვისება სცენარში კონტროლისათვის გამოიყენება, რათა არ მოხდეს სიის გარეთ გასვლა.

## მეთოდები history

- back() – ისტორიის სიიდან ჩატვირთავს წინა გვერდს;
- forvard() – ისტორიის სიიდან ჩატვირთავს მომდევნო გვერდს;
- go() – ისტორიის სიიდან ჩატვირთავს გვერდს ნომრით n (0-დან history.length-1-მდე) ან URL-მისამართით მითითებულ გვერდს;
- length() - აბრუნებს URL-ების (გვერდების) რაოდენობას ისტორიის სიაში.

მაგალითი:

ჯერ იტვირთება პირველი გვერდი, ხოლო შემდეგ მეხუთე გვერდი, ამასთან მოწმდება იგი არსებობს თუ არა მონახულებული გვერდების სიაში:

```
window . history . go (1)
if (window . history . length > 4)
        window . history . go (5)
```

ცხადია, რომ მეთოდი history.go(-1), history.back() მეთოდის, ხოლო მეთოდი history.go(1), history.forvard()-ის ეკვივალენტურია.

## ობიექტი location

ობიექტი location შეიცავს ინფორმაციას მიმდინარე გვერდის URL-მისამართის (და მისი კომპონენტების) შესახებ, აგრეთვე იმ მეთოდებს, რომლებიც გვერდების განახლების საშუალებას იძლევა.

### თვისებები location

- hash – დააყენებს ან დააბრუნებს URL-ის წამყვან ნაწილს (#).;
- host – დააყენებს ან დააბრუნებს URL-ის ჰოსტის სახელს და პორტის ნომერს;
- hostname – დააყენებს ან დააბრუნებს URL-ის ჰოსტის სახელს;
- href – დააყენებს ან დააბრუნებს მთელ URL-ს;
- origin - აბრუნებს URL-ის პროტოკოლს, ჰოსტის სახელს და პორტის ნომერს;
- pathname – დააყენებს ან დააბრუნებს URL-ის გზის სახელს;
- port – დააყენებს ან დააბრუნებს URL-ის პორტის ნომერი;
- protocol – დააყენებს ან დააბრუნებს URL-ის პროტო-კოლს, რომლის შემდეგაც ორი წერტილი დგას, მაგალითად „http:“;

- search – დააყენებს ან დააბრუნებს URL-ის მოთხოვნის სტრიქონს.

მაგალითად, თუ ჩვენ ჩავტვირთეთ გვერდი http://www.imedi.ge, მაშინ location . href-ის მნიშვნელობა იქნება ამ გვერდის მისამართი.

თუ href თვისებას მივანიჭებთ ახალ მნიშვნელობას, მაშინ ჩვენ შეგვიძლია შევცვალოთ ბრაუზერში ნაჩვენები გვერდი, მაგალითად:

```
window . location . href = "http://www.google.ge"
```

## მეთოდები location

- assign() – ჩატვირთავს სხვა გვერდს; ეს მეთოდი window.location. href თვისების ეკვივალენტურია;

- reload() – განაახლებს მიმდინარე გვერდს;

- replace() – ჩატვირთავს გვერდს URL-მისამართით მითითებულ პარამეტრში და ცვლის მიმდინარე გვერდის URL-მისამართს (location . href).

ზოგიერთი Web-საიტი შეიძლება შეიცავდეს ისეთ გვერდებს, რომლებიც მომხმარებლების სანახავად და ნანახი გვერდების სიაში შესატანად არ არის განკუთვნილი. მაგალითად, საიტებზე გადანაცვლებამ შეიძლება მომხმარებელი მიიყვანოს რომელიმე საშუალედო გვერდზე, რომელიც მას შემდგომში მეტი აღარ დასჭირდება. ამასთან, სასურველია რომ მომხმარებელმა ვეღარ შეძლოს დაუბრუნდეს მას Back ღილაკის საშუალებითაც. მითითებულ გვერდზე გადასასვლელად იყენებენ მეთოდს location.replace (URL-მისამართი), ამასთან ამ გვერდს პროგრამა სიაში არ შეიტანს, საიდანაც შეიძლება მასზე გადასვლა Back ღილაკის საშუალებით.

# ობიექტი navigator

ობიექტი navigator ინფორმაციას შეიცავს ბრაუზერის შემქნელის, მისი ვერსიისა და შესაძლებლობების შესახებ.

## თვისებები navigator

- appCodeName – აბრუნებს ბრაუზერის კოდის დასახელებას;
- appName – აბრუნებს ბრაუზერის დასახელებს;
- appVersion – აბრუნებს ბრაუზერის ვერსიას;
- cookieEnabled – ბრაუზერში განსაზღვრავს cookies გამოყენების შესაძლებლობას კლიენტის მხრიდან; ღებულობს მნიშვნელობას true, თუ გამოიყენება cookies;
- geolocation - აბრუნებს გეოლოკაციის ობიექტს მომხმარებლის მდებარეობის განსაზღვრისათვის;
  language - აბრუნებს ბრაუზერის ენას;
  onLine - თუ ბრაუზერი ინტერნეტშია, ბრაუზერი აბრუნებს true-ს;
  platform - აბრუნებს ბრაუზერის პლატფორმას;
  product - აბრუნებს ბრაუზერის ძრავას სახელს;
- userAgent – ბრაუზერის დასახელება, რომელიც იგზავნება http-პროტოკოლის დახმარებით.

## მეთოდები navigator

- javaEnabled – აბრუნებს true, თუ ბრაუზერში ჩართულია Java;
- taintEnabled – ამოღებულია JavaScript 1.2 ვერსიაში (1999).

## ობიექტი screen

ობიექტი screen შეიცავს ინფორმაციას მომხმარებლის ეკრანის შესაძლებლობების შესახებ და შეიძლება გამოვიყენოთ, მაგალითად, შესაქმნელი ფანჯრის ზომების განსაზღვრისა და გადასაცემი გრაფიკის ამოხსნადობის ასარჩევად (აზრი არა აქვს ჩაიტვირთოს 32 ბიტიანი გამოსახულება, თუ მომხმარებლის მონიტორი და ვიდეოადაპტერი აღიქვამს მხოლოდ 256 ფერს). ობიექტ screen-ს აქვს შემდეგი თვისებები:

- availHeight - აბრუნებს ეკრანის სიმაღლის (ამოცანათა პანელის გამოკლებით);
- availWidth - აბრუნებს ეკრანის სიგანის (ამოცანათა პანელის გამოკლებით);
- colorDepth - აბრუნებს მონიტორის ფერთა პალიტრის ბიტურ მნიშვნელობას გრაფიკული გამოსახულებების გამოტანის დროს;
- height – შედეგად აბრუნებს მომხმარებლის ეკრანის სიმაღლეს (პიქსელებში);
- pixelDepth - აბრუნებს ეკრანის ფერის გარჩევადობას (ბიტებში ერთ პიქსელზე);
- width – შედეგად აბრუნებს მომხმარებლის ეკრანის სიგანეს (პიქსელებში);

## ობიექტი document

ობიექტი document ობიექტური მოდელის იერარქიაში არის ცენტრალური ობიექტი და კოლექციებისა და თვისებების საშუალებით HTML-დოკუმენტზე მთელ ინფორმაციას მოიცავს.

ის, აგრეთვე, გვაწვდის მრავალ მეთოდსა და ხდომილებას დოკუმენტთან სამუშაოდ.

## თვისება document

| თვისება | ატრიბუტი | დანიშნულება |
|---|---|---|
| activeElement | | ახდენს აქტიური ელემენტის იდენტიფიკაციას |
| alinkColor | alink | გვერდზე აქტიური ბმულის ფერი |
| bgColor | bgcolor | განსაზღვრავს ელემენტის ფონის ფერს |
| body | | <body> ტეგში განსაზღვრული დოკუმენტის არაცხად ძირითად ობიექტზე მითითება მხოლოდ წაკითხვისათვის |
| cookie | | cookie-ჩანაწერის სტრიქონი. ამ თვისებისთვის ახალი მნიშვნელობის მინიჭებას, ბრაუზერის დახურვის შემდეგ, დისკზე cookie-ს ჩაწერამდე მივყავართ |
| domain | | აყენებს ან აბრუნებს დოკუმენტის დომენს მისი დაცვისა ან იდენტიფიკაციისათვის |
| fgColor | text | წინა პლანის ტექსტის ფერის დაყენება |

| | | |
|---|---|---|
| lastModified | | გვერდის ბოლო ცვლილების თარიღი, მისაწვდომია როგორც სტრიქონი |
| linkColor | link | გვერდზე არსებული ჯერ კიდევ მოუნახულებელი ჰიპერბმულის ფერი |
| location | | დოკუმენტის სრული URL |
| parentWindow | | აბრუნებს დოკუმენტის მშობლიურ ფანჯარას |
| readyState | | განსაზღვრავს ჩასატვირთი ობიექტის მიმდინარე მდგომარეობას |
| referer | | URL გვერდი, რომელმაც გამოიძახა მიმდინარე გვერდი |
| selection | | document-ისთვის selection შვილობილ ობიექტზე მითითება მხოლოდ წაკითხვისათვის |
| title | title | ელემენტისათვის განსაზღვრავს იმ საცნობარო ინფორმაციას, რომელიც გამოიყენება ჩატვირთვის დროს ან გამოჩნდება Popup prompt ფანჯარაში |
| url | URL | კლიენტის დოკუმენტის ან <meta> ტეგში URL-მისამართი |

| | | |
|---|---|---|
| vlinkColor | vlink | გვერდზე არსებული მონახულებული ბმულის ფერი |

## document კოლექციები

- all – დოკუმენტის ძირითად ნაწილში ყველა ტეგებისა და ელემენტების კოლექცია;
- anchors – დოკუმენტში არსებული ყველა სანიშნის კოლექცია;
- applets – დოკუმენტში არსებული ყველა ობიექტის კოლექცია, მათ შორის მართვის ჩაშენებული ელემენტები, გრაფიკული ელემენტები, აპლეტები და სხვა ობიექტები;
- embeds – დოკუმენტში არსებული ყველა დანერგილი ობიექტის კოლექცია;
- forms – გვერდზე არსებული ყველა ფორმის კოლექცია;
- images – გვერდზე გამოსახული ყველა გრაფიკული ელემენტის კოლექცია;
- links – გვერდზე არსებული ყველა მითითებისა და <area> ბლოკის კოლექცია;
- scripts – გვერდზე არსებული ყველა <script> განყოფილების კოლექცია;
- styleSheets – დოკუმენტში განსაზღვრული სტილის ყველა კონკრეტული თვისების კოლექცია.

## მეთოდები document

- clear – ასუფთავებს გამოყოფილ მონაკვეთს;
- close – ხურავს ბრაუზერის მიმდინარე ფანჯარას;
- createElement – გამოყოფილი ტეგისათვის ქმნის ელემენტის ეგზემპლარს;

- elementFromPoint – აბრუნებს ელემენტს მოცემული კოორდინატებით;
- execCommand – ასრულებს ბრძანებას (ოპერაციას) გამოყოფილ ობიექტზე ან არეზე;
- open – ხსნის დოკუმენტს, როგორც ნაკადს write და writeln მეთოდების გამოყენებით შედეგების დასამუშავებლად;
- queryCommandEnabled – იძლევა შეტყობინებას დაშვებულია თუ არა მოცემული ბრძანება;
- queryCommandIndeterm – იძლევა შეტყობინებას, თუ მოცემულ ბრძანებას აქვს განუსაზღვრელი სტატუსი;
- queryCommandState – აბრუნებს ბრძანების მიმდინარე მდგომარეობას;
- queryCommandSupported – იძლევა შეტყობინებას მხარდაჭერილია თუ არა მოცემული ბრძანება;
- queryCommandText – აბრუნებს სტრიქონს, რომელთანაც მუშაობს ბრძანება;
- queryCommandValue – აბრუნებს დოკუმენტის ან ობიექტ TextRange-ისათვის განსაზღვრული ბრძანების მნიშვნელობას;
- write – მითითებულ ფანჯარაში არსებულ დოკუმენტში ჩაწერს ტექსტს და HTML კოდს;
- writeln – ჩაწერს ტექსტს და HTML კოდს, რომელიც მთავრდება ახალ სტრიქონზე გადასვლით.

## ხდომილება document

- onafterupdate – წარმოიშობა მონაცემთა გადაცემის დამთავრების დროს;

- onbeforeupdate – წარმოიშობა გვერდის მიერ ფანჯრის დატოვების შემთხვევაში;
- onclick – ხდება მაუსის მარცხენა კლავიშზე დაწკაპუნების შემთხვევაში;
- ondblclick – ხდება მაუსის მარცხენა კლავიშზე ორჯერ დაწკაპუნების შემთხვევაში;
- ondragstart – ხდება, როდესაც მომხმარებელი იწყებს მაუსით ”გადათრევას”;
- onerror – შეცდომა გადაცემის დროს;
- onhelp – მომხმარებლის მიერ <F1> კლავიშზე ხელის დაჭერა;
- onkeydown – წარმოიშობა კლავიშზე ხელის დაჭერის შემთხვევაში;
- onkeypress – წარმოიშობა კლავიშზე ხელის დაჭერისა და მასზე ხელის დაჭერის მდგომარეობის შენარჩუნების შემთხვევაში;
- onkeyup – წარმოიშობა, როდესაც მომხმარებელი ხელს აიღებს კლავიშიდან;
- onload – წარმოიშობა დოკუმენტის მთლიანად ჩატვირთვის შემთხვევაში;
- onmousedown – ხდება მაუსის კლავიშზე ხელის დაჭერის შემთხვევაში;
- onmousemove – ხდება მაუსის მაჩვენებლის გადაადგილების შემთხვევაში;
- onmouseout – ხდება, როდესაც მაუსის მაჩვენებლი გადის ელემენტის საზღვრებს გარეთ;
- onmouseover – ხდება, როდესაც მაუსის მაჩვენებლი შედის ელემენტის საზღვრებში;

- onmouseup – ხდება, როდესაც მომხმარებელი ხელს აუშვებს მაუსის კლავიშს;
- onresdystatechange – წარმოიშვება readystate თვისების შეცვლის შემთხვევაში;
- onselectstart – ხდება, როდესაც მომხმარებელი პირველად უშვებს დოკუმენტის გამოყოფილ ნაწილს.

## ობიექტი event

ობიექტი event საშუალებას იძლევა მივიღოთ ინფორმაცია ბრაუზერში მომხდარი ხდომილების შესახებ. ეს ინფორმაცია მოცემულია შემდეგ თვისებებში:

- altKey – შედეგად აბრუნებს <Alt> კლავიშის მდგომარეობას, როდესაც ხდომილება ხდება;
- button – მაუსის კლავიში, რომელმაც გამოიწვია ხდომილება;
- cancelBubble – ხდება მოცემული ხდომილების აკრძალვა ობიექტური იერარქიის ზედა მიმართულებით;
- clientX – შედეგად აბრუნებს ელემენტის x კოორდინატას, გამოირიცხება ჩარჩოში ჩასმა, შეჭრები, გადაფურცვლის ზოლები და ა. შ.;
- clientY – შედეგად აბრუნებს ელემენტის y კოორდინატას, გამოირიცხება ჩარჩოში ჩასმა, შეჭრები, გადაფურცვლის ზოლები და ა. შ.;
- CtrlKey – <Ctrl> კლავიშის მდგომარეობა, როდესაც ხდება ხდომილება;
- fromElement – შედეგად აბრუნებს ელემენტს, რომლიდანაც onmouseover და onmouseout ხდომილებისათვის მაუსის კურსორმა გადაინაცვლა;

- keyCode – დაჭერილი კლავიშის ASCII კოდი; შესაძლებელია ობიექტზე გადასაცემი მნიშვნელობის შეცვლა;
- offsetX – შედეგად აბრუნებს მაუსის მაჩვენებლის x კოორდინატას პიქსელებში მისი შემცველი ელემენტის მიმართ ხდომილების წარმოქმნის დროს;
- offsetY – შედეგად აბრუნებს მაუსის მაჩვენებლის y კოორდინატას პიქსელებში მისი შემცველი ელემენტის მიმართ ხდომილების წარმოქმნის დროს;
- reason – მიუთითებს, რომ მონაცემთა გადანაცვლება წარმატებით განხორციელდა ან რის გამო მოხდა მისი შეწყვეტა;
- returnValue – ხდომილებისათვის განსაზღვრავს დასაბრუნებელ მნიშვნელობას;
- screenX – შედეგად აბრუნებს მაუსის მაჩვენებლის ჰორიზონტალურ კოორდინატას ეკრანის მიმართ, როდესაც ხდომილება ხდება;
- screenY – შედეგად აბრუნებს მაუსის მაჩვენებლის ვერტიკალურ კოორდინატას ეკრანის მიმართ, როდესაც ხდომილება ხდება;
- shiftKey – განსაზღვრავს <Shift> კლავიშის მდგომარეობას, ხდომილების წარმოშობის დროს;
- srcElement – შედეგად აბრუნებს ელემენტს, რომლიდანაც ხდომილების გასვლა დაიწყო;
- srcFilter – შედეგად აბრუნებს ფილტრს, რომელმაც onfilterchange ხდომილება გამოიწვია;
- toElement – შედეგად აბრუნებს ელემენტს, რომელზეც დადგა მაუსის მაჩვენებელი onmouseover და onmouseout ხდომილების წარმოშობის დროს;

- type – შედეგად აბრუნებს ხდომილების დასახელებას on წინსართის გარეშე სტრიქონის სახით;
- x – შედეგად მაუსის მაჩვენებლის x კოორდინატას შესაბამისად აბრუნებს მშობლიური ელემენტის ან ფანჯრის მიმართ;
- y – შედეგად მაუსის მაჩვენებლის y კოორდინატას შესაბამისად აბრუნებს მშობლიური ელემენტის ან ფანჯრის მიმართ;

event ობიექტის თვისებები ხდომილების გასვლის მომენტში წარმოიშვება და მათი უმეტესობა განკუთვნილია მხოლოდ წაკითხვისათვის (არ შეიძლება მათი შეცვლა). თუმცა არის ორი თვისება keyCode და returnValue, რომელთა შეცვლაც შესაძლებელია.

# მარტივი ვიზუალური ეფექტები

## გამოსახულების შეცვლა

Web-დიზაინში ხშირად წარმოიშობა ერთი გამოსახულების მეორეთი შეცვლის აუცილებლობა. გამოსახულების შეცვლის არსი მდგომარეობს <img> ტეგის src ატრიბუტის მნიშვნელობის სცენარის დახმარებით შეცვლაში. თუ <img> ტეგის ელემენტი, რომელიც გვაძლევს გამოსახულებას, არის HTML-დოკუმენტში, მაშინ ობიექტურ მოდელში გვაქვს ამ ელემენტის src თვისების მქონე ობიექტი. ამ თვისების მნიშვნელობა შეიძლება შეიცვალოს სცენარში. ამასთან, ბრაუზერის ფანჯარაში სცენარის მიერ ჩაიტვირთება შესაბამისი გრაფიკული ფაილი, თუ რა თქმა უნდა იპოვნის მას.

მაგალითი:

```
<!DOCTYPE html>
<html>
<img id = "myimg" src = 'pict1.jpg'
onclick = "document . all . myimg . src = 'pict2.jpg' ">
</html>
```

მოყვანილ მაგალითში პირველ სურათზე მაუსის ერთხელ დაწკაპუნება გამოიწვევს მისი მეორე სურათით შეცვლას. გამოსახულების შეცვლა ამ მაგალითში მხოლოდ ერთხელ მოხდება. მაუსის შემდგომი დაწკაპუნება არავითარ ცვლილებას არ გამოიწვევს, ვინაიდან მეორე გამოსახულება ისევ იგივეთი იცვლება. იმისათვის, რომ განმეორებითმა დაწკაპუნებამ გამოიწვიოს წინა სურათის დაბრუნება, მოცემული სცენარი მნიშვნელოვნად უნდა შეიცვალოს: უნდა მოხდეს ცვლადი

ტრიგერის (ე. წ. ალამი) შექმნა, რომელიც ორიდან ერთ-ერთის მნიშვნელობას მიიღებს. ალმის მიმდინარე მნიშვნელობით სცენარი განსაზღვრავს ორიდან რომელი გამოსახულების გამოტანა უნდა მოხდეს. გამოსახულების შეცვლასთან ერთად აუცილებლად უნდა მოხდეს ალმის მნიშვნელობის შეცვლაც.

```
<!DOCTYPE html>
<html>
<img id = "myimg" src = 'pict1.jpg' onclick = "imgchange ( ) ">
<SCRIPT>
var flag = false
function imgchange ( ) {
if (flag) document . all . myimg . src = "pict1.jpg"
          else document . all . myimg . src = "pict2.jpg"
flag = !flag
}
</script>
</html>
```

ახლა განვიხილოთ მაგალითი, როდესაც Web-გვერდზე განთავსებულია მინიატურული გალერეა (thumbnails – ფართომასშტაბიანი გამოსახულების შემცირებული ასლები). შემცირებულ ასლზე მაუსის დაწკაპუნებით მისი გადიდება უნდა მოხდეს, ხოლო გადიდებულზე დაწკაპუნებით კი ისევ შემცირება. ამ ამოცანის გადასაწყვეტად საჭირო იქნება იმდენი ალამი, რამდენი გამოსახულებაც გვექნება.

```
<!DOCTYPE html>
<html>
<script>
```

```
let apict1 = new Array (" pict1.jpg ", . . . );  /* საწყისი ფაილების
    სახელების მასივი */
let apict2 = new Array (" pict2.jpg ", . . . ); /* შემცვლელი ფაილების
    სახელების მასივი */
let aflag = new Array (apict1 . length);  /* ალმების მასივი */
let xstr = " ";
for ( i = 0; i < apict1 . length; i++)  {
xstr += ' <img id = " i ' + i + ' " src = " ' + apict1 [i] + ' " onclick = "
    imgchange ( ) " >'
}
document . write (xstr);
function imgchange ( )  {
let xid = event . srcElement . id;
let n = parseInt (xid . substr (1) );
if (aflag [n]) document . all [xid] . src = apict1 [n]
    else document . all [xid] . src = apict2 [n];
aflag [n] = !aflag [n];
}
</script>
</html>
```

ყურადღება იმას უნდა მიექცეს, რომ src თვისებაზე მიმართვა ხდება document.all [xid].src-ით, და არა document.all.xid.src-ით ან document.all ["xid"].src-ით. ეს აიხსნება იმით, რომ xid არის სტრიქონული ცვლადი, რომელიც შეიცავს ID იდენტიფიკატორის მნიშვნელობას და არ არის უშუალოდ ამ იდენტიფიკატორის მნიშვნელობა.

## ღილაკებისა და ტექსტის გამონათება

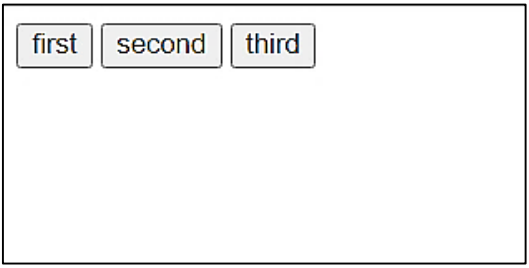
განვიხილოთ ღილაკის ფერის შეცვლის ამოცანა, მასთან მაუსის მაჩვენებლის მიყვანის დროს. მაჩვენებლის ღილაკიდან გადაწევის შემთხვევაში მას უნდა დაუბრუნდეს თავდაპირველი ფერი. ეს არის ე. წ. ღილაკების გამონათება.

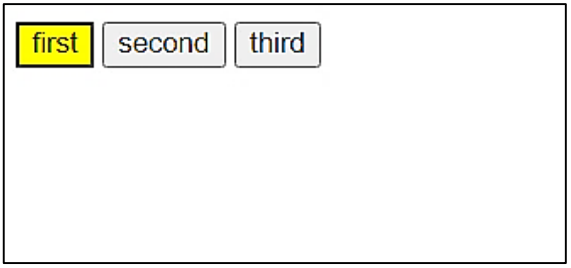
მაგალითი:

მოცემულია სამი ღილაკი კონტეინერული ფორმის <form> სახით. ამ კონტეინერთან მიმაგრებულია ხდომილების დამამუშავებლები onmouseover (მაუსის მაჩვენებლის დაყენება ობიექტზე) და onmouseout (მაუსის მაჩვენებლის გადატანა ობიექტიდან). ამგვარად, ამ ხდომილების ინიციატორი (მიმღები) შეიძლება იყოს ფორმის ნებისმიერი ელემენტი (ჩვენს მაგალითში ღილაკები). ჩვეულებრივ მდგომარეობაში ღილაკების ფერი ნაცრისფერია, მოცემული თექვსმეტობითი a0a0a0 კოდით. მასთან მაუსის მაჩვენებლის მიყვანით ღილაკების ფერი იცვლება და ხდება ყვითელი (yellow).

```
<!DOCTYPE html>
<html>
<style>
mystyle  {font-weight:bold; background-color:a0a0a0}
</style>
<form  onmouseover = "colorchange ('yellow')"
onmouseout = "colorchange ('a0a0a0')">
<input   type = "button" value = "first" class = "mystyle" onclick =
    "alert('you have pressed the button - 1')">
<input type = "button" value = "second" class  = "mystyle" onclick =
    "alert('you have pressed the button - 2')">
```

```
<input type = "button" value = "third" class = "mystyle" onclick =
    "alert('you have pressed the button - 3')">
</form>
<script>
function colorchange  (color) {
if (event . srcElement . type == "button")
    event . srcElement . style . backgroundColor=color;
}
</script>
</html>
```




ფანჯარაში არსებულ რომელიმე ღილაკზე მაუსის დაჭერის შემდეგ ამონათდება შეტყობინების ფანჯარა, ხოლო თავად ღილაკი შეფერადდება ყვითელი ფერით. მაგალითად, პირველ ღილაკზე დაჭერისას მივიღებთ:


აქ colorchange() ფუნქციაში მოწმდება ხდომილების ინიციატორი იყო თუ არა ობიექტი button. თუ ეს ასეა, მაშინ ღილაკის ფერი იცვლება, ხოლო წინააღმდეგ შემთხვევაში არა. ამ შემოწმების გარეშე შეიცვლებოდა არა მარტო ღილაკების ფერი, არამედ მთელი ფონიც. ანალოგიურად შეიძლება ნებისმიერი სხვა ელემენტის ფერის შეცვლაც, მაგალითად, ტექსტის ფრაგმენტის. თუ საჭიროა ტექსტის გამონათება, მაშინ ის მოთავსებული უნდა იყოს რომელიმე კონტეინერულ ტეგებს შორის, მაგალითად, ტეგებში <p>, <b>, <i> ან . შემდეგ მაგალითში <b> ტეგში მოთავსებული ტექსტის ფერი მასზე მაუსის მაჩვენებლის მიყვანის შემთხვევაში ლურჯი ფერიდან იცვლება წითლით:

```
<!DOCTYPE html>
<html>
<b onmouseover = "colorch('red')" onmouseout = "colorch('blue')"
style = "color:blue">bold text</b>
<script>
function colorch (color) {
event . srcElement . style . color=color;
}
</script>
```

```
</html>
```

ეკრანზე გამონათებულ ჩანაწერს ექნება ლურჯი, მუქი შრიფტით ჩაწერილი ტექსტი.

## მოცულობითი სათაურები

მოცულობითი სათაურები ძალზე ეფექტურია Web-გვერდების მომზადების დროს. მისი შექმნის იდეა მარტივია: საკმარისია შეიქმნას ერთნაირი რამდენიმე წარწერა და დაედოს ერთმანეთს მცირე წანაცვლებით. საჭიროა მინიმუმ ორი ასეთი წარწერა. ერთი მათგანი გამოიყენება ჩრდილის ეფექტის შესაქმნელად, ხოლო მეორე განთავსდება მასზე. შეიძლება გამოყენებულ იქნეს ასეთი მესამე წარწერაც ქვემოდან განათების ეფექტის შესაქმნელად. საუკეთესო ეფექტს იძლევა ფონის ფერის გათვალისწინებით ამ წარწერების ფერების შერჩევა. ქვემოთ მოყვანილ მაგალითში მოცულობითი სათაური მიღებულია სამი წარწერის ზედდებით. სტილების ცხრილი განისაზღვრება <p> ტეგით. მოცემულ ცხრილში განსაზღვრულია წარწერის შრიფტის ფერი და ზომა. წარწერის პოზიციები კი განისაზღვრება კონტეინერული  ტეგის style ატრიბუტის პარამეტრებით.

```
<!DOCTYPE html>
<html>
<head><title> 3d effect</title><head>
<style>
p {font-family:sans-serif; font-size:72; font-weight:800; color:00aaaa}
p.highlight {color:silver}
p.shadow {color:darkred}
```

```
</style>
<body bgcolor = aeb98d>
<div style = "position:absolute; top:25; left:25">
<p class = shadow>Volumetric Heading</p>
</div>
<div style = "position:absolute; top:20; left:20">
<p class = highlight>Volumetric Heading</p>
</div>
<div style = "position:absolute; top:22; left:22">
<p>Volumetric Heading</p>
</div>
</body>
</html>
```

ამ პროგრამის შესრულებით მიიღება შემდეგი შედეგი:


ახლა განვიხილოთ მაგალითი, სადაც გამოყენებული იქნება ფუნქცია, რომელიც ქმნის სათაურს მოცემული პარამეტრების მიხედვით.

```
<!DOCTYPE html>
```

```
<html>
<head><title> 3d effect</title><head>
<script>
function d3(text, x, y, tcolor, fsize, fweight, ffamily, zind) {
if (!text) return null;
if (!ffamily) ffamily='arial';
if (!fweight) fweight=800;
if (!fsize) fsize=36;
if (!tcolor) tcolor='00aaff';
if (!y) y=0;
if (!x) x=0;
let sd=5, hd=2;
let xzind=" ";
if (!zind) xzind=":z-index:"+zind;
let xstyle='font-family:' + ffamily + ';font-size:' + fsize + ';font-
    weight:' + fweight + ';color:' + tcolor + ';'
let xstr = '<div style = "position:absolute; top:' + (y + sd) + ';left:' + (x
    + sd) + xzind + '">';
xstr += '<p style = "' + xstyle + 'color:darkred">' + text + '</p></div>';
xstr += '<div style = "position:absolute; top:' + y + '; left:' + x + xzind +
    '">';
xstr += '<p style = "' + xstyle + 'color:silver">' + text + '</p></div>';
xstr += '<div style = "position:absolute; top:' + (y + hd) + '; left:' + (x +
    hd) + xzind + '">';
xstr += '<p style = " ' + xstyle + 'color:' + tcolor + ' ">' + text +
    '</p></div>';
document . write (xstr);
}
```

```
d3 ("HELLO!", 50, 15, 'red', 30, 800, 'arial');
d3 ("es ar aris grafiki", 50, 50, 'blue', 52, 800, 'acadnusx');
d3 ("es ubralod teqstis stilia", 10, 120, '00ff00', 40, 900, 'acadmtavr', -
    7);
</script>
</html>
```

ამ პროგრამის შესრულებით მიიღება:


ამ ფუნქციაში სტრიქონები იქმნება, რომელიც შემდეგ ჩაიწერება HTML-დოკუმენტში ტეგის სახით. ფუნქციის პარამეტრებს შორის ბოლო z-Index-ია, რომლის საშუალებითაც შეიძლება ფენის დონის განსაზღვრა, რომელშიც უნდა განთავსდეს სათაური. ელემენტი, რომლის z-Index-ის მნიშვნელობა მეტია, მდებარეობს ზემოთ, ვიდრე ელემენტი, რომლის z-Index-ის მნიშვნელობა ნაკლებია. ტოლი მნიშვნელობისათვის მიმდევრობა განისაზღვრება HTML-დოკუმენტში ტეგების მიმდევრობით. sd და hd პარამეტრებით განისაზღვრება ჩრდილისა და განათების არეალის ზომები.

# ფორმის მონაცემთა დამუშავება

HTML-დოკუმენტის ისეთი ელემენტები, როგორიცაა მონაცემთა შეტანის ველები, ტექსტური არეები, გადამრთველები და ალმები, ჩამოშლადი სიები და ღილაკები, შეიძლება გავაერთიანოთ ფორმაში. ფორმა <form> კონტეინერული ტეგის დახმარებით იქმნება, რომლის შიგნით ამ ფორმის ელემენტების ტეგებია განთავსებული. დოკუმენტის ობიექტურ მოდელში თითოეულ ფორმას შეესაბამება forms კოლექციაში შემავალი თავისი ობიექტი. ნებისმიერი ეს ელემენტი შეიძლება გამოყენებულ იქნეს ფორმის გარეშეც, თუმცა ის მხოლოდ უბრალოდ კონტეინერი კი არ არის, არამედ ამ ფორმის ელემენტებში არსებული ყველა მონაცემის სერვერზე გაგზავნის ორგანიზებისათვის განკუთვნილი კონტეინერი და ობიექტია.

როგორც ადრე, ასევე ახლაც მონაცემების სერვერზე გასაგზავნად სცენარი არ არის აუცილებელი. მონაცემების გასაგზავნად საკმარისია <form> ტეგში მივუთითოთ action ატრიბუტი, ხოლო ფორმაში დავაყენოთ Submit ტიპის ღილაკი. ამ ღილაკზე დაწკაპუნება გამოიწვევს მონაცემთა გაგზავნის ინიცირებას. თუ არ არის მითითებული action ატრიბუტი ან მისი მნიშვნელობა ცარიელია, ფორმის მონაცემები არ გაიგზავნება. ზოგადად, ეს შეიძლება იყოს ფაილის ან CGI-პროგრამის URL-მისამართი, რომელიც მიიღებს და დაამუშავებს გაგზავნილ მონაცემებს. მაგალითად,

```
ACTION = "http://www.myserver/cgi/myprogram.pl".
```

თუ ჩვენ გვსურს ელექტრონული ფოსტით ფორმის მონაცემების გაგზავნა, მაშინ ACTION-ის მნიშვნელობა უნდა ჩაიწეროს შემდეგი სტრიქონის სახით:

```
mailto : e-mail მისამართი
```

ასევე შეიძლება იყოს მითითებული შეტყობინების თემა:

```
mailto : e-mail მისამართი?subject = შეტყობინების თემა
```

<form> ტეგში action ატრიბუტის გარდა საჭიროა მითითებული იყოს კიდევ ორი ატრიბუტი: method და enctype. method ატრიბუტს შეუძლია მიიღოს ორი მნიშვნელობა post ან get. მნიშვნელობის არჩევა მხოლოდ ფორმაზე აისახება, რომელშიც მონაცემები გადაიცემა. მომხმარებლისათვის უფრო მოსახერხებელია თუ post მნიშვნელობას ავირჩევთ. enctype ატრიბუტს მივანიჭოთ მნიშვნელობა "text/plain". ამ შემთხვევაში გაგზავნილ შეტყობინებას ექნება

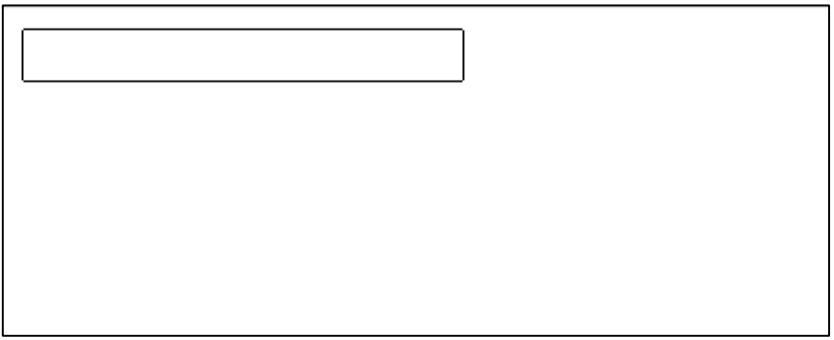
```
ელემენტის სახელი = მნიშვნელობა
```

წყვილის სახე. აქ „ელემენტის სახელი“ ეს არის ელემენტის ტეგში name ატრიბუტის მნიშვნელობა, ხოლო „მნიშვნელობა“ - იმავე ტეგში value ატრიბუტის მნიშვნელობა. თუ არ მივუთითებთ enctype ატრიბუტს, მაშინ შეტყობინება იქნება წარმოდგენილი კოდირებული სახით.

ქვემოთ მოყვანილია HTML-დოკუმენტის მაგალითი ფორმით, რომელიც შეიცავს მონაცემთა შეტანის ველსა და Submit ტიპის ღილაკს:

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

```
<form method = post action = "mailto:kapr@mail.ru" enctype =
    "text/plain">
<input name = "message" type = "text" value = "">
</form>
</html>
```

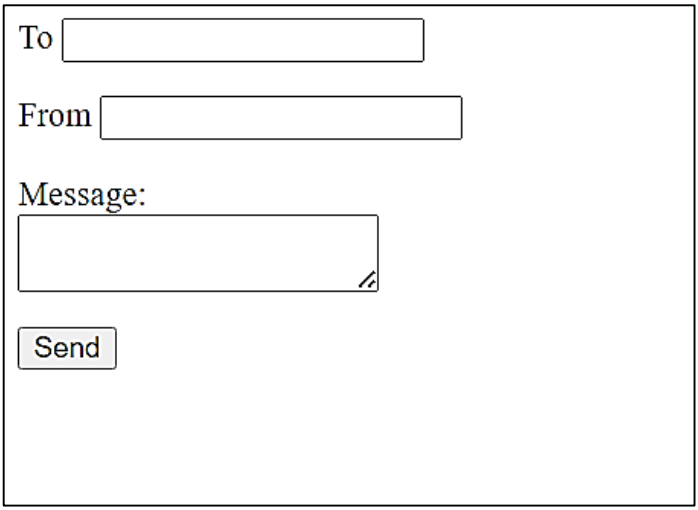


მონაცემთა გაგზავნა განხორციელდება Submit ტიპის ღილაკზე დაწკაპუნებით, რომელზეც ჩვენს მაგალითში იქნება წარწერა Send. მიმღების მისამართი მითითებულია როგორც ACTION ატრიბუტის მნიშვნელობა.

თუ მონაცემთა გაგზავნის წინ საჭიროა მათი შემოწმება ან რაიმე სხვა მოქმედებების ჩატარება, მაშინ უნდა დაიწეროს სცენარი. შემდეგ მაგალითში მოწმდება არის თუ არა ელექტრონული ფოსტის მისამართის ველში სიმბოლო “@” და ხომ არ არის ცარიელი თვით შეტყობინების ველი. ამ დროს არ მოხდება შეტყობინების გაგზავნა.

```
<!DOCTYPE html>
<html>
<form id = "myform" method = POST action = "" enctype =
    "text/plain" style = "background:'e0e0e0' ">
```

```
to
<input name = "email_to" type = "text" value = "">
<p>
from
<input name = "email_from" type = "text" value = "">
<p>
message: <br>
<textarea name = "message" type = "text" value = ""></textarea>
<p>
<! button-type submit >
<input name = "submit" type = "submit" value = "send">
</form>
<script>
function myform.onsubmit ( ) {
let noemail = myform.email_to.value.indexOf ('@') == - 1;
let notext = !myform.Message.value;
let xtext = "\nThe letter was not sent";
if (noemail || notext) {
    event.returnValue = false;
    if (noemail)
        alert ("Invalid email recipient" + xtext);
    else
        alert ("No text messiges" + xtext);
} else
    myform.action = "mailto: " + myform.email_to.value;
}
</script>
</html>
```




ხშირად WEB-გვერდებზე განათავსებენ მიმართვას ან ღილაკს, რომლის საშუალებითაც გაიხსნება გვერდის ავტორისათვის ელექტრონული ფოსტით შეტყობინების გაგზავნის ფორმა. ამ შემთხვევაში საჭირო არ არის მიმღების საფოსტო მისამართის მითითება. თუ ჩვენ გვსურს ამ მისამართის გასაიდუმლოება, საჭიროა მივიღოთ ზოგიერთი ზომები. ყველაზე მარტივი მეთოდია, რომ შევინახოთ მისამართის ცალკეული კომპონენტები ცალკ-ცალკე ცვლადებში და შემდეგ კონკატენაციის საშუალებით მოვახდინოთ მათი გაერთიანება. შედეგად, სპეციალური პროგრამა-რობოტი HTML-დოკუმენტში ვერ იპოვის ელექტრონული ფოსტის მისამართის სტრუქტურის მქონე სტრიქონს. ქვემოთ მოვიყვანოთ ამ ტიპის მაგალითი:

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

```
<form id = "myform" method = POST action = "" enctype =
    "text/plain" onsubmit = "return validator ( )" style =
    "background:'e0e0e0' ">
<h2> send mail to autor</h2>
from
<input name = "email_from" type = "text" value = "">
<p>
message: <br>
<textarea name = "message" type = "text" value = ""></textarea>
<p>
<! button-type submit >
<input name = "submit" type = "submit" value = "send">
</form>
<script>
let first = "xxx" , second = "gmail";
function validator ( ) {
if (!myform.message.value) {
    alert ("no text messiges\nthe letter was not sent");
    return false;
} else
    myform . action = "mailto: " + first + "@" + second + ".com";
    return true ;
}
</script>
</html>
```


მაგალითში validator ფუნქცია აბრუნებს true ან false შედეგს, იმის მიხედვით შეგვიძლია გავაგზავნოთ თუ არა შეტყობინება.

