Devid Flanaqanın "The Definitive Guide" kitabının əsasında hazırlanmışdır.

Aktiv veb-saytların hazırlanması

JavaScript

geniş izah,
ətraflı baxış,
əsas anlayışlar

O'REILLY®

Məcidov Abbas

JavaScript

The Definitive Guide

Firth Edition

David Flanagan

O'REILLY®

JavaScript

Ətraflı izah

Beşinci buraxılış

Abbas Məcidov

(tərcüməçi və təşkilatçı)

Bakı – Gəncə

2015 - 2016

Mündəricat

1

JavaScript-ə giriş

JavaScript – obyektyönümlü proqramlaşdırma imkanları olan interprerativ proqramlaşdırma dilidir. Sintaksis nöqteyi-nəzərindən JavaScript-in baza dili C, C++ və Java PD-lərinin proqram konstruksiyalarına bənzəyir (məs.: if, while və && operatort). Ancaq bu oxşarlıq yalnız sintaktis oxşarlıqilə məhdudlaşır.

Şəkil 1.1. *JavaScript-in qeyri-rəsmi loqotipi*

JavaScript –tipləşdirilməmiş dildir , yəni bu dildə dəyişənlərin tiplərini müəyyən etmək tələb olunur. JavaScript-də obyektlər, ad xüsusiyyətlərində sərbəst qiymətləri əks etdirir. Bu xüsusiyyətinə görə Perl PD-sinin assosiativ massivlərini, C strukturlarını və ya C++ və ya Java obyektlərini xatırladırlar. JavaScript dilinin nüvəsi sadə məlumat tipləri, ədədlər, sətirlər və Bull qiymətlərini dəstəkləyir.

Bundan başqa bu dil massivlər, tarixlərin və müntəzəm ifadə obyektlərinin inteqrasiya edilmiş dəstinə malikdir. Adətən JavaScript veb-brauzerlərdə tətbiq edilir. Obyektlərin tətbiqi nəticəsində bu dilin imkanları daha genişdir. JavaScript, istifadəçi ilə qarşılıqlı təsiri təşkil etməyə, veb-brauzeri idarə etməyə və veb-brauzerin pəncərəsinin daxilində əks etdirilən sənədin tərkibini dəyişdirməyə imkan verir. JavaScript veb səhifələrin HTML-koduna tətbiq edilmiş ssenarilər vasitəsilə inteqrasiya edir. Bir qayda olaraq, bu versiya JavaScript-in kliyent dili adlanır. Qeyd etmək lazımdır ki, ssenari kliyenti veb-serverdə deyil, kompüterinizdə (oflayn rejimdə) icra edilir. JavaScript və bu dilin dəstəklədiyi məlumat tipləri dili beynəlxalq standartlara əsaslanır. Bunun sayəsində reallaşdırmalar arasında çox gözəl uyğunluq yaranır.

Bunun sayəsində reallaşdırmalar arasında çox gözəl uyğunluq yaranır. JavaScript-kliyentinin bəzi hissələri rəsmi standart, bəzi hissələri hissələr de-fakto standartdır, amma kliyentin elə hissələri də var ki, brauzerin konkret versiyasına uyğun spesifik genişlənmədir. Müxtəlif

brauzerlərdə JavaScript reallaşdırmalarının uyğunluğu JavaScript-in kliyent dilindən istifadə edən proqramçıları tez-tez düşündürür və narahat edir. Bu fəsildə dilin imkanlarının faktiki öyrənməsinə keçməzdən əvvəl JavaScript-in qısa icmalı və giriş informasiyası verilir.
Bundan başqa, praktik veb-proqramlaşdırma JavaScript-in kliyent dilində bir neçə kod fraqmenti bu fəsildə nümayiş etdirilir.

1.1. JavaScript nədir

JavaScript-in haqqında çoxlu dezinformasiyalar və qarışıq məlumatlar mövcuddur. JavaScript-in öyrənilməsindən əvvəl, bu dillə bağlı yayılmış bəzi mifik məlumatları aydınlaşdıraq.

1.1.1. JavaScript – Java deyil

JavaScript haqqında ən geniş yayılmış yanlış fikirlərdən biri ondan ibarətdir ki, guya bu dil Sun Microsystems şirkəti tərəfindən hazırlanmış Java proqramlaşdırma dilinin sadələşdirilmiş versiyasıdır. Bəzi sintaktis oxşarlıq və veb-brauzerə icra edilə bilən tərkib mənimsətmək qabiliyyətindən savayı, bu iki dil arasında heç bir bağlılıq yoxdur. Adlarının oxşarlığına gəlincə isə, bu sadəcə marketinq siyasətinin nəticəsidir (dilin ilkin adı LiveScript olmuşdur, sonralar isə bu dil JavaScript adlandırdırıldı). Ancaq JavaScript və Java bir-birinə qarşılıqlı təsir edə bilər.

1.1.2. JavaScript sadə dil deyil

Bir halda ki, JavaScript izah edilən dildir və bu dil adətən proqramlaşdırma dili kimi deyil ssenarilər şəklində yerləşdirilir, bu halda nəzərə almaq lazımdır ki, ssenari dilləri təkcə, çox mürəkkəb şəraitdə işləməyi bacaran proqramçıdan tutmuş, elementar proqramlaşdırma bilikləri olan proqramçı üçün yönəlməmişdir, hətta bu dil adi istifadəçi üçün də nəzərdə tutulmuşdur. Əslində, JavaScript tipləşdirilməmiş xüsusiyyəti sayəsində, təcrübəsiz proqramçılar tərəfindən tiplərin təyini zamanı buraxılan səhvləri güzəştə gedir. Buna görə də əksər veb-dizaynerlər dəqiq reseptlər üzrə yerinə yetirilən məsələlərin məhdudlaşmış əhatədə həlli üçün JavaScript-dən istifadə edir. Ancaq

JavaScript-in zahiri sadəliyi baxmayaraq, daxilində bir o qədər də mürəkkəb və dəyərli proqramlaşdırma dili gizlənir. JavaScript imkanlarını tam anlamadan bu dilin köməyilə qeyri-adi məsələləri həll etməyə çalışan proqramçılar hazırlanma prosesində tez-tez məyus olurlar. Bu kitabda, JavaScript-in hərtərəfli təsviri verilmişdir. Əgər bu kitabdan əvvəl hazır məzmunu ehtiva edən JavaScript məlumat kitabçalarından istifadə etmisinizsə, bu kitabdakı fəsillərinin dərinliyi və ifadə təfərrüatı yəqin ki, sizi təəccübləndirəcək.

1.2. JavaScript kliyenti

JavaScript interpretatoru veb-brauzerə JavaScript kliyenti vasitəsilə inteqrasiya edir. Elə buna görə də, JavaScript dedikdə, insanların ağlına ilk öncə JavaScript kliyenti gəlir. Bu kitabda JavaScript-in kliyent dili bu dilin alt çoxluğunu təşkil edən JavaScript-bazası ilə birlikdə təsvir edilir.

JavaScript kliyentinin daxilində JavaScript interpretatorunu və obyektiveb-brauzerlə müəyyən edilən sənədin obyekt modeli (***Document Object Model, DOM***) yerləşir.