## მენიუ

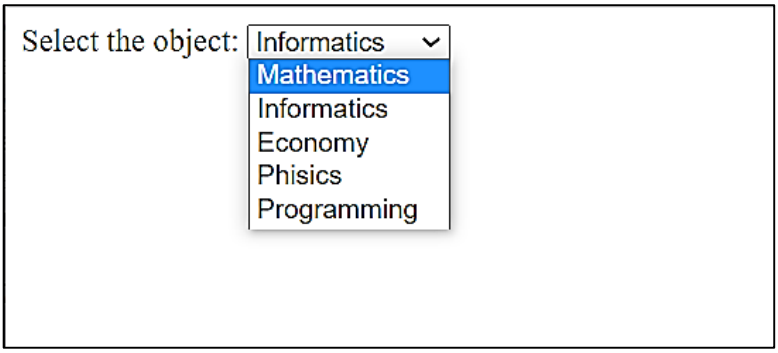
### ჩამოშლადი სია

მარტივი მენიუ შეიძლება შეიქმნას <select> და <option> ტეგების დახმარებით. ჩვეულებრივ ასეთ კონსტრუქციებს უწოდებენ ჩამოშლად სიებს. ქვემოთ მოყვანილი არის ჩამოშლადი სიის გამოყენების მარტივი მაგალითი. ამ მაგალითში ჩამოშლადი სია მოცემულია კოდის სახით, ხოლო ამ სიიდან ამორჩეული ელემენტის დამუშავება ხდება სცენარით. მისი საშუალებით ხდება მხოლოდ სიიდან ამორჩეული ელემენტის ნომრის განსაზღვრა. მაგალითში ეს არის შესაბამისი შეტყობინების ფანჯრიის გამოტანა. მომხმარებლის მიერ სიის ელემენტზე ამორჩევა ხდება მაუსის მარცხენა ღილაკის დაწკაპუნებით. ამასთან, შესაბამისი <select> ტეგის დოკუმენტის

ელემენტის ობიექტის selectedIndex თვისება მნიშვნელობად ენიჭება ამორჩეული ელემენტის ნომერი სიაში (ნუმერაცია 0-დან იწყება). მომხმარებლის ამორჩევის დამუშავების ინიციაცია onchange ხდომილებით ხდება (ელემენტთა სიაში მოხდა ცვლილება). ამ ხდომილების დამუშავება ხორციელდება myselection() ფუნქციით. ჩამოშლად სიაში ელემენტის საწყისი გამოყოფა და გამოჩენა <option> ტეგის selected ატრიბუტითაა მოცემული. ჩვენს მაგალითში გამოყოფილია ელემენტი „Informatics“.

```
<!DOCTYPE html>
<html>
Select the object:
<select name = "test" onchange = "myselection ( )">
    <option> Mathematics
    <option selected> Informatics
    <option> Economy
    <option> Phisics
    <option> Programming
</select>
<script>
function myselection ( ) {
var testname, testnumber;
testnumber = document . all . TEST . selectedIndex;
if (testnumber == 0)
    testname="Fundamentals of Mathematics";
else {
    if (testnumber == 1)
        testname="Basics algorithmization and Sciences";
```

```
    else {
        if (testnumber == 2)
            testname="Economy";
    else {
        if (testnumber == 3)
            testname="Elementary Physics";
    else {
        testname = "Programming language JavaScript";
      }
     }
    }
   }
alert ("You will pass: " + testname);
}
</script>
</html>
```




სისტემაში ნაგულისხმევი წესის თანახმად, ჩამოშლად სიაში ჩანს მხოლოდ ერთი ელემენტი. იმისათვის, რომ ჩამოშლადი სია იყოს ნაწილობრივ გახსნილი (ჩანდეს

რამდენიმე ელემენტი), საჭიროა <select> ტეგში მივუთითოთ ატრიბუტი size = გამოსაჩენი ელემენტების რაოდენობა. მაგალითად:

```
<html>
Select the object:
<script>
let N_sel = 1
let aoptions = new Array ( );
aoptions[0] = "Mathematics";
aoptions[1] = "Informatics";
aoptions[2] = "Economy";
aoptions[3] = "Phisics";
aoptions[4] = "Programming";
xstr = '<select id = "test" onchange = "myselection ( )">';
for (i = 0; i < aoptions . length; i++)  {
xprefix = (i == N_sel) ? 'selected =' + N_sel: '';
xstr += '<option ' + xprefix + '>' + aoptions[i] ;
}
xstr += '</select>';
document . write (xstr);
function myselection ( )  {
var testname, testnumber;
testnumber = document . all . test . selectedIndex;
if (testnumber == 0)
     testname="Fundamentals of Mathematics";
else  {
     if (testnumber == 1)
          testname="Basics algorithmization and Sciences";
```

```
    else {
        if (testnumber == 2)
            testname="Economy";
    else {
        if (testnumber == 3)
            testname="Elementary Physics";
    else {
        testname = "Programming language JavaScript";
      }
     }
    }
   }
alert ("You will pass: " + testname);
}
</script>
</html>
```

ჩამოშლილი მენიუდან რომელიმე პუნქტის შერჩევის შემდეგ ეკრანზე გამოჩნდება შეტყობინების ფანჯარა, შერჩეული თემის შესაბამისი ტექსტით. მაგალითად, პროგრამირების არჩევის შემდეგ გვექნება:

## მენიუ

მენიუ, რომლის შექმნასაც ჩვენ ახლა განვიხილავთ, შედგება მთავარი ჰორიზონტალური მენიუსა და რამდენიმე ვერტიკალური ქვემენიუსაგან. ქვემენიუ ჩამოიშლება მაუსის მაჩვენებლის მთავარი მენიუს ოფციაზე მიყვანის დროს. აუცილებელი არ არის, რომ მთავარი მენიუს ყველა ოფციას შეესაბამებოდეს ქვემენიუ: ზოგიერთი ოფცია შეიძლება იყოს ტერმინალური, ანუ არ შეიცავდეს ქვემენიუს. ოფციაზე მაუსის მარცხენა ღილაკით დაწკაპუნებით განხორციელდება სცენარით განსაზღვრული რაიმე ქმედება. ეს შეიძლება იყოს WEB-გვერდის URL-მისამართი ან JavaScript-ის კოდის შემცველი სტრიქონი.

ჩვენ ასეთი მენიუს სცენარის კოდი გავაკეთოთ უფრო უნივერსალური და გავაფორმოთ ორი ფაილის სახით, რომლის გაფართოებაა .js. პირველი ფაილი, menu_prm.js, შეიცავს მენიუს პარამეტრებს, რომლის დანართის შექმნის დროს მისი შინაარსი შეიძლება შევცვალოთ კონკრეტული ამოცანის მიხედვით. მაგალითად, ამ ფაილში განისაზღვრება ოფციების დასახელება და მათთვის კონკრეტული ქმედებები, ფერი და სხვა პარამეტრები. ამგვარად, ეს არის მენიუს აღწერის ცვლადი ნაწილი. მეორე ფაილი, menu_bld.js, შეიცავს მენიუს აგების მექანიზმისა და ფუნქციონირების აღწერას. მასში გამოიყენება პირველ ფაილში განსაზღვრული პარამეტრები. ეს არის მენიუს აღწერის მუდმივი ნაწილი. რა თქმა უნდა, მომხმარებელს შეუძლია თავისი შეხედულების მიხედვით მოახდინოს ამ ფაილის შინაარსის კორექტირება.

იმისათვის, რომ შევქმნათ მენიუ საკმარისია HTML-დოკუმენტში უბრალოდ ჩავწეროთ შემდეგი სტრიქონები:

```
<script src = "menu_prm.js"></script>
<script src = "menu_bld.js"></script>
<script> buildmenu ( ) </script>
```

სადაც buildMenu() – menu_bld.js ფაილში განსაზღვრული ფუნქციაა. მას menu_prm.js ფაილში მოცემული პარამეტრების მიხედვით ეკრანზე გამოჰყავს მენიუ.

განვიხილოთ მაგალითი რომელშიც მთავარი (ჰორიზონტალური) მენიუ შეიცავს სამ ოფციას და მათ შორის პირველ ორს შეესაბამება ვერტიკალური ქვემენიუ. ბოლო ოფცია არის ტერმინალური. ოფციათა დასახელება შერჩეულია ისე, რომ ადვილად მისახვედრი იყოს რომელი რას მიეკუთვნება.

პირველ ფაილში ჩვენ ვაწვდით ფერებისა და შრიფტის პარამეტრებს, აგრეთვე ოფციის დასახელებისა და მოქმედებების შედგენილობას და კოორდინატებს. მენიუს ოფციათა დასახელება, მათი პოზიციონირება, სტატუსის ზოლის მოქმედებები და შედგენილობა მოცემული იქნება მასივის საშუალებით. ასე, რომ მენიუს სტრუქტურა მოცემული იქნება ორგანზომილებიანი მასივის სახით.

შენიშვნა. თუ მოქმედება არ არის URL-მისამართი, არამედ არის JavaScript ენაზე ჩაწერილი რაიმე კოდი, მაშინ იგი "JavaScript:" პრეფიქსით უნდა იწყებოდეს, რომელსაც მოსდევს წერტილ-მძიმით გამოყოფილი გამოსახულებანი.

მეორე ფაილში განსაზღვრულია ზოგიერთი ფუნქციები, რომლის დახმარებითაც მენიუ ჩამოიშლება და დაიწყებს ფუნქციონირებას. მენიუს ასახვა ხდება ცხრილების განმსაზღვრელი HTML-ტეგების გამოყენების საფუძველზე. HTML-კოდის ფრაგმენტები, რომლისაგანაც იკრიბება HTML-

დოკუმენტის გენერირებული სტრიქონები, განსაზღვრულია მასივის ელემენტების სახით.

menu_prm.js ფაილის კოდი:

```
/* მენიუს პარამეტრები */
let clBorder = 'blue';          // ჩარჩოს ფერი
let clBgInact = '#83d6f5';      // ფონის ფერი
let clBgAct = '3d6f4fe';        // ოფციის ფონის განათების ფერი
let clFnInact = 'blue';         // ჩვეულებრივი წარწერის ფერი
let clFnAct = 'black';          // წარწერის ფერი ოფციის
    განათების დროს
let cFontSize = '18';           // შრიფტის ზომა
let cFontFamily = 'times';      // შრიფტის ტიპი
let closeTimeout = 500;         // მენიუს შეყოვნების დრო, მლ.წმ
let selfPos = false;  // მენიუს ოფციის პოზიციონირება
// თვით მენიუ
let menu = new Array ( );
menu [0] = new Array ( );
menu [0] [0] = new Array ("Menu 1", "", "Choose something from the
    submenu", 50, 10, 80, 50, 120, 120);
menu [0] [1] = new Array ("Submenu 1.1", "javascript:history.go(-1)",
    "Back");
menu [0] [2] = new Array ("Submenu 1.2", "http://www.yahoo.com",
    "Yahoo")
menu [0] [3] = new Array ("Submenu 1.3", " javascript:alert('Hello!
    ')", "Welcome Window");
menu [1] = new Array ( );
menu [1] [0] = new Array ("Menu 2", "", "", 130, 10, 80, 130, 120, 0);
menu [1] [1] = new Array ("Submenu 2.1", "", "");
```

```
menu [1] [2] = new Array ("Submenu 2.2", "", "");
menu [2] = new Array ( );
menu [2] [0] = new Array ("Menu 3", "http://www.Internet.ge",
    "INTERNET.GE", 210, 10, 80, 0, 0, 0);
```

menu_bld.js ფაილის კოდი:

```
let ie = document . all ? true ; false;
let overBox = ' ';
let timerID;
let barHtml = new Array ( );
barHtml [0] = '<DIV ID = "divbarpos';
barHtml [1] = ' " STYLE = "position:absolute; left:';
barHtml [2] = 'px; top:';
barHtml [3] = 'px; " onmouseover = " openbox ( ';
barHtml [4] = ') " onmouseout = " closebox ( ';
barHtml [5] = ') " onclick = " clickbox ( ';
barHtml [6] = ') "><table cellpadding=0 cellspacing=0><tr><td
    bgcolor= " ';
barHtml [7] = ' "><table cellpadding= "0" cellspacing= "1" border =
    "0"><tr><td class = "mnubarpos" id = "mnubarpos';
barHtml [8] = ' " width = " ';
barHtml [9] = ' " bgcolor = " ';
barHtml [10] = ' " style = "color:';
barHtml [11] = '; font-size: ' + cFontSize + '; font-family: ' +
    cFontFamily + '; ">';
barHtml [12] = '</td></tr></table></td></<tr></table></div>';
var boxHtml = new Array ( );
boxHtml [0] = '<div id = "divbox';
```

```
boxHtml [1] = ' " style = "position:absolute; visibility:hidden; left:';
boxHtml [2] = ' px; top:';
boxHtml [3] = ' px; font-family: ' + cFontFamily + '; " onmouseout =
    " closebox ( ';
boxHtml [4] = ') "><table cellpadding=0 cellspacing=0><tr><td
    bgcolor= " ';
boxHtml [5] = ' "><table class = "mnubox" ID = "mnubox';
boxHtml [6] = ' " cellpadding= "0" cellspacing= "1" border = "0" > ';
boxHtml [7] = '<span id = "divboxpos ';
boxHtml [8] = ' " onmouseover = "openboxpos ( ';
boxHtml [9] = ') " onmouseout = "closeboxpos ( ';
boxHtml [10] = ') " onclick = "clickboxpos ( ';
boxHtml [11] = ') "><tr><td class = "mnuboxpos " id = "mnuboxpos ';
boxHtml [12] = ' " width = " ';
boxHtml [13] = ' " bgcolor = " ';
boxHtml [14] = ' " style = "color: ';
boxHtml [15] = '; font-size: ' + cFontSize + ' "> ';
boxHtml [16] = '</td></tr></span>;
boxHtml [17] = '</table></td></tr></table></div>;
function buildMenu ( ) {      // მენიუს აგება
    if (ie) {
        buildMenuBar ( );
        buildSubMenu ( );
        if (selfPos) PosMenu ( );
        ResizeSubMenu ( );
    }
}
function buildMenuBar ( ) {    // ჰორიზონტალური მენიუს აგება
```

```
    for (i = 0; i < menu . length; i++)  {
let s = barHtml [0] + i + barHtml [1] + menu [i] [0] [3] + barHtml [2]
    + menu [i] [0] [4] + barHtml [3] + i + barHtml [4] + i + barHtml
    [5] + i + barHtml [6] + clBorder + barHtml [7] + i + barHtml [8] +
    menu [i] [0] [5] + barHtml [9] + clBgInact + barHtml [10] +
    clFnInact + barHtml [11] + menu [i] [0] [0] + barHtml [12];
        document . writeln (s);
    }
}
function buildSubMenu ( )  {              // ქვემენიუს აგება
    for (i = 0; i < menu . length; i++)  {
        if (menu[i] . length > 1)  {
let s = boxHtml [0] + i + boxHtml [1] + menu [i] [0] [6] + boxHtml [2]
    + menu [i] [0] [7] + boxHtml [3] + i + boxHtml [4] + clBorder+
    boxHtml [5] + i + boxHtml [6];
        for (j = 1; j < menu [i] . length; j++)  {
            let s1 = i + ' , ' + j;
            let s2 = i + ' _ ' + j;
s += boxHtml [7] + s2 + boxHtml [8] + s1 + boxHtml [9] + s1 +
    boxHtml [10] + s1 + boxHtml [11] + s2+ boxHtml [12] + menu [i]
    [0] [8] + boxHtml [13] + clBgInact + boxHtml [14] + clFnInact +
    boxHtml [15] + menu [i] [j] [0] + boxHtml [16];
        }
        s += boxHtml [17];
        document . writeln (s);
    }
   }
}
```

```
function PosMenu ( ) {
    for (i = 0; i < menu . length; i++) {
        barCurr = document . all ['divbarpos' + i ];
        if (i > 0) {
            barPrev = document . all ['divbarpos' + (i - 1) ];
            barCurr . style . left = barPrev . offsetLeft + barPrev .
    offsetWidth – 1;
        }
        if (menu [i] . length > 1) {
            boxCurr = document . all ['divbox' + 1 ];
            boxCurr . style . pixelTop = barCurr . offsetTop + barCurr
    . offsetHeight – 1;
            boxCurr . style . pixelLeft = barCurr . offsetLeft;
        }
    }
}
function ResizeSubMenu ( ) {
    for (i = 0; i < menu . length; i++) {
        if (menu [i] . length > 1 && menu [i] [0] [8] <= 0) {
                el = document . all ['mnubox' + i];
                w = document . all ['divbarpos' + i] . offsetWidth;
                if (el . offsetWidth < w) el . style . width = w;
        }
    }
}
function openbox (x) {
    paintLayer ('mnubarpos' + x, active = true);
    showLayer ('divbox' + x);
```

```
    for (i = 0; i < menu . length; i++)  if (i != x) closebox (i, 0);
    overBox = 'divbox' + x;
    window . status = menu [x] [0] [2];
}
function closebox (x, timeout)  {
    paintLayer ('mnubarpos' + x, active = false);
    clearTimeout (timerID);
    if (timeout ++ 0) hideLayer ('divbox' + x);
        else timerID = setTimeout ('hideLayer ("divbox' + x + ' ")',
    closeTimeout);
    overBox = ' ' ;
    window . status = defaultStatus;
}
function clickbox (x)  {
    clickMenu (menu [x] [0] [1] );
}
function openboxpos (x, y)  {
    paintLayer ('mnuboxpos' + x + '_' + y, active = true);
    overBox = 'divbox' + x;
    window . status = menu [x] [y] [2];
}
function closeboxpos (x, y)  {
    window . status = defaultStatus;
    paintLayer ('mnuboxpos' + x + '_' + y, active = false);
}
function clickboxpos (x, y)  {
    clickMenu (menu [x] [y] [1] );
}
```

```
function clickMenu (s)  {
    if (s . indexOf ('javascript:') == 0) eval (s);
        else if (s != ' ') window . location . href = s;
}
function paintLayer (layerID, active)  {
    if (layer = document . all [layerID])  {
        if (active)  { clBg = clBgAct; clFn = clFnAct ;}
                else { clBg = clBgInact; clFn = clFnInact; }
        layer . style . backgroundColor = clBg;
        layer . style . color = clFn;
    }
}
function showLayer (layerID)  {
    if (layer = document . all [layerID] )  layer . style . visibility =
    'visible';
}
function hideLayer (layerID)  {
    if (layer = document . all [layerID] && (overBox != layerID) );
        document . all [layerID] > style . visibility = 'hidden';
}
```

ყურადღება უნდა მიექცეს იმას, რომ მენიუსა და ქვე-მენიუების პოზიციონირება ერთმანეთისაგან დამოუკიდებლად ხდება. ეს საშუალებას იძლევა მენიუ ეკრანზე ვიზუალურად წარმოვადგინოთ სივრცეში გაბნეული ნაწილების სახით. ამგვარად, რამდენიმე მენიუს გასაკეთებლად აუცილებელი არ არის თითოეული მათგანისათვის შევქმნათ და ჩავტვირთოთ სპეციალური ფაილები აღწერებით.

## ძებნის ოპერაციები ტექსტურ არეში

Web-გვერდებზე განთავსებული დიდი მოცულობის ტექსტები, ჩვეულებრივ საძიებო სისტემითაა აღჭურვილი. გარეგნულად იგი გამოიყურება როგორც საძიებო ტექსტის შესატანი ველი და ღილაკი, რომელზეც მაუსის დაწკაპუნებით ჩართავს ძიების პროცედურას. ძებნის უმარტივესი ვარიანტის დროს ეს პროცედურა ფანჯარაში ტექსტს ისე გადაფურცლავს, რომ ნაპოვნი საძიებო ტექსტი გამოჩნდეს და იყოს მონიშნული. თუ ძებნა უშედეგოდ დამთავრდა, მაშინ ტექსტი ფანჯარაში არ იცვლის მდებარეობას და შესაძლებელია შესაბამისი შეტყობინებაც გამოჩნდეს.

ტექსტში ძებნის პროცესის გადასაწყვეტად გამოიყენება ობიექტი TextRange. ეს ობიექტი ხელმისაწვდომს ხდის ტექსტურ ინფორმაციას, რომლებიც განთავსებულია <body>, <textarea>, <button> და <input type = "text"> ტეგებში მოთავსებულ ობიექტებში. ქვემოთ მოყვანილია ძებნის მაგალითი. იგი შეიცავს myfind() ფუნქციას, რომელიც ღილაკზე მაუსის დაწკაპუნებით იწყებს ძებნის ოპერაციას. ეს ფუნქცია createTextRange() მეთოდით ქმნის საძიებო ტექსტურ ველს. შემდეგ findText() მეთოდით ხდება ტექსტში ძებნა, და ბოლოს scrollIntoView() მეთოდით ფანჯარა ისე გადაიფურცლება, რომ ნაპოვნი ტექსტი გამოჩნდეს. select() მეთოდის დახმარებით მოხდება ნაპოვნი ტექსტის სხვა ფერით გამოყოფა.

```
<!DOCTYPE html>
<html>
<body>
<input type ="text" name ="word" value ="" size =20>
```

```
<button onclick = "mufind ( ) ">Search</button>
<p>
<! Here is a text that search >
</body>
<script>
function myfind ( ) {
obj = document . body . createTextRange ( );
obj . findText (WORD . value);
obj . scrollIntoView ( );
obj . select ( );
}
</script>
</html>
```


თუ ძებნა უშედეგოდ დამთავრდა, მაშინ არაფერი არ მოხდება. მაგრამ თუ მომხმარებელმა საძიებო ტექსტის შეყვანის გარეშე ხელი დააჭირა ღილაკს, მაშინ წარმოიშვება შეცდომა. ეს რომ არ მოხდეს საჭიროა ნაწილობრივ შეიცვალოს პროგრამა შემდეგნაირად:

```
function myfind ( ) {
if (!WORD . value) return;
```

```
obj = document . body . createTextRange ( );
obj . findText (WORD . value);
obj . scrollIntoView ( );
obj . select ( );
}
```

განვიხილოთ ვარიანტი, როდესაც ეკრანზე ტექსტთან ერთად გვაქვს მხოლოდ ღილაკი Search. მასზე დაწკაპუნებით ეკრანზე გამოჩნდება ფორმა, სადაც მომხმარებელი ჩაწერს საძიებო ტექსტს. თუ ეს ველი ცარიელი არ იქნება, მაშინ განხორციელდება ძებნა, ხოლო ცარიელი ველის შემთხვევაში არაფერი არ მოხდება.

```
function myfind ( ) {
WORD = prompt ("Enter that finding: ", "");
if (!WORD) return;
obj = document . body . createTextRange ( );
obj . findText (WORD);
obj . scrollIntoView ( );
obj . select ( );
}
```

ქვემოთ მოყვანილია ძებნის მაგალითი არა დოკუმენტში, არამედ <textarea> ტეგით მოცემულ ტექსტურ არეში.

```
<!DOCTYPE html>
<html>
<input type = "text" name = "word" value = "" size = 20 >
<button onclick = "mufind ( ) ">Search</button>
<p>
```

```
<textarea id = "mytext">
<! Here is a text that search >
</textarea>
<script>
function myfind ( ) {
if (!WORD . value) return;
obj = document . all . mytext . createTextRange ( );
obj . findText (WORD . value);
obj . scrollIntoView ( );
obj . select ( );
}
</script>
</html>
```


## ცხრილები

ცხრილები Web-დიზაინში ერთ-ერთ ყველაზე ხშირად გამოსაყენებელი ელემენტია. ეს განპირობებულია ცხრილების შესაქმნელი ტეგების, როგორიცაა <table>, <tr>, <td> და სხვა,

გამოყენების სიმარტივითა და მოხერხებულობით. ჩვენ ფანჯრის მთელი არე უნდა დავყოთ მართკუთხა უჯრედებად ხილული და უხილავი საზღვრებით და მათში განვათავსოთ დოკუმენტის ელემენტები (გამოსახულებები, ტექსტები, მიმართვები, ღილაკები, სხვა ცხრილები და ა. შ.). ამგვარად, ცხრილები ასრულებს დოკუმენტის კარკასის მოვალეობას.

## ცხრილის ელემენტებზე მიმართვა

ცხრილს, როგორც დოკუმენტის ობიექტს აქვს ორი კოლექცია, რომლის საშუალებითაც მის შიგთავსზე ხდება მიმართვა. მათ შორის პირველი არის სტრიქონების კოლექცია rows, ხოლო მეორე – უჯრების კოლექცია Cells. კოლექცია Rows შეიცავს ცხრილის ყველა სტრიქონს <thead> და <tfoot> ტეგების შესაბამისი განყოფილებების ჩათვლით. კოლექცია cells შეიცავს <th> და <td> ტეგების მიერ შექმნილი ცხრილების ყველა ელემენტს. კოლექციის ელემენტებზე მიმართვა ხორციელდება ან ინდექსის ან შესაბამის ტეგში id ატრიბუტის მნიშვნელობის მიხედვით. ამგვარად, ცხრილის სტრიქონზე მიმართვისათვის შეიძლება გამოვიყენოთ <tr> ტეგში id ატრიბუტის მნიშვნელობა, ხოლო უჯრედზე მიმართვისათვის <th> ან <td> ტეგებში id ატრიბუტის მნიშვნელობა. ინდექსის (ნომერის) გამოყენების შემთხვევაში უნდა გვახსოვდეს, რომ ნუმერაცია იწყება 0-დან. ამასთან, ცხრილის უჯრედების ნუმერაცია ხდება მარცხნიდან მარჯვნივ და ზემოდან ქვემოთ.

განვიხილოთ მარტივი ცხრილის მაგალითი, რომელიც შეიცავს სამ სვეტსა და ოთხ სტრიქონს:

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

```
<table id = "mytab">
<thead>MY TABLE</thead>
<th> Surname </th><th> Name</th><th> Position</th>
<tr  id = "r1">
<td>ბერიძე</td><td>გიორგი</td><td>Director</td>
</tr>
<tr  id = "r2">
<td>მელაძე</td><td>თამაზი</td><td>Deputy Director</td>
</tr>
<tr  id = "r3">
<td>დვალი</td><td>მანანა</td><td>Secretary</td>
</tr>
<tr  id = "r4">
<td>ქერქაძე</td><td>ირაკლი</td><td>Driver</td>
</tr>
</table>
</html>
```

MY TABLE

| Surname | Name | Position |
|---|---|---|
| ბერიძე | გიორგი | Director |
| მელაძე | თამაზი | Deputy Director |
| დვალი | მანანა | Secretary |
| ქერქაძე | ირაკლი | Driver |

ქვემოთ მოყვანილია ცხრილის ელემენტებზე მიმართვის რამდენიმე მაგალითი:

```
document . all . mytab . rows [0]
document . all . mytab . rows [1]
document . all . mytab . rows ["r1"]
document . all . mytab . cells [3]
document . all . mytab . rows [2] . cells [1]
document . all . mytab . rows ["r1"] . cells [2]
```

იმისათვის, რომ მივმართოთ ცხრილის სტრიქონს ან უჯრედს საჭიროა გავითვალისწინოთ ობიექტის იერარქია: ჯერ ჩვენ მივმართავთ დოკუმენტის ყველა ელემენტების all კოლექციას და მხოლოდ ამის შემდეგ – ცხრილის ელემენტებს. ყურადღება უნდა მიექცეს იმ გარემოებას, რომ ეს მიმართვები აბრუნებს არა ცხრილის ელემენტების შიგთავსს, არამედ მხოლოდ მიმართვას ელემენტებზე, როგორც ობიექტებზე.

ზოგჯერ საჭირო ხდება ცხრილის ელემენტების შეცვლა ახალი მნიშვნელობით ან გამოსახულებით, რაც შესაძლებელია შემდეგნაირად:

```
document . all . mytab . rows [ინდექსი1] . cells [ინდექსი 2] .
    innerText = "ახალი მნიშვნელობა"
document . all . mytab . rows [ინდექსი1] . cells [ინდექსი 2] .
    innerHTML = "<IMG SRC = 'pict.jpg'>"
```

სტრიქონებისა და უჯრედების კოლექციას, ისევე როგორც ნებისმიერ მასივს აქვს length თვისება, რომლის მნიშვნელობა კოლექციაში ელემენტების რაოდენობაა:

```
document . all . mytab . rows . length
```

```
document . all . mytab . cells . length
document . all . mytab . rows [2] . cells . length
```

## ცხრილში სტრიქონის დამატება და წაშლა

ცხრილში ახალი სტრიქონის დამატება insertRow() მეთოდის დახმარებით ხდება. ეს მეთოდი მიმართვას აბრუნებს ახალ შექმნილ სტრიქონზე, რომელიც შემდეგ გამოიყენება უჯრედების ჩასასმელად. უჯრედები სტრიქონში ისმება insertCell (უჯრედის ინდექსი) მეთოდის საშუალებით. მოცემული მეთოდი აბრუნებს მიმართვას ახალ შექმნილ უჯრედზე, რომელიც შემდგომ გამოიყენება უჯრედის შიგთავსის მოსაცემად. უჯრედები სტრიქონში ისმება მიმდევრობით, გამოტოვების გარეშე დაწყებული ნულიდან. მაგალითად,

```
newrow = document . all . mytab . insertRow ( )
newcell = newrow . insertCell (0)
newcell . innerText = "Hello"
newcell = newrow . insertCell (1)
newcell . innerHTML = "<img src = 'pict.jpg'>"
newcell = newrow . insertCell (2)
newcell . innerHTML = "<b>guard</b>"
```

ცხრილის სტრიქონის წასაშლელად გამოიყენება deleteRow (სტრიქონის ინდექსი) მეთოდი. პარამეტრი მიუთითებს წასაშლელი სტრიქონის ნომერს. მაგალითად,

```
document . all . mytab . deleteRow (2)
```

## ცხრილების გენერაცია სცენარის დახმარებით

HTML ტეგების საშუალებით ცხრილების შექმნაში ერთ-ერთ ყველაზე უხერხული მომენტი ამ ტეგების დიდი რაოდენობა არის. ამასთან დიდი ყურადღებაა საჭირო მათი სწორად განლაგებისა და კორექტირების დროს. საქმეს ძლიერ ამარტივებს უჯრების შიგთავსის შენახვა მასივების სახით და ცხრილების გენერაცია სცენარის დახმარებით. ქვემოთ მოყვანილია ამის მაგალითი:

```
<!DOCTYPE html>
<html>
<script>
ah = new Array ("Surname", "Name", "Position");
ad = new Array ( );
    ad[0] = new Array ("ბერიძე","გიორგი","Director");
    ad[1] = new Array ("მელაძე","თამაზი","Deputy Director");
    ad[2] = new Array ("დვალი","მანანა","Secretary");
    ad[3] = new Array ("ქერქაძე","ირაკლი","Driver");
strtab = "<table>";
for (i = 0; i < ah . length; i++)  {
    strtab += "<th>" + ah [i] + "</th>";
}
for (i = 0; i < ad . length; i++)  {
    strtab += "<tr>";
    for (j = 0; j < ad [i] . length; j++)  {
        strtab += "<td>" + ad [i] [j] + "</td>";
    }
    strtab += "</tr>";
```

```
}
strtab += "</table>";
document . write (strtab);
</script>
</html>
```

| Surname | Name | Position |
|---|---|---|
| ბერიძე | გიორგი | Director |
| მელაძე | თამაზი | Deputy Director |
| დვალი | მანანა | Secretary |
| ქერქაძე | ირაკლი | Driver |

## მონაცემთა მარტივი ბაზები

ცხრილების ფორმირების ერთ-ერთი ეფექტური საშუალება ეს არის მართვის სპეციალური ელემენტი ActiveX - Simple Tabular Data (STD - მარტივი ცხრილური მონაცემები). ეს ელემენტი დოკუმენტში ჩაიწერება <object> ტეგის დახმარებით და საშუალებას იძლევა მარტივად ვმართოთ ჩვეულებრივ ტექსტურ ფაილში ჩაწერილი მონაცემები. STD-ს გამოყენებით შეგვიძლია შევცვალოთ, დავუმატოთ, წავშალოთ, მოვაწესრიგოთ, მოვძებნოთ და ამოვარჩიოთ (მოვახდინოთ ფილტრაცია) მონაცემები, მაგრამ მისი მთავარი ღირსება არის დიდი ცხრილების მარტივად შექმნა.

მონაცემები ამ დროს ინახება დისკზე ტექსტური ფაილების სახით. ამასთან, მათი ცხრილური სტრუქტურა დაცულია გამყოფი-სიმბოლოს საშუალებით. სტრიქონების გამყოფად ჩვეულებრივ გამოიყენება სტრიქონის გადაყვანის სიმბოლო (კლავიში <Enter>). ცალკეულ უჯრედებში მდგომი მონაცემების გამოსაყოფად გამოიყენება ნებისმიერი სიმბოლო, რომელიც სხვა მიზნებისათვის არ გამოიყენება ან არ გვხვდება მონაცემებში (მაგალითად, მძიმე ან ვერტიკალური ხაზი). ასეთი ფაილების შექმნა შესაძლებელია ნებისმიერი ტექსტური რედაქტორით (მაგალითად, Notepad++).

ტექსტური ფაილის პირველ სტრიქონში სვეტების დასახელებებია მოცემული. მაგალითად,

```
Surname|Name|Position
Beridze|Giorgi|Director
Meladze|Tamaz|Deputy Director
Dvalishvili|Manana|Secretary
Kerkadze|Irakly|Driver
```

STD - მონაცემთა მართვის ელემენტი ActiveX, ჩაშენებულია ბრაუზერში, ხოლო მისი დოკუმენტში ჩასასმელად საჭიროა შემდეგი ტეგების გამოყენება:

```
<object id = "mydbcontrol" classid =
    "CLSID:333C7BC4-460F-11D0-BC04-0080C7055A83">
    <param name = "FieldDelim" value = "|">
    <param name = "DataURL" value = "mydb.txt">
    <param name = "UseHeader" value = true>
    <param name = "RowDelim" value = "&# ASCII – kodi;">
```

```
</object>
```

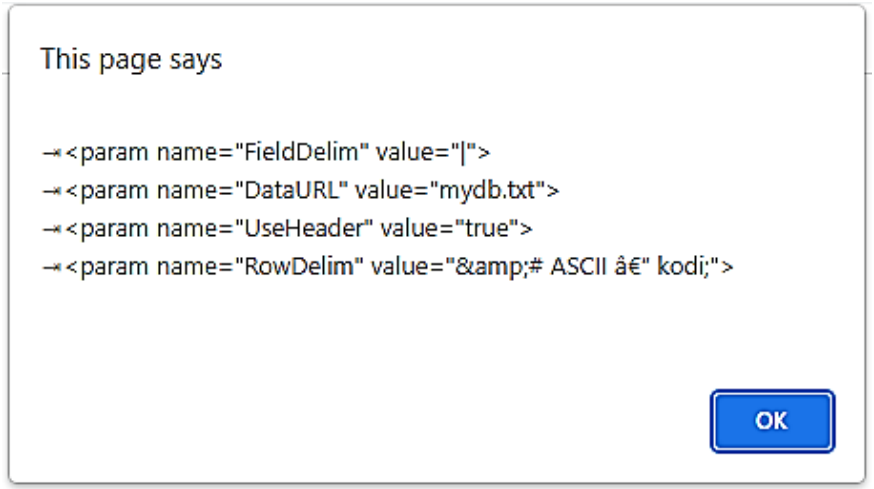
კონტეინერული ტეგი <object> შეიცავს მრავალ <param> ტეგს, რომლის საშუალებითაც მოცემულია პარამეტრები. ქვემოთ მოცემულია ზოგიერთი მათგანის მნიშვნელობა:

- FieldDelim – ველების (უჯრედების) გამყოფი სიმბოლო;
- DataUrl – მონაცემთა ტექსტური ფაილის ადგილმდებარეობა (URL-მისამართი);
- UseHeader – განსაზღვრავს, შეიცავს თუ არა ტექსტური ფაილის პირველი სტრიქონი ველების დასახელებას;
- RowDelim – ტექსტურ ფაილში სტრიქონების გამყოფი სიმბოლო. გაჩუმების პრინციპით value ატრიბუტის მნიშვნელობაა „&#10“, ანუ სტრიქონის გადაყვანის სიმბოლოს კოდი.

გაჩუმების პრინციპით STD-ს მართვის ელემენტების ყველა პარამეტრების დათვალიერების მიზნით საკმარისია ბრაუზერში ჩაიტვირთოს შემდეგი HTML-დოკუმენტი:

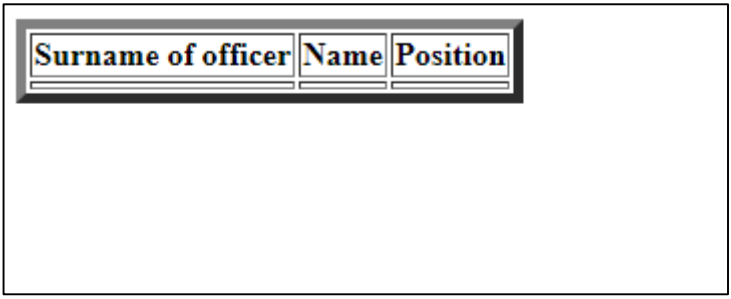
```
<!DOCTYPE html>
<html>
<object id = "mydbcontrol" classid =
    "CLSID:333C7BC4-460F-11D0-BC04-0080C7055A83">
</object>
<script>
alert (mydbcontrol . innerHTML);
</script>
</html>
```

ეკრანზე გამოვა შემდეგი სახის დიალოგური ფანჯარა:




იმისათვის, რომ ბრაუზერის ფანჯარაში მონაცემები mydb.txt ტექსტური ფაილიდან აისახოს ცხრილში საჭიროა იმავე დოკუმენტში ჩაიწეროს შემდეგი კოდი:

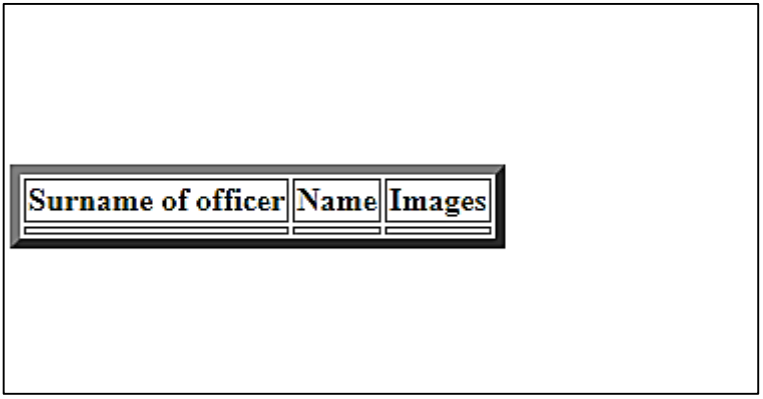
```
<table datasrc = #mydbcontrol border =5>
<thead>
<th>Surname of officer</th><th>Name</th><th>Position</th>
</thead>
<tr>
<td><span datafld = "Surname"></span></TD>
<td><span datafld = "Name"></span></td>
<td><span datafld = "Position"></span></td>
</tr>
</table>
```




მონაცემთა ტექსტურ ფაილში ტექსტური ინფორმაციის გარდა შეიძლება იყოს HTML-კოდიც. იმისათვის, რომ ეს კოდები გამოისახოს ცხრილში არა მარტო როგორც ტექსტი, ამისათვის საჭიროა <td> ტეგში დაემატოს ატრიბუტი dataformatas = "html". ამასთან, თუ ცხრილის უჯრედი არ შეიცავს ტეგებს, მაშინ მასში უბრალო ტექსტი აისახება, წინააღმდეგ შემთხვევაში კი HTML-კოდის შესრულების შედეგი. ეს საშუალებას მოგვცემს ცხრილის უჯრედში ჩავსვათ გამოსახულება, ჰიპერმიმართვა, ღილაკი და სხვა ელემენტი. ქვემოთ მოყვანილია ამის მაგალითი:

```
<!DOCTYPE html>
<html>
<object id = "mydbcontrol" classid =
    "CLSID:333C7BC4-460F-11D0-BC04-0080C7055A83">
    <param name = "FieldDelim" value = "|">
    <param name = "DataURL" value = " mydb.txt">
    <param name = "UseHeader" value = true>
</object>
<table datasrc = #mydbcontrol border = 5>
<thead>
<th>Surname of officer</th><th>Name</th><th>Images</th>
```

```
</thead>
<tr>
<td><span datafld = "Surname"></span></td>
<td><span datafld = "Name"></span></td>
<td><span datafld = "Portrait"  dataformatas = "html"></span> </td>
</tr>
</table>
</html>
```




STD-ს მართვის ელემენტს საშუალება აქვს გამოიყენოს სხვადასხვა სახის პარამეტრები. კერძოდ, არის ფილტრისა (ამორჩევის) და დალაგების (მოწესრიგების) მომართვის პარამეტრები.

ფილტრის მოსამართად გათვალისწინებულია შემდეგი სამი პარამეტრი:

```
<param name = "FilterColumn" value = "ველის სახელი">
<param name = "FilterCriterion" value = "შედარების ოპერატორი">
<param name = "FilterValue" value = "ნიმუში">
```

შედარების ოპერატორად გამოიყენება: =, ==, !=, >, <, >= და <= ოპერატორები.

მაგალითად:

```
<param name = "FilterColumn" value = "Surname">
<param name = "FilterCriterion" value = "==">
<param name = "FilterValue" value = "Beridze">
```

იმისათვის, რომ ფილტრი გამოვრთოთ და მთელი ცხრილი მისაწვდომი გავხადოთ საჭიროა პირობა გამოვიყენოთ, რომელსაც აკმაყოფილებს ცხრილის ყველა სტრიქონი. მაგალითად:

```
<param name = "FilterCriterion" value = "=">
<param name = "FilterValue" value = "">
```

სისტემაში ნაგულისხმევი წესის თანახმად, ფილტრი რეგისტრზეც რეაგირებს, რომელშიც მონაცემებია შეტანილი. ჩვენ შეგვიძლია რეგისტრის მართვაც. ამისათვის შემდეგი პარამეტრი გამოიყენება:

```
<param name = "CaseSensitive" value = "მნიშვნელობა">
```

value ატრიბუტს შეუძლია მიიღოს მნიშვნელობები 0/false (არ არის დამოკიდებული) ან 1/true (დამოკიდებულია).

მონაცემთა დასალაგებლად (მოსაწესრიგებლად) შემდეგი პარამეტრი გამოიყენება:

```
<param name = "SortColumn" value = "ველის სახელი">
```

სისტემაში ნაგულისხმევი წესის თანახმად, დალაგება ხდება სიმბოლოთა კოდის ზრდადობის მიხედვით, მაგრამ ჩვენ შეგვიძლია იგი შემდეგი პარამეტრით შევცვალოთ:

```
<param name = "SortAscending" value = 0>
```

საწყის მდგომარეობას დავუბრუნდებით როცა value = 1.

ცხრილის სტრიქონებს შორის გადაადგილების საშუალებას გვაძლევს recordset ობიექტი. მისი სინტაქსია:

```
ობიექტის id . recordset . მეთოდი
```

მონაცემთა ბაზის ჩანაწერებს შორის გადაადგილების მეთოდებია:

- movePrevios() - წინა ჩანაწერზე გადასვლა;
- moveNext() - მომდევნო ჩანაწერზე გადასვლა;
- moveFirst() - პირველ ჩანაწერზე გადასვლა;
- moveLast() - ბოლო ჩანაწერზე გადასვლა;

მომდევნო moveNext() და movePrevios() წინა ჩანაწერზე გადასვლამდე უნდა შემოწმდეს eof (მონაცემთა ფაილის დასასრული) და bof (მონაცემთა ფაილის დასაწყისი) თვისებათა მნიშვნელობა. ვინაიდან, თუ მიმდინარე სტრიქონი არის ბოლო, მაშინ არ შეიძლება moveNext() მეთოდის გამოყენება და ანალოგიურად, თუ მიმდინარე სტრიქონი არის პირველი, მაშინ არ შეიძლება movePrevios() მეთოდის გამოყენება.

ქვემოთ მოყვანილია HTML-კოდის მაგალითი, სადაც ეკრანზე გამოდის მონაცემთა ბაზის ერთი ჩანაწერი ნავიგაციის ღილაკებით:

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

```
<header><title> Example forms of STD</title></header>
<object id = "mydbcontrol" classid =
    "CLSID:333C7BC4-460F-11D0-BC04-0080C7055A83">
    <param name = "FieldDelim" value = "|">
    <param name = "DataURL" value = " mydb.txt">
    <param name = "UseHeader" value = true>
</object>
<!  Field Data>
<table width = 75%>
<tr><th>Surname</th>
<td>
< input type = "text" datasrc = #mydbcontrol datafld = "Surname">
</td></tr>
<tr><th>Name</th>
<td>< input type = "text" datasrc = #mydbcontrol datafld = "Name">
</td></tr>
<tr><th>Portrait</th>
<td><span datasrc = #mydbcontrol datafld = "Portrait"
    dataformatas = "html"></span>
 </td></tr>
</table>
<p>
<! Buttons displaced by recordings >
<input  name = "cmdFirst" type = "button" value = "<<" onclick = "First
    ( ) ">
<input  name = "cmdPrevious" type = "button" value = "<" onclick =
    "Previous ( ) ">
```

```
<input  name = "cmdNext" type = "button" value = ">" onclick =
    "Next ( ) ">
<input  name = "cmdLast" type = "button" value = ">>" onclick =
    "Last ( ) ">
<script>
function First ( ) {
mydbcontrol . recordset . moveFirst ( );
}
function Previous ( ) {
if (!mydbcontrol . recordset . bof);
mydbcontrol . recordset . movePrevious ( );
}
function Next ( ) {
if (!mydbcontrol . recordset . eof)
mydbcontrol . recordset . moveNext ( );
}
function Last ( ) {
mydbcontrol . recordset . moveLast ( );
}
</script>
</html>
```

## ცხრილის მონაცემთა დალაგება (მოწესრიგება)

განვიხილოთ ცხრილის სტრიქონების დალაგება ამა თუ იმ სვეტის მნიშვნელობის მიხედვით. ჩვენს მაგალითში მონაცემთა მოწესრიგების მიზნით საკმარისია მაუსით დავაწკაპუნოთ ცხრილის საჭირო სვეტის სათაურზე.

```
<!DOCTYPE html>
<html>
<object id = "mydbcontrol" classid =
    "CLSID:333C7BC4-460F-11D0-BC04-0080C7055A83">
    <param name = "FieldDelim" value = "|">
    <param name = "DataURL" value = " mydb.txt">
    <param name = "UseHeader" value = true>
    <param name = "SortColumn" value = "Surname">
    <param name = "SortAscending" value = 1>
</object>
<table  datasrc = #mydbcontrol  border = 5>
<thead >
<th  onclick = "sort ('Surname')">Surname of officer</th>
<th  onclick = "sort ('Name')">Name</th>
<th>Portrait</th>
</thead >
<tr>
<td><span datafld = "Surname"></span></td>
<td><span datafld = "Name"></span></td>
<td><span  datafld = "Portrait" dataformatas = "html"> </span>
</td></tr>
</table>
<script>
let x = document . all . mydbcontrol . innerHTML;
let obj = '<object id = "mydbcontrol" classid =
    "CLSID:333C7BC4-460F-11D0-BC04-0080C7055A83">' + x;
function  sort (field)  {
let  y = document . all . mydbcontrol;
```

```
y . outerHTML = obj + '<param  name = "SortColumn"  value = " ' +
    field + ' ">;
</object>
}
</script>
</html>
```

მოცემულ მაგალითში მონაცემების დალაგება ტექსტური ველების მიხედვით ზრდადობით ხდება. იგივე მაგალითი შეიძლება ისე გადავაკეთოთ, რომ მონაცემების დალაგება განხორციელდეს საწინააღმდეგო მიმდევრობით, რისთვისაც სცენარში უნდა ჩავწეროთ <param name = "SortAscending" value = 0> პარამეტრი.

## ცხრილის მონაცემთა ფილტრაცია

მოცემული კრიტერიუმით ცხრილის მონაცემების ფილტრაციისათვის მოვიყვანოთ მაგალითი, სადაც ბრაუზერის ფანჯარაში გამოტანილია ცხრილი მონაცემებით და ელემენტებით, რომლის საშუალებითაც ფილტრის მარტივი პირობის ფორმირებისათვის პროგრამას შეიძლება მივაწოდოთ ველი (სვეტი) და მნიშვნელობა, შემდეგი სახით:

```
ველის სახელი = მნიშვნელობა
```

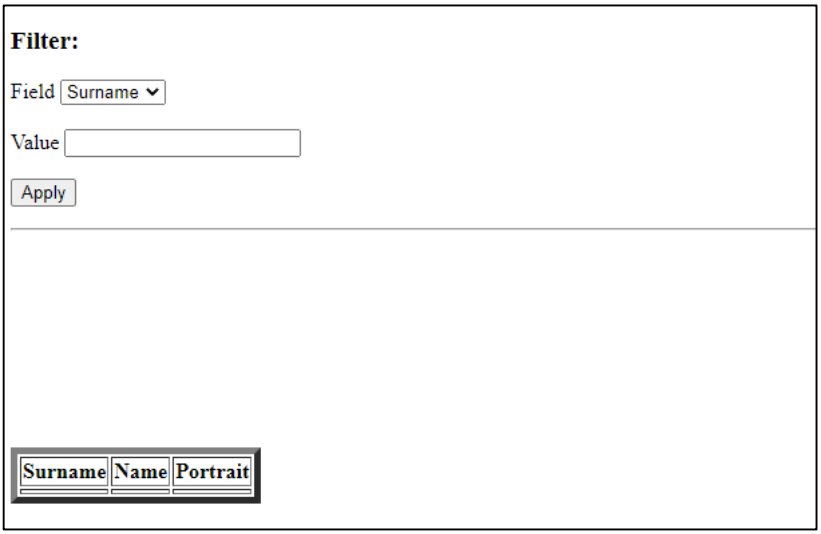
ველის სახელი ამოირჩევა ჩამოშლადი სიიდან, ხოლო მნიშვნელობა კლავიატურის დახმარებით შეგვყავს. ფილტრის დასაყენებლად მაუსით უნდა დავაწკაპუნოთ Apply ღილაკზე. თუ მნიშვნელობის შეტანის ველი ცარიელია, მაშინ ამ ღილაკზე დაწკაპუნება გამოიწვევს მთელი ცხრილის გამოტანას. აქვე უნდა

აღინიშნოს, რომ ჩვენს მაგალითში შედეგი არ არის დამოკიდებული რეგისტრზე.

```
<!DOCTYPE html>
<html>
<h3>Filter:</h3>
Field
<! A drop-down list the field names >
<select  name = "FLD">
<option  value = "Surname">Surname
<option  value = "Name">Name
</select>
<br>
Value
<! The input field values to filter button >
<input  name = "INP" value = " "  type = "text">
<p>
<button  onclick = "filter ( ) "> Apply</button>
<hr>
<! Control STD >
<object  id  =  "mydbcontrol"  classid  =
    "CLSID:333C7BC4 460F 11D0 BC04 0080C7055A83">
    <param name = "FieldDelim" value = "|">
    <param name = "DataURL" value = "mydb.txt">
        <param name = "UseHeader" value = true>
</object>
<! Table to display the data >
<table  datasrc = #mydbcontrol  border = 5>
<thead >
```

```
<th>Surname</th>
<th>Name</th>
<th>Portrait</th>
</thead >
<tr>
<td><span datafld = "Surname"></span></td>
<td><span datafld = "Name"></span></td>
<td><span  datafld = "Portrait"  dataformatas = "html"></span>
</td></tr>
</table>
<script>
let obj = "<object  id = 'mydbcontrol'  ";
obj      +=      "classid      =      'CLSID:333C7BC4-460F-11D0-BC04-
    0080C7055A83'>";
obj += "<param name = 'FieldDelim' value = ' | '>";
obj += "<param name = 'DataURL' value = 'mydb.txt'>";
obj += "<param name = 'UseHeader' value = true>";
function  filter ( )  {
let  cpar = " ";
if (INP . value)  {
    cpar = "<param name = 'FilterColumn'  value = ' " + FLD . value +
    " ' >";
    cpar += "<param name = 'FilterValue'  value = ' " + INP . value + "
    ' >";
    cpar += "<param name = 'FilterCriterion'  value = '=' >";
    cpar += "<param  name = 'CaseSensitive'  value = false >";
    }
document . all . mydbcontrol . outerHTML = obj + cpar + "</object>";
```

```
}
</script>
</html>
```




## ძებნა საიტზე

საძიებო სისტემის საფუძველია მონაცემთა ბაზა, რომელიც შეიცავს საძიებო მონაცემებსა (გამხსნელი სიტყვა) და Web-გვერდზე მიმართვებს შორის დამოკიდებულებას. ეს საძიებო მონაცემთა ბაზა ჩვეულებრივ ტექსტურ ფაილში იქმნება, რომელსაც ჩვენს მაგალითში search.txt ჰქვია. მასში არის მხოლოდ ორი სვეტი p1 და p2. პირველი სვეტი შეიცავს გამხსნელ სიტყვებს, ხოლო მეორე – იმ HTML-დოკუმენტებზე მიმართვებს, რომლებშიც არის შესაბამისი გამხსნელი სიტყვები. ტექსტურ ფაილში ველების გამყოფად გამოიყენება ვერტიკალური ხაზი. ამგვარად, მონაცემთა ბაზის სტრუქტურას აქვს შემდეგი სახე:

```
p1|p2
გამხსნელი სიტყვა|მიმართვა
```

უპირველეს ყოვლისა, კარგად უნდა იყოს გააზრებული გამხსნელი სიტყვების სისტემა, რადგან ძებნა იყოს ეფექტური. შემდეგ უნდა დამუშავდეს საძიებო სისტემის მუშაობის ალგორითმი. კერძოდ, უნდა გადაწყდეს ძებნა დამოკიდებული იქნება თუ არა შეტანილი საძიებო სახის რეგისტრზე, ძებნის შედეგად გამოიტანოს თუ არა პირველივე ნაპოვნი დოკუმენტი, თუ გამოიტანოს ყველა ნაპოვნ დოკუმენტზე მიმართვა და ა. შ.

```
<!DOCTYPE html>
<html>
<! The control for the table of links >
<object id = "fnd1" classid =
    "CLSID:333C7BC4-460F-11D0-BC04-0080C7055A83">
    <param name = "FieldDelim" value = "|">
    <param name = "DataURL" value = "search.txt">
    <param name = "UseHeader" value = true>
    <param name = "CaseSensitive" value = false>
</object>
<! The control for the keyword table >
<object id = "fnd2" classid =
    "CLSID:333C7BC4-460F-11D0-BC04-0080C7055A83">
    <param name = "FieldDelim" value = "|">
    <param name = "DataURL" value = " search.txt">
    <param name = "UseHeader" value = true>
    <param name = "CaseSensitive" value = false>
    <param name = "CharSet" value = "windows-1251">
```

```
</object>
<! The interface of the search engine >
<! Field input image >
<input  type ="text" Name ="WORD" value ="" size =20>
<! Button start searching >
<button  onclick = "findkeyword ( ) ">Search</button><br>
<! It will appear in the search results >
<b  id = "rezult"></b>
<! Table of keywords >
<table  style = "visibility : hidden">
<tr>
<td><input  name = "f1" datasrc = #fnd2  datafld = "p1"> </td>
</tr>
</table>
<! Table of links >
<table  id = "mytab" datasrc = #fnd1  width = 350  align = left  style =
    "visibility : hidden">
<tr>
<td><span  datafld = "p2" dataformatas = "html"> </span></td>
</tr>
</table>
<script>
let obj = "<object  id = 'fnd1'  ";
obj     +=     "classid     =     'CLSID:333C7BC4-460F-11D0-BC04-
    0080C7055A83'>";
obj += "<param name = 'FieldDelim' value = ' | '>";
obj += "<param name = 'DataURL' value = 'search.txt'>";
obj += "<param name = 'UseHeader' value = true>";
```

```
obj += "<param name = "CaseSensitive" value = false>";
function  findkeyword()  {
if (!WORD . value) return;
let  xfltr = "" . y . cpar;
fnd2 . recordset . moveFirst()
while (!fnd2 . recordset . eof)  {
     y = f1 . value . toLowerCase()
     if (y . indexOf (WORD . value . toLowerCase() ) >= 0)  {
          xfltr = f1 . value;
                    break;
     }
     fnd2 . recordset . moveNext();
}
if (!xfltr)
          document . all . rezult . innerText = "Nothing found";
     else
          document . all . rezult . innerText = "";
cpar = obj + "<param name = 'FilterColumn'  value = 'p1' >";
cpar += "<param name = 'FilterCriterion'  value = '=' >";
cpar  += "<param  name  =  'FilterValue'   value  =  '  "  +  xfltr  +  "  '  >
     </object>";
document . all . fnd1 . outerHTML = cpar;
mytab . style . visibility = "visible";
}
</script>
</html>
```

განხილულ მაგალითში ძებნა დამოკიდებული არ არის საძიებო სახის რეგისტრზე, სისტემა სტრიქონ-სტრიქონ

ათვალიერებს მონაცემთა ბაზის პირველი ველის მნიშვნელობას, ამოწმებს შეიცავს თუ არა პირველი ველის მნიშვნელობა მომხმარებლის მიერ შეტანილ საძიებო სახეს. თუ შეიცავს, მაშინ ძებნა წყდება, ხოლო წინააღმდეგ შემთხვევაში ძებნა გრძელდება. ძებნის შედეგად დოკუმენტზე გამოდის მიმართვა. მოცემულ მიმართვაზე მაუსით დაწკაპუნება ამ დოკუმენტს ბრაუზერის ფანჯარაში გამოიტანს. თუ ძებნა უშედეგოდ დასრულდა, მაშინ შესაბამის შეტყობინებას გამოიტანს.

ძებნის ინტერფეისი მარტივია: საძიებო სახის შესატანი ველი და ძებნის პროცედურის დაწყების ღილაკი. შედეგს გამოიტანს შეტანის ველის ქვემოთ.

ჩვენ განვიხილეთ საძიებო სისტემის ყველაზე მარტივი ვარიანტი. მისი სრულყოფა შეიძლება რამდენიმე მიმართულებით განხორციელდეს. ქვემოთ მოვიყვანოთ მაგალითი, სადაც საძიებო სისტემა დაათვალიერებს მონაცემთა ბაზის ყველა ჩანაწერს და შედეგად გამოიტანს საძიებო სახის შესაბამის ყველა მიმართვას. ამ შემთხვევაში მონაცემთა ფილტრაცია ამოვარდება და კოდი უფრო გამარტივდება.

```
<!DOCTYPE html>
<html>
<! Control >
<object id = "fnd1" classid =
    "CLSID:333C7BC4-460F-11D0-BC04-0080C7055A83">
    <param name = "FieldDelim" value = "|">
    <param name = "DataURL" value = "search.txt">
    <param name = "UseHeader" value = true>
    <param name = "CaseSensitive" value = false>
</object>
```

```
<! The interface of the search engine >
<! Field input image >
<input  type ="text" Name ="WORD" value ="" size =20>
<! Button start searching >
<button  onclick = "findkeyword ( ) ">Search</button><br>
<! It will appear in the search results >
<b  id = "rezult"></b>
<! Table searchable database>
<table  style = "visibility : hidden">
<tr>
<td><input  name = "f1" datasrc = #fnd1  datafld = "p1"> </TD>
<td><input  name = "f2" datasrc = #fnd1  datafld = "p2" dataformatas
    = "html"></TD>
</tr>
</table>
<script>
function  findkeyword ( )  {
if (!WORD . value) return;
let  xrezult = "" . y;
fnd1 . recordset . moveFirst();
while (!fnd1 . recordset . eof)  {
    y = f1 . value . toLowerCase();
    if (y.indexOf (WORD.value.toLowerCase() ) >= 0)  {
        xrezult += f2 . value + "<br>";
    }
    fnd1.recordset.moveNext();
}
if (!xrezult)
```

```
            document.all.rezult.innerText = "Nothing found";
    else
            document.all.rezult.innerText = "";
}
</script>
</html>
```

## HTML-დოკუმენტის ჩასმა ცხრილში

დოკუმენტი გარე HTML-ფაილიდან შეიძლება ჩავსვათ ცხრილში. ეს შესაძლებელია იმიტომ, რომ HTML-ფაილი ჩვეულებრივ ტექსტური ფაილია. ვინაიდან ამ ფაილის პირველ სტრიქონში ჩაწერილი <html> ტეგი, ამიტომ ჩვენ იგი შეგვიძლია მონაცემთა ბაზის ველის სახელის სახით გამოვიყენოთ. ამგვარად, ჩასასმელ HTML-ფაილს განვიხილავთ როგორც ერთ ველიან (სვეტი) მონაცემთა ბაზას. ამ ფაილის მიმართ გვექნება მხოლოდ ერთი ძირითადი მოთხოვნა: ნებისმიერი კონტეინერული ტეგი მთლიანად უნდა განთავსდეს ერთ სტრიქონში. დასაწყისში განვიხილოთ ცხრილში ერთი HTML-დოკუმენტის ჩასმის მაგალითი:

```
<!DOCTYPE html>
<html>
<! Control >
<object id = "mydbcontrol" classid =
    "CLSID:333C7BC4-460F-11D0-BC04-0080C7055A83">
    <param name = "FieldDelim" value = "|">
    <param name = "DataURL" value = "mydocum.htm">
```

```
    <param name = "UseHeader" value = true>
</object>
<table  datasrc =#mydbcontrol  width = 350>
<tr>
<td><span  datafld = "<html>" dataformatas = "html"> </span>
</td></tr>
</table>
</html>
```

იგულისხმება, რომ დოკუმენტში <html> ტეგი მთავრული ასოებითაა ჩაწერილი. თუ ის პატარა ასოებითაა ჩაწერილი, მაშინ იგი უნდა აისახოს ატრიბუტ datafld = "<html>" მნიშვნელობაში.

ახლა განვიხილოთ ცხრილში რამდენიმე HTML-დოკუმენტის ჩასმის მაგალითი:

```
<script>
let  aurl = new Array ( );
aurl [0] = "html1.htm";
aurl [1] = "html2.htm";
let  tabwidth = 800;
let  xobject = new Array ( );
xobject [0] = "<object  id = 'idobj" ;
xobject [1] = " 'classid =
    "CLSID:333C7BC4-460F-11D0-BC04-0080C7055A83">';
xobject [1] += '<param name = "FieldDelim" value = "|">';
xobject [1] += '<param name = "UseHeader" value = true>';
xobject [1] += '<param name = "CharSet" value = "windows-1251">';
xobject [1] += '<param name = "DataURL" value = " ';
xobject [2] += ' "></object>';
```

```
let  xtab = new Array ( );
xtab [0] = '<table  width =' + tabwidth + ' datasrc = #';;
xtab  [1]  =  '><tr><td><span   datafld  =  "<html>"  dataformatas  =
    "html">';
xtab [1] += '></span></td></tr></table>';
let  sobj = "";
for (i = 0; i < aurl . length; i ++)  {
sobj += xobject [0] + i + xobject [1] + aurl [i] + xobject [2] + xtab [0] +
    'idobj' + i + xtab [1];
}
document.write (sobj);
</script>
```

საცდელად ავიღოთ შემდეგი ორი მარტივი HTML-დოკუმენტი:

html1.htm ფაილი:

```
<!DOCTYPE html>
<html>
<h3>Document 1</h3>
<image  src = "pict.jpg">This is - just a picture
<a  href = "db0.htm">Description of databases in text files</a>
<button   onclick  =  "alert  ('Hello!!!  ')  ">Click</button>  This  is  a  –
    button
</html>
```

და html2.htm ფაილი:

```
<html>
<h3>Document 2</h3>
```

```
<i>This - the contents of a document from the second file. </i>
It just the text
</html>
```

ჩვენ შეგვიძლია ძირითად დოკუმენტში განვათავსოთ მართვის ელემენტები (მიმართვები, ღილაკები), რომელთა საშუალებითაც ერთი და იგივე ცხრილში სხვადასხვა HTML-დოკუმენტები ჩაიტვირთება. ამისათვის, აუცილებელია შესაბამისად შეიცვალოს DataUrl პარამეტრის მნიშვნელობა. ზემოთ განხილული მაგალითი წარმოადგენს HTML-დოკუმენტის ჩასმის კიდევ ერთ საშუალებას („მცურავი“ ჩარჩოს <iframe> გარდა) კლიენტის მხრიდან ანუ მომხმარებლის ბრაუზერით, და არა სერვერით.

## ცხრილის მონაცემების დამუშავება

ხშირად მონაცემთა ბაზებში ინახება პირველადი მონაცემები, ხოლო ეკრანზე აისახება მათი დამუშავების შედეგები. ჩვეულებრივ ხდება რიცხვითი მონაცემების დამუშავება. მაგალითად, საჭიროა ცხრილის სვეტების ჯამის გამოთვლა ან უნდა ვიპოვოთ მაქსიმალური ან მინიმალური მნიშვნელობა. არსებობს უფრო რთული ამოცანებიც. ნებისმიერ შემთხვევაში უფრო მოსახერხებელია ეს მონაცემები ცხრილიდან გადავიტანოთ მასივში.

როდესაც მონაცემთა ცხრილი მთლიანად შექმნილია HTML-ის ტეგების დახმარებით (STD მართვის ელემენტის გამოყენების გარეშე), მაშინ წაკითხვის ამოცანა მარტივია. ქვემოთ მოყვანილია ამ ამოცანის გადაწყვეტის სცენარი, სადაც

<table> ელემენტს აქვს id = "mytab" და საჭიროა მასივში წავიკითხოთ იმ სვეტების მნიშვნელობა, რომელთა ინდექსია n:

```
myarray = new Array ( );
for (i = 0; i < document.all.mytab.rows.length; i ++)  {
myarray [i] = parseFloat (document.all.mytab.rows [i].cells [n]);
}
```

აქ რიცხვით ტიპად გარდაქმნისათვის გამოყენებულია ფუნქცია parseFloat, რათა შენარჩუნებულ იქნეს რიცხვის წილადი ნაწილი.

განვიხილოთ ცხრილის სვეტის მნიშვნელობების წაკითხვის ამოცანა STD მართვის ელემენტის გამოყენებით. როგორც ცნობილია ასეთი ცხრილის მონაცემთა წყაროს წარმოადგენს ტექსტური ფაილი:

```
<!DOCTYPE html>
<html>
<! Control >
<object id = "mydatacontrol" classid =
    "CLSID:333C7BC4-460F-11D0-BC04-0080C7055A83">
    <param name = "FieldDelim" value = "|">
    <param name = "DataURL" value = "mydata.txt">
    <param name = "UseHeader" value = true>
</object>
<! Table>
<table  id = "mytab">
<tr>
<td><input  name = "zp" datasrc = #mydatacontrol datafld = "Salary"
    onchange = "zpchange ( ) ">
```

```
</td>
<! There may be a definition the other columns of the table >
</tr>
</table>
<P>
The amount of payment <b  id ="sum"></b>
<script>
let  azp = new Array ( );
zpchange ( );
function  zpchange ( )  {
let  S = 0, i = 0;
mydatacontrol . recordset . moveFirst ( );
while (!mydatacontrol . recordset . eof)  {
azp [i] = parseFloat (document . all mytab . zp . value);
S += azp [i];
mydatacontrol . recordset . moveNext ( );
i ++;
}
document . all . sum . innerText = S;
}
</script>
</html>
```

აქ იგულისხმება, რომ მონაცემთა წყარო განთავსებულია mydata.txt ფაილში და ამ ფაილში არსებობს ველი ხელფასი (Salary), ხოლო ველები ერთმანეთისაგან გამოყოფილია ვერტიკალური ხაზით.

# Web-გვერდების დაცვა პაროლის საშუალებით

თუ ჩვენ გვინდა საიტი ან მისი ცალკეული გვერდი დავიცვათ პაროლის საშუალებით, მაშინ მთავარია უზრუნველვყოთ პაროლისა და აგრეთვე, დაცული გვერდების მისამართების შენახვა. ნათელია, რომ ისინი ცხადი სახით არ უნდა ფიგურირებდეს HTML-დოკუმენტებსა და სცენარებში. ამ ამოცანის საკმაოდ მარტივი და საიმედო გადაწყვეტა იმაში მდგომარეობს, რომ პაროლი დაემთხვეს სერვერზე განთავსებულ რომელიმე ფაილის სახელს (გაფართოების გარეშე). წინააღმდეგ შემთხვევაში, მომხმარებელს ფაილზე მიმართვაზე უარი უნდა ეთქვას. ფაილის არსებობის შემოწმება ხდება FileExists() მეთოდით.

პაროლით დაცვის ამოცანის გადაწყვეტის მეორე ვარიანტი ისევ ეფუძნება პაროლისა და ფაილის სახელის დამთხვევას. მაგრამ ამ ვარიანტში არა ფაილების სისტემის ობიექტი გამოიყენება, არამედ ტექსტური მონაცემთა ბაზის მართვის ActiveX ელემენტი. ფაილის სახელი, რომელიც ემთხვევა პაროლს არის ტექსტური ფაილი და შეიცავს ორ სვეტს (ველი) და ორ სტრიქონს (ჩანაწერი). პირველ მონაცემთა ბაზის ველის იდენტიფიკატორები. მეორე სტრიქონში ასევე ორი სიტყვა – ამ ტექსტური ფაილის სახელი გაფართოების გარეშე და HTML-ფაილის სახელი ან გადასასვლელი გვერდის URL-მისამართი.

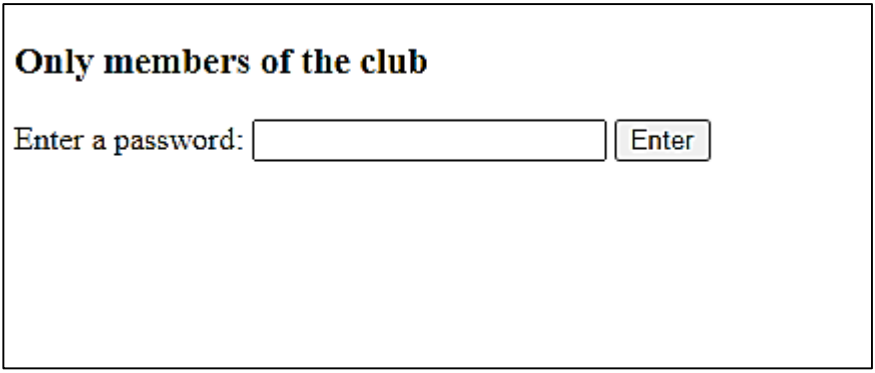
მაგალითად,

```
password|myspecurl
myspecfile|http://www.myweb.ge/mypage.htm
```

ქვემოთ მოყვანილია მაგალითი, სადაც HTML-დოკუმენტში განთავსებულია შეტანის ტექსტური ველი და ღილაკი:

```
<!DOCTYPE html>
<html>
<h3>Only members of the club</h3>
Enter a password:
<input  id = "pw1"  type = "password"  value = "">
<button  id ="pwenter">Enter</button>
<b  id = "dbelem"></b>
<script>
function  pwenter . onclick ( )  {
if (!document . all . pw1 . value);
    return;
dbstr  =  '<object  id  =  "mydbcontrol"  '  +  'classid  =
    "CLSID:333C7BC4-460F-11D0-BC04-0080C7055A83">'  +
    '<param name = "DataURL" value = " ' + document . all .pw1 .
    value + ' .txt"> ' + '<param name = "FieldDelim" value = "|">' +
    '<param name = "UseHeader" value = true></object>' +
      '<table  style = "visibility : hidden"><tr><td>' +
      '<input  id = "pw2" type = "password" datasrc = #mydbcontrol"
    ' +
      ' datafld = "password"></td><td>' +
      '<input  id = "xurl" type = "text" datasrc = #mydbcontrol" ' +
      ' datafld = "myspecurl"></td></tr></table>' ;
document . all . dbelem . innerHTML = dbstr;
setTimeout ("validation ( ) ", 1000);
}
```

```
function validation ( ) {
if (document . all . pw1 . value == document . all . pw2 . value) {
    document . all . pw1 . value = " ";
    window . location . href = document . all . xurl . value;
} else
    alert ("Password is not correct!");
}
</script>
</html>
```




სურვილის შემთხვევაში, შესაძლებელია ამ სცენარის სრულყოფა. თუ მომხმარებლის მიერ შეყვანილი პაროლი სწორი აღმოჩნდა, მაშინ მისი შენახვა cookie-ფაილში შეიძლება, იმისათვის რომ მომხმარებლის მიერ ამ გვერდის შემდგომში მონახულების დროს ხელახლა არ დასჭირდეს პაროლის შეყვანა.

ქვემოთ ამის მაგალითია მოყვანილი:

```
<!DOCTYPE html>
<html>
<h3>Only members of the club</h3>
```

```
<a  href = "#"  id = "myref"  onclick = "pwvalid ( )">CLICK</a>
<b  id = "dbelem"></b>
<script>
let  pw;
function  pwvalid ( )  {
pw = readCookie ("myspecrecord");
if  (!pw)
     pw = pwinput ( );
else
     pwenter ( );
}
function  pwinput ( )  {
let  inpstr = 'Enter a password:<input  id = "pw1"  type = " password"
     value = "">'  +
     '<button  onclick = "pw = document . all . pw1 . value; pwenter (
     )">Enter </button>';
function  pwenter ( )  {
if (!pw)
     return;
dbstr  =  '<object  id  =  "mydbcontrol"  '  +  'classid  =
     "CLSID:333C7BC4-460F-11D0-BC04-0080C7055A83">'  +
     '<param name = "DataURL" value = " ' + pw + ' .txt"> ' +
     '<param name = "FieldDelim" value = "|">' +
     '<param name = "UseHeader" value = true></object>' +
       '<table  style = "visibility : hidden"><tr><td>' +
       '<input  id = "pw2" type = "text" datasrc = #mydbcontrol" ' +
       ' datafld = "password"></td><td>' +
       '<input  id = "xurl" type = "text" datasrc = #mydbcontrol" ' +
```

```
        ' datafld = "myspecurl"></td></tr></table>' ;
document . all . dbelem . innerHTML = dbstr;
setTimeout ("validation ( ) ", 1000);
}
function validation ( )  {
if (pw == document . all . pw2 . value)  {
      window . location . href = document . all . xurl . value;
      let  d = new Date ( );
      let  d2 = d . getTime ( ) + (365 * 24 * 60 * 60 * 1000);
      d . setTime (d2);
      writeCookie ("myspecrecord", pw, d);
}  else
      alert ("Password is not correct!");
}
function  readCookie (name)  {
let  xname + name + "=";
let  xlen = xname . length;
let  clen = document . cookie . length;
let  i = 0;
while (i < clen)  {
let  j = i + xlen;
if  (document . cookie . substring (i, j) == xname)
      return  getCookieVal (j);
i = document . cookie . indexOf (" ", 1) + 1;
if (i == 0)  break;
}
return  null;
}
```

```
function  getCookieVal (n)  {
let  endstr = document . cookie . indexOf (";", n);
if (endstr == -1)
     endstr = document . cookie . length;
return  unescape (document . cookie . substring (n, endstr));
}
function  writeCookie (name, value, expires, path, domain, secure)  {
     document . cookie = name + "=" + escape (value) +
     ((expires) ? "; expires =" + expires . toGMTString ( ) ; "") +
     ((path) ? "; path =" + path ; "") +
     ((domain) ? "; domain =" + domain ; "") +
     ((secure) ? "; secure" ; "");
}
</script>
</html>
```

ამ მაგალითში პაროლის შენახვის ვადა არის 1 წელი.

# პროტოტიპური მემკვიდრეობა

დაპროგრამებაში ხშირად გვსურს რომელიმე არსებული ობიექტის მეთოდების აღება და მის საფუძველზე ახალის შექმნა და გაფართოება.

მაგალითად, ჩვენ გვაქვს ობიექტი user თავისი თვისებებითა და მეთოდებით და გვინდა შევქმნათ ობიექტები admin და guest, მისი ოდნავ შეცვლილი ვარიანტები. ჩვენ გვსურს ხელახლა გამოვიყენოთ ის თვისებები, რაც აქვს user ობიექტს, არა მისი მეთოდების კოპირება/შეცვლა, არამედ უბრალოდ მის საფუძველზე შევქმნათ ახალი ობიექტი.

პროტოტიპური მემკვიდრეობა ეს არის ენის შესაძლებლობა, რომელიც ამაში დაგვეხმარება.

## [[Prototype]]

JavaScript-ში ობიექტებს აქვთ სპეციალური დამალული თვისება [[Prototype]] (როგორც მას უწოდებენ სპეციფიკაციაში), რომელიც ან null-ის ტოლია ან მიმართავს სხვა ობიექტს. ამ ობიექტს „პროტოტიპი“ეწოდება:

როდესაც ჩვენ გვინდა წავიკითხოთ თვისება object-იდან და ის არ გვაქვს, JavaScript-ი მას ავტომატურად იღებს პროტოტიპიდან. დაპროგრამირებაში ამ მექანიზმს „პროტოტიპური მემკვიდრეობა“ ეწოდება. ამას ეფუძნება ენისა და დაპროგრამების ტექნიკის ბევრი საინტერესო თვისება.

[[Prototype]] თვისება არის შიდა და ფარული, მაგრამ მისი მინიჭების მრავალი გზა არსებობს.

ერთ-ერთი არის __proto__ თვისების გამოყენება, მაგალითად:

```
let animal = {
  eats: true
};
let rabbit = {
  jumps: true
};
rabbit.__proto__ = animal;
```

თუ ჩვენ ვეძებთ თვისებას rabbit-ში და ის მას აკლია, JavaScript ავტომატურად იღებს მას animal-იდან.

მაგალითი:

```
let animal = {
  eats: true
};
let rabbit = {
  jumps: true
};

rabbit.__proto__ = animal; // (*)

// ახლა შეგვიძლია ორივე თვისება rabbit-ში ვიპოვოთ:
alert( rabbit.eats ); // true (**)
alert( rabbit.jumps ); // true
```

აქ (*) სტრიქონი animal-ს თვისებას ანიჭებს, როგორც rabbit-ის პროტოტიპს.

შემდეგ, როდესაც alert-ი ცდილობს წაიკითხოს rabbit.eats თვისება (**), მაგრამ ის rabbit-ში არ გვაქვს, ამიტომ JavaScript-ი

[[Prototype]] ბმულს მიჰყვება და მას animal-ში პოულობს (პროგრამას მიყევით ქვემოდან ზევით):

აქ შეგვიძლია ვთქვათ, რომ „animal არის rabbit-ის პროტოტიპი“ ან „rabbit animal-სგან პროტოტიპურად მემკვიდრეობით იღებს თვისებას“.

ასე რომ, თუ animal-ს აქვს ბევრი სასარგებლო თვისება და მეთოდი, მაშინ ისინი ავტომატურად ხელმისაწვდომი გახდება rabbit-თვისაც. ასეთ თვისებებს „მემკვიდრეობითს“ უწოდებენ.

თუ ჩვენ გვაქვს მეთოდი animal-ში, მაშინ ის შეიძლება გამოძახებული იქნას rabbit-ისთვისაც:

```
let animal = {
  eats: true,
  walk() {
    alert("Animal walk");
  }
};

let rabbit = {
  jumps: true,
  __proto__: animal
};

// walk აღებულია პროტოტიპიდან
rabbit.walk(); // Animal walk
```

მეთოდის აღება ავტომატურად პროტოტიპიდან ხდება. პროტოტიპების ჯაჭვი შეიძლება საკმაოდ გრძელი იყოს:

```
let animal = {
```

```
  eats: true,
  walk() {
    alert("Animal walk");
  }
};

let rabbit = {
  jumps: true,
  __proto__: animal
};

let longEar = {
  earLength: 10,
  __proto__: rabbit
};

// walk აღებულია პროტორიბების ჯაჭვიდან
longEar.walk(); // Animal walk
alert(longEar.jumps); // true (rabbit-დან)
```

ახლა, თუ რაიმეს წავიკითხავთ longEar-დან რაიმეს წაკითხვა გვინდა და ის მასში არ არის, მაშინ JavaScript მას ჯერ rabbit-ში და შემდეგ animal-ში მოძებნის.

არსებობს მხოლოდ ორი შეზღუდვა:

1. ბმულების გადასვლა წრეზე არ უნდა ხდებოდეს. ამ დროს JavaScript მოგვცემს შეცდომას;

2. __proto__-ის მნიშვნელობა შეიძლება იყოს ობიექტი ან null. სხვა ტიპების იგნორირება ხდება.

შეიძლება იყოს მხოლოდ ერთი [[Prototype]]. ობიექტს არ შეუძლია მემკვიდრეობა მიიღოს ორი სხვა ობიექტისგან.

__proto__ თვისება [[Prototype]]-სთვის ისტორიულად არის განსაზღვრული გეტერი/სეტერი.

გაითვალისწინეთ, რომ __proto__ არ არის იგივე, რაც არის შიდა თვისება [[Prototype]]. ეს არის გეტერი/სეტერი [[Prototype]]-ისთვის.

__proto__ თვისება ოდნავ მოძველებულია, ის არსებობს ისტორიული მიზეზების გამო. თანამედროვე JavaScript გვთავაზობს, რომ პროტოტიპის მიღების/დაყენების ნაცვლად უნდა გამოვიყენოთ Object.getPrototypeOf/Object.setPrototypeOf ფუნქციები.

სპეციფიკაციის მიხედვით, __proto__ მხოლოდ ბრაუზერების მიერ უნდა იყოს მხარდაჭერილი, მაგრამ სინამდვილეში ყველა გარემო, სერვერის გარემოს ჩათვლით, მხარს უჭერს მას. ასე რომ, ჩვენ შეგვიძლია მისი საკმაოდ უსაფრთხოდ გამოყენება.

შემდეგში ჩვენ გამოვიყენებთ __proto__-ს მაგალითებში, რადგან ეს არის პროტოტიპის დაყენებისა და წაკითხვის ყველაზე მოკლე და ინტუიციური გზა.

პროტოტიპი გამოიყენება მხოლოდ თვისებების წასაკითხად.

ჩაწერის/წაშლის ოპერაციები მოქმედებს უშუალოდ ობიექტზე.

ქვემოთ მოცემულ მაგალითში ჩვენ rabbit-ს ვანიჭებთ walk-ის საკუთარ მეთოდს:

```
let animal = {
  eats: true,
```

```
  walk() {
    /* ეს მეთოდი rabbit-ში არ იქნება გამოყენებული */
  }
};

let rabbit = {
  __proto__: animal
};

rabbit.walk = function() {
  alert("Rabbit! Bounce-bounce!");
};

rabbit.walk(); // Rabbit! Bounce-bounce!
```

ახლა rabbit.walk()-ის გამოძახება პოულობს მეთოდს უშუალოდ ობიექტში და პროტოტიპის გამოყენების გარეშე ახორციელებს მას:

თვისება-აქსესორი გამონაკლისია, რადგან მათში ჩაწერა ფუნქცია-სეტერის საშუალებით ხდება. ანუ, ეს არის რეალურად ფუნქციის გამოძახება.

ამ მიზეზით, admin.fullName ქვემოთ მოცემულ კოდში მუშაობს სწორად:

```
let user = {
  name: "ნიკოლოზ",
  surname: "ბერიძე",

  set fullName(value) {
```

```
    [this.name, this.surname] = value.split(" ");
  },

  get fullName() {
    return `${this.name} ${this.surname}`;
  }
};

let admin = {
  __proto__: user,
  isAdmin: true
};

alert(admin.fullName); // ნიკოლოზ ბერიძე (*)

// მუშაობს სეტერი!
admin.fullName = "მარიამ ხელაძე"; // (**)
alert(admin.name); // მარიამ
alert(admin.surname); // ხელაძე
```

აქ (*) სტრიქონში user პროტოტიპში admin.fullName თვისებას აქვს გეტერი, ამიტომ ხდება მისი გამოძახება. (**) სტრიქონშიც პროტოტიპში ასევე არის სეტერი და რომლის გამოძახებაც მოხდება.

ზემოთ მოყვანილ მაგალითში შეიძლება წარმოიშვას საინტერესო კითხვა: რა არის this-ის მნიშვნელობა set fullName(value)-ის შიგნით? სად იწერება this.name და this.surname თვისებები: user-ში თუ admin-ში?

პასუხი მარტივია: პროტოტიპები არანაირად არ მოქმედებს this-ზე.

არ აქვს მნიშვნელობა სად მდებარეობს მეთოდი: ობიექტში თუ მის პროტოტიპში. this მეთოდის გამოძახებისას, ობიექტი ყოველთვის არის წერტილის წინ.

ამრიგად, admin.fullName= სეტერის გამოძახება this-ის სახით იყენებს admin-ს და არა user-ს.

ეს რეალურად ძალიან მნიშვნელოვანი დეტალია, რადგან ჩვენ შეგვიძლია გვქონდეს დიდი ობიექტი მრავალი მეთოდით, რომლიდანაც შეიძლება მემკვიდრეობით მივიღოთ. შემდეგ მემკვიდრეობით ობიექტებს შეუძლიათ გამოიძახონ მისი მეთოდები, მაგრამ ისინი შეცვლიან მათ მდგომარეობას და არა მშობლიური ობიექტის მდგომარეობას.

მაგალითად, აქ animal-ი წარმოადგენს „მეთოდების საცავს“, ხოლო rabbit-ი იყენებს მას.

rabbit.sleep()-ის გამოძახება rabbit ობიექტისათვის დააყენებს this.isSleeping-ს:

```
// animal მეთოდები
let animal = {
  walk() {
    if (!this.isSleeping) {
      alert('I walk');
    }
  },
  sleep() {
    this.isSleeping = true;
  }
};
```

```
let rabbit = {
  name: "White Rabbit",
  __proto__: animal
};

// rabbit.isSleeping-ის მოდიფიცირებას ახდენს
rabbit.sleep();

alert(rabbit.isSleeping); // true
alert(animal.isSleeping); // undefined (პროტოტიპში ასეთი
    თვისება არ არის)
```

ჩვენ რომ გვქონდეს animal-სგან მემკვიდრეობით მიღებული სხვა ობიექტები, როგორიცაა bird, snake და ა.შ., მაშინ ისინიც მიიღებდნენ animal-ის მეთოდებზე წვდომას. მაგრამ ყოველი მეთოდის გამოძახებისას this შეესაბამება ობიექტს (წერტილის წინ), რომლის გამოძახებაც ხდება და არა animal-ს. ამიტომ, როდესაც ჩვენ მონაცემებს this-ში ვწერთ, ისინი ამ ობიექტებში ინახება.

შედეგად, მეთოდები საერთოა, მაგრამ ობიექტის მდგომარეობა არა.

## ციკლი for…in

for..in ციკლი გაივლის არა მხოლოდ საკუთარ, არამედ ობიექტის მემკვიდრეობით მიღებულ თვისებებს.

მაგალითად:

```
let animal = {
```

```
  eats: true
};

let rabbit = {
  jumps: true,
  __proto__: animal
};

// Object.keys აბრუნებს მხოლოდ საკუთარ გასაღებებს
alert(Object.keys(rabbit)); // jumps

// for..in გაივლის როგორც საკუთარ, ისე მემკვიდრეობით
    გასაღებებს
for(let prop in rabbit) alert(prop); // jumps, შემდეგ eats
```

თუ არ გვჭირდება მემკვიდრეობითი თვისებები, მათი გაფილტვრა შეგვიძლია obj.hasOwnProperty(key) ჩაშენებული მეთოდის გამოყენებით: ის აბრუნებს true-ს, თუ obj-ს აქვს საკუთარი, არამემკვიდრეობითი თვისება სახელით key.

ქვემოთ მოცემულია ასეთი ფილტრაციის მაგალითი:

```
let animal = {
  eats: true
};

let rabbit = {
  jumps: true,
  __proto__: animal
};
```

```
for(let prop in rabbit) {
  let isOwn = rabbit.hasOwnProperty(prop);

  if (isOwn) {
    alert(‘Our: ${prop}’); // Our: jumps
  } else {
    alert(‘Inherited: ${prop}’); // Inherited: eats
  }
}
```

ამ მაგალითში მემკვიდრეობის ჯაჭვი ასე გამოიყურება: rabbit-ი იღებს მემკვიდრეობას animal-ისგან, რომელიც მემკვიდრეობით იღებს Object.prototype-სგან (რადგან animal-ი არის პირდაპირი ობიექტი {...}, ეს ნაგულისხმევად), და შემდეგ null სულ ზედა ნაწილში:

აღვნიშნოთ კიდევ ერთი დეტალი. საიდან გაჩნდა rabbit.hasOwnProperty მეთოდი? ჩვენ ცალსახად არ განგვისაზღვრავს იგი. თუ დააკვირდებით პროტოტიპის ჯაჭვს, დაინახავთ, რომ ის Object.prototype.hasOwnProperty-დან არის აღებული. ანუ მემკვიდრეობითაა მიღებული.

მაგრამ რატომ არ ჩნდება hasOwnProperty-ი for..in ციკლში, eats-ისა და jumps-ისაგან განსხვავებით? იგი ჩამოთვლის ყველა მემკვიდრეობით თვისებას.

პასუხი მარტივია: ის არ არის ჩამოთვლადი. ანუ მას აქვს შიდა enumerable-ის ალამი დაყენებულია false-ზე, ისევე როგორც სხვა Object.prototype თვისებებშიც. ამიტომ, ის არ ჩანს ციკლში.

თითქმის ყველა სხვა მეთოდი, რომელიც იღებს გასაღებებს/მნიშვნელობებს, როგორიცაა Object.keys, Object.values და სხვა, ისინი უგულებელყოფენ მემკვიდრეობით მიღებულ თვისებებს. ისინი ითვალისწინებენ მხოლოდ თავად ობიექტის თვისებებს და არა მის პროტოტიპს.

## F.prototype

როგორც ვიცით, ახალი ობიექტების შექმნა შესაძლებელია new F() კონსტრუქტორი-ფუნქციის გამოყენებით.

თუ F.prototype-ში არის ობიექტი, ოპერატორი new ახალი ობიექტისთვის ადგენს მას, როგორც [[Prototype]]-ს.

JavaScript პროტოტიპურ მემკვიდრეობას შექმნის დღიდან იყენებს. ეს არის ენის ერთ-ერთი ძირითადი მახასიათებელი. ადრე, ობიექტის პროტოტიპზე პირდაპირი წვდომა არ იყო. საიმედოდ მხოლოდ ფუნქცია-კონსტრუქტორის „prototype“ თვისება მუშაობდა. ამიტომ, იგი ბევრ სკრიპტში გამოიყენება.

გაითვალისწინეთ, რომ F.prototype ნიშნავს ჩვეულებრივ თვისებას F-სათვის სახელად „prototype“. ეს ჯერ არ არის „ობიექტის პროტოტიპი“, არამედ ჩვეულებრივი F თვისება ამ სახელით.

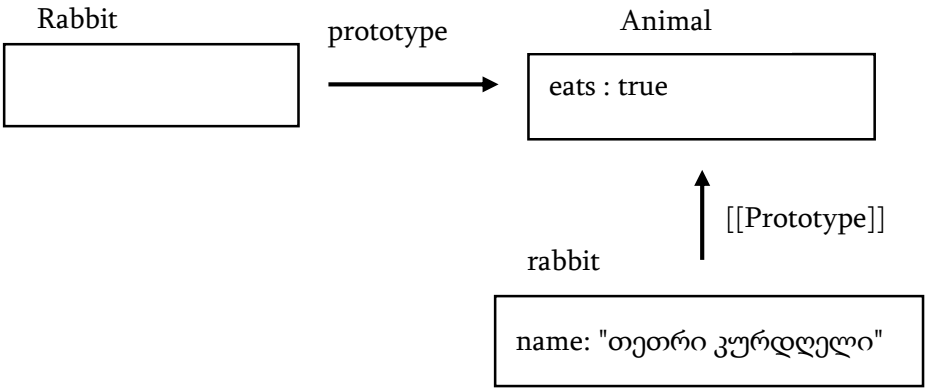
მოვიყვანოთ მაგალითი:

```
let animal = {
  eats: true
};

function Rabbit(name) {
  this.name = name;
```

```
}

Rabbit.prototype = animal;

let rabbit = new Rabbit("თეთრი კურდღელი"); //  rabbit.__proto__
    == animal

alert( rabbit.eats ); // true
```

Rabbit.prototype = animal მინიჭება ინტერპრეტატორს სიტყვასიტყვით ეუბნება: „new Rabbit()-ის საშუალებით ობიექტის შექმნის პროცესში, მას animal-ი ჩაუწერეთ [[Prototype]]-ში“.

შედეგი ასე გამოიყურება:



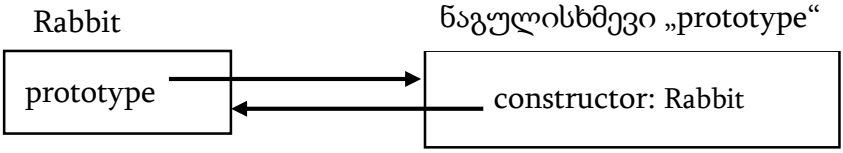

სურათზე „prototype“ არის ჰორიზონტალური ისარი, რომელიც „F“-ის ჩვეულებრივ თვისებაზე მიუთითებს და [[Prototype]] ვერტიკალური ისარი კი მიუთითებს თვისებაზე, რომელიც animal-ისგან მემკვიდრეობით მიიღო rabbit-მა.

F.prototype გამოიყენება მხოლოდ new F()-ის გამოძახების დროს და [[Prototype]] თვისების სახით ენიჭება ახალი ობიექტს. ამის შემდეგ F.prototype-ს და ახალ ობიექტს არაფერს არ აკავშირებს. ეს უნდა გავიგოთ, როგორც ობიექტისთვის „ერთჯერადი საჩუქარი“.

შექმნის შემდეგ F.prototype-ი შეიძლება შეიცვალოს და new F()-ით შექმნილ ახალ ობიექტებს [[Prototype]]-ის სახით ექნებათ სხვა ობიექტი, მაგრამ უკვე არსებული ობიექტები შეინარჩუნებენ ძველს.

ყველა ფუნქციას ნაგულისხმევად უკვე აქვს „prototype“ თვისება. სისტემაში ნაგულისხმევი წესის თანახმად, „prototype“ არის ობიექტი ერთადერთი constructor თვისებით, რომელიც მიმართავს ფუნქცია-კონსტრუქტორს.

```
function Rabbit() {}

/* ნაგულისხმევი პროტოტიპი
Rabbit.prototype = { constructor: Rabbit };
*/
```




შევამოწმოთ ეს:

```
function Rabbit() {}
// ნაგულისხმევი:
```

```
// Rabbit.prototype = { constructor: Rabbit }

alert( Rabbit.prototype.constructor == Rabbit ); // true
```

შესაბამისად, თუ ჩვენ არაფერს არ ვცვლით, მაშინ [[Prototype]]-ის მეშვეობით constructor თვისება ყველა Rabbit-ისთვის ხელმისაწვდომი იქნება:

```
function Rabbit() {}
// ნაგულისხმევი:
// Rabbit.prototype = { constructor: Rabbit }

let rabbit = new Rabbit(); // მემკვიდრეობით {constructor: Rabbit}-
    სგან

alert(rabbit.constructor == Rabbit); // true (თვისება
    პროტოტიპისაგანაა მიღებული)
```

ჩვენ შეგვიძლია არსებული ობიექტის constructor თვისება გამოვიყენოთ ახალი ობიექტის შესაქმნელად. მაგალითად:

```
function Rabbit(name) {
  this.name = name;
  alert(name);
}

let rabbit = new Rabbit("თეთრი კურდღელი");

let rabbit2 = new rabbit.constructor("შავი კურდღელი");
```

ეს მოსახერხებელია, როდესაც გვაქვს ობიექტი, მაგრამ არ ვიცით რომელი კონსტრუქტორი გამოიყენეს მის შესაქმნელად (მაგალითად, მისი აღება შეიძლება სხვა ბიბლიოთეკიდან), ხოლო ჩვენ აუცილებელია შევქმნათ კიდევ სხვა ასეთი ობიექტი.

მაგრამ, ალბათ, ყველაზე მნიშვნელოვანი „constructor“ თვისებაში ის არის, რომ თვითონ JavaScript-ი „constructor“ სწორი მნიშვნელობის შესახებ გარანტიას არ იძლევა.

ფუნქციისთვის ეს არის ნაგულისხმევი თვისება prototype-ში, მაგრამ რა დაემართება მას მოგვიანებით, ეს მხოლოდ ჩვენზეა დამოკიდებული.

კერძოდ, თუ ნაგულისხმევ პროტოტიპს სხვა ობიექტით შევცვლით, მაშინ „constructor“ თვისება მასში აღარ იქნება.

მაგალითად:

```
function Rabbit() {}
Rabbit.prototype = {
  jumps: true
};

let rabbit = new Rabbit();
alert(rabbit.constructor === Rabbit); // false
```

ამგვარად, იმისათვის, რომ შევინარჩუნოთ „constructor“ თვისება, იმის მაგივრად, რომ სრულად გადავწეროთ ის, საჭიროა ან დავუმატოთ ან წავშალოთ ან შევცვალოთ ნაგულისხმევი პროტოტიპის თვისებას:

```
function Rabbit() {}

// სრულად არ გადავწეროთ Rabbit.prototype,
```

```
// არამედ დავუმატოთ მას თვისება
Rabbit.prototype.jumps = true
// ნაგულისხმევი პროტოტიპის შენარჩუნებულია,
// ჩვენ ისევ გვაქვს მასზე წვდომა Rabbit.prototype.constructor
```

ან შეგვიძლია ხელახლა შევქმნათ „constructor“ თვისება:

```
Rabbit.prototype = {
  jumps: true,
  constructor: Rabbit
};
// ახლა „constructor“ თვისება ისევ სწორია, რადგან დავამატეთ
```

# კლასები

ობიექტზე ორიენტირებულ დაპროგრამებაში კლასი არის გაფართოებადი კოდის შაბლონი ობიექტების შესაქმნელად, რომელიც მათში ადგენს საწყის მნიშვნელობებს (თვისებებს) და რეალიზების ქცევებს (მეთოდებს).

პრაქტიკაში, ხშირად, ერთი და იგივე ტიპის მრავალი ობიექტის შექმნა გვჭირდება, როგორიცაა მომხმარებლები, პროდუქტები ან სხვა.

როგორც უკვე ვიცით, ამაში დაგეხმარებათ ოპერატორი new function.

თანამედროვე JavaScript-ს ასევე აქვს უფრო მოწინავე კონსტრუქცია class, რომელიც ახალ შესაძლებლობებს გვთავაზობს, რომლებიც ობიექტზე ორიენტირებული დაპროგრამებისთვის არის სასარგებლო.

კონსტრუქცია class-ის სინტაქსი ასე გამოიყურება:

```
class MyClass {
  // კლასის მეთოდი
  constructor() { ... }
  method1() { ... }
  method2() { ... }
  method3() { ... }
  ...
}
```

შემდეგ ახალი ობიექტის შესაქმნელად გამოიყენეთ new MyClass() გამოძახება ყველა ჩამოთვლილი მეთოდით.

ამ შემთხვევაში, ავტომატურად მოხდება constructor() მეთოდის გამოძახება, რომელშიც შეგვიძლია ობიექტის ინიცია-ლიზაცია განვახორციელოთ.

მაგალითად:

```
class User {

  constructor(name) {
    this.name = name;
  }

  sayHi() {
    alert(this.name);
  }

}

// გამოყენება:
let user = new User("ალექსანდრე");
user.sayHi();
```

როდესაც მოხდებ new User("ალექსანდრე") გამოძახება:

1. იქმნება ახალი ობიექტი;
2. მოცემული არგუმენტით ხდება constructor-ის გაშვება და ინახავს მას this.name-ში.

…შემდეგ შეგიძლიათ ობიექტზე გამოიძახოთ მეთოდები, როგორიცაა user.sayHi().

კლასში მეთოდები ერთმანეთისაგან მძიმით არ არის გამოყოფილი.

კლასის სინტაქსი განსხვავდება ობიექტის სინტაქსისგან. კლასებში მძიმეები არ არის საჭირო.

რა არის კლასი? ეს არ არის სრულიად ახალი ენობრივი ერთეული, როგორც ეს ერთი შეხედვით შეიძლება ჩანდეს.

სინამდვილეში JavaScript-ში კლასი არის ერთგვარი ფუნქცია.

დააკვირდით:

```
class User {
  constructor(name) { this.name = name; }
  sayHi() { alert(this.name); }
}

// მტკიცებულება: User - ეს ფუნქციაა
alert(typeof User);   // function
```

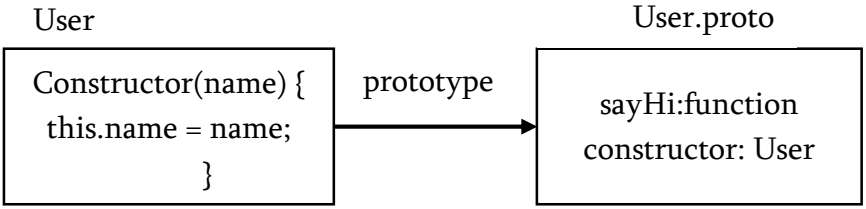
აი, რას აკეთებს class User {...} კონსტრუქცია:

1. ქმნის ფუნქციას სახელად User, რომელიც ხდება კლასის გამოცხადების შედეგი. ფუნქციის კოდი აღებულია მეთოდიდან constructor (ის ცარიელი იქნება, თუ ასეთი მეთოდი არ არსებობს).

2. ინახავს ყველა მეთოდს, როგორიცაა sayHi - User.prototype-ში.

როდესაც მოხდება new User ობიექტის მეთოდის გამოძახება, ის აღებული იქნება პროტოტიპიდან, როგორც ეს F.prototype თავშია აღწერილი. ამრიგად, new User ობიექტებს აქვთ წვდომა კლასის მეთოდებზე.

სურათზე ნაჩვენებია მომხმარებლის კლასის გამოცხადების შედეგი:




ზემოთ თქმული შეიძლება შემოწმდეს მოცემული კოდით:

```
class User {
  constructor(name) { this.name = name; }
  sayHi() { alert(this.name); }
}

// კლასი - ეს ფუნქციაა
alert(typeof User); // function

// ... ან, უფრო ზუსტად ეს constructor მეთოდია
alert(User === User.prototype.constructor); // true

// მეთოდი განთავსებულია User.prototype-ში, მაგალითად:
alert(User.prototype.sayHi); // sayHi() { alert(this.name); }

// prototype-ში ზუსტად 2 მეთოდია
alert(Object.getOwnPropertyNames(User.prototype));                    //
    constructor, sayHi
```

კლასი JavaScript-ში გამოიყენება სინტაქსის კოდის წაკითხვის გასაუმჯობესებლად, მაგრამ ფუნდამენტურად ახალს

არაფერს აკეთებს, რადგან ჩვენ იგივეს გაკეთება class კონსტრუქციის გარეშეც შეგვიძლია:

```
  // გადავწეროთ User კლასი სუფთა ფუნქციების გამოყენებით

  // 1. ფუნქცია constructor-ის შექმნა
function User(name) {
  this.name = name;
}
  // ფუნქციის თითოეულ პროტოტიპს ნაგულისხმევად აქვს
     constructor-ის თვისება,
  // ასე რომ, ჩვენ არ გვჭირდება მისი შექმნა

  // 2. პროტოტიპში მეთოდის დამატება
User.prototype.sayHi = function() {
  alert(this.name);
};

  // გამოყენება:
let user = new User("ალექსანდრე");
user.sayHi();
```

ამ კოდების შედეგი ერთნაირია.

თუმცა, არსებობს მნიშვნელოვანი განსხვავებები:

1. პირველ რიგში, class-ით შექმნილი ფუნქცია აღინიშნება სპეციალური შიდა თვისებით [[IsClassConstructor]]: true. ამიტომ, ეს არ არის ზუსტად იგივე, რაც მისი ხელით შექმნა.

ჩვეულებრივი ფუნქციებისგან განსხვავებით, კლასის კონსტრუქტორს ვერ გამოიძახებთ new-ს გარეშე:

```
class User {
  constructor() {}
}

alert(typeof User); // function
User(); // Error: Class constructor User cannot be invoked without
    'new'
```

ამის გარდა, კლასის კონსტრუქტორის სტრიქონული წარმოდგენა JavaScript ძრავების უმეტესობაში იწყება "class..."-ით.

```
class User {
  constructor() {}
}

alert(User); // class User { ... }
```

1. კლასის მეთოდები უამრავია. კლასის განმარტება enumerable ალამს აყენებს false-ზე ყველა მეთოდისთვის "prototype"-ში.

და ეს კარგია, რადგან თუ ჩვენ ობიექტზე ვიყენებთ for..in ციკლს, მაშინ ჩვეულებრივ არ გვჭირდება კლასის მეთოდების მიღება.

2. კლასები ყოველთვის იყენებს use strict. კლასში არსებული ყველა კოდი ავტომატურად მკაცრ რეჟიმში იმყოფება.

ასევე, გარდა ზემოთ აღწერილი ძირითადი ფუნქციებისა, class-ის სინტაქსი კიდევ უამრავ სხვა საინტერესო ფუნქციას გვთავაზობს.

## Class Expression

ფუნქციების მსგავსად, კლასები შეიძლება განისაზღვროს სხვა გამოსახულებაში, გადასცეს, დააბრუნოს, მიანიჭოს და ა.შ.

Class Expression მაგალითი (Function Expression-ის ანალოგიური):

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

Named Function Expression-ის მსგავსად, Class Expression-ს შეიძლება ჰქონდეს სახელი.

თუ Class Expression-ს აქვს სახელი, მაშინ ის მხოლოდ კლასის შიგნით ჩანს:

```
// "Named Class Expression"
// (სპეციფიკაციაში ასეთი ტერმინი არ არის, მაგრამ რაც ხდება
     Named Function Expression-ის მსგავსია)
let User = class MyClass {
  sayHi() {
    alert(MyClass); // სახელი MyClass მხოლოდ კლასის შიგნით
      ჩანს
  }
};

new User().sayHi(); // მუშაობს, გამოაქვს განსაზღვრება MyClass
```

```
alert(MyClass); // შეცდომაა, სახელი MyClass არ ჩანს კლასის
    გარეთ
```

ჩვენ შეგვიძლია შევქმნათ კლასები დინამიურად „მოთხოვნით“:

```
function makeClass(phrase) {
  // გამოვაცხადებთ კლასს და ვაბრუნებთ უკან
  return class {
    sayHi() {
      alert(phrase);
    };
  };
}

// ვქმნით ახალ კლასს
let User = makeClass("Hello!");

new User().sayHi(); // Hello!
```

## გეტერები/სეტერები, სხვა აბრევიატურები

როგორც ჩვეულებრივი ობიექტების შემთხვევაში, კლასებშიც შეგიძლიათ გამოაცხადოთ გამოთვლილი თვისებები, გეტერები/სეტერები და ასე შემდეგ.

ქვემოთ მოცემულია user.name-ის მაგალითი, რომელიც განხორციელებულია get/set-ის გამოყენებით:

```
class User {

  constructor(name) {
```

```
    // გამოიძახებს სეტერს
    this.name = name;
  }

  get name() {
    return this._name;
  }

  set name(value) {
    if (value.length < 4) {
      alert("სახელი ძალზე მოკლეა.");
      return;
    }
    this._name = value;
  }

}

let user = new User("გიორგი");
alert(user.name); // გიორგი

user = new User(""); // სახელი ძალზე მოკლეა.
```

კლასის გამოცხადების შემთხვევაში, გეტერები/სეტერები User.prototype-ზე ასე იქმნება:

```
  name: {
    get() {
      return this._name
```

```
  },
  set(name) {
    // ...
  }
 }
});
```

მაგალითი ფრჩხილებში [...] განთავსებული გამოთვლადი თვისებით:

```
class User {

 ['say' + 'Hi']() {
  alert("Hello!");
 }

}

new User().sayHi();
```

კლასის თვისებები ენას ახლახან დაემატა.

ზემოთ მოყვანილ მაგალითში User კლასს მხოლოდ მეთოდები ჰქონდა. დავამატოთ თვისება:

```
class User {
 name = "Anonym";

 sayHi() {
  alert('Hello!, ${this.name}!');
 }
```

```
}

new User().sayHi();
```

თვისება name არ არის დაყენებული User.prototype. ამის ნაცვლად, ის კონსტრუქტორის გაშვების წინ იქმნება new ოპერატორის საშუალებით, ეს მხოლოდ ობიექტის თვისებაა.

## კლასების მემკვიდრეობა

კლასების მემკვიდრეობა არის ერთი კლასის მეორე კლასით გაფართოების გზა.

ამრიგად, ჩვენ შეგვიძლია უკვე არსებულს ახალი ფუნქცია დავამატოთ.

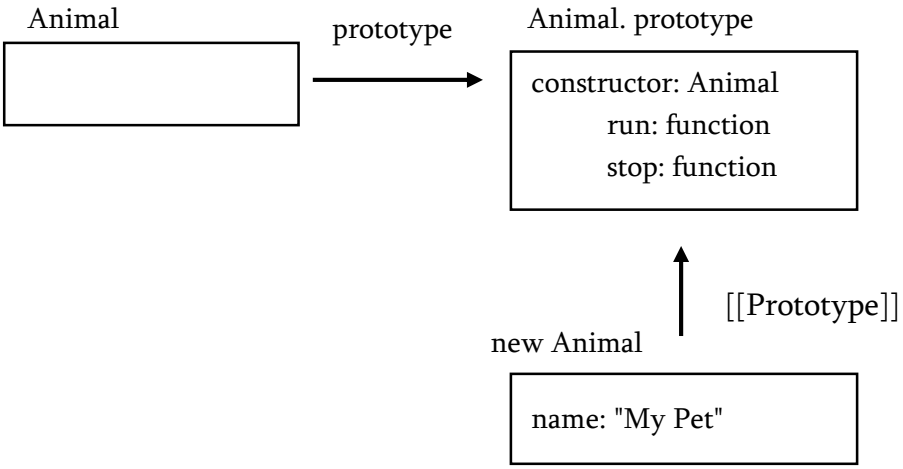
### საკვანძო სიტყვა extends

ვთქვათ, გვაქვს Animal კლასი:

```
class Animal {
  constructor(name) {
    this.speed = 0;
    this.name = name;
  }
  run(speed) {
    this.speed = speed;
    alert(‘${this.name} Runs at speed ${this.speed}.’);
  }
  stop() {
    this.speed = 0;
```

```
    alert(‘${this.name} Stands still.’);
  }
}

let animal = new Animal("My Pet");
```

ობიექტი animal და კლასი Animal გრაფიკულად შეიძლება ასე წარმოვადგინოთ:




შევქმნათ კიდევ ერთი class Rabbit.

ვინაიდან კურდღლები ცხოველები არიან, class Rabbit დაფუძნებული უნდა იყოს Animal-ზე და ჰქონდეს წვდომა ცხოველების მეთოდებზე, რათა კურდღლებს შეეძლოთ ისეთი რამის გაკეთება, რისი გაკეთებაც „ზოგად“ ცხოველებს შეუძლიათ.

სხვა კლასის გაფართოების სინტაქსია: class Child extends Parent.

შევქმნათ Rabbit კლასი, რომელიც Animal-ის მემკვიდრეა:

```
class Rabbit extends Animal {
  hide() {
    alert(‘${this.name} იმალება!’);
  }
}

let rabbit = new Rabbit("თეთრი კურდღელი");

rabbit.run(5); // თეთრი კურდღელი გარბის სიჩქარით 5.
rabbit.hide(); // თეთრი კურდღელი იმალება!
```

Rabbit კლასის ობიექტს აქვს წვდომა როგორც Rabbit მეთოდებზე, როგორიცაა rabbit.hide(), ასევე, Animal მეთოდებზე, როგორიცაა rabbit.run().

საკვანძო სიტყვის extends შიგნით ძველი პროტოტიპის მექანიკის მსგავსად მუშაობს. ის Animal.prototype-ში დააყენებს Rabbit.prototype.[[Prototype]]-ს. ამრიგად, თუ Rabbit.prototype-ში მეთოდი არ არის, მაშინ მას JavaScript-ი Animal.prototype-დან აიღებს.

მაგალითად, rabbit.run მეთოდის საპოვნელად, ძრავა ამოწმებს (ქვემოდან ზემოთ):

1. Rabbit ობიექტს (არ აქვს არც hide და არც run მეთოდი);
2. მისი პროტოტიპი ანუ Rabbit.prototype (აქვს hide, მაგრამ არ აქვს run მეთოდი);
3. მისი პროტოტიპი, ანუ (extends-ის გამო) Animal.prototype, რომელსაც საბოლოოდ აქვს run მეთოდი.

როგორც გვახსოვს ჩაშენებული პროტოტიპებიდან, თავად JavaScript ჩაშენებული ობიექტებისთვის იყენებს პროტოტიპის

მემკვიდრეობას. მაგალითად, Date.prototype.[[Prototype]] არის Object.prototype, ამიტომ თარიღებს აქვს ობიექტის უნივერსალური მეთოდები.

კლასის შექმნის სინტაქსი საშუალებას გაძლევთ extends-ის შემდეგ მიუთითოთ არა მხოლოდ კლასის, არამედ ნებისმიერი გამოსახულება.

ფუნქციის გამოძახების მაგალითი, რომელიც მშობლიურ კლასს ქმნის:

```
function f(phrase) {
  return class {
    sayHi() { alert(phrase); }
  };
}

class User extends f("Hello!") {}

new User().sayHi(); // Hello!
```

აქ class User გამოძახების შედეგიდან მემკვიდრეობით იღებს f ("Hello!").

ეს შეიძლება იყოს სასარგებლო დაპროექტების მოწინავე საშუალებებისთვის, სადაც ჩვენ შეგვიძლია კლასების გენერირებისთვის გამოვიყენოთ ფუნქციები სხვადასხვა მრავალ პირობებზე დამოკიდებულებით და შემდეგ მათგან მემკვიდრეობით მივიღოთ ისინი.

## მეთოდების შეცვლა

ახლა მოდით შევცვალოთ მეთოდი. სისტემაში ნაგულისხმევი წესის თანახმად, ყველა მეთოდი, რომელიც არ არის მითითებული Rabbit კლასში, უშუალოდ აღებულია Animal კლასიდან.

მაგრამ თუ Rabbit-ში მივუთითებთ საკუთარ მეთოდს, მაგალითად stop(), მაშინ ის გამოყენებული იქნება მის ნაცვლად:

```
class Rabbit extends Animal {
  stop() {
    // ...შემდეგში stop()-ის ნაცვლად Animal კლასიდან
    // ეს იქნება გამოყენებული rabbit.stop()-სათვის
     }
}
```

თუმცა, ჩვენ, როგორც წესი, არ გვინდა მთლიანად შევცვალოთ მშობლიური მეთოდი, არამედ მის საფუძველზე შევქმნათ ახალი, შევცვალოთ ან გავაფართოვოთ მისი ფუნქციები. ჩვენ ვწერთ პროგრამულ კოდს მეთოდში და ვიძახებთ მშობლიურ მეთოდს ადრე/შემდეგ ან პროცესის მიმდინარეობისას.

კლასებს ასეთი შემთხვევებისთვის აქვთ საკვანძო სიტყვა super.

- super.method(...) მშობლიურ მეთოდს იძახებს;
- super(...) მშობლიური კონსტრუქტორის გამოსაძახებლად (მუშაობს მხოლოდ ჩვენი კონსტრუქტორის შიგნით).

ჩვენი „კურდღელი“ ავტომატურად დაიმალოს, როდესაც ის გაჩერდება:

```
class Animal {

  constructor(name) {
    this.speed = 0;
    this.name = name;
  }

  run(speed) {
    this.speed = speed;
    alert('${this.name} გარბის სიჩქარით ${this.speed}.');
  }

  stop() {
    this.speed = 0;
    alert('${this.name} დგას გაუნძრევლად.');
  }

}

class Rabbit extends Animal {
  hide() {
    alert('${this.name} იმალება!');
  }

  stop() {
    super.stop(); // ვიძახებთ stop მშობლიურ მეთოდს
    this.hide(); // ხოლო შემდეგ hide-ს
  }
```

```
}

let rabbit = new Rabbit("თეთრი კურდღელი");

rabbit.run(5); // თეთრი კურდღელი გარბის სიჩქარით 5.
rabbit.stop(); // თეთრი კურდღელი დგას გაუნძრევლად. თეთრი
    კურდღელი იმალება!
```

Rabbit კლასს ახლა აქვს მეთოდი stop, რომელიც გაშვების დროს იძახებს მშობლიურ super.stop()-ს.

ისრის ფუნქციებს არ აქვს super.

ისრის ფუნქციის super-ზე მიმართვის დროს მისი აღება გარე ფუნქციიდან ხდება:

```
class Rabbit extends Animal {
  stop() {
    setTimeout(() => super.stop(), 1000); // 1 წამის შემდეგ იძახებს
      მშობლიურ stop
  }
}
```

მაგალითში, ისრის ფუნქციაში super არის იგივე, რაც stop(), ამიტომ მეთოდი მუშაობს ისე, როგორც მოსალოდნელია. აქ რომ მივუთითოთ ჩვეულებრივი ფუნქცია, იქნებოდა შეცდომა:

```
// Unexpected super
setTimeout(function() { super.stop() }, 1000);
```

## კონსტრუქტორის შეცვლა

კონსტრუქტორების შეცვლა ცოტა უფრო რთულია.

აქამდე Rabbit-ს არ ჰქონდა საკუთარი კონსტრუქტორი.

სპეციფიკაციის მიხედვით, თუ კლასი აფართოებს სხვა კლასს და არ აქვს კონსტრუქტორი, მაშინ ავტომატურად იქმნება ასეთი „ცარიელი“ კონსტრუქტორი:

```
class Rabbit extends Animal {
// შთამომავალი კლასებისთვის, რომლებსაც არ აქვთ საკუთარი
    კონსტრუქტორი ხდება გენერირება
  constructor(...args) {
    super(...args);
  }
}
```

როგორც ვხედავთ, ის უბრალოდ იძახებს მშობლიური კლასის კონსტრუქტორს. ეს გაგრძელდება მანამ, სანამ არ შევქმნით საკუთარ კონსტრუქტორს.

დავამატოთ კონსტრუქტორი Rabbit-ისთვის. ის name-ს დამატებით დაუყენებს earLength-ს:

```
class Animal {
  constructor(name) {
    this.speed = 0;
    this.name = name;
  }
  // ...
}

class Rabbit extends Animal {

  constructor(name, earLength) {
```

```
    this.speed = 0;
    this.name = name;
    this.earLength = earLength;
  }

  // ...
}

// არ მუშაობს!
let rabbit = new Rabbit("თეთრი კურდღელი", 10); // Error: this is
    not defined.
```

Rabbit-ის შექმნისას - შეცდომაა! რა მოხდა?

მემკვიდრეობით კლასებში კონსტრუქტორებმა ყოველთვის უნდა გამოიძახონ super(...) და ეს უნდა მოხდეს this-ის გამოყენებამდე.

JavaScript-ში არის განსხვავება „მემკვიდრეობითი კლასის ფუნქცია-კონსტრუქტორსა“ და სხვა ყველაფერს შორის. მემკვიდრეობით კლასში შესაბამისი კონსტრუქტორის ფუნქცია აღინიშნება სპეციალური შიდა თვისებით [[ConstructorKind]]:" derived".

განსხვავება შემდეგია:

- როდესაც ჩვეულებრივი კონსტრუქტორი სრულდება, ის ქმნის ცარიელ ობიექტს და ანიჭებს მას this-ს;
- როდესაც მემკვიდრე კლასის კონსტრუქტორის გაშვება ხდება, ის ამას არ აკეთებს. ამის ნაცვლად, ის ელოდება, რომ ამას გააკეთებს მშობლიური კლასის კონსტრუქტორი.

ამიტომ, თუ ჩვენ შევქმნით საკუთარ კონსტრუქტორს, უნდა გამოვიძახოთ super, წინააღმდეგ შემთხვევაში this-ისთვის ობიექტი არ შეიქმნება და მივიღებთ შეცდომას.

იმისათვის, რომ Rabbit კონსტრუქტორმა იმუშაოს, მან this-ის გამოყენებამდე უნდა გამოიძახოს super(), რათა არ მოხდეს შეცდომა:

```
class Animal {

  constructor(name) {
    this.speed = 0;
    this.name = name;
  }

  // ...
}

class Rabbit extends Animal {

  constructor(name, earLength) {
    super(name);
    this.earLength = earLength;
  }

  // ...
}

// ახლა იმუშავებს
let rabbit = new Rabbit("თეთრი კურდღელი", 10);
```

```
alert(rabbit.name); // თეთრი კურდღელი
alert(rabbit.earLength); // 10
```

## კლასის ველების შეცვლა

ჩვენ შეგვიძლია შევცვალოთ არა მხოლოდ მეთოდები, არამედ კლასის ველებიც.

თუმცა, როდესაც ჩვენ ვიღებთ მშობლიურ კონსტრუქტორში შეცვლილ ველზე წვდომას, ეს ქცევა დაპროგრამების მრავალი სხვა ენებისგან განსხვავდება.

განვიხილოთ ეს მაგალითი:

```
class Animal {
  name = 'animal';

  constructor() {
    alert(this.name); // (*)
  }
}

class Rabbit extends Animal {
  name = 'rabbit';
}

new Animal(); // animal
new Rabbit(); // animal
```

აქ Rabbit კლასი აფართოებს Animal-ს და საკუთარი მნიშვნელობით ცვლის name ველს.

Rabbit-ს არ აქვს საკუთარი კონსტრუქტორი, ამიტომ Animal კონსტრუქტორს იძახებს.

საინტერესოა, რომ ორივე შემთხვევაში: new Animal() და new Rabbit(), alert სტრიქონში (*) animal-ს აჩვენებს.

სხვა სიტყვებით რომ ვთქვათ, მშობლიური კონსტრუქტორი ყოველთვის იყენებს საკუთარი ველის მნიშვნელობას და არა შეცვლილს.

რა არის ამაში უცნაური? თუ ჯერ არ არის გასაგები, შეადარეთ მეთოდებს.

აქ არის იგივე კოდი, მაგრამ this.name ველის ნაცვლად, ჩვენ this.showName() მეთოდს ვიძახებთ:

```
class Animal {
  showName() {  // this.name = 'animal'-ის ნაცვლად
    alert('animal');
  }

  constructor() {
    this.showName(); // alert(this.name);-ის ნაცვლად
  }
}

class Rabbit extends Animal {
  showName() {
    alert('rabbit');
  }
}

new Animal(); // animal
```

```
new Rabbit(); // rabbit
```

შედეგი ახლა განსხვავებული მივიღეთ. ეს არის ის, რასაც ჩვენ ველოდით. როდესაც მშობლიური კონსტრუქტორის გამოძახება ხდება წარმოებულ კლასში, ის იყენებს შეცვლილ მეთოდს. უნდა ავღნიშნოთ, რომ კლასის ველებისთვის ეს ასე არ არის. მშობლიური კონსტრუქტორი ყოველთვის იყენებს მშობლიურ ველს. მიზეზი ველების ინიციალიზაციაში მდგომარეობს. ხდება კლასის ველის ინიციალიზაცია :

- საბაზისო კლასის კონსტრუქტორამდე (რომელიც არაფერს არ აფართოებს);
- წარმოებული კლასისთვის super()-ის შემდეგ.

ჩვენს შემთხვევაში, Rabbit არის შეცვლილი კლასი. მას არ აქვს constructor() კონსტრუქტორი. როგორც უკვე აღვნიშნეთ, ეს იგივეა, რომ იყოს ცარიელი კონსტრუქტორი, რომელიც მხოლოდ super(...args)-ს შეიცავს.

ასე რომ, new Rabbit() იძახებს super(), ამგვარად, ასრულებს მშობლიურ კონსტრუქტორს და (წარმოებული კლასების წესის შესაბამისად) მხოლოდ ამის შემდეგ ხდება მისი კლასის ველების ინიციალიზაცია. მშობლიური კონსტრუქტორის შესრულების დროს ჯერ არ არის Rabbit კლასის ველები, ამიტომ Animal ველები გამოიყენება.

ველებსა და მეთოდებს შორის ეს განსხვავება JavaScript-სთვისაა დამახასიათებელი.

საბედნიეროდ, ხდება მხოლოდ მაშინ, როდესაც შეცვლილი ველი მშობლიურ კონსტრუქტორში გამოიყენება. თუ ეს პრობლემად იქცევა, მაშინ მისი გადაჭრა ველების ნაცვლად მეთოდების ან გეტერების/სეტერების გამოყენებითაა შესაძლებელი.

## [[HomeObject]]

წინა ქვეთავში არსებული პრობლემის გადასაჭრელად JavaScript-ში ფუნქციებისთვის დამატებული იყო სპეციალური შიგა თვისება: [[HomeObject]].

როდესაც ფუნქცია გამოცხადებულია კლასში ან ობიექტში როგორც მეთოდი, მისი [[HomeObject]] თვისება ამ ობიექტის ტოლი ხდება. შემდეგ super იყენებს მას მშობლიური პროტოტიპისა და მეთოდების მისაღებად.

მარტივი ობიექტების გამოყენებით ვნახოთ, როგორ მუშაობს ეს თვისება:

```
let animal = {
  name: "ცხოველი",
  eat() {         // animal.eat.[[HomeObject]] == animal
    alert(‘${this.name} ест.’);
  }
};

let rabbit = {
  __proto__: animal,
  name: "კურდღელი",
  eat() {         // rabbit.eat.[[HomeObject]] == rabbit
    super.eat();
  }
};

let longEar = {
  __proto__: rabbit,
```

```
  name: "გრძელყურა",
  eat() {       // longEar.eat.[[HomeObject]] == longEar
    super.eat();
  }
};

// მუშაობს
longEar.eat();  // გრძელყურა ჭამს.
```

ეს მუშაობს ისე, როგორც იყო ჩაფიქრებული [[HomeObject]]-ის გამოყენებით. მეთოდმა, როგორიცაა longEar.eat, იცის თავისი [[HomeObject]] და მისი პროტოტიპიდან იღებს მშობლიურ მეთოდს. ზოგადად, this-ის გამოყენების გარეშე.

ვიცით, რომ JavaScript-ში ფუნქციები „თავისუფალია“ და არ არის მიბმული ობიექტებთან. ობიექტებს შორის შესაძლებელია მათი კოპირება და გამოძახება ნებისმიერი this-ით.

მაგრამ [[HomeObject]]-ის არსებობა არღვევს ამ პრინციპს, რადგან მეთოდებს მათი ობიექტები იმახსოვრებს. [[HomeObject]]-ის შეცვლა შეუძლებელია, ეს ურთიერთობა მუდმივია.

ერთადერთი ადგილი ენაში, სადაც [[HomeObjct]] გამოიყენება არის super. ამიტომ, თუ მეთოდი super-ს არ იყენებს, მაშინ ჩვენ მაინც შეგვიძლია მივიჩნიოთ იგი თავისუფლად და მოვახდინოთ მისი კოპირება ობიექტებს შორის. მაგრამ თუ კოდში არის super, მაშინ ამან შესაძლებელია გამოიწვიოს შეცდომა.

[[HomeObject]] თვისება შეიძლება განისაზღვროს როგორც კლასების, ასევე ჩვეულებრივი ობიექტებისათვის. მაგრამ ობიექტებისთვის, მეთოდები უნდა გამოცხადდეს როგორც method(), და არა "method: function()". ეს განსხვავება მნიშვნელოვანია JavaScript ენისთვის.

# JavaScript HTML დოკუმენტის ობიექტური მოდელი (DOM)

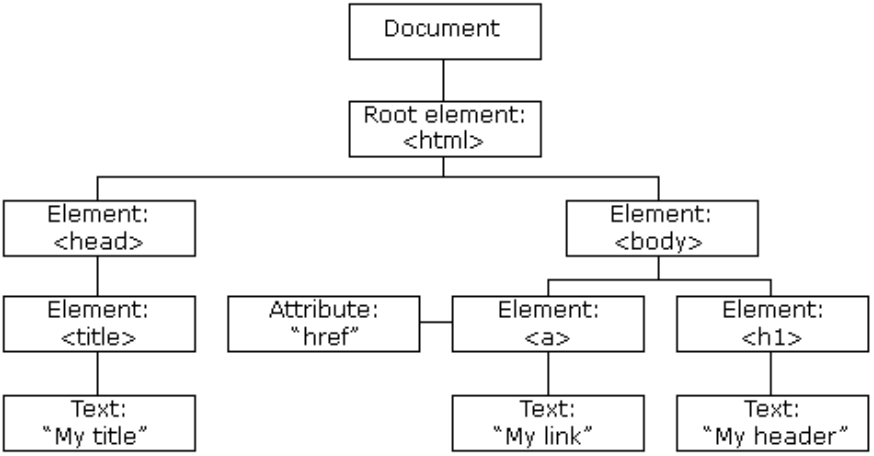
HTML DOM-ით (DOM - Document Object Model - დოკუმენტის ობიექტური მოდელი) JavaScript-ს შეუძლია მიიღოს HTML-დოკუმენტის ყველა ელემენტზე წვდომა და შეცვლა.

## HTML DOM

როდესაც ვებგვერდი იტვირთება, ბრაუზერი ქმნის გვერდის დოკუმენტის ობიექტურ მოდელს.

HTML DOM მოდელი აგებულია როგორც „ობიექტების ხე“.

HTML DOM-ის ობიექტის ხე



ობიექტური მოდელის წყალობით, JavaScript-ი იღებს ყველა იმ შესაძლებლობას, რომელიც საჭიროა დინამიური HTML-დოკუმენტის შესაქმნელად:

- JavaScript-ს შეუძლია ვებგვერდზე შეცვალოს ყველა HTML-ელემენტი;
- JavaScript-ს შეუძლია ვებგვერდზე შეცვალოს ყველა HTML-ატრიბუტი;
- JavaScript-ს შეუძლია შეცვალოს ყველა CSS სტილი;
- JavaScript-ს შეუძლია წაშალოს არსებული HTML-ელემენტები და ატრიბუტები;
- JavaScript-ს შეუძლია ახალი HTML ელემენტების და ატრიბუტების დამატება;
- JavaScript-ს შეუძლია ვებგვერდზე ყველა არსებულ HTML-ქმედებებზე მოახდინოს რეაგირება;
- JavaScript-ს შეუძლია ვებგვერდზე შექმნას ახალი HTML-ქმედებები.

## რა არის DOM

DOM არის W3C (World Wide Web Consortium) სტანდარტი.

DOM განსაზღვრავს დოკუმენტზე წვდომის სტანდარტს:

„W3C Document Object Model (DOM) არის პლატფორმა და ენისგან დამოუკიდებელი ინტერფეისი, რომელიც პროგრამებსა და სკრიპტებს საშუალებას აძლევს დოკუმენტის შინაარსზე, სტრუქტურაზე და სტილზე დინამიურად მიიღონ წვდომა და განახლების შესაძლებლობა“.

W3C DOM სტანდარტი 3 სხვადასხვა ნაწილად არის დაყოფილი:

- Core DOM - სტანდარტული მოდელი ყველა ტიპის დოკუმენტისთვის;
- XML DOM – XML-დოკუმენტების სტანდარტული მოდელი;
- HTML DOM – HTML-დოკუმენტების სტანდარტული მოდელი.

### რა არის HTML DOM

HTML DOM არის სტანდარტული ობიექტური მოდელი და პროგრამული ინტერფეისი HTML-ისთვის. იგი განსაზღვრავს:

- HTML ელემენტებს, როგორც ობიექტებს;
- ყველა HTML-ელემენტის თვისებებს;
- HTML-ის ყველა ელემენტზე წვდომის მეთოდებს;
- ქმედებებს HTML-ის ყველა ელემენტისთვის.

სხვა სიტყვებით რომ ვთქვათ: HTML DOM არის HTML-ელემენტების მიღების, შეცვლის, დამატების ან წაშლის სტანდარტი.

## JavaScript - HTML DOM მეთოდები

HTML DOM მეთოდები არის ის მოქმედებები, რომელთა შესრულებაც შეგიძლიათ HTML ელემენტებზე.

HTML DOM თვისებები არის მნიშვნელობები (HTML ელემენტები), რომელთა მინიჭება ან შეცვლა შეგიძლიათ.

### DOM დაპროგრამების ინტერფეისი

HTML DOM-ზე წვდომა შესაძლებელია JavaScript-ის (და დაპროგრამების სხვა ენების) გამოყენებით.

DOM-ში ყველა HTML ელემენტი განისაზღვრება როგორც ობიექტები.

დაპროგრამების ინტერფეისი არის თითოეული ობიექტის თვისებები და მეთოდები.

თვისება არის მნიშვნელობა, რომელიც შეგიძლიათ მიიღოთ ან მიანიჭოთ (მაგ. შეცვალოთ HTML ელემენტის შინაარსი).

მეთოდი არის მოქმედება, რომელიც შეგიძლიათ შეასრულოთ (მაგ. დაამატოთ ან წაშალოთ HTML ელემენტი).

მაგალითი: მოცემულ მაგალითში, innerHTML ელემენტის შინაარსი ( ) <p> ტეგში შეცვლილია id="demo"-ის საშუალებით:

```
<!DOCTYPE html>
<html>
<body>

<h2>My First Page</h2>

<p id="demo"></p>

<script>
document.getElementById("demo").innerHTML = "Hello World!";
</script>

</body>
</html>
```

HTML ელემენტზე წვდომის ყველაზე გავრცელებული გზა id ელემენტის გამოყენებაა.

ზემოთ მოცემულ მაგალითში, getElementById არის მეთოდი და innerHTML არის თვისება. getElementById მეთოდის გამოყენება ხდება id="demo" ელემენტის საპოვნელად.

ელემენტის შინაარსის მისაღებად ყველაზე მარტივი გზა არის innerHTML თვისების გამოყენება. innerHTML თვისება სასარგებლოა HTML ელემენტების შინაარსის მისაღებად ან შესაცვლელად.

innerHTML თვისება შეიძლება გამოყენებულ იქნას ნებისმიერი HTML ელემენტის მისაღებად ან შესაცვლელად, მათ შორის ტეგების <html> და <body>.

## JavaScript HTML DOM დოკუმენტი

HTML DOM დოკუმენტის ობიექტი არის თქვენს ვებგვერდზე ყველა სხვა ობიექტის მფლობელი. დოკუმენტის ობიექტი წარმოადგენს თქვენს ვებგვერდს. თუ გსურთ HTML გვერდის ნებისმიერ ელემენტზე წვდომის მიღება, ამას ყოველთვის იწყებთ დოკუმენტის ობიექტზე წვდომით.

ქვემოთ მოცემულია რამდენიმე მაგალითი იმისა, თუ როგორ შეგიძლიათ გამოიყენოთ დოკუმენტის ობიექტი HTML-ზე წვდომისა და მართვისთვის.

### HTML ელემენტების ძებნა

| მეთოდი | აღწერა |
|---|---|
| document.getElementById(id) | ელემენტის ძებნა id ელემენტით |
| document.getElementsByTagName(name) | ელემენტების ძებნა ტეგის |

| | |
|---|---|
| | სახელის მიხედვით |
| document.getElementsByClassName(name) | იპოვეთ ელემენტები კლასის სახელის მიხედვით |

## HTML ელემენტების შეცვლა

| თვისება | აღწერა |
|---|---|
| element.innerHTML = new html content | შეცვალეთ inner HTML-ელემენტის შინაარსი |
| element.attribute = new value | შეცვალეთ HTML-ელემენტის ატრიბუტის მნიშვნელობა |
| element.style.property = new style | შეცვალეთ HTML-ელემენტის სტილი |
| მეთოდი | აღწერა |
| element.setAttribute(attribute, value) | შეცვალეთ HTML-ელემენტის ატრიბუტის მნიშვნელობა |

## ელემენტების დამატება და წაშლა

| მეთოდი | აღწერა |
|---|---|
| document.createElement(element) | HTML ელემენტის შექმნა |
| document.removeChild(element) | HTML ელემენტის წაშლა |

| | |
|---|---|
| document.appendChild(element) | HTML ელემენტის დამატება |
| document.replaceChild(new, old) | HTML ელემენტის შეცვლა |
| document.write(text) | HTML-ის გამომავალ ნაკადში ჩაწერა |

## ხდომილების დამუშავების დამატება

| მეთოდი | აღწერა |
|---|---|
| document.getElementById(id).onclick = function(){code} | Onclick ხდომილებაში ხდომილების დამუშავების კოდის დამატება |

## HTML-ობიექტების ძებნა

HTML DOM 1-ის პირველმა დონემ (1998) განსაზღვრა 11 HTML-ობიექტი, ობიექტების კოლექცია და თვისებები. ისინი კვლავ მოქმედებს HTML5-ში. მოგვიანებით, HTML DOM მე-3 დონეზე, უფრო მეტი ობიექტი, კოლექციები და თვისებები დაემატა.

| თვისება | აღწერა | DOM |
|---|---|---|
| document.anchors | აბრუნებს ყველა <a> ელემენტს, რომელსაც აქვს სახელის ატრიბუტი | 1 |
| document.applets | მოძველებულია | 1 |
| document.baseURI | აბრუნებს დოკუმენტის | 3 |

| | | |
|---|---|---|
| | აბსოლუტურ საბაზისო URI-ს | |
| document.body | აბრუნებს <body> ელემენტს | 1 |
| document.cookie | აბრუნებს დოკუმენტის cookie-ფაილს | 1 |
| document.doctype | აბრუნებს დოკუმენტის ტიპს | 3 |
| document.documentElement | აბრუნებს <html> ელემენტს | 3 |
| document.documentMode | აბრუნებს ბრაუზერის მიერ გამოყენებულ რეჟიმს | 3 |
| document.documentURI | აბრუნებს დოკუმენტის URI-ს | 3 |
| document.domain | აბრუნებს დოკუმენტის სერვერის დომენის სახელს | 1 |
| document.domConfig | მოძველებული. | 3 |
| document.embeds | აბრუნებს ყველა <embed> ელემენტს | 3 |
| document.forms | აბრუნებს ყველა <form> ელემენტს | 1 |
| document.head | აბრუნებს <head> ელემენტს | 3 |

| | | |
|---|---|---|
| document.images | აბრუნებს ყველა <img> ელემენტს | 1 |
| document.implementation | აბრუნებს DOM-ის რეალიზებას | 3 |
| document.inputEncoding | აბრუნებს დოკუმენტის დაშიფვრას (სიმბოლოების ნაკრები) | 3 |
| document.lastModified | აბრუნებს დოკუმენტის განახლების თარიღსა და დროს | 3 |
| document.links | აბრუნებს ყველა <area> და <a> ელემენტებს, რომლებსაც აქვთ href ატრიბუტი | 1 |
| document.readyState | აბრუნებს დოკუმენტის (ჩატვირთვის) სტატუსს | 3 |
| document.referrer | აბრუნებს რეფერენტის URI-ს (დამაკავშირებელი დოკუმენტი) | 1 |
| document.scripts | აბრუნებს ყველა <script> ელემენტს | 3 |

| | | |
|---|---|---|
| document.strictErrorChecking | ბრუნდება, თუ შეცდომის შემოწმება განხორციელდება | 3 |
| document.title | აბრუნებს <title> ელემენტს | 1 |
| document.URL | აბრუნებს დოკუმენტის სრულ URL-ს | 1 |

## HTML-ელემენტების ძებნა

აქ გაეცნობით, HTML-გვერდზე თუ როგორ უნდა მოძებნოთ და წვდომა გქონდეთ HTML-ელემენტებზე.

ხშირად JavaScript-ით საჭიროა HTML ელემენტებით მანიპულირება. ამისათვის, ჯერ უნდა იპოვოთ ელემენტები. ამის გაკეთების რამდენიმე გზა არსებობს:

- HTML ელემენტების იდენტიფიკატორით ძებნა;
- HTML ელემენტების ტეგის სახელით ძებნა;
- HTML ელემენტების კლასის სახელით ძებნა;
- HTML ელემენტების CSS სელექტორების გამოყენებით ძებნა;
- HTML ელემენტების HTML ობიექტების კოლექციებში ძებნა.

### HTML ელემენტის იდენტიფიკატორით ძებნა

DOM-ში HTML ელემენტის ძებნის ყველაზე მარტივი გზა ელემენტის იდენტიფიკატორის გამოყენებაა.

ქვემოთ მოყვანილ მაგალითში ხდება ელემენტის ძებნა id="intro"-თი:

მაგალითი:

```
<!DOCTYPE html>
<html>
<body>

<h2>JavaScript HTML DOM</h2>

<p id="intro"> HTML ელემენტების id-ით ძებნა </p>
<p> ეს არის <b>getElementsById</b> მეთოდის გამოყენების
    მაგალითი.</p>

<p id="demo"></p>

<script>
const element = document.getElementById("intro");

document.getElementById("demo").innerHTML =
"ტექსტი შესავალი აბზაციდან ასე გამოიყურება:" +
    element.innerHTML;

</script>

</body>
</html>
```

თუ ელემენტი მოიძებნება, მეთოდი დააბრუნებს ელემენტს, როგორც ობიექტს (ელემენტში), ხოლო თუ ელემენტი ვერ მოიძებნა, მაშინ ელემენტი მიიღებს მნიშვნელობას null-ს.

## HTML ელემენტის ტეგის სახელით ძებნა

ამ მაგალითში ხდება ყველა <p> ტეგის მოძებნა.

მაგალითი:

```
<!DOCTYPE html>
<html>
<body>

<h2>JavaScript HTML DOM</h2>

<p> HTML ელემენტების ტეგის სახელით ძებნა.</p>
<p>ეს არის <b>getElementsByTagName</b> მეთოდის გამოყენების მაგალითი.</p>

<p id="demo"></p>

<script>
const element = document.getElementsByTagName("p");

document.getElementById("demo").innerHTML = ' ტექსტი პირველ აბზაცში (index 0) არის: ' + element[0].innerHTML;

</script>

</body>
```

```
</html>
```

ეს მაგალითი კი ჯერ პოულობს ელემენტს id="main"-ით და შემდეგ პოულობს ყველა <p> ელემენტს „main“-ში:

```
<!DOCTYPE html>
<html>
<body>

<h2>JavaScript HTML DOM</h2>

<div id="main">
<p> HTML ელემენტების ტეგის სახელით ძებნა. </p>
<p>ეს არის <b>getElementsByTagName</b> მეთოდის
    გამოყენების მაგალითი. </p>
</div>

<p id="demo"></p>

<script>
const x = document.getElementById("main");
const y = x.getElementsByTagName("p");

document.getElementById("demo").innerHTML =
'პირველი აბზაცი (index 0) არის "main"-ის შიგნით : ' +
    y[0].innerHTML;

</script>
```

```
</body>
</html>
```

## HTML ელემენტების კლასის სახელით ძებნა

თუ გსურთ იპოვოთ ყველა HTML-ელემენტი კლასის ერთნაირი სახელით, გამოიყენეთ getElementsByClassName() მეთოდი.

ეს მაგალითი აბრუნებს ყველა ელემენტის ჩამონათვალს class="intro" გაფართოებით:

```
<!DOCTYPE html>
<html>
<body>

<h2>JavaScript HTML DOM</h2>

<p> HTML ელემენტების კლასის სახელით ძებნა.</p>
<p class="intro">Hello World!</p>
<p class="intro">ეს მაგალითი <b>getElementsByClassName</b>
    მეთოდის დემონსტრირებას ახდენს.</p>

<p id="demo"></p>

<script>
const x = document.getElementsByClassName("intro");
document.getElementById("demo").innerHTML =
'პირველი აბზაცი (index 0) class="intro"-სთან არის: ' +
    x[0].innerHTML;
```

```
</script>

</body>
</html>
```

## HTML ელემენტების CSS სელექტორების გამოყენებით ძებნა

თუ გსურთ იპოვოთ ყველა HTML ელემენტი, რომელიც შეესაბამება მოცემულ CSS სელექტორს (იდენტიფიკატორი, კლასების სახელი, ტიპები, ატრიბუტები, ატრიბუტების მნიშვნელობები და ა.შ.), ამისათვის გამოიყენეთ querySelectorAll() მეთოდი.

ეს მაგალითი აბრუნებს ყველა <p> ელემენტის სიას class="intro" გაფართოებით:

```
<!DOCTYPE html>
<html>
<body>

<h2>JavaScript HTML DOM</h2>

<p> HTML ელემენტების CSS სელექტორების გამოყენებით
    ძებნა</p>
<p class="intro">Hello World!.</p>
<p class="intro">ეს მაგალითი <b>querySelectorAll</b> მეთოდის
    დემონსტრირებას ახდენს.</p>

<p id="demo"></p>

<script>
```

```
const x = document.querySelectorAll("p.intro");
document.getElementById("demo").innerHTML =
'პირველი აბზაცი (index 0) class="intro"-სთან არის: ' +
    x[0].innerHTML;
</script>

</body>
</html>
```

## HTML ელემენტების HTML ობიექტების კოლექციებში ძებნა

ეს მაგალითი პოულობს ფორმის ელემენტს id="frm1"-ით ფორმების კოლექციაში და გამოაქვს ელემენტის ყველა მნიშვნელობა:

```
<!DOCTYPE html>
<html>
<body>

<h2>JavaScript HTML DOM</h2>
<p>HTML-ელემენტების ძებნა <b>document.forms</b> მეთოდის
    საშუალებით.</p>

<form id="frm1" action="/action_page.php">
 First name: <input type="text" name="fname"
    value="ნიკოლოზ"><br>
 Last name: <input type="text" name="lname"
    value="სტურუა"><br><br>
 <input type="submit" value="Submit">
</form>
```

```
<p> ეს არის თითოეული ელემენტის მნიშვნელობა
    ფორმაში:</p>

<p id="demo"></p>

<script>
const x = document.forms["frm1"];
let text = "";
for (let i = 0; i < x.length ;i++) {
  text += x.elements[i].value + "<br>";
}
document.getElementById("demo").innerHTML = text;
</script>

</body>
</html>
```

HTML ელემენტების ძებნისათვის ასევე ხელმისაწვდომია შემდეგი HTML ობიექტები (და ობიექტების კოლექციები):

- document.anchors
- document.body
- document.documentElement
- document.embeds
- document.forms
- document.head
- document.images
- document.links

- document.scripts
- document.title

## HTML-ის შინაარსის შეცვლა

HTML DOM JavaScript-ს საშუალებას აძლევს HTML-ელემენტების შინაარსი შეცვალოს.

HTML-ელემენტის შინაარსის შეცვლის უმარტივესი გზა არის innerHTML თვისების გამოყენება.

HTML ელემენტის შინაარსის შესაცვლელად გამოიყენეთ შემდეგი სინტაქსი:

document.getElementById(id).innerHTML = new HTML

ქვემოთ მოყვანილი მაგალითი ცვლის <p> ელემენტის შინაარსს:

```
<!DOCTYPE html>
<html>
<body>

<h2>JavaScript-ი ცვლის HTML-ს</h2>

<p id="p1">Hello World!</p>

<script>
document.getElementById("p1").innerHTML = "New text!";
</script>

<p> ზემოთ მოცემული აბზაცი შეიცვალა სკრიპტით.</p>
```

```
</body>
</html>
```

ზემოთ მოყვანილი მაგალითი შემდეგნაირად მუშაობს:

- მოცემული HTML-დოკუმენტი შეიცავს <p> ელემენტს id="p1" იდენტიფიკატორით;
- ჩვენ ვიყენებთ HTML DOM-ს ელემენტის id="p1" იდენტიფიკატორით მისაღებად;
- JavaScript ცვლის ამ ელემენტის შინაარსს (innerHTML) „New text!“.

შემდეგი მაგალითი ცვლის <h1> ელემენტის შინაარსს:

```
<!DOCTYPE html>
<html>
<body>

<h1 id="id01">ძველი სათაური</h1>

<script>
const element = document.getElementById("id01");
element.innerHTML = "ახალი სათაური";
</script>

<p>JavaScript-ის საშუალებით "ძველ სათაური" იცვლება
    "ახალი სათაური"-თ.</p>

</body>
</html>
```

## ატრიბუტის მნიშვნელობის შეცვლა

HTML-ის ატრიბუტის შესაცვლელად შემდეგი სინტაქსი გამოიყენეთ:

document.getElementById(id).attribute = new value

ამ მაგალითში <img> ელემენტის src ატრიბუტის მნიშვნელობა იცვლება:

```
<!DOCTYPE html>
<html>
<body>

<h2>JavaScript HTML DOM</h2>
<img id="image" src="pict0.jpg" width="260" height="220">

<script>
document.getElementById("image").src = "pict1.jpg";
</script>

<p>საწყისი სურათი იყო pict0.jpg, მაგრამ სკრიპტმა იგი pict1.jpg
    სურათით შეცვალა </p>

</body>
</html>
```

- ზემოთ მოცემული HTML დოკუმენტი შეიცავს <img> ელემენტს id="myImage" იდენტიფიკატორით;
- ჩვენ ვიყენებთ HTML DOM-ს ელემენტის id="myImage" იდენტიფიკატორით მისაღებად;

- JavaScript ცვლის ამ ელემენტის src ატრიბუტს "smiley.gif"-ს "landscape.jpg"-ით.

## დინამიური HTML-შინაარსი

JavaScript-ს შეუძლია შექმნას დინამიური HTML-შინაარსი.

მაგალითი:

```
<!DOCTYPE html>
<html>
<body>

<p id="demo"></p>

<script>
document.getElementById("demo").innerHTML = "Date : " + Date();
</script>

</body>
</html>
```

პროგრამის მუშაობის შედეგი იქნება:

```
Date : Fri May 26 2023 16:19:25 GMT+0400 (Georgia Standard Time).
```

## document.write()

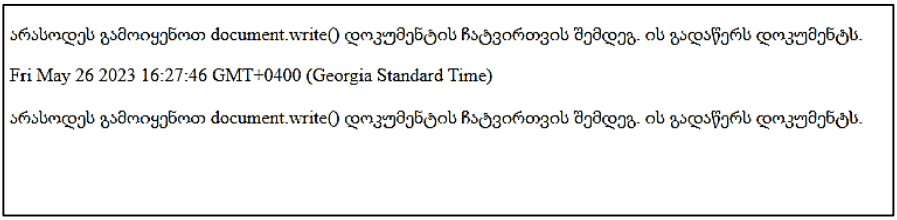
JavaScript-ში document.write() შეიძლება გამოყენებულ იქნას უშუალოდ HTML-ის გამოტანის ნაკადში ჩასაწერად.

მაგალითი:

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

```
<body>

<p> არასოდეს გამოიყენოთ document.write() დოკუმენტის
ჩატვირთვის შემდეგ. ის გადაწერს დოკუმენტს.</p>

<script>
document.write(Date());
</script>

<p> არასოდეს გამოიყენოთ document.write() დოკუმენტის
ჩატვირთვის შემდეგ. ის გადაწერს დოკუმენტს.</p>

</body>
</html>
```

შედეგი იქნება შემდეგი:




არასოდეს გამოიყენოთ document.write() დოკუმენტის ჩატვირთვის შემდეგ. ის გადაწერს დოკუმენტს.

## JavaScript- ფორმები

HTML ფორმის შემოწმება შეიძლება განხორციელდეს JavaScript საშუალებით.

თუ ფორმის ველი (fname) ცარიელია, ამ ფუნქციას გამოაქვს შეტყობინება და აბრუნებს false-ს, რათა არ მოხდეს ფორმის გაგზავნა.

JavaScript მაგალითი:

```
function validateForm() {
      let x = document.forms["myForm"]["fname"].value;
      if (x == "") {
        alert("სახელი აუცილებლად უნდა შეივსოს");
        return false;
      }
    }
```

ფუნქციის გამოძახება ფორმის გაგზავნითაა შესაძლებელი.

HTML-ფორმის მაგალითი:

```
<!DOCTYPE html>
<html>
<head>
<script>
function validateForm() {
  let x = document.forms["myForm"]["fname"].value;
  if (x == "") {
    alert("სახელი აუცილებლად უნდა შეივსოს");
    return false;
  }
}
</script>
</head>
<body>
```

```
<h2>JavaScript Validation</h2>

<form name="myForm" action="/action_page.php"
    onsubmit="return validateForm()" method="post">
  Name: <input type="text" name="fname">
  <input type="submit" value="Submit">
</form>

</body>
</html>
```

JavaScript ხშირად გამოიყენება რიცხვითი მონაცემების შეტანის შესამოწმებლად:

```
<!DOCTYPE html>
<html>
<body>

<h2>JavaScript Validation</h2>

<p> გთხოვთ, შეიყვანოთ რიცხვი 1-დან 10-მდე:</p>

<input id="numb">

<button type="button" onclick="myFunction()">Submit</button>

<p id="demo"></p>
```

```
<script>
function myFunction() {
 // მიიღეთ ინფორმაციის შეყვანის ველის მნიშვნელობა
    id="numb" იდენტიფიკატორით
 let x = document.getElementById("numb").value;
 // თუ x არ არის რიცხვი ან ერთზე ნაკლებია ან 10-ზე მეტი
 let text;
 if (isNaN(x) || x < 1 || x > 10) {
   text = "მონაცემი არასწორადაა შეტანილი";
 } else {
   text = "მონაცემი სწორადაა შეტანილი";
 }
 document.getElementById("demo").innerHTML = text;
}
</script>

</body>
</html>
```

HTML ფორმის შემოწმება ბრაუზერის მიერ შეიძლება ავტომატურად განხორციელდეს. თუ ფორმის fname ველი ცარიელია, მაშინ required ატრიბუტი ხელს შეუშლის ამ ფორმის გაგზავნას.

HTML ფორმის მაგალითი:

```
<!DOCTYPE html>
<html>
<body>
```

```
<h2>JavaScript Validation</h2>

<form action="/action_page.php" method="post">
  <input type="text" name="fname" required>
  <input type="submit" value="Submit">
</form>

<p> თუ ხელი დააჭირეთ Submit ღილაკს, ტექსტის ველის
    შევსების გარეშე,
თქვენი ბრაუზერი შეცდომის შეტყობინებას გამოიტანს.</p>

</body>
</html>
```

მონაცემთა შემოწმება არის პროცესი იმის უზრუნველ-საყოფად, რომ მომხმარებლის მიერ შეტანილი ინფორმაცია არის სუფთა, სწორი და სასარგებლო.

შემოწმების ტიპიური ამოცანებია:

- შეავსო თუ არა მომხმარებელმა ყველა საჭირო ველი?
- შეიყვანა თუ არა მომხმარებელმა სწორი თარიღი?
- შეიყვანა თუ არა მომხმარებელმა ტექსტი რიცხვით ველში?

ყველაზე ხშირად, მონაცემთა შემოწმების მიზანი იმის უზრუნველყოფაა, რომ მომხმარებლის მიერ შეტანილი ინფორმაცია სწორია.

შემოწმება შეიძლება სხვადასხვა გზითა და სხვადასხვა სახით განხორციელდეს.

სერვერის მხარეს შემოწმება ვებ-სერვერის მიერ მას შემდეგ ხორციელდება, რაც შეტანილი მონაცემები გაიგზავნა სერვერზე.

კლიენტის მხრიდან შემოწმება, ვებ-ბრაუზერის მიერ შეტანილი მონაცემების ვებ-სერვერზე გადაგზავნამდე ხორციელდება.

HTML5-ში შემოიტანეს ახალი HTML შემოწმების კონცეფცია, რომელსაც შეზღუდვების შემოწმება ეწოდება.

HTML შეზღუდვების შემოწმება ეფუძნება:

- HTML ინფორმაციის შეტანის ატრიბუტების შეზღუდვების შემოწმებას;
- CSS ფსევდო-სელექტორების შეზღუდვის შემოწმება;
- DOM თვისებების და მეთოდების შეზღუდვების შემოწმება.

## HTML ინფორმაციის შეტანის ატრიბუტების შეზღუდვების შემოწმება

| ატრიბუტი | აღწერა |
|---|---|
| disabled | მიუთითებს, რომ შესატანი ელემენტის ველი უნდა იყოს გამორთული |
| max | განსაზღვრავს შესატანი ელემენტის მაქსიმალურ მნიშვნელობას |
| min | განსაზღვრავს შესატანი ელემენტის მინიმალურ მნიშვნელობას |
| pattern | განსაზღვრავს შესატანი ელემენტის მნიშვნელობის ნიმუშს |
| required | მიუთითებს, რომ შესატანი ელემენტის ველი აუცილებლად უნდა იყოს შევსებული |
| type | განსაზღვრავს შესატანი ელემენტის ტიპს |

### CSS ფსევდო-სელექტორების შეზღუდვის შემოწმება

| სელექტორი | აღწერა |
|---|---|
| :disabled | ირჩევს შესატან ელემენტებს მითითებული „disabled“ ატრიბუტით |
| :invalid | ირჩევს შესატან ელემენტებს არასწორი მნიშვნელობებით |
| :optional | ირჩევს შესატან ელემენტებს „required“ ატრიბუტის მითითების გარეშე |
| : required | ირჩევს შესატან ელემენტებს მითითებული „required“ ატრიბუტით |
| :valid | ირჩევს შესატან ელემენტებს სწორი მნიშვნელობებით |

## HTML სტილის შეცვლა

HTML DOM საშუალებას აძლევს JavaScript-ს შეცვალოს HTML ელემენტების სტილი. HTML ელემენტის სტილის შესაცვლელად შემდეგი სინტაქსი გამოიყენეთ:

document.getElementById(id).style.property = new style

შემდეგ მაგალითში <p> ელემენტის სტილი იცვლება:

```
<!DOCTYPE html>
<html>
<body>

<h2>JavaScript HTML DOM</h2>
```

```
<p> HTML სტილის შეცვლა:</p>

<p id="p1">Hello World!</p>
<p id="p2">Hello World!</p>

<script>
document.getElementById("p2").style.color = "blue";
document.getElementById("p2").style.fontFamily = "Arial";
document.getElementById("p2").style.fontSize = "larger";
</script>

</body>
</html>
```

HTML DOM საშუალებას იძლევა შესრულდეს კოდი, როდესაც რაიმე ხდომილება ხდება. ხდომილება ირთვება ბრაუზერის მიერ, როდესაც HTML ელემენტებთან „რაღაც ხდება“:

- ელემენტზე მაუსის დაწკაპუნება;
- გვერდის ჩატვირთვა;
- ინფორმაციის შეყვანის ველები შეიცვალა.

ამ მაგალითში, HTML ელემენტს სტილი აქვს id="id1", როდესაც მომხმარებელი ღილაკზე მაუსით დააწკაპუნებს:

```
<!DOCTYPE html>
<html>
<body>

<h1 id="id1">My Heading 1</h1>
```

```
<button type="button"
onclick="document.getElementById('id1').style.color = 'red'">
Click Me!</button>

</body>
</html>
```

## JavaScript HTML DOM ანიმაცია

შევისწავლოთ JavaScript-ით როგორ შექმნათ HTML ანიმაცია. იმის საჩვენებლად, თუ JavaScript-ით როგორ შევქმნათ HTML ანიმაციები, ჩვენ მარტივ ვებგვერდს გამოვიყენებთ:

```
<!DOCTYPE html>
<html>
<body>

<h1>My First JavaScript Animation</h1>

<div id="animation">ჩემი ანიმაცია აქ შესრულდება</div>

</body>
</html>
```

ყველა ანიმაცია კონტეინერის ელემენტთან უნდა იყოს დაკავშირებული. კონტეინერის ელემენტი უნდა შეიქმნას style="position: relative"-ით, ხოლო ანიმაციის ელემენტი კი style="position: absolute" სტილით.

```
<!Doctype html>
<html>
<style>
#container {
  width: 400px;
  height: 400px;
  position: relative;
  background: yellow;
}
#animate {
  width: 50px;
  height: 50px;
  position: absolute;
  background: red;
}
</style>
<body>

<h2>My First JavaScript Animation</h2>

<div id="container">
<div id="animate"></div>
</div>

</body>
</html>
```

JavaScript ანიმაციები მიიღწევა დაპროგრამებით ელემენტზე სტილის თანდათანობითი ცვლილებების გზით.

ცვლილებები ხდება ტაიმერის მიერ. როდესაც ტაიმერის ინტერვალი მცირეა, ანიმაცია უწყვეტად გამოიყურება.

მთავარი კოდი:

```
id = setInterval(frame, 5);

function frame() {
    if (/* ტესტის დასრულება */) {
        clearInterval(id);
    } else {
        /* ელემენტის სტილის შესაცვლელი კოდი */
    }
}
```

JavaScript-ით შექმნილ ანიმაციას შემდეგი სახე ექნება:

```
<!DOCTYPE html>
<html>
<style>
#container {
  width: 400px;
  height: 400px;
  position: relative;
  background: yellow;
}
#animate {
  width: 50px;
  height: 50px;
  position: absolute;
```

```
  background-color: red;
}
</style>
<body>

<p><button onclick="myMove()">Click Me</button></p>

<div id ="container">
  <div id ="animate"></div>
</div>

<script>
function myMove() {
  let id = null;
  const elem = document.getElementById("animate");
  let pos = 0;
  clearInterval(id);
  id = setInterval(frame, 5);
  function frame() {
    if (pos == 350) {
      clearInterval(id);
    } else {
      pos++;
      elem.style.top = pos + "px";
      elem.style.left = pos + "px";
    }
  }
}
```

```
</script>

</body>
</html>
```

## JavaScript HTML DOM ხდომილება

HTML DOM-ი JavaScript-ს საშუალებას აძლევს უპასუხოს HTML ხდომილებას.

JavaScript შეიძლება შესრულდეს, როდესაც რაიმე ხდომილება ხდება, მაგალითად, როდესაც მომხმარებელი HTML ელემენტზე მაუსით დააწკაპუნებს.

კოდის შესასრულებლად, როდესაც მომხმარებელი მაუსით ელემენტზე დააწკაპუნებს HTML მოვლენის ატრიბუტში უნდა დაამატოთ JavaScript კოდი:

```
onclick=JavaScript
```

HTML ხდომილების მაგალითები:

- როდესაც მომხმარებელი მაუსით დააწკაპუნებს;
- როდესაც მოხდება ვებგვერდის ჩატვირთვა;
- როდესაც მოხდება სურათის ჩატვირთვა;
- როდესაც მაუსი გადაადგილდება ელემენტზე;
- როდესაც ინფორმაციის შეტანის ველი იცვლება;
- HTML ფორმის გაგზავნისას;
- როდესაც მომხმარებელი მაუსით დააწკაპუნებს ღილაკზე.

ქვემოთ მოყვანილ მაგალითში, <h1> ელემენტის შინაარსი იცვლება, როდესაც მომხმარებელი დააწკაპუნებს მასზე:

```
<!DOCTYPE html>
<html>
<body>

<h2>JavaScript HTML ხდომილება</h2>
<h2 onclick="this.innerHTML='ოოო!'">მაუსით დააწკაპუნე ამ
    ტექსტზე!</h2>

</body>
</html>
```

ხოლო ამ მაგალითში ფუნქცია გამოძახებულია ხდომილების დამმუშავებლისგან:

```
<!DOCTYPE html>
<html>
<body>

<h2>JavaScript HTML ხდომილება</h2>
<h2 onclick="changeText(this)"> მაუსით დააწკაპუნე ამ
    ტექსტზე!</h2>

<script>
function changeText(id) {
  id.innerHTML = "Ooops!";
}
</script>

</body>
```

```
</html>
```

## HTML ხდომილების ატრიბუტები

HTML ელემენტებისთვის ხდომილების მინიჭებისთვის, შეგიძლიათ გამოიყენოთ ხდომილების ატრიბუტები.

მიამაგრეთ onclick ხდომილება ღილაკის ელემენტს:

```
<!DOCTYPE html>
<html>
<body>

<h2>JavaScript HTML Events</h2>
<p>თარიღის სანახავად მაუსით დააწკაპუნეთ ღილაკზე.</p>

<button onclick="displayDate()">რა დროა ახლა?</button>

<script>
function displayDate() {
  document.getElementById("demo").innerHTML = Date();
}
</script>

<p id="demo"></p>

</body>
</html>
```

ზემოთ მოცემულ მაგალითში ფუნქცია სახელად displayDate შესრულდება მაუსით ღილაკზე დაწკაპუნების დროს.

## ხდომილების მინიჭება HTML DOM-ის გამოყენებით

HTML DOM საშუალებას გაძლევთ JavaScript-ის გამოყენებით HTML ელემენტებს მიაკუთვნოთ ხდომილება:

onclick ხდომილება ღილაკის ელემენტს მიამაგრეთ:

```
<!DOCTYPE html>
<html>
<body>

<h2>JavaScript HTML Events</h2>
<p>მაუსით დააწკაპუნეთ ღილაკზე "სცადეთ", რათა
    შეასრულოთ displayDate() ფუნქცია.</p>

<button id="myBtn">სცადეთ</button>

<p id="demo"></p>

<script>
document.getElementById("myBtn").onclick = displayDate;

function displayDate() {
  document.getElementById("demo").innerHTML = Date();
}
</script>
```

```
</body>
</html>
```

ზემოთ მოცემულ მაგალითში ფუნქცია სახელად displayDate ენიჭება HTML ელემენტს id="myBtn" გაფართოებით. ფუნქცია შესრულდება ღილაკზე მაუსით დაწკაპუნების შემდეგ.

## onload და onunload ხდომილება

onload და onunload ხდომილება ირთვება, როდესაც მომხმარებელი შედის ან ტოვებს გვერდზე.

onload ხდომილება შეიძლება გამოყენებულ იქნას ვიზიტორის ბრაუზერის ტიპისა და ვერსიის შესამოწმებლად და ინფორმაციის საფუძველზე ვებგვერდის სწორი ვერსიის ჩასატვირთად.

onload და onunload ხდომილება გამოყენება შესაძლებელია cookie-ფაილებთან სამუშაოდ.

```
<!DOCTYPE html>
<html>
<body onload="checkCookies()">

<h2>JavaScript HTML Events</h2>

<p id="demo"></p>

<script>
function checkCookies() {
  var text = "";
  if (navigator.cookieEnabled == true) {
```

```
  text = "cookie-ფაილები ჩართულია.";
 } else {
  text = "cookie-ფაილები გამორთულია";
 }
 document.getElementById("demo").innerHTML = text;
}
</script>

</body>
</html>
```

## onchange ხდომილება

onchange ხდომილება ხშირად გამოიყენება ინფორმაციის შეყვანის ველის შემოწმებასთან ერთად. ქვემოთ მოცემულია მაგალითი onchange-ის გამოყენებით. upperCase() ფუნქციის გამოძახება მოხდება, როდესაც მომხმარებელი შეცვლის ინფორმაციის შეყვანის ველის შინაარსს.

```
<!DOCTYPE html>
<html>
<body>

<h2>JavaScript HTML Events</h2>
შეიყვანეთ თქვენი სახელი: <input type="text" id="fname"
    onchange="upperCase()">
<p> ინფორმაციის შეყვანის ველიდან გასვლისას ამოქმედდება
    ფუნქცია, რომელიც შეყვანილ ტექსტს ზედა რეგისტრში
    გადაიყვანს.</p>
```

```
<script>
function upperCase() {
  const x = document.getElementById("fname");
  x.value = x.value.toUpperCase();
}
</script>

</body>
</html>
```

### onmouseover და onmouseout ხდომილება

onmouseover და onmouseout ხდომილება შეიძლება გამოყენებულ იქნას ფუნქციის გასააქტიურებლად, როდესაც მომხმარებელი მაუსის მაჩვენებელს დააყენებს HTML ელემენტზე ან გამოვა მის გარეთ:

```
<!DOCTYPE html>
<html>
<body>

<div onmouseover="mOver(this)" onmouseout="mOut(this)"
style="background-
    color:#D94A38;width:180px;height:40px;padding:20px;">
მაუსის მაჩვენებელი ობიექტის გარეთაა</div>

<script>
function mOver(obj) {
  obj.innerHTML = "გმადლობთ!"
}
```

```
function mOut(obj) {
  obj.innerHTML = "მაუსის მაჩვენებელი ობიექტის გარეთაა"
}
</script>

</body>
</html>
```

## onmousedown, onmouseup და onclick ხდომილება

onmousedown, onmouseup და onclick ხდომილება მაუსის დაწკაპუნების ნაწილია. ჯერ მაუსის ღილაკზე დაწკაპუნებისას ირთვება onmousedown ხდომილება, შემდეგ მაუსით ღილაკის გათავისუფლებისას ირთვება onmouseup ხდომილება, ბოლოს, როდესაც მაუსის დაწკაპუნება დასრულდება, onclick ხდომილება ირთვება.

```
<!DOCTYPE html>
<html>
<body>

<div onmousedown="mDown(this)" onmouseup="mUp(this)"
style="background-
    color:#D94A38;width:120px;height:20px;padding:20px;">
დააწკაპუნეთ</div>

<script>
function mDown(obj) {
  obj.style.backgroundColor = "#1ec5e5";
```

```
  obj.innerHTML = "გამათავისუფლე";
}

function mUp(obj) {
  obj.style.backgroundColor="#D94A38";
  obj.innerHTML="გმადლობთ!";
}
</script>

</body>
</html>
```

## JavaScript HTML DOM EventListener

### addEventListener() მეთოდი

დაამატეთ ხდომილების მსმენელი, რომელიც ირთვება, როდესაც მომხმარებელი მაუსს დააჭერს ღილაკზე:

```
document.getElementById("myBtn").addEventListener("click",
    displayDate);
```

მაგალითი:

```
<!DOCTYPE html>
<html>
<body>

<h2>JavaScript addEventListener()</h2>
```

```
<p> ეს მაგალითი იყენებს addEventListener() მეთოდს ღილაკზე
    მაუსის დაწკაპუნების ხდომილების დასამაგრებლად.</p>

<button id="myBtn">სცადე</button>

<p id="demo"></p>

<script>
document.getElementById("myBtn").addEventListener("click",
    displayDate);

function displayDate() {
  document.getElementById("demo").innerHTML = Date();
}
</script>

</body>
</html>
```

addEventListener() მეთოდი მითითებულ ელემენტს მიანიჭებს ხდომილების დამმუშავებელს. addEventListener() მეთოდი ხდომილების დამმუშავებელს ელემენტს მიანიჭებს, არსებული ღონისძიების დამმუშავებლების გადაწერის გარეშე. თქვენ შეგიძლიათ მრავალი ხდომილების დამმუშავებელი ერთ ელემენტს დაამატოთ.

თქვენ შეგიძლიათ დაამატოთ ერთი და იგივე ტიპის მრავალი ხდომილების დამმუშავებელი ერთ ელემენტს, მაგალითად, მაუსის ორი „დაწკაპუნება“.

თქვენ შეგიძლიათ დაამატოთ ხდომილების მსმენელები ნებისმიერ DOM ობიექტს და არა მხოლოდ HTML ელემენტებს. ანუ, ფანჯრის ობიექტი.

addEventListener() მეთოდი აადვილებს კონტროლს, თუ როგორ რეაგირებს ხდომილება ზემოთ ასვლაზე.

addEventListener() მეთოდის გამოყენებისას JavaScript-ი გამოყოფილია HTML-ის მარკირებისგან უკეთესი წაკითხვისთვის და საშუალებას გაძლევთ დაამატოთ ხდომილების მსმენელები მაშინაც კი, თუ არ გაქვთ HTML-ის მარკირებაზე კონტროლი.

თქვენ შეგიძლიათ მარტივად წაშალოთ ხდომილების მსმენელი removeEventListener() მეთოდის გამოყენებით.

მისი სინტაქსია:

```
element.addEventListener(event, function, useCapture);
```

პირველი პარამეტრი არის ხდომილების ტიპი (მაგალითად, „click“ ან „mousedown“ ან ნებისმიერი სხვა HTML DOM ხდომილება).

მეორე პარამეტრი არის ფუნქცია, რომელიც გვინდა გამოვიძახოთ ხდომილების დადგომისას.

მესამე პარამეტრი არის ლოგიკური მნიშვნელობა, რომელიც მიუთითებს, გამოიყენოს ხდომილების ინფორმაცია. ეს პარამეტრი არჩევითია.

გაითვალისწინეთ, რომ თქვენ არ იყენებთ „on“ პრეფიქსს ხდომილებისთვის; გამოიყენეთ „click“-ი „onclick“-ის ნაცვლად.

მაგალითი. შეტყობინება „Hello World!“ გამოჩნდება, როდესაც მომხმარებელი მაუსით დააწკაპუნებს ღილაკზე:

```
<!DOCTYPE html>
```

```
<html>
<body>

<h2>JavaScript addEventListener()</h2>

<p> ეს მაგალითი იყენებს addEventListener() მეთოდს, როდესაც
    მომხმარებელი მაუსით დააწკაპუნებს ღილაკზე.</p>

<button id="myBtn">სცადე</button>

<script>
document.getElementById("myBtn").addEventListener("click",
    function() {
  alert("Hello World!");
});
</script>

</body>
</html>
```

თქვენ ასევე შეგიძლიათ მიმართოთ გარე ფუნქციას, მაგალითად:

```
<!DOCTYPE html>
<html>
<body>

<h2>JavaScript addEventListener()</h2>
```

```
<p> ეს მაგალითი იყენებს addEventListener() მეთოდს ფუნქციის
    შესასრულებლად, როდესაც მომხმარებელი მაუსით
    დააწკაპუნებს ღილაკზე.</p>

<button id="myBtn">Try it</button>

<script>
document.getElementById("myBtn").addEventListener("click",
    myFunction);

function myFunction() {
  alert ("Hello World!");
}
</script>

</body>
</html>
```

addEventListener() მეთოდი საშუალებას გაძლევთ დაამატოთ რამდენიმე ხდომილება იმავე ელემენტში არსებული ხდომილების გადაწერის გარეშე.

მაგალითი:

```
<!DOCTYPE html>
<body>

<h2>JavaScript addEventListener()</h2>
```

```
<p> ეს მაგალითი იყენებს addEventListener() მეთოდს იმავე
    ღილაკზე მაუსით ორი დაწკაპუნების ხდომილების
    დასამატებლად.</p>

<button id="myBtn">სცადე</button>

<script>
let x = document.getElementById("myBtn");
x.addEventListener("click", myFunction);
x.addEventListener("click", someOtherFunction);

function myFunction() {
  alert ("Hello World!");
}

function someOtherFunction() {
  alert ("ეს ფუნქციაც შესრულდა!");
}
</script>

</body>
</html>
```

თქვენ შეგიძლიათ სხვადასხვა ტიპის ხდომილებები ერთსა და იმავე ელემენტს დაამატოთ.

მაგალითი:

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

```
<body>

<h2>JavaScript addEventListener()</h2>

<p> ეს მაგალითი იყენებს addEventListener() მეთოდს ერთი და
    იმავე ღილაკზე მრავალი ხდომილების დასამატებლად.</p>

<button id="myBtn">სცადე</button>
<p id="demo"></p>

<script>
let x = document.getElementById("myBtn");
x.addEventListener("mouseover", myFunction);
x.addEventListener("click", mySecondFunction);
x.addEventListener("mouseout", myThirdFunction);

function myFunction() {
  document.getElementById("demo").innerHTML  +=  "Moused
    over!<br>";
}

function mySecondFunction() {
  document.getElementById("demo").innerHTML += "Clicked!<br>";
}

function myThirdFunction() {
  document.getElementById("demo").innerHTML  +=  "Moused
    out!<br>";
```

```
}
</script>

</body>
</html>
```

დაამატეთ ხდომილების დამამუშავებელი ფანჯრის ობიექტს

addEventListener() მეთოდი საშუალებას გაძლევთ დაამატოთ ხდომილების დამმუშავებელი ნებისმიერ HTML DOM ობიექტს, როგორიცაა HTML ელემენტები, HTML დოკუმენტი, ფანჯრის ობიექტი ან სხვა ხდომილების მხარდამჭერი ობიექტი, როგორიცაა xmlHttpRequest ობიექტი.

მაგალითი. დაამატეთ ხდომილების დამმუშავებელი, რომელიც მაშინვე ჩაირთვება, როდესაც მომხმარებელი ფანჯრის ზომას შეცვლის:

```
<!DOCTYPE html>
<html>
<body>

<h2>JavaScript addEventListener()</h2>

<p> ეს მაგალითი ფანჯრის ობიექტისთვის იყენებს
    addEventListener() მეთოდს.</p>

<p> სცადეთ შეცვალოთ ბრაუზერის ფანჯრის ზომა, რათა
    გააქტიუროთ "resize" ხდომილება.</p>
```

```
<p id="demo"></p>

<script>
window.addEventListener("resize", function(){
  document.getElementById("demo").innerHTML = Math.random();
});
</script>

</body>
</html>
```

## პარამეტრების გადაცემა

პარამეტრის მნიშვნელობების გადაცემისას გამოიყენეთ „ანონიმური ფუნქცია“, რომელიც იძახებს მითითებულ ფუნქციას პარამეტრებით. მაგალითი:

```
<!DOCTYPE html>
<html>
<body>
<h2>JavaScript addEventListener()</h2>

<p> ეს მაგალითი გვიჩვენებს, თუ როგორ უნდა გადაიტანოთ
    პარამეტრების მნიშვნელობები addEventListener() მეთოდის
    გამოყენებისას.</p>
<p>დააწკაპუნეთ ღილაკზე გაანგარიშების
    შესასრულებლად.</p>
<button id="myBtn">სცადე</button>

<p id="demo"></p>
```

```
<script>
let p1 = 5;
let p2 = 7;
document.getElementById("myBtn").addEventListener("click",
    function() {
  myFunction(p1, p2);
});

function myFunction(a, b) {
  document.getElementById("demo").innerHTML = a * b;
}
</script>

</body>
</html>
```

HTML DOM-ში ხდომილების გავრცელების ორი გზა არსებობს: Bubbling (ბუშტუკები) და Capturing (ხელის დაჭერა).

ხდომილების გავრცელება არის საშუალება, რათა დადგინდეს ელემენტების რიგი, როდესაც ხდება ხდომილება. თუ თქვენ გაქვთ <p> ელემენტი  ელემენტში და მომხმარებელი დააწკაპუნებს <p> ელემენტზე, მაუსით რომელ ელემენტზე დაწკაპუნება უნდა დამუშავდეს პირველ რიგში?

ბუშტუკების დროს, ჯერ მუშავდება ყველაზე შიდა ელემენტის ხდომილება, შემდეგ კი ყველაზე გარე ელემენტის: ჯერ მუშავდება <p> მაუსით ელემენტზე დაწკაპუნების ხდომილება, შემდეგ  ელემენტზე დაწკაპუნების ხდომილება.

ხელის დაჭერის შემთხვევაში ჯერ დამუშავდება ყველაზე გარე ელემენტის შემდეგ კი შიდა ელემენტის ხდომილება: ჯერ დამუშავდება  ელემენტზე დაწკაპუნების ხდომილება, შემდეგ კი <p> ელემენტზე დაწკაპუნების ხდომილება.

addEventListener() მეთოდით შეგიძლიათ მიუთითოთ რიგითობის ტიპი "useCapture" პარამეტრის გამოყენებით:

```
addEventListener(event, ფუნქცია, useCapture);
```

ნაგულისხმევი მნიშვნელობა არის false, რომელიც გამოიყენებს pop-up გავრცელებას. თუ მნიშვნელობაა true, ხდომილება ხელის დაჭერას იყენებს.

მაგალითი:

```
<!DOCTYPE html>
<html>
<head>
<style>
#myDiv1, #myDiv2 {
  background-color: coral;
 padding: 50px;
}
#myP1, #myP2 {
  background-color: white;
  font-size: 20px;
  border: 1px solid;
  padding: 20px;
}
</style>
```

```
meta content="text/html; charset=utf-8" http-equiv="Content-
    Type">
</head>
<body>
<h2>JavaScript addEventListener()</h2>
<div id="myDiv1">

  <h2>Bubbling:</h2>
  <p id="myP1">Click me!</p>
</div>
<br>
<div id="myDiv2">
  <h2>Capturing:</h2>
  <p id="myP2">Click me!</p>
</div>
<script>
document.getElementById("myP1").addEventListener("click",
    function() {
  alert("თქვენ დააწკაპუნეთ თეთრ ელემენტზე!");
}, false);
document.getElementById("myDiv1").addEventListener("click",
    function() {
  alert("თქვენ დააწკაპუნეთ ნარინჯისფერ ელემენტზე!");
}, false);
document.getElementById("myP2").addEventListener("click",
    function() {
  alert("თქვენ დააწკაპუნეთ თეთრ ელემენტზე!");
}, true);
```

```
document.getElementById("myDiv2").addEventListener("click",
    function() {
  alert("თქვენ დააწკაპუნეთ ნარინჯისფერ ელემენტზე!");
}, true);
</script>
</body>
</html>
```

## removeEventListener() მეთოდი

removeEventListener() მეთოდი წაშლის ხდომილების დამმუშავებლებს, რომელთა მიმაგრება addEventListener() მეთოდის გამოყენებით მოხდა. მაგალითი:

```
<!DOCTYPE html>
<html>
<head>
<style>
#myDIV {
  background-color: coral;
  border: 1px solid;
  padding: 50px;
  color: white;
  font-size: 20px;
}
</style>
</head>
<body>

<h2>JavaScript removeEventListener()</h2>
```

```
<div id="myDIV">
  <p> ამ div ელემენტს აქვს onmousemove ხდომილების
     დამმუშავებელი, რომელიც ყოველ ჯერზე შემთხვევით
     რიცხვს აჩვენებს, როცა მაუსის მაჩვენებელით ამ
     ნარინჯისფერ ველში გადაადგილდებით.</p>
   <p>დააწკაპუნეთ ღილაკზე div-ის ხდომილების
     დამმუშავებლის წასაშლელად.</p>
  <button onclick="removeHandler()" id="myBtn">წაშალე</button>
</div>

<p id="demo"></p>

<script>
document.getElementById("myDIV").addEventListener("mousemo
     ve", myFunction);

function myFunction() {
  document.getElementById("demo").innerHTML = Math.random();
}

function removeHandler() {

     document.getElementById("myDIV").removeEventListener("m
     ousemove", myFunction);
}
</script>
```

```
</body>
</html>
```

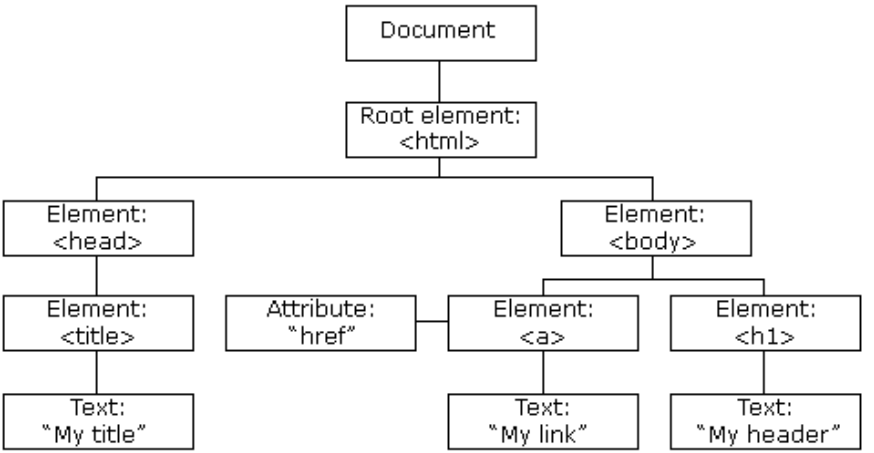
## JavaScript HTML DOM-ნავიგაცია

HTML DOM საშუალებით შეგიძლიათ „კვანძის ხე“-ზე გადაადგილდეთ კვანძებს შორის დამოკიდებულების გამოყენებით.

### DOM-კვანძები

W3C HTML DOM სტანდარტის მიხედვით, HTML-დოკუმენტში ყველაფერი არის კვანძი:

- მთელი დოკუმენტი არის დოკუმენტის კვანძი;
- თითოეული HTML ელემენტი არის ელემენტის კვანძი;
- ტექსტი HTML ელემენტების შიგნით არის ტექსტის კვანძები;
- ყოველი HTML ატრიბუტი არის ატრიბუტის კვანძი (მოძველებულია);
- ყველა კომენტარი არის კომენტარის კვანძი;




HTML DOM დახმარებით, „კვანძის ხე“-ს ყველა კვანძზე წვდომა JavaScript-ის გამოყენებითაა შესაძლებელი.

თქვენ შეგიძლიათ შექმნათ ახალი კვანძები და ყველა კვანძი შეიძლება შეიცვალოს ან წაიშალოს.

## კვანძებს შორის დამოკიდებულება

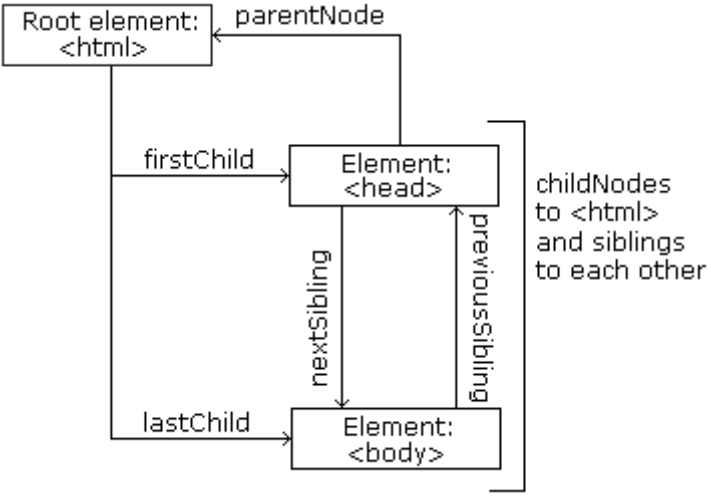
„კვანძის ხე“-ში კვანძებს ერთმანეთთან იერარქიული დამოკიდებულება აქვთ.

ტერმინები „მშობლიური“ და „შვილობილი“ გამოიყენება ურთიერთობების აღსაწერად.

- „კვანძის ხე“-ში ყველაზე ზედა კვანძს ფესვი (ან ძირეული კვანძი) ეწოდება;
- თითოეულ კვანძს აქვს ზუსტად ერთი მშობლიური კვანძი, გარდა ფესვისა (რომელსაც არ ჰყავს მშობლიური კვანძი);
- კვანძს შეიძლება ჰყავდეს რამდენიმე შვილობილი კვანძი;

მაგალითი:

```
<html>

<head>
  <title>DOM Tutorial</title>
</head>

<body>
  <h1>DOM Lesson one</h1>
  <p>Hello world!</p>
</body>

</html>
```




ზემოთ მოყვანილი HTML კოდიდან შეგიძლიათ წაიკითხოთ:

- <html> არის ძირეული კვანძი;
- <html>-ს არ აქვს მშობლიური კვანძი;
- <html> არის <head>-ისა და <body>-ის მშობლიური კვანძი;
- <head> არის პირველი შვილობილი კვანძი <html>;
- <body> ბოლო შვილობილი კვანძია <html>.

და:

- <head> ჰყავს ერთი შვილობილი კვანძი: <title>;
- <title> ჰყავს ერთი შვილობილი ელემენტი (ტექსტური კვანძი): „DOM Tutorial“;
- <body> ჰყავს ორი შვილობილი კვანძი: <h1> და <p>;
- <h1> ჰყავს ერთი შვილობილი კვანძი: „DOM გაკვეთილი პირველი“;
- <p> ჰყავს ერთი შვილობილი კვანძი: „გამარჯობა სამყარო!“;
- <h1> და <p> შვილობილი კვანძებია.

კვანძებს შორის ნავიგაცია

JavaScript-ის გამოყენებით კვანძებს შორის ნავიგაციისთვის კვანძის შემდეგი თვისებები შეგიძლიათ გამოიყენოთ:

- parentNode;
- childNodes[nodenumber];
- firstChild;
- lastChild;
- nextSibling;
- previousSibling.

## შვილობილი კვანძები და კვანძების მნიშვნელობა

გავრცელებული შეცდომა DOM-ის დამუშავებისას არის მოლოდინი, რომ ელემენტის კვანძი შეიცავს ტექსტს.

```
<title id="demo"> DOM Tutorial </title>
```

<title> ელემენტის კვანძი არ შეიცავს ტექსტს.

ის შეიცავს ტექსტურ კვანძს მნიშვნელობით „DOM Tutorial“.

ტექსტური კვანძის მნიშვნელობაზე წვდომა შესაძლებელია კვანძის innerHTML თვისების გამოყენებით:

```
myTitle = document.getElementById("demo").innerHTML;
```

innerHTML თვისებაზე წვდომა იგივეა, რაც nodeValue - პირველ შვილობილ ელემენტზე წვდომა:

```
myTitle = document.getElementById("demo").firstChild.nodeValue;
```

პირველ შვილობილზე წვდომა შეიძლება ასევე ასე განხორციელდეს:

```
myTitle =
    document.getElementById("demo").childNodes[0].nodeValue;
```

შემდეგი სამივე მაგალითი ამოიღებს <h1> ელემენტის ტექსტს და მის კოპირებას <p> ელემენტში მოახდენს:

```
      <html>
<body>

<h1 id="id01">My First Page</h1>
<p id="id02"></p>
```

```
<script>
document.getElementById("id02").innerHTML =
document.getElementById("id01").innerHTML;
</script>

</body>
</html>
```

```
        <html>
<body>

<h1 id="id01">My First Page</h1>
<p id="id02"></p>

<script>
document.getElementById("id02").innerHTML =
document.getElementById("id01").firstChild.nodeValue;
</script>

</body>
</html>
```

```
        <html>
<body>

<h1 id="id01">My First Page</h1>
<p id="id02">Hello!</p>
```

```
<script>
document.getElementById("id02").innerHTML =
document.getElementById("id01").childNodes[0].nodeValue;
</script>

</body>
</html>
```

## DOM მირეული კვანძები

არსებობს ორი სპეციალური თვისება, რომელიც საშუალებას გაძლევთ მიიღოთ სრული დოკუმენტი:

- document.body - დოკუმენტის ტანი;
- document.documentElement - სრული დოკუმენტი.

```
<html>
 <body>

 <h2>JavaScript HTML DOM</h2>
 <p>Displaying document.body</p>

 <p id="demo"></p>

 <script>
 document.getElementById("demo").innerHTML =
 document.body.innerHTML;
 </script>
```

```
</body>
</html>
```

```
        <html>
<body>

<h2>JavaScript HTMLDOM</h2>
<p>Displaying document.documentElement</p>

<p id="demo"></p>

<script>
document.getElementById("demo").innerHTML =
document.documentElement.innerHTML;
</script>

</body>
</html>
```

## nodeName თვისება

nodeName თვისება მიუთითებს კვანძის სახელს:

- nodeName არის მხოლოდ წაკითხვისათვის;
- nodeName ელემენტის კვანძი ემთხვევა ტეგის სახელს;
- nodeName ატრიბუტის კვანძი - ეს არის ატრიბუტის სახელი;
- nodeName ტექსტის კვანძი ყოველთვის არის #text;
- nodeName დოკუმენტის კვანძი ყოველთვის არის #document.

```
<!DOCTYPE html>
<html>
<body>

<h1 id="id01">My First Page</h1>
<p id="id02"></p>

<script>
document.getElementById("id02").innerHTML =
    document.getElementById("id01").nodeName;
</script>

</body>
</html>
```

შენიშვნა: nodeName ყოველთვის შეიცავს HTML-ელემენტის ტეგის სახელს ზედა რეგისტრში.

## nodeValue თვისება

nodeValue თვისება განსაზღვრავს კვანძის მნიშვნელობას:

- nodeValue ელემენტის კვანძებისთვის არის null;
- nodeValue ტექსტური კვანძებისთვის თვით ტექსტი არის;
- nodeValue ატრიბუტის კვანძებისთვის არის ატრიბუტის მნიშვნელობა.

## nodeType თვისება

nodeType თვისება არის მხოლოდ წაკითხვისათვის. იგი აბრუნებს კვანძის ტიპს.

```
<!DOCTYPE html>
<html>
<body>

<h1 id="id01">My First Page</h1>
<p id="id02"></p>

<script>
document.getElementById("id02").innerHTML =
    document.getElementById("id01").nodeType;
</script>

</body>
</html>
```

ყველაზე მნიშვნელოვანი nodeType-ის თვისებებია:

| კვანძი | ტიპი | მაგალითი |
|---|---|---|
| ELEMENT_NODE | 1 | <h1 class="heading">W3Schools</h1> |
| ATTRIBUTE_NODE | 2 | class = "heading" (deprecated) |
| TEXT_NODE | 3 | W3Schools |
| COMMENT_NODE | 8 | <!-- This is a comment --> |
| DOCUMENT_NODE | 9 | The HTML document itself (the parent of <html>) |
| DOCUMENT_TYPE_ NODE | 10 | <!Doctype html> |

HTML DOM-ში ტიპი 2 მოძველებულია (მაგრამ მუშაობს). ის არ არის მოძველებული XML DOM-ში.

# JavaScript HTML DOM ელემენტები (კვანძები)

## ახალი HTML ელემენტების (კვანძების) შექმნა

HTML DOM-ში ახალი ელემენტის დასამატებლად ჯერ უნდა შექმნათ ელემენტი (ელემენტის კვანძი) და შემდეგ დაუმატოთ ის არსებულ ელემენტში.

მაგალითი:

```
<!DOCTYPE html>
<html>
<body>

<h2>JavaScript HTML DOM</h2>
<p>დაამატეთ ახალი HTML ელემენტი.</p>

<div id="div1">
<p id="p1">ეს არის აბზაცი.</p>
<p id="p2">ეს არის კიდევ მეორე აბზაცი.</p>
</div>

<script>
const para = document.createElement("p");
const node = document.createTextNode("ეს არის ახალი აბზაცი.");
para.appendChild(node);
const element = document.getElementById("div1");
element.appendChild(para);
</script>
```

```
</body>
</html>
```

ეს კოდი ქმნის ახალ <p> ელემენტს:

```
const para = document.createElement("p");
```

იმისათვის, რომ <p> ელემენტს დაემატოს ტექსტი, ჯერ უნდა შექმნათ ტექსტური კვანძი. ეს კოდი ქმნის ტექსტურ კვანძს:

```
const node = document.createTextNode("ეს არის ახალი აბზაცი.");
```

შემდეგ თქვენ უნდა დაამატოთ ტექსტური კვანძი <p> ელემენტს:

```
para.appendChild(node);
```

და ბოლოს, თქვენ უნდა დაამატოთ ახალი ელემენტი არსებულ ელემენტს. ეს კოდი პოულობს არსებულ ელემენტს:

```
const element = document.getElementById("div1");
```

ეს კოდი არსებულ ელემენტს ამატებს ახალ ელემენტს:

```
element.appendChild(para);
```

## ახალი HTML-ელემენტების შექმნა - insertBefore()

წინა მაგალითში appendChild() მეთოდის საშუალებით დაემატა ახალი ელემენტი, როგორც მშობლიური ელემენტის ბოლო შვილობილი ელემენტი.

თუ ამ მეთოდის გამოყენება არ გსურთ, შეგიძლიათ გამოიყენოთ insertBefore() მეთოდი. მაგალითი:

```
<!DOCTYPE html>
<html>
<body>

<h2>JavaScript HTML DOM</h2>
<p>დაამატეთ ახალი HTML ელემენტი.</p>

<div id="div1">
<p id="p1">ეს არის აბზაცი.</p>
<p id="p2">ეს არის კიდევ მეორე აბზაცი.</p>
</div>

<script>
const para = document.createElement("p");
const node = document.createTextNode("ეს არის ახალი აბზაცი.");
para.appendChild(node);

const element = document.getElementById("div1");
const child = document.getElementById("p1");
element.insertBefore(para,child);
</script>

</body>
</html>
```

## არსებული HTML-ელემენტების წაშლა

იმისათვის, რომ HTML-ელემენტი წაშალოთ, გამოიყენეთ remove() მეთოდი.

მაგალითი:

```
<!DOCTYPE html>
<html>
<body>

<h2>JavaScript HTML DOM</h2>
<h3> HTML ელემენტის წაშლა.</h3>

<div>
<p id="p1">ეს არის აბზაცი.</p>
<p id="p2">ეს არის კიდევ მეორე აბზაცი.</p>
</div>

<button onclick="myFunction()">წაშალე ელემენტი</button>

<script>
function myFunction() {
document.getElementById("p1").remove();
}
</script>

</body>
</html>
```

HTML დოკუმენტი შეიცავს  ელემენტს ორი შვილობილი კვანძით (ორი <p> ელემენტი):

```
<div>
<p id="p1">ეს არის აბზაცი.</p>
```

```
<p id="p2">ეს არის კიდევ მეორე აბზაცი.</p>
</div>
```

იპოვეთ ელემენტი, რომლის წაშლა გსურთ:

```
const elmnt = document.getElementById("p1");
```

შემდეგ შეასრულეთ remove() მეთოდი ამ ელემენტზე:

```
elmnt.remove();
```

ეს remove() მეთოდი არ მუშაობს ძველ ბრაუზერებში, იხილეთ ქვემოთ მოცემული მაგალითი, თუ როგორ გამოიყენოთ removeChild().

## შვილობილი კვანძის წაშლა

ბრაუზერებისთვის, რომლებიც არ უჭერენ მხარს remove() მეთოდს, თქვენ უნდა იპოვოთ მშობლიური კვანძი, რათა წაშალოთ ელემენტი. მაგალითი:

```
<!DOCTYPE html>
<html>
<body>

<h2>JavaScript HTML DOM</h2>
<p>შვილობილი ელემენტის წაშლა</p>

<div>
<p id="p1">ეს არის აბზაცი.</p>
<p id="p2">ეს არის კიდევ მეორე აბზაცი.</p>
</div>
```

```
<script>
const parent = document.getElementById("div1");
const child = document.getElementById("p1");
parent.removeChild(child);
</script>

</body>
</html>
```

ეს HTML-დოკუმენტი შეიცავს  ელემენტს ორი შვილობილი კვანძით (ორი <p> ელემენტი):

```
<div id="div1">
  <p id="p1">ეს არის აბზაცი.</p>
  <p id="p2">ეს არის კიდევ ერთი აბზაცი.</p>
</div>
```

იპოვეთ ელემენტი id="div1"-ით:

```
const parent = document.getElementById("div1");
```

იპოვეთ <p> ელემენტი id="p1"-ით:

```
const child = document.getElementById("p1");
```

მშობლიურისგან შვილობილი ელემენტის წაშლა:

```
parent.removeChild(child);
```

აქ არის ჩვეულებრივი გამოსავალი: იპოვეთ შვილობილი, რომლის ამოღებაც გსურთ და გამოიყენეთ მისი parentNode თვისება მშობლიურის საპოვნელად:

```
const child = document.getElementById("p1");
```

```
child.parentNode.removeChild(child);
```

## HTML-ელემენტების შეცვლა

HTML DOM-ში ელემენტის შესაცვლელად გამოიყენეთ replaceChild() მეთოდი.

მაგალითი:

```
<!DOCTYPE html>
<html>
<body>

<h2>JavaScript HTML DOM</h2>
<h3>შეცვალეთ HTML-ელემენტი.</h3>

<div>
<p id="p1">ეს არის აბზაცი.</p>
<p id="p2">ეს არის კიდევ მეორე აბზაცი.</p>
</div>

<script>
const parent = document.getElementById("div1");
const child = document.getElementById("p1");
const para = document.createElement("p");
const node = document.createTextNode("ეს არის ახალი აბზაცი.");
para.appendChild(node);
parent.replaceChild(para,child);
</script>

</body>
```

```
</html>
```

# JavaScript HTML DOM კოლექცია

## HTMLCollection ობიექტი

getElementsByTagName() მეთოდი HTMLCollection ობიექტს აბრუნებს.

HTMLCollection ობიექტი HTML-ელემენტების მასივის სიის (კოლექცია) მსგავსია.

შემდეგი კოდი დოკუმენტში ირჩევს ყველა <p> ელემენტს.

მაგალითი:

```
const myCollection = document.getElementsByTagName("p");
```

კოლექციის ელემენტებზე წვდომა ინდექსის ნომრითაა შესაძლებელი.

მეორე <p> ელემენტზე წვდომისთვის შეგიძლიათ დაწეროთ:

myCollection[1]

```
<!DOCTYPE html>
<html>
<body>

<h2>JavaScript HTML DOM</h2>

<p>Hello World!</p>

<p>Hello Georgia!</p>
```

```
<p id="demo"></p>

<script>
const myCollection = document.getElementsByTagName("p");

document.getElementById("demo").innerHTML = "The innerHTML
    of the second paragraph is: " + myCollection[1].innerHTML;

</script>

</body>
</html>
```

ინდექსის ნუმერაცია ნულიდან იწყება.

## HTML-კოლექციის სიგრძე

length თვისება HTMLCollection-ში განსაზღვრავს ელემენტების რაოდენობას. მაგალითი:

```
<!DOCTYPE html>
<html>
<body>

<h2>JavaScript HTML DOM</h2>

<p>Hello World!</p>

<p>Hello Georgia!</p>
```

```
<p id="demo"></p>

<script>
const myCollection = document.getElementsByTagName("p");

document.getElementById("demo").innerHTML = "This document
    contains " + myCollection.length + " paragraphs.";

</script>

</body>
</html>
```

length თვისება სასარგებლოა, როდესაც კოლექციის ელემენტების მოწესრიგება გსურთ.

მაგალითი: შეცვალეთ ყველა <p> ელემენტის ტექსტის ფერი:

```
<!DOCTYPE html>
<html>
<body>

<h2>JavaScript HTML DOM</h2>

<p>Hello World!</p>

<p>Hello Georgia!</p>
```

```
<p>მაუსით დააწკაპუნეთ ღილაკზე ყველა p ელემენტის ფერის
    შესაცვლელად.</p>

<button onclick="myFunction()">სცადე</button>

<script>
function myFunction() {
  const myCollection = document.getElementsByTagName("p");
  for (let i = 0; i < myCollection.length; i++) {
    myCollection[i].style.color = "red";
  }
}
</script>

</body>
</html>
```

HTMLCollection არ არის მასივი!

HTMLCollection შეიძლება მასივის მსგავსად გამოიყურებოდეს, მაგრამ ეს ასე არ არის.

შეგიძლიათ სიას გაჰყვეთ და მიმართოთ ელემენტს ნომრით (ისევე როგორც მასივს).

თუმცა, თქვენ არ შეგიძლიათ HTMLCollection-ის დროს მასივის მეთოდების გამოყენება, როგორიცაა valueOf(), pop(), push(), ან join().

# JavaScript HTML DOM კვანძების სია

## HTML DOM NodeList ობიექტი

NodeList ობიექტი არის დოკუმენტიდან ამოღებული კვანძების სია (ნაკრები).

NodeList ობიექტი თითქმის იგივეა, რაც HTMLCollection ობიექტი.

ზოგიერთი (ძველი) ბრაუზერი ისეთი მეთოდებისთვის, როგორიცაა getElementsByClassName() HTMLCollection-ის ნაცვლად NodeList ობიექტს აბრუნებს.

ყველა ბრაუზერი childNodes თვისებისთვის NodeList ობიექტს აბრუნებს.

ბრაუზერების უმეტესობა querySelectorAll() მეთოდისთვის NodeList ობიექტს აბრუნებს.

მაგალითი. შემდეგი კოდი დოკუმენტში არჩევს ყველა <p> კვანძს:

```
const myNodeList = document.querySelectorAll("p");
```

NodeList-ში ელემენტებზე წვდომა ინდექსის ნომრითაა შესაძლებელი.

მეორე <p> კვანძზე წვდომისთვის შეგიძლიათ დაწეროთ:

```
<!DOCTYPE html>
<html>
<body>

<h2>JavaScript HTML DOM</h2>
```

```
<p>Hello World!</p>

<p>Hello Georgia!</p>

<p id="demo"></p>

<script>
const myNodelist = document.querySelectorAll("p");

document.getElementById("demo").innerHTML = "The innerHTML
    of the second paragraph is: " + myNodelist[1].innerHTML;

</script>

</body>
</html>
```

ინდექსი იწყება ნულით.

## HTML DOM კვანძების სიის სიგრძე

length თვისება კვანძების სიაში კვანძების რაოდენობას განსაზღვრავს. მაგალითი:

```
<!DOCTYPE html>
<html>
<body>

<h2>JavaScript HTML DOM</h2>
```

```
<p>Hellow World!</p>

<p>Hellow Georgia!</p>

<p id="demo"></p>

<script>
const myNodelist = document.querySelectorAll("p");

document.getElementById("demo").innerHTML = "This document
    contains " + myNodelist.length + " paragraphs.";

</script>

</body>
</html>
```

length თვისება სასარგებლოა, როდესაც კვანძების სიაში გსურთ კვანძების მოწესრიგება.

მაგალითი. კვანძების სიაში შეცვალეთ ყველა <p> ელემენტის ფერი:

```
<!DOCTYPE html>
<html>
<body>

<h2>JavaScript HTML DOM</h2>

<p>Hello World!</p>
```

```
<p>Hello Georgia!</p>

<p>მაუსით დააწკაპუნეთ ღილაკზე ყველა p ელემენტის ფერის
    შესაცვლელად.</p>

<button onclick="myFunction()">Try it</button>

<script>
function myFunction() {
  const myNodelist = document.querySelectorAll("p");
  for (let i = 0; i < myNodelist.length; i++) {
    myNodelist[i].style.color = "red";
  }
}
</script>

</body>
</html>
```

## HTMLCollection და NodeList შორის სხვაობა

NodeList და HTMLCollection ერთი და იგივეა.

ორივე არის დოკუმენტიდან ამორჩეული კვანძების (ელემენტების) მასივის მსგავსი კოლექციები (სიები). კვანძებზე წვდომა შესაძლებელია რიგითი ნომრით. ინდექსი იწყება 0-დან.

ორივეს აქვს length თვისება, რომელიც აბრუნებს ელემენტების რაოდენობას სიაში (ნაკრებში).

HTMLCollection არის დოკუმენტის ელემენტების კოლექცია.

NodeList არის დოკუმენტის კვანძების კრებული (ელემენტების კვანძები, ატრიბუტების კვანძები და ტექსტური კვანძები).

HTMLCollection-ის ელემენტებზე წვდომა შესაძლებელია მათი სახელით, იდენტიფიკატორებით ან ინდექსის ნომრით.

NodeList ელემენტების წვდომა შესაძლებელია მხოლოდ მათი ინდექსის ნომრით.

HTMLCollection ყოველთვის „ცოცხალი“ კოლექციაა. მაგალითად, თუ DOM-ში სიაში დაამატებთ <li> ელემენტს, HTMLCollection-ის სიაც ასევე შეიცვლება.

NodeList ყველაზე ხშირად სტატიკური კოლექციაა. მაგალითად, თუ DOM-ში სიაში დაამატებთ <li> ელემენტს, NodeList-ში სია არ შეიცვლება.

getElementsByClassName() და getElementsByTagName() მეთოდები აბრუნებს ცოცხალ კოლექციას - HTMLCollection-ს.

querySelectorAll() მეთოდი აბრუნებს სტატიკურ NodeList-ს.

childNodes თვისება აბრუნებს აქტიურ NodeList-ს.

# Call Back

JavaScript-ში ნაგულისხმევი წესის თანახმად მოქმედებებები ასინქრონულია.

მაგალითად, განიხილეთ loadScript(src) ფუნქცია:

```
function loadScript(src) {
  let script = document.createElement('script');
  script.src = src;
  document.head.append(script);
}
```

ეს ფუნქცია გვერდზე ახალ სკრიპტს ატვირთავს. როდესაც <script src="…"> კონსტრუქცია დოკუმენტის ტანში დაემატება, ბრაუზერი ჩამოტვირთავს სკრიპტს და შეასრულებს მას.

აქ მოცემულია ამ ფუნქციის გამოყენების მაგალითი:

```
// ჩამოტვირთე და შეასრულე სკრიპტი
loadScript('/my/script.js');
```

ასეთ ფუნქციებს „ასინქრონულს“ უწოდებენ, რადგან მოქმედება (სკრიპტის ჩატვირთვა) არ დასრულდება ახლა, არამედ მოგვიანებით.

თუ loadScript(…) გამოძახების შემდეგ რაიმე კოდი არის, მაშინ ის სკრიპტის ჩატვირთვას არ დაელოდება.

```
loadScript('/my/script.js');
// კოდი, რომელიც ჩაწერილია loadScript ფუნქციის
    გამოძახების შემდეგ,
// არ დაელოდება სკრიპტის სრულად ჩატვირთვას
// ...
```

ჩვენ გვსურს გამოვიყენოთ ახალი სკრიპტი, როგორც კი ის ჩაიტვირთება. ვთქვათ, ის აცხადებს ახალ ფუნქციას, რომლის შესრულებაც გვინდა.

მაგრამ თუ ჩვენ უბრალოდ გამოვიძახებთ ამ ფუნქციას loadScript(...)-ის შემდეგ, არაფერი გამოვა:

```
loadScript('/my/script.js'); // სკრიპტში არის "function newFunction() {…}"

newFunction(); // ასეთი ფუნქცია არ არსებობს!
```

მართლაც, ბრაუზერს არ ჰქონდა დრო სკრიპტის ჩასატვირთად. ახლა loadScript ფუნქცია არ გვაძლევს საშუალებას ვაკონტროლოთ ჩატვირთვის მომენტი. სკრიპტი იტვირთება და შემდეგ სრულდება. მაგრამ ჩვენ ზუსტად გვინდა ვიცოდეთ, როდის მოხდება ეს, რათა გამოვიყენოთ ამ სკრიპტის ფუნქციები და ცვლადები.

მოდით loadScript ფუნქციაში მეორე არგუმენტად გადავცეთ callback ფუნქცია, იმისათვის, რომ სკრიპტის ჩატვირთვის დროს გამოვიძახოთ იგი:

```
function loadScript(src, callback) {
  let script = document.createElement('script');
  script.src = src;
  script.onload = () => callback(script);
  document.head.append(script);
}
```

onload ხდომილება ძირითადად ასრულებს ფუნქციას სკრიპტის ჩატვირთვისა და შესრულების შემდეგ.

ახლა, თუ გვინდა გამოვიძახოთ ფუნქცია სკრიპტიდან, ეს callback-ში უნდა გავაკეთოთ:

```
loadScript('/my/script.js', function() {
  // ამ ფუნქციის გამოძახება მოხდება სკრიპტის ჩატვირთვის
     შემდეგ
  newFunction(); // ახლა ყველაფერი მუშაობს
  ...
});
```

მისი არსი შემდეგში მდგომარეობს: მეორე არგუმენტად გადაეცემა ფუნქცია (ჩვეულებრივ ანონიმური), რომელიც სრულდება მოქმედების დასრულების შემდეგ.

მაგალითისთვის ავიღოთ რეალური სკრიპტი ფუნქციების ბიბლიოთეკით:

```
function loadScript(src, callback) {
  let script = document.createElement('script');
  script.src = src;
  script.onload = () => callback(script);
  document.head.append(script);
}

loadScript('https://cdnjs.cloudflare.com/ajax/libs/lodash.js/3.2.0/loda
     sh.js', script => {
  alert(`გამარჯობა, სკირპი  ${script.src} იტვირთება`);
  alert( _ ); // ფუნქცია ჩატვირთულ სკრიპტშია გამოცხადებული
});
```

ასეთ ჩაწერას ეწოდება ასინქრონული პროგრამირება callback-ის გამოყენებით. ფუნქციებში, რომლებიც ასრულებენ ნებისმიერ ასინქრონულ ოპერაციებს, გადაეცემა არგუმენტი callback-ი - ფუნქცია, რომლის გამოიძახებაც მოხდება ასინქრონული მოქმედების დასრულების შემდეგ.

ჩვენ იგივე გავაკეთეთ loadScript-ში, მაგრამ ეს, რა თქმა უნდა, გავრცელებული მიდგომაა.

## Call Back-ი Call Back-ში

როგორ ჩავტვირთოთ ორი სკრიპტი ერთმანეთის მიყოლებით: ჯერ პირველი და მერე მეორე?

პირველი, რაც უნდა გავაკეთოთ ეს არის, კიდევ ერთხელ loadScript-ის გამოძახება ისევ callback-ის შიგნით:

```
loadScript('/my/script.js', function(script) {

  alert(`Great, the ${script.src} script has loaded, let's load another
     one `);

  loadScript('/my/script2.js', function(script) {
    alert(`Great, the second script has loaded `);
  });

});
```

როდესაც გარე loadScript ფუნქცია შესრულდება, გამოძახება მოხდება callback-ის შიგნით.

რა მოხდება, თუ კიდევ ერთი სკრიპტის ჩატვირთვა დაგვჭირდება?..

```
loadScript('/my/script.js', function(script) {

  loadScript('/my/script2.js', function(script) {

    loadScript('/my/script3.js', function(script) {
      // და ა. შ., ვიდრე ყველა სკრიპტი არ ჩაიტვირთება
    });

  })

});
```

ყოველი ახალი მოქმედება ჩვენ უნდა გამოვიძახოთ callback-ის შიგნით. ეს ვარიანტი გამოდგება, როდესაც გვაქვს ერთი ან ორი მოქმედება, მაგრამ მეტისთვის ის უკვე აღარ არის მოსახერხებელი.

## შეცდომების აღმოჩენა

ზემოთ მოყვანილ მაგალითებში ჩვენ არ გვიფიქრია შეცდომებზე. რა მოხდება, თუ სკრიპტი ვერ ჩაიტვირთა? callback-ს უნდა შეეძლოს უპასუხოს შესაძლო პრობლემებს.

ქვემოთ მოცემულია loadScript-ის გაუმჯობესებული ვერსია, რომელსაც შეუძლია აკონტროლოს ჩატვირთვის შეცდომები:

```
function loadScript(src, callback) {
  let script = document.createElement('script');
```

```
  script.src = src;

  script.onload = () => callback(null, script);
  script.onerror = () => callback(new Error(`ვერ მოხერხდა ${src}
     სკრიპტის ჩატვირთვა`));

  document.head.append(script);
}
```

ჩვენ ვიძახებთ callback(null, script) წარმატებული ჩატვირთვის შემთხვევაში და callback(error)-ს სკრიპტის წარუმატებლად ჩატვირთვის შემთხვევაში.

მაგალითი:

```
loadScript('/my/script.js', function(error, script) {
  if (error) {
    // შეცდომის დამუშავება
  } else {
    // სკრიპტი წარმატებით ჩაიტვირთა
  }
});
```

ისევ და ისევ, მიდგომა, რომელსაც ჩვენ ვიყენებდით loadScript-ში, ასევე გავრცელებულია და ეწოდება „error-first callback“ (შეცდომის პირველი გამოძახება).

წესები ასეთია:

1. callback ფუნქციის პირველი არგუმენტი დარეზერვირებულია შეცდომისთვის. ამ შემთხვევაში გამოძახება ასე გამოიყურება: callback(err).

2. მეორე და შემდგომი არგუმენტები შესრულების შედეგებისათვის არის. ამ შემთხვევაში, გამოძახება ასე გამოიყურება: callback(null, result1, result2…).

ერთი და იგივე callback ფუნქცია გამოიყენება როგორც შეცდომის შესახებ ინფორმირებისთვის, ასევე შედეგების გადასაცემად.

## Promise

დავუშვათ, მომხმარებლებს სურთ მუდმივად მიიღონ გარკვეული ინფორმაცია, რისთვისაც საჭიროა დარეგისტრირდნენ. მათ შეუძლიათ დატოვონ თავიანთი ელექტრონული ფოსტის მისამართი, რომელზედაც გაიგზავნება სასურველი ინფორმაცია, ამ ინფორმაციის შექმნისთანავე. კიდევ უფრო მეტი: თუ რამე არასწორედ მოხდება, ისინიც მიიღებენ შეტყობინებას ამის შესახებ.

ეს არის რეალური ცხოვრების ანალოგია სიტუაციებისთვის, რომლებსაც ხშირად ვხვდებით დაპროგრამების პროცესში:

1. არის კოდის „შემქმნელი“, რომელიც რაღაცას აკეთებს, რასაც გარკვეული დრო სჭირდება. მაგალითად, მონაცემთა ჩამოტვირთვა ქსელში;

2. არის „მომხმარებელი“ კოდი, რომელსაც სურს მიიღოს „შემქმნელი“ კოდის შედეგი, როცა ის მზად იქნება. ის შეიძლება დასჭირდეს ერთზე მეტ ფუნქციას;

3. Promise (ინგლისურად დაპირება, პირობა, ასეთ ობიექტს დავარქმევთ „promise“) არის სპეციალური ობიექტი JavaScript-ში, რომელიც ერთმანეთთან აკავშირებს „შემქმნელის“

და „მომხმარებლის“ კოდებს. ჩვენი ანალოგიის ტერმინების მიხედვით, ეს არის „ხელმომწერთა სია“. „შემქმნელი“ კოდი შეიძლება იმდენჯერ შესრულდეს, რამდენიც შედეგის მისაღებადაა საჭირო, ხოლო როცა შედეგი მზად იქნება promise ობიექტი შედეგს ხელმისაწვდომს ხდის კოდისთვის, რომლსაც ის გამოიწერს.

ანალოგია არ არის მთლად ზუსტი, რადგან JavaScript-ში Promise ობიექტი ბევრად უფრო რთულია, ვიდრე ხელმოწერთა მარტივი სია: მას აქვს დამატებითი ფუნქციები და შეზღუდვები.

Promise-ის შექმნის სინტაქსია:

```
let promise = new Promise(function(resolve, reject) {
  // ფუნქცია-შემსრულებელი (executor)
});
```

new Promise კონსტრუქციაზე გადაცემულ ფუნქციას შემსრულებელი (executor) ეწოდება. როდესაც Promise შეიქმნება, ის ავტომატურად გაიშვება. ის უნდა შეიცავდეს „შემქმნელ“ კოდს, რომელიც საბოლოოდ გამოიღებს შედეგს.

მისი resolve და reject არგუმენტები არის callback, რომელიც თავად JavaScript-ის მიერარის მოწოდებული. ჩვენი კოდი მხოლოდ შემსრულებლის შიგნით არის.

როდესაც ის მიიღებს შედეგს, ადრე თუ გვიან, არ აქვს მნიშვნელობა, მან უნდა გამოიძახოს ერთ-ერთი ასეთი callback:

- resolve(value) - თუ ოპერაცია წარმატებით დამთავრდა, შედეგით value;
- reject(error) - თუ მოხდა შეცდომა, error - შეცდომის ობიექტი.

ასე რომ, შემსრულებელი ავტომატურად იწყებს მუშაობას, მან უნდა შეასრულოს სამუშაო, ხოლო შემდეგ გამოიძახოს resolve ან reject.

new Promise კონსტრუქტორის მიერ დაბრუნებულ Promise ობიექტს აქვს შიდა თვისებები:

- state („მდგომარეობა“) - დასაწყისში „pending“ („მოლოდინი“), შემდეგ იცვლება „fulfilled“ („წარმატებით დასრულდა“) resolve-ს გამოძახების დროს ან „rejected“-ზე („შეცდომით დასრულდა“) reject-ის გამოძახების დროს;
- result („შედეგი“) - თავდაპირველად undefined-ი იყო, შემდეგ შეიცვალა value-ით, resolve(value)-ს გამოძახების დროს ან error-ით, reject(error)-ის გამოძახების დროს.

ასე რომ, შემსრულებელს საბოლოოდ promise-ს ორიდან ერთ-ერთ მდგომარეობაში გადაჰყავს:

ამ ცვლილებების შესახებ როგორ გაიგებენ მომხმარებლები მოგვიანებით ვნახავთ.

ქვემოთ მოცემულია Promise კონსტრუქტორის მაგალითი და მარტივი შემსრულებელი კოდით, რომელიც შედეგს იძლევა დაყოვნებით (setTimeout-ის მეშვეობით):

```
let promise = new Promise(function(resolve, reject) {
  // ეს ფუნქცია new Promise-ის გამოძახებით ავტომატურად შესრულდება

  // 1 წამის შემდეგ მიიღება შეტყობინება, რომ დავალება დასრულებულია შედეგით "done"
  setTimeout(() => resolve("done"), 1000);
});
```

ჩვენ შეგვიძლია დავაკვირდეთ ორ რამეს ზემოთ მოცემული კოდის გაშვებით:

1. შემსრულებელი ფუნქცია ამოქმედდება მაშინვე, როდესაც new Promise-ის გამოძახება მოხდება;

2. შემსრულებელი იღებს ორ არგუმენტს: resolve და reject - JavaScript-ში ჩაშენებული ფუნქციებია, ამიტომ მათი ჩაწერა არ გვჭირდება. ჩვენ უბრალოდ უნდა დავრწმუნდეთ, რომ შემსრულებელი მზად ყოფნის შემთხვევაში გამოიძახებს ერთ-ერთ მათგანს.

პროგრამის გაშვების ერთი წამის შემდეგ, შემსრულებელი გამოიძახებს resolve("done") ფუნქციას შედეგის გადასაცემად:

ეს იყო წარმატებით შესრულებული დავალების მაგალითი, საბოლოოდ მივიღეთ, რომ დავალება დასრულებულია შედეგით „done“.

ახლა კი მაგალითი, რომელშიც შემსრულებელი შეატყობინებს, რომ დავალება შეცდომით დასრულდა:

```
let promise = new Promise(function(resolve, reject) {
  // 1 წამის შემდეგ მიიღება შეტყობინება, რომ დავალება
     დასრულებულია შეცდომით
  setTimeout(() => reject(new Error("Whoops!")), 1000);
});
```

შემსრულებელი ასრულებს დავალებას (რასაც ჩვეულებრივ დრო სჭირდება), შემდეგ იძახებს resolve-ს ან reject-ს, რათა შეცვალოს შესაბამისი Promise-ის მდგომარეობა.

თავდაპირველი „მოლოდინში“ მყოფი Promise-ისგან განსხვავებით, წარმატებითაც და უარყოფითად დასრულებულ Promise-ს ეძახიან „შესრულებულს“.

შეიძლება იყოს ერთი რამ: ან შედეგი ან შეცდომა.

შემსრულებელმა უნდა გამოიძახოს ერთ-ერთი: resolve ან reject. Promise-ის მდგომარეობა შეიძლება შეიცვალოს მხოლოდ ერთხელ.

resolve-ის ან reject-ის ყველა შემდგომი გამოძახება იგნორირებული იქნება:

```
let promise = new Promise(function(resolve, reject) {
  resolve("done");

  reject(new Error("…")); // იგნორირებულია
  setTimeout(() => resolve("…")); // იგნორირებულია
});
```

## მომხმარებლები then, catch

ობიექტი Promise არის როგორც დამაკავშირებელი რგოლი შემსრულებელს („შემქმნელი“ კოდი) და ფუნქცია-მომხმარებელს შორის, რომელიც მიიღებს ან შედეგს ან შეცდომას. ფუნქცია-მომხმარებელი შეიძლება იყოს დარეგისტრირებული (ხელმოწერილი) .then და .catch მეთოდების გამოყენებით.

.then ყველაზე მნიშვნელოვანი და ფუნდამენტური მეთოდია.

მისი სინტაქსია:

```
promise.then(
  function(result) { /* დაამუშავებს წარმატებულ შედეგს */ },
  function(error) { /* დაამუშავებს შეცდომას */ }
);
```

.then მეთოდის პირველი არგუმენტი არის ფუნქცია, რომელიც სრულდება, როდესაც Promise-ი გადადის „შესრულებულია წარმატებით“ მდგომარეობაში და მიიღებს შედეგს.

.then მეთოდის მეორე არგუმენტი არის ფუნქცია, რომელიც სრულდება, როდესაც Promise-ი გადადის „შესრულებულია შეცდომით“ მდგომარეობაში და მიიღებს შეცდომას.

მაგალითად, აქ არის რეაქცია წარმატებით შესრულებულ Promise-ზე:

```
let promise = new Promise(function(resolve, reject) {
  setTimeout(() => resolve("done!"), 1000);
});

// resolve გაუშვებს პირველ ფუნქციას, რომელიც გადაეცემა
    .then-ს
promise.then(
  result => alert(result), // 1 წამში შედეგად გამოიტანს "done!"
  error => alert(error) // არ იქნება გაშვებული
);
```

შესრულდა პირველი ფუნქცია.

შეცდომის შემთხვევაში Promise-ში - შესრულდება მეორე:

```
let promise = new Promise(function(resolve, reject) {
  setTimeout(() => reject(new Error("Whoops!")), 1000);
});

// reject გაუშვებს მეორე ფუნქციას, რომელიც გადაეცემა .then-
    ს
```

```
promise.then(
  result => alert(result), // არ იქნება გაშვებული
  error => alert(error) // 1 წამში შედეგად გამოიტანს "Error: Whoops!"
);
```

თუ ჩვენ გვაინტერესებს მხოლოდ დავალების წარმატებით შესრულება, მაშინ then-ს მხოლოდ ერთი ფუნქცია შეიძლება გადაეცეს:

```
let promise = new Promise(resolve => {
  setTimeout(() => resolve("done!"), 1000);
});

promise.then(alert); // 1 წამში შედეგად გამოიტანს "done!"
```

თუ ჩვენ გვსურს მხოლოდ შეცდომის დამუშავება, მაშინ შეგვიძლია პირველ არგუმენტად გამოვიყენოთ null: .then(null, errorHandlingFunction). ან შეგვიძლია გამოვიყენოთ .catch(errorHandlingFunction) მეთოდი, რომელიც იგივეს გააკეთებს:

```
let promise = new Promise((resolve, reject) => {
  setTimeout(() => reject(new Error("შეცდომაა!")), 1000);
});

// .catch(f) ეს იგივეა რაც promise.then(null, f)
promise.catch(alert); // 1 წამში შედეგად გამოიტანს "Error: Whoops!"
```

.catch(f)-ის გამოძახება .then(null, f) ფუნქციის შემოკლებული ვარიანტია.

## finally

finally ბლოკის ანალოგიურად ჩვეულებრივ try {...} catch {...}-ს, Promise-ებს ასევე აქვს finally მეთოდი.

.finally(f)-ის გამოძახება .then(f, f)-ის მსგავსია, იმ აზრით, რომ f-ი ნებისმიერ შემთხვევაში შესრულდება, როდესაც Promise-ი წარმატებით ან შეცდომით დამთავრდება.

finally-ის იდეა იმაში მდგომარეობს, რომ დამმუშავებელი შესრულებისათვის მოიმართოს გასუფთავება/დასრულები-სათვის წინა ოპერაციების დასრულების შემდეგ.

მაგალითად, ჩატვირთვის ინდიკატორების შეჩერება, ისეთი დაკავშირების დახურვა, რომლებიც აღარ არის საჭირო და ა.შ. კოდი შეიძლება ასე გამოიყურებოდეს:

```
new Promise((resolve, reject) => {
  /* გააკეთე ის, რასაც დრო სჭირდება და შემდეგ გამოიძახოთ
     resolve-ი ან შესაძლოა reject-ი */
})
  // შესრულდება, როდესაც Promise-ი დასრულდება,
     წარმატებული იქნება თუ არა
  .finally(() => ჩატვირთვის ინდიკატორის გაჩერება)
  // ასე რომ, ჩატვირთვის ინდიკატორი ყოველთვის ჩერდება,
     ვიდრე გავაგრძელებთ
  .then (result => შედეგის ჩვენება, err => შეცდომის ჩვენება)
```

გაითვალისწინეთ, რომ finally(f)-ი არ არის ზუსტად then(f,f)-ის ფსევდონიმი, როგორც შეიძლება იფიქროთ.

მნიშვნელოვანი განსხვავებებია:

1. finally-იდან გამოძახებულ დამმუშავებელს არგუმენტი არ აქვს. finally-ში ჩვენ არ ვიცით, როგორ შესრულდა Promise-ი. და ეს ნორმალურია, რადგან, როგორც წესი, ჩვენი ამოცანაა „ზოგადი“ საბოლოო პროცედურების შესრულება;

2. დამმუშავებელი finally-ი „გამოტოვებს“ შედეგს ან შეცდომას შემდეგ, შემდგომი დამმუშავებლებისკენ.

მაგალითად, აქ შედეგი finally-ის მეშვეობით გადადის then-ზე:

```
new Promise((resolve, reject) => {
  setTimeout(() => resolve("value"), 2000);
})
  .finally(() => alert("Promise-ი დამთავრებულია")) // პირველად
     შესრულდება
  .then(result => alert(result)); // <-- .then აჩვენებს "value"-ს
```

როგორც ხედავთ, პირველი Promise-ით დაბრუნებული მნიშვნელობა finally-ის საშუალებით შემდეგ then-ს გადაეცემა.

ეს ძალიან მოსახერხებელია, რადგან finally-ი განკუთვნილი არ არის Promise-ის შედეგის დასამუშავებლად. როგორც უკვე ითქვა, ეს არის ზოგადი გასუფთავებისათვის, საბოლოო შედეგისაგან დამოუკიდებლად.

აქ კი Promise-ის შეცდომა finally-ის მეშვეობით catch-ზე გადის:

```
new Promise((resolve, reject) => {
  throw new Error("error");
})
```

```
  .finally(() => alert("Promise-ი დამთავრებულია")) // პირველად
     შესრულდება
  .catch(err => alert(err));  // <-- .catch აჩვენებს შეცდომას
```

3. finally დამმუშავებელმა ასევე არაფერი არ უნდა დააბრუნოს. თუ ასეა, მაშინ დაბრუნებული მნიშვნელობა იგნორირებული იქნება.

ამ წესის ერთადერთი გამონაკლისი არის ის, როდესაც finally-ის დამმუშავებელი შეცდომას გამოიტანს. ეს შეცდომა ნებისმიერი წინა შედეგის ნაცვლად შემდეგ დამმუშავებელს გადაეცემა.

თუ Promise-ი მოლოდინის მდგომარეობაშია, .then/catch/finally დამმუშავებლები მას დაელოდებიან.

ზოგჯერ შეიძლება მოხდეს, რომ Promise-ი უკვე შესრულებულია, როცა მას დამმუშავებელს დავამატებთ. ასეთ შემთხვევაში, ეს დამმუშავებლები უბრალოდ დაუყოვნებლივ გაიშვება:

```
// Promise-ი შექმნისას დაუყოვნებლივ გადადის
    „წარმატებულად დასრულებულ“ მდგომარეობაში
let promise = new Promise(resolve => resolve("მზადაა!"));

promise.then(alert); // მზადაა! (მაშინვე გამოვა)
```

## Promise-ების ჯაჭვი

დავუბრუნდეთ სიტუაციას callback-იდან: ჩვენ გვაქვს ასინქრონული ამოცანების თანმიმდევრობა, რომლებიც უნდა

შესრულდეს ერთმანეთის მიყოლებით. მაგალითად, შეგვიძლია ვისაუბროთ სკრიპტების ჩატვირთვაზე.

Promise-ები ამ ამოცანის შესრულების რამდენიმე გზას გვთავაზობს.

აქ ჩვენ განვიხილავთ Promise-ები ჯაჭვს.

ის ასე გამოიყურება:

```
new Promise(function(resolve, reject) {

  setTimeout(() => resolve(1), 1000); // (*)

}).then(function(result) { // (**)

  alert(result); // 1
  return result * 2;

}).then(function(result) { // (***)

  alert(result); // 2
  return result * 2;

}).then(function(result) {

  alert(result); // 4
  return result * 2;

});
```

იდეა ისაა, რომ პირველი Promise-ის შედეგი ჯაჭვურად გადაეცემა .then დამმუშავებელს. შესრულების ნაკადი ასეთია:

1. საწყისი Promise-ი წარმატებით შესრულდება 1 წამში (*);
2. შემდეგ მოხდება .then დამმუშავებლის გამოძახება (**);
3. მის მიერ დაბრუნებული მნიშვნელობა შემდეგ .then დამმუშავებელს გადაეცემა (***);
4. ... და ასე შემდეგ.

შედეგად, შედეგი გადაეცემა დამმუშავებლებს ჯაჭვის გასწვრივ და ჩვენ ვხედავთ მიმდევრობით რამდენიმე alert-ს, რომლებსაც გამოაქვს: 1 → 2 → 4.

ეს ყველაფერი ასე მუშაობს, რადგან promise.then გამოძახება ასევე Promise-ს აბრუნებს, ასე რომ, ჩვენ შემდეგი .then-ის გამოძახება შეგვიძლია.

როდესაც დამმუშავებელი რაიმე მნიშვნელობას დააბრუნებს, ის შესაბამისი Promise-ის შესრულების შედეგი ხდება და შემდეგ .then-ს გადაეცემა.

## ვაბრუნებთ Promise-ს

.then( handler)-ში გადაცემულ handler დამმუშავებელს Promise-ის დაბრუნება შეუძლია.

ამ შემთხვევაში, შემდგომი დამმუშავებლები მის შესრულებას ელიან და შემდეგ მიიღებენ მის შედეგს.

მაგალითად:

```
new Promise(function(resolve, reject) {

  setTimeout(() => resolve(1), 1000);
```

```
}).then(function(result) {

  alert(result); // 1

  return new Promise((resolve, reject) => { // (*)
    setTimeout(() => resolve(result * 2), 1000);
  });

}).then(function(result) { // (**)

  alert(result); // 2

  return new Promise((resolve, reject) => {
    setTimeout(() => resolve(result * 2), 1000);
  });

}).then(function(result) {

  alert(result); // 4

});
```

აქ პირველი .then აჩვენებს 1-ს და ახალ Promise-ს აბრუნებს new Promise(...) (*) სტრიქონში. ერთი წამის შემდეგ, ეს Promise-ი წარმატებით შესრულდება და მისი შედეგი (არგუმენტი resolve-ში, ე.ი. result * 2) შემდეგ .then დამმუშავებელს გადაეცემა. ის (**) სტრიქონში მდებარეობს, აჩვენებს 2-ს და იგივეს აკეთებს.

ამრიგად, როგორც წინა მაგალითში, გამოიტანს 1 → 2 → 4, მაგრამ ახლა გამოძახებებს შორის არის 1 წამის პაუზა.

Promise-ის დაბრუნებით, ჩვენ ასინქრონული მოქმედებების ჯაჭვის აგება შეგვიძლია.

## Promise API

Promise კლასს აქვს 6 სტატიკური მეთოდი. გავეცნოთ თითოეულ მათგანს.

### Promise.all

ვთქვათ, რამდენიმე Promise-ი პარალელურად უნდა შევასრულოთ და დაველოდოთ, სანამ ყველა შესრულდება.

მაგალითად, პარალელურად ჩაიტვირთა რამდენიმე ფაილი და დაამუშავდეს შედეგები, როცა ის მზად იქნება.

ამისათვის, Promise.all მეთოდი გამოდგება.

მისი სინტაქსია:

let soz = Promise.all(iterable);

Promise.all მეთოდი იღებს Promise-ების მასივს (მას შეუძლია მიიღოს ნებისმიერი გამეორებადი ობიექტი, მაგრამ მასივი ჩვეულებრივ გამოიყენება) და ახალ Promise-ს აბრუნებს.

ახალი Promise-ი შესრულდება, როდესაც Promise-ების მთელი გადაცემული სია დასრულდება და შედეგი იქნება მისი შედეგების მასივი.

მაგალითად, ქვემოთ მოცემული Promise.all 3 წამის შემდეგ შესრულდება და შედეგად მიიღება [1, 2, 3] მასივი:

```
Promise.all([
  new Promise(resolve => setTimeout(() => resolve(1), 3000)), // 1
```

```
  new Promise(resolve => setTimeout(() => resolve(2), 2000)), // 2
  new Promise(resolve => setTimeout(() => resolve(3), 1000))  // 3
]).then(alert); // როდესაც ყველა Promise-ი შესრულდება,
    შედეგი იქნება 1,2,3
// ყოველი Promise-ი მასივის ელემენტს იძლევა
```

გაითვალისწინეთ, რომ მასივის ელემენტების თანმიმდევრობა ზუსტად ემთხვევა საწყისი Promise-ის თანმიმდევრობას. მაშინაც კი, თუ პირველ Promise-ის შესრულებას ყველაზე დიდი დრო დასჭირდება, მისი შედეგი მასივში მაინც პირველი იქნება.

ხშირად სჯობს, რომ მონაცემთა მასივი გატარებული იყოს map-ფუნქციის საშუალებით, რომელიც ყოველი ელემენტისთვის ამოცანა-Promise-ს შექმნის და შემდეგ მიღებულ მასივს Promise.all-ში გადაიტანს.

მაგალითად, თუ ჩვენ გვაქვს ბმულების მასივი, მაშინ მათი ჩატვირთვა ასე შეგვიძლია:

```
let urls = [
  'https://api.github.com/users/iliakan',
  'https://api.github.com/users/remy',
  'https://api.github.com/users/jeresig'
];

// თითოეულ URL-ს Promise-ში გარდაქმნის, რომელსაც fetch-
    ში დააბრუნებს
let requests = urls.map(url => fetch(url));

// Promise.all ყველა Promise-ის შესრულებას დაელოდება
```

```
Promise.all(requests)
  .then(responses => responses.forEach(
    response => alert(`${response.url}: ${response.status}`)
  ));
```

ქვემოთ მოცემულია მომხმარებლების მასივიდან GitHub მომხმარებლების შესახებ ინფორმაციის მიღების მაგალითი (ანალოგიურად შეიძლება მივიღოთ პროდუქტების მასივი მათი იდენტიფიკატორებით, ლოგიკა იგივეა):

```
let names = ['iliakan', 'remy', 'jeresig'];

let requests = names.map(name =>
    fetch(`https://api.github.com/users/${name}`));

Promise.all(requests)
  .then(responses => {
    // ყველა Promise-ი წარმატებით დამთავრდა
    for(let response of responses) {
      alert(`${response.url}: ${response.status}`); // 200-ს აჩვენებს
      თითოეული ბმულისათვის
    }

    return responses;
  })
  // response პასუხების მასივს გარდაქმნის response.json()-ში,
  // თითოეულის შინაარსის წასაკითხად
  .then(responses => Promise.all(responses.map(r => r.json())))
```

```
// ყველა JSON-პასუხი დამუშავებულია, users - მასივი
   შედეგებით
.then(users => users.forEach(user => alert(user.name)));
```

თუ რომელიმე Promise-ი შეცდომით დამთავრდა, მაშინ დაბრუნებული Promise.all-ი დაუყოვნებლივ დამთავრდება ამ შეცდომით.

შეცდომის შემთხვევაში, დანარჩენი შედეგები იგნორირებულია.

## Promise.allSettled

ეს ფუნქცია ენას ახლახანს დაემატა. ძველ ბრაუზერებში პოლიფილი[1] გამოიყენება.

მისი სინტაქსია:

```
let promise = Promise.allSettled(iterable);
```

Promise.all შეცდომით მთავრდება, თუ ეს ნებისმიერ გადაცემულ Promise-ში მოხდება. ეს მისაღებია სიტუაციისთვის „ყველაფერი ან არაფერი“, სადაც გასაგრძელებლად ყველა შედეგი გვჭირდება:

```
Promise.all([
  fetch('/template.html'),
  fetch('/style.css'),
  fetch('/data.json')
```

[1] პოლიფილი არის კოდი, რომელიც ახორციელებს რაიმე სახის ფუნქციონირებას, რომელიც არ არის მხარდაჭერილი ზოგიერთი ბრაუზერის მიერ. საკუთარი პოლიფილის რეალიზება ერთნაირი შედეგების მიღებას უზრუნველყოფს სხვადასხვა ბრაუზერში.

```
]).then(render); // ყველა fetch-ის შედეგი ჭირდება render მეთოდს
```

Promise.allSettled მეთოდი ყოველთვის ყველა Promise-ის დამთავრებას ელოდება. შედეგების მასივში იქნება:

- {status:"fulfilled",value:შედეგი} წარმატებული დამთავრებისათვის;
- {status:"rejected", reason:შეცდომა} შეცდომისათვის.

მაგალითად, გვსურს ჩავტვირთოთ მრავალი მომხმარებლის ინფორმაცია. მიუხედავად იმისა, თუ რომელიმეში შეცდომაა, ჩვენ დანარჩენი მაინც გვაინტერესებს.

ამისათვის გამოვიყენოთ Promise.allSettled:

```
let urls = [
  'https://api.github.com/users/iliakan',
  'https://api.github.com/users/remy',
  'https://no-such-url'
];

Promise.allSettled(urls.map(url => fetch(url)))
  .then(results => { // (*)
    results.forEach((result, num) => {
      if (result.status == "fulfilled") {
        alert(`${urls[num]}: ${result.value.status}`);
      }
      if (result.status == "rejected") {
        alert(`${urls[num]}: ${result.reason}`);
      }
    });
```

```
});
```

results მასივი (*) სტრიქონში ასეთი იქნება:

```
[
 {status: 'fulfilled', value: ...объект ответа...},
 {status: 'fulfilled', value: ...объект ответа...},
 {status: 'rejected', reason: ...объект ошибки...}
]
```

ანუ ყოველი Promise-ისთვის გვაქვს მისი სტატუსი და მნიშვნელობა/შეცდომა.

### პოლიფილი

თუ ბრაუზერს არ აქვს Promise.allSettled-ის მხარდაჭერა, მისთვის ძალზე მარტივია პოლიფილის (polyfill - მრავალჯერადი) გამოყენება:

```
if(!Promise.allSettled) {
 Promise.allSettled = function(promises) {
  return Promise.all(promises.map(p =>
   Promise.resolve(p).then(value => ({
   status: 'fulfilled',
   value: value
  }), error => ({
   status: 'rejected',
   reason: error
  })));
 };
}
```

ამ კოდში promises.map  იღებს არგუმენტებს, აქცევს მათ Promise-ად (ყოველ შემთხვევისათვის) და თითოეულს ამატებს .then დამმუშავებელს.

ეს დამმუშავებელი წარმატებულ value შედეგს {state:'fulfilled', value: value}-ად, ხოლო error შეცდომას {state:'rejected', reason: error}-ად გადააქცევს. ზუსტად ეს არის Promise.allSettled შედეგების ფორმატი.

შემდეგ ჩვენ შეგვიძლია გამოვიყენოთ Promise.allSettled ყველა Promise-ის შედეგების მისაღებად, თუნდაც რომელიმე მათგანის დროს წარმოიშვება შეცდომა.

## Promise.race

მეთოდი ძალიან ჰგავს Promise.all-ს, მაგრამ მხოლოდ პირველ შესრულებულ Promise-ს ელოდება, საიდანაც იღებს შედეგს (ან შეცდომას).

მისი სინტაქსია:

```
let promise = Promise.race(iterable);
```

მაგალითად, აქ შედეგი იქნება 1:

```
Promise.race([
  new Promise((resolve, reject) => setTimeout(() => resolve(1), 1000)),
  new Promise((resolve, reject) => setTimeout(() => reject(new Error("შეცდომაა!")), 2000)),
  new Promise((resolve, reject) => setTimeout(() => resolve(3), 3000))
]).then(alert); // 1
```

პირველი Promise-ი ყველაზე სწრაფად შესრულდა და შედეგიც გამოიღო. ამის შემდეგ სხვა Promise-ები იგნორირებული იქნებიან.

## Promise.any

მეთოდი ძალიან ჰგავს Promise.race-ს, მაგრამ ელოდება მხოლოდ პირველ წარმატებით დასრულებულ Promise-ს, საიდანაც იღებს შედეგს.

თუ არცერთი გავლილი Promise-ი წარმატებით არ დამთავრდა, მაშინ დაბრუნებული Promise ობიექტი შეცდომის სპეციალური ობიექტის - AggregateError-ის საშუალებით, რომელიც ყველა Promise-ის შეცდომას თავის errors თვისებებში ინახავს, უარყოფილი იქნება.

მისი სინტაქსია:

```
let promise = Promise.any(iterable);
```

მაგალითად, აქ შედეგი იქნება 1:

```
Promise.any([
  new Promise((resolve, reject) => setTimeout(() => reject(new
     Error("Ошибка!")), 1000)),
  new Promise((resolve, reject) => setTimeout(() => resolve(1),
     2000)),
  new Promise((resolve, reject) => setTimeout(() => resolve(3), 3000))
]).then(alert); // 1
```

ამ მაგალითში პირველი Promise-ი ყველაზე სწრაფად შესრულდა, მაგრამ ის უარყოფილ იქნა, ამიტომ შედეგი გახდა მეორე. მას შემდეგ, რაც პირველი წარმატებული Promise-ი შესრულდა, ყველა შემდგომი შედეგი იგნორირებული იქნება.

ქვემოთ მოყვანილია მაგალითი, სადაც ყველა დაპირება უარყოფილია:

```
Promise.any([
  new Promise((resolve, reject) => setTimeout(() => reject(new
    Error("შეცდომაა!")), 1000)),
  new Promise((resolve, reject) => setTimeout(() => reject(new
    Error("კიდევ შეცდომა!")), 2000))
]).catch(error => {
  console.log(error.constructor.name); // AggregateError
  console.log(error.errors[0]); // Error: შეცდომაა!
  console.log(error.errors[1]); // Error: კიდევ შეცდომა!
});
```

როგორც ხედავთ, უარყოფილი Promise-ების შეცდომის ობიექტები AggregateError ობიექტის errors თვისებაშია ხელმისაწვდომი.

## Promise.resolve და Promise.reject

Promise.resolve და Promise.reject მეთოდები იშვიათად გამოიყენება თანამედროვე კოდში, რადგან async/wait სინტაქსი მათ საერთოდ არასაჭიროს ხდის.

ჩვენ მათ აქ სრული სისრულისთვის გავაშუქებთ და ასევე მათთვის, ვინც რაიმე მიზეზით ვერ გამოიყენებს async/wait-ს.

Promise.resolve(value) ქმნის წარმატებით შესრულებულ value შედეგით.

ეს იგივეა რაც:

```
let promise = new Promise(resolve => resolve(value));
```

ეს მეთოდი გამოიყენება თავსებადობისთვის: როდესაც მოსალოდნელია, რომ ფუნქცია ზუსტად Promise-ს დააბრუნებს.

მაგალითად, ქვემოთ მოცემული loadCached ფუნქცია URL-ს ჩატვირთავს და დაიმახსოვრებს (ქეშირებს) მის შინაარსს. იმავე URL-ზე მომავალი გამოძახებების დროს, ის მაშინვე კითხულობს წინა შიგთავსს ქეშ-მეხსიერებიდან, მაგრამ იყენებს Promise.resolve-ს, რათა მისგან შექმნას Promise-ი, რათა დაბრუნებული მნიშვნელობა ყოველთვის Promise-ი იყოს:

```
let cache = new Map();

function loadCached(url) {
  if (cache.has(url)) {
    return Promise.resolve(cache.get(url)); // (*)
  }

  return fetch(url)
    .then(response => response.text())
    .then(text => {
      cache.set(url,text);
      return text;
    });
}
```

ჩვენ შეგვიძლია დავწეროთ loadCached(url).then(…), რადგან loadCached ფუნქცია ყოველთვის აბრუნებს Promise-ს. ჩვენ loadCached-ის შემდეგ ყოველთვის შეგვიძლია გამოვიყენოთ .then . ზუსტად ეს არის Promise.resolve-ის გამოყენების მიზანი (*) სტრიქონში.

Promise.reject(error) ქმნის Promise-ს, რომელიც error შეცდომით მთავრდება.

ეს იგივეა რაც:

```
let promise = new Promise((resolve, reject) => reject(error));
```

პრაქტიკაში ეს მეთოდი თითქმის არასოდეს არ გამოიყენება.

## ასინქრონული ფუნქციები Async/await

Promise-ებთან სამუშაოდ არსებობს სპეციალური სინტაქსი, სახელწოდებით async/await.

დავიწყოთ async საკვანძო სიტყვიდან. ის უნდა ჩაიწეროს ფუნქციის წინ შემდეგნაირად:

```
async function f() {
  return 1;
}
```

სიტყვა async-ს აქვს ერთი მარტივი მნიშვნელობა: ეს ფუნქცია ყოველთვის აბრუნებს Promise-ს. სხვა ტიპის მნიშვნელობები წარმატებით დამთავრებულ Promise-ად ავტომატურად გარდაიქმნება.

მაგალითად, ეს ფუნქცია დააბრუნებს დასრულებულ Promise-ს შედეგით 1:

```
async function f() {
  return 1;
}
```

```
f().then(alert); // 1
```

Promise-ის აშკარა დაბრუნებაც შეიძლება, შედეგი იგივე იქნება:

```
async function f() {
  return Promise.resolve(1);
}
f().then(alert); // 1
```

ასე რომ, async საკვანძო სიტყვა ფუნქციის წინ უზრუნველყოფს, რომ ფუნქცია მაინც Promise-ს დააბრუნებს. მაგრამ ეს ყველაფერი არ არის. არის კიდევ ერთი საკვანძო სიტყვა - await, რომლის გამოყენება მხოლოდ async-ფუნქციის შიგნითაა შესაძლებელი.

მისი სინტაქსია:

```
let value = await promise;
```

await საკვანძო სიტყვა JavaScript-ის ინტერპრეტატორის ლოდინს გამოიწვევს, ვიდრე await-ის მარჯვნივ Promise-ი არ შესრულდება. ამის შემდეგ ის დააბრუნებს თავის შედეგს და კოდის შესრულება გაგრძელდება.

ამ მაგალითში Promise 1 წამში შესრულდება:

```
async function f() {

  let promise = new Promise((resolve, reject) => {
    setTimeout(() => resolve("მზადაა!"), 1000)
  });

```

```
  let result = await promise; // დაელოდება, ვიდრე Promise-ი არ
     შესრულდება (*)

  alert(result); // "მზადაა!"
}

f();
```

ამ მაგალითში, ფუნქციის შესრულება (*) სტრიქონზე Promise-ის შესრულებამდე შეჩერდება. ეს მოხდება ფუნქციის დაწყებიდან 1 წამში. ამის შემდეგ, Promise-ის შესრულების შედეგი ჩაიწერება result ცვლადში და ბრაუზერი alert-ფანჯარაში აჩვენებს "მზადაა!".

გაითვალისწინეთ, რომ სანამ await აიძულებს JavaScript-ს დაელოდოს Promise-ის დასრულებას, ის არ იკავებს პროცესორის რესურსებს. სანამ Promise-ი არ შესრულდება, JavaScript ძრავას შეუძლია სხვა ამოცანების შესრულება: სხვა სკრიპტების შესრულება, მოვლენების მართვა და ა.შ.

await არ შეიძლება გამოყენებულ იქნას ჩვეულებრივ ფუნქციებში.

თუ ჩვენ ვეცდებით await გამოვიყენოთ async-ის გარეშე გამოცხადებული ფუნქციის შიგნით, მივიღებთ სინტაქსურ შეცდომას:

```
function f() {
  let promise = Promise.resolve(1);
  let result = await promise; // SyntaxError
}
```

არ იქნება შეცდომა, თუ ფუნქციის გამოცხადების წინ მივუთითებთ async საკვანძო სიტყვას. როგორც უკვე აღვნიშნეთ, await-ის გამოყენება შესაძლებელია მხოლოდ ასინქრონული ფუნქციების შიგნით.

await-ის გამოყენება promise-ის შიგნით განსაკუთრებით მნიშვნელოვანია ისეთ სიტუაციებში, როდესაც კლიენტის მხარეზე JavaScript-ის კოდში ველოდებით მონაცემებს სერვერის მხრიდან, ე.წ. API-დან, დალოდების პროცესში შეიძლება გაეშვას სხვა რაიმე სკრიპტი მაგალითად დალოდების ე.წ. სპინერი, ხოლო მონაცემების მიღების შემდეგ კი მოხდეს შესაბამისი ამოცანის შესრულება.

# გენერატორები

ჩვეულებრივი ფუნქციები აბრუნებს მხოლოდ ერთადერთ მნიშვნელობას (ან არაფერს).

გენერატორებს საჭიროებისამებრ (yield - სარგებელი) მრავალი მნიშვნელობის ერთმანეთის მიყოლებით გენერაცია შეუძლიათ. გენერატორი მშვენივრად მუშაობს გადასამუშავებელ ობიექტებთან და აადვილებენ მონაცემთა ნაკადების შექმნას.

## ფუნქცია-გენერატორი

გენერატორის გამოსაცხადებლად გამოიყენება სპეციალური სინტაქსური კონსტრუქცია: function*, რომელსაც „ფუნქცია-გენერატორს“ უწოდებენ.

ეს ასე გამოიყურება:

```
function* generateSequence() {
```

```
  yield 1;
  yield 2;
  return 3;
}
```

„ფუნქცია-გენერატორი“ განსხვავებულად იქცევა, ვიდრე ჩვეულებრივი ფუნქცია. როდესაც ასეთი ფუნქცია გამოიძახება, ის არ ასრულებს თავის კოდს. ამის ნაცვლად, ის აბრუნებს სპეციალურ ობიექტს, ეგრეთ წოდებულ „გენერატორს“, რათა გააკონტროლოს მისი შესრულება.

მაგალითი:

```
function* generateSequence() {
  yield 1;
  yield 2;
  return 3;
}

// "ფუნქცია-გენერატორი" ქმნის ობიექტს "გენერატორი"
let generator = generateSequence();
alert(generator); // [object Generator]
```


ფუნქციის კოდის შესრულება ჯერ არ დაწყებულა:

გენერატორის მთავარ მეთოდს წარმოადგენს next(). გამოძახების დროს ის იწყებს კოდის შესრულებას უახლოეს yield <value> ინსტრუქციამდე (მნიშვნელობა შეიძლება არ იყოს, ამ შემთხვევაში მისი მნიშვნელობა undefined-ის ტოლი იქნება). როდესაც მიიღწევა yield-ს მნიშვნელობა ფუნქციის შესრულება შეჩერდება და შესაბამისი მნიშვნელობა უბრუნდება გარე კოდს:

next() მეთოდის შედეგი ყოველთვის არის ობიექტი ორი თვისებით:

- value: მნიშვნელობა yield-დან;
- done: true, თუ ფუნქციის შესრულება დასრულებულია, წინააღმდეგ შემთხვევაში false.

მაგალითად, აქ ჩვენ ვქმნით გენერატორს და ვიღებთ დასაბრუნებელი მნიშვნელობიდან პირველ მნიშვნელობას:

```
function* generateSequence() {
  yield 1;
  yield 2;
  return 3;
}

let generator = generateSequence();

let one = generator.next();

alert(JSON.stringify(one)); // {value: 1, done: false}
```


ამ მომენტში, ჩვენ მივიღეთ მხოლოდ პირველი მნიშვნელობა, ფუნქციის შესრულება მეორე სტრიქონზეა შეჩერებული:

generator.next()-ის ხელახლა გამოძახება განაახლებს კოდის შესრულებას და შედეგად დააბრუნებს შემდეგ yield-ს:

```
let two = generator.next();

alert(JSON.stringify(two)); // {value: 2, done: false}
```

და ბოლოს, ბოლო გამოძახება დაასრულებს ფუნქციის შესრულებას და დააბრუნებს შედეგს:

```
let three = generator.next();

alert(JSON.stringify(three)); // {value: 3, done: true}
```

ახლა გენერატორი სრულად დასრულებულია. ჩვენ შეგვიძლია ეს დავინახოთ done:true თვისებაში და მივიღოთ value:3 , როგორც საბოლოო შედეგი.

generator.next()-ის ახალ გამოძახებას აზრი აღარ აქვს. თუმცა, თუ ეს მოხდება, ისინი შეცდომას არ გამოიწვევს, მაგრამ დააბრუნებს იმავე ობიექტს: {done: true}.

გენერატორის ჩაწერის სინტაქსი function* f(…)-ს თუ function *f(…)-ს?

განსხვავება არ არის, ორივე სინტაქსი სწორია.

მაგრამ, როგორც წესი, უპირატესობა ენიჭება პირველ ვარიანტს, რადგან ფიფქი მიეკუთვნება გამოცხადებულ ტიპს (function* - „ფუნქცია-გენერატორი“) და არა მის სახელს, ამიტომ უკეთესი იქნება, რომ ის განთავსდე სიტყვასთან function.

## გენერატორების გამეორებადობა

როგორც თქვენ ალბათ უკვე მიხვდით next() მეთოდის არსებობით, გენერატორები გამეორებადი ობიექტებია.

მათ მიერ დაბრუნებული მნიშვნელობები შეიძლება გამეორდეს:

```
function* generateSequence() {
  yield 1;
  yield 2;
  return 3;
}

let generator = generateSequence();

for(let value of generator) {
  alert(value); // 1, შემდეგ 2
}
```

გაცილებით ლამაზად გამოიყურება, ვიდრე .next().value-ის გამოყენების დროს.

მაგრამ გაითვალისწინეთ, რომ ზემოთ მოყვანილი მაგალითი გამოიტანს მნიშვნელობას 1-ს, შემდეგ 2-ს. მნიშვნელობა 3-ს არ გამოიტანს!

ეს იმიტომ ხდება, რომ for..of გამეორება უგულებელყოფს ბოლო მნიშვნელობას, სადაც done: true. ამიტომ, თუ გვსურს გვქონდეს ყველა მნიშვნელობა for..of-ის გამეორებისას, მაშინ ჩვენ ისინი yield-ით უნდა დავაბრუნოთ:

```
function* generateSequence() {
  yield 1;
  yield 2;
  yield 3;
}

let generator = generateSequence();

for(let value of generator) {
  alert(value); // 1, შემდეგ 2, შემდეგ 3
}
```

ვინაიდან გენერატორები გამეორებადი ობიექტებია, ჩვენ შეგვიძლია გამოვიყენოთ მათთან დაკავშირებული ყველა ფუნქცია, როგორიცაა გაფართოების ოპერატორი:

```
function* generateSequence() {
  yield 1;
  yield 2;
  yield 3;
}
```

```
let sequence = [0, ...generateSequence()];

alert(sequence); // 0, 1, 2, 3
```

ზემოთ მოცემულ კოდში, generateSequence() ფუნქცია გამეორებადი გენერატორის ობიექტს ელემენტების მასივად გადააქცევს.

## გენერატორების კომპოზიცია

გენერატორების კომპოზიცია გენერატორების განსაკუთრებული მახასიათებელია, რომელიც საშუალებას გვაძლევს გამჭვირვალედ „ჩავსვათ“ გენერატორები ერთმანეთში.

მაგალითად, გვაქვს რიცხვების მიმდევრობის გენერირების ფუნქცია:

```
function* generateSequence(start, end) {
  for (let i = start; i <= end; i++) yield i;
}
```

ჩვენ გვინდა გამოვიყენოთ ის უფრო რთული მიმდევრობის გენერირებისათვის:

- პირველად ციფრები 0..9 (სიმბოლოების კოდებით 48…57);
- რომელსაც მოჰყვება ასოები A..Z სედა რეგისტრში (სიმბოლოების კოდები 65…90);
- შემდეგ მოჰყვება ანბანის ასოები a..z (სიმბოლოების კოდები 97…122).

ჩვენ შეგვიძლია ასეთი თანმიმდევრობა გამოვიყენოთ პაროლების გენერირებისთვის, მისგან ამოვირჩიოთ

სიმბოლოები (შეიძლება კიდევ დავამატოთ პუნქტუაციის სიმბოლოები), მაგრამ ჯერ მისი გენერირებაა საჭირო.

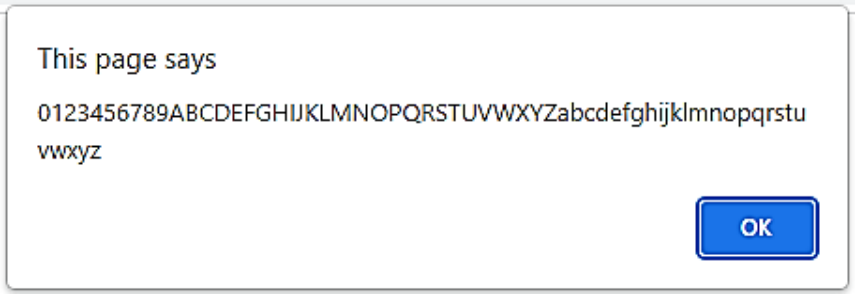
ჩვეულებრივ ფუნქციაში, რამდენიმე სხვა ფუნქციის შედეგების გაერთიანებისთვის, ჩვენ ვიძახებთ მათ, შუალედურ შედეგებს ვინახავთ და შემდეგ ბოლოს ვაერთიანებთ.

გენერატორებისთვის არის სპეციალური yield* სინტაქსი, რომელიც საშუალებას გვაძლევს გენერატორები ერთმანეთში „ჩავალაგოთ“ (მოვახდინოთ მათი კომპოზიცია).

ქვემოთ მოცემულია გენერატორი კომპოზიციით:

```
function* generateSequence(start, end) {
  for (let i = start; i <= end; i++) yield i;
}

function* generatePasswordCodes() {

  // 0..9
  yield* generateSequence(48, 57);

  // A..Z
  yield* generateSequence(65, 90);

  // a..z
  yield* generateSequence(97, 122);

}

let str = '';
```

```
for(let code of generatePasswordCodes()) {
  str += String.fromCharCode(code);
}

alert(str); // 0..9A..Za..z
```




yield* დირექტივა შესრულებას გადასცემს სხვა გენერატორს. ეს ტერმინი ნიშნავს, რომ yield* gen იმეორებს gen გენერატორს და გამჭვირვალედ მიმართავს გარეთ მის გამოტანას. თითქოს მნიშვნელობები გარე გენერატორის მიერ იქნა გენერირებული.

შედეგი იგივეა, თითქოს ჩვენ ჩადგმული გენერატორებიდან ჩავსვით კოდი:

```
function* generateSequence(start, end) {
  for (let i = start; i <= end; i++) yield i;
}

function* generateAlphaNum() {

  // yield* generateSequence(48, 57);
  for (let i = 48; i <= 57; i++) yield i;
```

```
  // yield* generateSequence(65, 90);
  for (let i = 65; i <= 90; i++) yield i;

  // yield* generateSequence(97, 122);
  for (let i = 97; i <= 122; i++) yield i;

}

let str = '';

for(let code of generateAlphaNum()) {
  str += String.fromCharCode(code);
}

alert(str); // 0..9a..zA..Z
```

გენერატორის კომპოზიცია ბუნებრივი გზაა ერთი გენერატორის გამომავალი შედეგების მეორე გენერატორის ნაკადში ჩასართავად. ის არ იყენებს დამატებით მეხსიერებას შუალედური შედეგების შესანახად.

ამ მომენტამდე გენერატორები ძალიან ჰგავდნენ განმეორებით ობიექტებს, მნიშვნელობების გენერირების სპეციალური სინტაქსით. მაგრამ სინამდვილეში, ისინი ბევრად უფრო ძლიერი და მოქნილები არიან.

საქმე იმაშია, რომ yield ორმხრივ მიმართული გზაა: ის არა მარტო აბრუნებს შედეგს გარედან, არამედ შეუძლია გარედან გადასცეს მნიშვნელობა გენერატორს.

ამისათვის ჩვენ უნდა გამოვიძახოთ generator.next(arg) არგუმენტით. ეს არგუმენტი yield-ის შედეგი ხდება.

მოდით ვაჩვენოთ ეს მაგალითზე:

```
function* gen() {
  // კითხვის გადავცემთ გარე კოდს და ველოდებით პასუხს
  let result = yield "2 + 2 = ?";

  alert(result);
}

let generator = gen();

let question = generator.next().value; // <-- yield აბრუნებს პასუხს

generator.next(4); // --> შედეგს ვუგზავნით გენერატორს
```

პირველი გამოძახება generator.next() ყოველთვის არგუმენტის გარეშეა, ის იწყებს შესრულებას და აბრუნებს პირველ yield "2+2=?" შედეგს. ამ დროს გენერატორი აჩერებს შესრულებას.

შემდეგ, როგორც ზემოთაა ნაჩვენები, yield-ის შედეგი გადაეცემა გარე კოდის question ცვლადს.

generator.next(4)-ით გენერატორი განაახლებს შესრულებას და 4 გამოდის როგორც მინიჭების შედეგი: let result = 4.

გაითვალისწინეთ, რომ გარე კოდი არ არის ვალდებული, რომ დაუყონებლივ გამოიძახოს next(4). მას შეიძლება დასჭირდეს დრო. ეს არ არის პრობლემა, გენერატორი დაელოდება.

მაგალითად:

```
// განაახლეთ გენერატორი გარკვეული დროის შემდეგ
setTimeout(() => generator.next(4), 1000);
```

იმისათვის, რომ უფრო ცხადი გახდეს, რა ხდება, აქ კიდევ ერთ მაგალითს მოვიყვანთ დიდი რაოდენობის გამოძახებებით:

```
function* gen() {
  let ask1 = yield "2 + 2 = ?";

  alert(ask1); // 4

  let ask2 = yield "3 * 3 = ?"

  alert(ask2); // 9
}

let generator = gen();

alert( generator.next().value ); // "2 + 2 = ?"

alert( generator.next(4).value ); // "3 * 3 = ?"

alert( generator.next(9).done ); // true
```

1. პირველი .next() იწყებს შესრულებას. ის მიდის პირველ yield-მდე;
2. შედეგი უბრუნდება გარე კოდს;

3. მეორე .next(4) გადასცემს 4-ს გენერატორს როგორც პირველი yield-ის შედეგს და განაახლებს შესრულებას;

4. შემდეგ მიდის მეორე yield-მდე, რომელიც იქნება.next(4)-ის შედეგი;

5. მესამე next(9) გადასცემს 9-ს გენერატორს როგორც მეორე yield-ის შედეგს და განაახლებს შესრულებას, რომელიც მთავრდება ფუნქციის დასასრულით, ასე, რომ done: true.

გამოდის ასე: ყოველი next(value) გადასცემს გენერატორს მნიშვნელობას, რომელიც ხდება მიმდინარე yield-ის შედეგი, განაახლებს შესრულებას და იღებს გამოსახულებას შემდეგი yield-დან.

## generator.throw

როგორც ზემოთ მაგალითებში ვნახეთ, გარე კოდს შეუძლია გადასცეს მნიშვნელობა გენერატორს როგორც yield-ის შედეგი.

მაგრამ თქვენ შეგიძლიათ გადასცეთ არა მხოლოდ შედეგი, არამედ მოახდინოთ შეცდომის ინიცირება. ეს ბუნებრივია, რადგან შეცდომა ერთგვარი შედეგია.

იმისათვის, რომ შეცდომა გადასცეთ yield-ში, უნდა გამოვიძახოთ generator.throw(err). ამ შემთხვევაში, err-ის გამონაკლისი yield-ის სტრიქონში წარმოიქმნება.

მაგალითად, აქ yield "2 + 2 =?" გამოიწვევს შეცდომას:

```
function* gen() {
  try {
    let result = yield "2 + 2 = ?"; // (1)
```

```
    alert("პროგრამის შესრულება ამ სტრიქონამდე არ მივა,
      ვინაიდან ზემოთ წარმოიშვება შეცდომა");
  } catch(e) {
    alert(e); // აჩვენებს შეცდომას
  }
}

let generator = gen();

let question = generator.next().value;

generator.throw(new Error("ჩემს მონაცემთა ბაზაში პასუხი ვერ
    ვიპოვე")); // (2)
```

შეცდომა, რომელიც გადაცემულია გენერატორში (2) სტრიქონში, გამონაკლისის სახით მივყავართ (1) სტრიქონზე yield-ით. ზემოთ მოყვანილ მაგალითში try..catch იჭერს და ასახავს მას. თუ ჩვენ არ გვინდა მისი დაჭერა, მაშინ ის, როგორც ნებისმიერი ჩვეულებრივი გამონაკლისი, გენერატორიდან გარე კოდს გადაეცემა.

გამოძახების კოდის მიმდინარე სტრიქონი ეს არის სტრიქონი generator.throw-ით, მონიშნულია (2). ასე რომ, ჩვენ შეგვიძლია დავიჭიროთ ის გარე კოდში ასე:

```
function* generate() {
  let result = yield "2 + 2 = ?"; // შეცდომა ამ სტრიქონშია
}

let generator = generate();
```

```
let question = generator.next().value;

try {
  generator.throw(new Error("ჩემს მონაცემთა ბაზაში პასუხი ვერ
     ვიპოვე"));
} catch(e) {
  alert(e); // აჩვენებს შეცდომას
}
```

თუ შეცდომა იქ არ არის დაფიქსირებული, მაშინ შემდეგში - ჩვეულებისამებრ, ის გამოჩნდება გარეთ და, თუ არ იქნა დაფიქსირებული, სკრიპტი „ჩაგვივარდება“.

## ასინქრონული იტერატორები და გენერატორები

ასინქრონული იტერატორები საშუალებას გვაძლევს გადავარჩიოთ მონაცემები, რომლებიც ასინქრონულად მოდის. მაგალითად, როცა რაღაცას ნაწილ-ნაწილ გადმოვტვირთავთ ქსელში. ასინქრონული გენერატორები ამ ძიებას კიდევ უფრო მოსახერხებელს ხდის.

სინტაქსის მარტივი მაგალითი განვიხილოთ.

ასინქრონული იტერატორები ჩვეულებრივი იტერატორე-ბის მსგავსია, მაგრამ გარკვეული სინტაქსური განსხვავებები აქვთ.

ჩვეულებრივი გამეორებადი ობიექტი ასე გამოიყურება:

```
let range = {
  from: 1,
```

```
  to: 5,

  // for..of ამ მეთოდს დასაწყისში მხოლოდ ერთხელ იძახებს
  [Symbol.iterator]() {
    // ...აბრუნებს ობიექტს-იტერატორს:
    // შემდეგში for..of მხოლოდ ამ ობიექტთან მუშაობს, შემდეგი
      მნიშვნელობის მოთხოვნა next()-ის გამოძახებით ხდება
    return {
      current: this.from,
      last: this.to,

      // next()-ის გამოძახება for..of ციკლის ყოველ იტერაციაზე
      ხორციელდება
      next() { // (2)
        // მნიშვნელობა {done:.., value :...} ობიექტის სახით უნდა
      დააბრუნოს
        if (this.current <= this.last) {
          return { done: false, value: this.current++ };
        } else {
          return { done: true };
        }
      }
    };
  }
};

for(let value of range) {
  alert(value); // 1, შემდეგ 2, შემდეგ 3, შემდეგ 4, შემდეგ 5
```

```
}
```

ობიექტის ასინქრონულად გამეორება რომ მოხდეს:

1. Symbol.iterator-ის ნაცვლად უნდა გამოვიყენოთ Symbol.asyncIterator;
2. next()-მა უნდა დააბრუნოს Promise;
3. ასეთ ობიექტის გადასარჩევად გამოიყენება for await (let item of iterable) ციკლი.

მოდით შევქმნათ იტერაციული range ობიექტი, როგორც წინა მაგალითში, მაგრამ ახლა ის ასინქრონულად დააბრუნებს წამში ერთ მნიშვნელობას:

```
let range = {
  from: 1,
  to: 5,

  // for await..of ამ მეთოდს დასაწყისში მხოლოდ ერთხელ
     იძახებს
  [Symbol.asyncIterator]() { // (1)
    // ...აბრუნებს ობიექტს-იტერატორს:
    // შემდეგში for await..of მხოლოდ ამ ობიექტთან მუშაობს,
    // მასთან შემდეგი მნიშვნელობის მოთხოვნა next()-ის
      გამოძახებით ხდება
    return {
      current: this.from,
      last: this.to,

      // next()-ის გამოძახება for await..of ციკლის ყოველ
      იტერაციაზე ხორციელდება
```

```
    async next() { // (2)
      // მნიშვნელობა {done:.., value :...} ობიექტის სახით უნდა
    დააბრუნოს
      // (async დახმარებით ავტომატურად აბრუნებს Promise-ს)

      // ასინქრონულობისათვის შიგნით შეიძლება
    გამოვიყენოთ await:
      await new Promise(resolve => setTimeout(resolve, 1000)); // (3)

      if (this.current <= this.last) {
        return { done: false, value: this.current++ };
      } else {
        return { done: true };
      }
    }
  };
 }
};

(async () => {

 for await (let value of range) { // (4)
  alert(value); // 1,2,3,4,5
 }

})()
```

როგორც ხედავთ, სტრუქტურა ჩვეულებრივი იტერატორის მსგავსია:

1. იმისათვის, რომ ობიექტი ასინქრონული იტერატორი იყოს, მას უნდა ჰქონდეს Symbol.asyncIterator (1) მეთოდი;

2. ამ მეთოდმა უნდა დააბრუნოს ობიექტი next() მეთოდით, რომელიც თავის მხრივ Promise-ს აბრუნებს (2);

3. next() მეთოდი აუცილებელი არ არის იყოს ასინქრონული, ის შეიძლება იყოს ჩვეულებრივი მეთოდი რომელიც Promise-ს დააბრუნებს, მაგრამ async გაძლევთ საშუალებას გამოიყენოთ await, ამიტომ ის უფრო მოსახერხებელია. უბრალოდ, ჩვენ აქ ვაკეთებთ პაუზას ერთი წამით (3);

4. იტერაციისთვის ვიყენებთ for await-ს (let value of range) (4), for-ის შემდეგ ვამატებთ wait-ს. ის ერთხელ გამოიძახებს range [Symbol.asyncIterator]() და შემდეგ მის next() მეთოდს მნიშვნელობების მისაღებად.

| | იტერატორები | ასინქრონული იტერატორები |
|---|---|---|
| იტერაციული ობიექტის შესაქმნელი მეთოდი | Symbol.iterator | Symbol.asyncIterator |
| next() აბრუნებს | ნებისმიერი მნიშვნელობა | Promise |
| ციკლისათვის გამოიყენება | for..of | for await..of |

ფუნქციები, რომლებიც საჭიროებენ ნორმალურ სინქრონულ იტერატორებს, არ მუშაობს ასინქრონულთან.

მაგალითად, გაფართოების ოპერატორი (სამი წერტილი ...) არ იმუშავებს:

```
alert( [...range] ); // შეცდომაა, არ არის Symbol.iterator
```

ეს ბუნებრივია, რადგან ის მოელის Symbol.iterator-ს, ისევე როგორც for..of-ს await გარეშე. Symbol.asyncIterator მას არ გამოიყენება.

როგორც უკვე ვიცით, JavaScript-ს აქვს გენერატორები და ისინი გამეორებადია.

გავიხსენოთ მიმდევრობის გენერატორი. ის ახდენს მნიშვნელობების მიმდევრობის გენერირებას start-დან end-მდე:

```
function* generateSequence(start, end) {
  for (let i = start; i <= end; i++) {
    yield i;
  }
}

for(let value of generateSequence(1, 5)) {
  alert(value); // 1, შემდეგ 2, შემდეგ 3, შემდეგ 4, შემდეგ 5
}
```

ჩვეულებრივ გენერატორებში ჩვენ ვერ გამოვიყენებთ await-ს. ყველა მნიშვნელობა უნდა მიეწოდოს სინქრონულად: for..of-ში შეფერხების ადგილი, ეს არის სინქრონული კონსტრუქცია. მაგრამ, თუ ჩვენ გენერატორის ტანში await-ის გამოყენება გვჭირდება? მაგალითად, ქსელის მოთხოვნების შესასრულებლად. პრობლემა არ არის, უბრალოდ დასაწყისში async დაამატეთ:

```
async function* generateSequence(start, end) {

  for (let i = start; i <= end; i++) {

    // შეიძლება await-ის გამოყენება!
    await new Promise(resolve => setTimeout(resolve, 1000));

    yield i;
  }

}

(async () => {

  let generator = generateSequence(1, 5);
  for await (let value of generator) {
    alert(value); // 1, შემდეგ 2, შემდეგ 3, შემდეგ 4, შემდეგ 5
  }

})();
```

ჩვენ ახლა გვაქვს ასინქრონული გენერატორი, რომლის გამეორება for await ... of-ის დახმარებითაა შესაძლებელი. ეს მართლაც ძალიან მარტივია. ჩვენ ვამატებთ async საკვანძო სიტყვას და გენერატორში ახლა შეგიძლიათ გამოიყენოთ await, ასევე Promise-ი და სხვა ასინქრონული ფუნქციები.

ტექნიკური თვალსაზრისით, ასინქრონული გენერატორის კიდევ ერთი განსხვავება ის არის, რომ მისი generator.next() მეთოდი ახლა ასევე ასინქრონულია და Promise-ს აბრუნებს.

ჩვეულებრივი გენერატორიდან result = generator.next()-ის საშუალებით შეგვიძლია მივიღოთ მნიშვნელობები. ასინქრონულისთვის, თქვენ უნდა დაამატოთ await, ასე:

```
result = await generator.next(); // result = {value: ..., done: true/false}
```

## ასინქრონულად გამეორებადი ობიექტები

როგორც უკვე ვიცით, ობიექტი გავხადოთ გამეორებადი, მას უნდა დავამატოთ Symbol.iterator.

```
let range = {
  from: 1,
  to: 5,
  [Symbol.iterator]() {
    return <ობიექტი next-ით, რომ გავხადოთ range-ი
      გამეორებადი>
  }
}
```

Symbol.iterator-ისთვის ჩვეულებრივი პრაქტიკა არის გენერატორის დაბრუნება და არა მარტივი ობიექტის დაბრუნება next-ით, როგორც წინა მაგალითში.

გავიხსენოთ მაგალითი გენერატორების თავიდან:

```
let range = {
  from: 1,
  to: 5,

  *[Symbol.iterator]() { // შემოკლებული ფორმა [Symbol.iterator]:
      function*()-თვის
```

```
    for(let value = this.from; value <= this.to; value++) {
      yield value;
    }
  }
};

for(let value of range) {
  alert(value); // 1, შემდეგ 2, შემდეგ 3, შემდეგ 4, შემდეგ 5
}
```

აქ შექმნილი range ობიექტი განმეორებადია და *[Symbol.iterator] გენერატორი მნიშვნელობების ჩამოთვლის ლოგიკას ახორციელებს.

თუ გვსურს გენერატორში დავამატოთ ასინქრონული მოქმედებები, მაშინ Symbol.iterator-ი ასინქრონული Symbol.asyncIterator-ით უნდა შევცვალოთ:

```
let range = {
  from: 1,
  to: 5,

  async *[Symbol.asyncIterator]() { // იგივეა რაც
      [Symbol.asyncIterator]: async function*()
    for(let value = this.from; value <= this.to; value++) {

      // მოლოდინი, პაუზა მნიშვნელობებს შორის
      await new Promise(resolve => setTimeout(resolve, 1000));

      yield value;
```

```
    }
  }
};

(async () => {

  for await (let value of orange) {
    alert(value); // 1, შემდეგ 2, შემდეგ 3, შემდეგ 4, შემდეგ 5
  }

})();
```

ახლა მნიშვნელობების მიწოდება ერთი წამის დაყოვნებით განხორციელდება.

# 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%20Book.pdf

გადაეცა წარმოებას 20.11.2023. ხელმოწერილია დასაბეჭდად 27.11.2023.
ქაღალდის ზომა 60X84 1/16. პირობითი ნაბეჭდი თაბახი 17,7.

„ევროპის უნივერსიტეტი“,
თბილისი, დ. გურამიშვილის 76

Verba volant,
scripta manent

ი.მ. „გოჩა დალაქიშვილი“,
თბილისი, ვარკეთილი 3, კორპ. 333, ბინა 38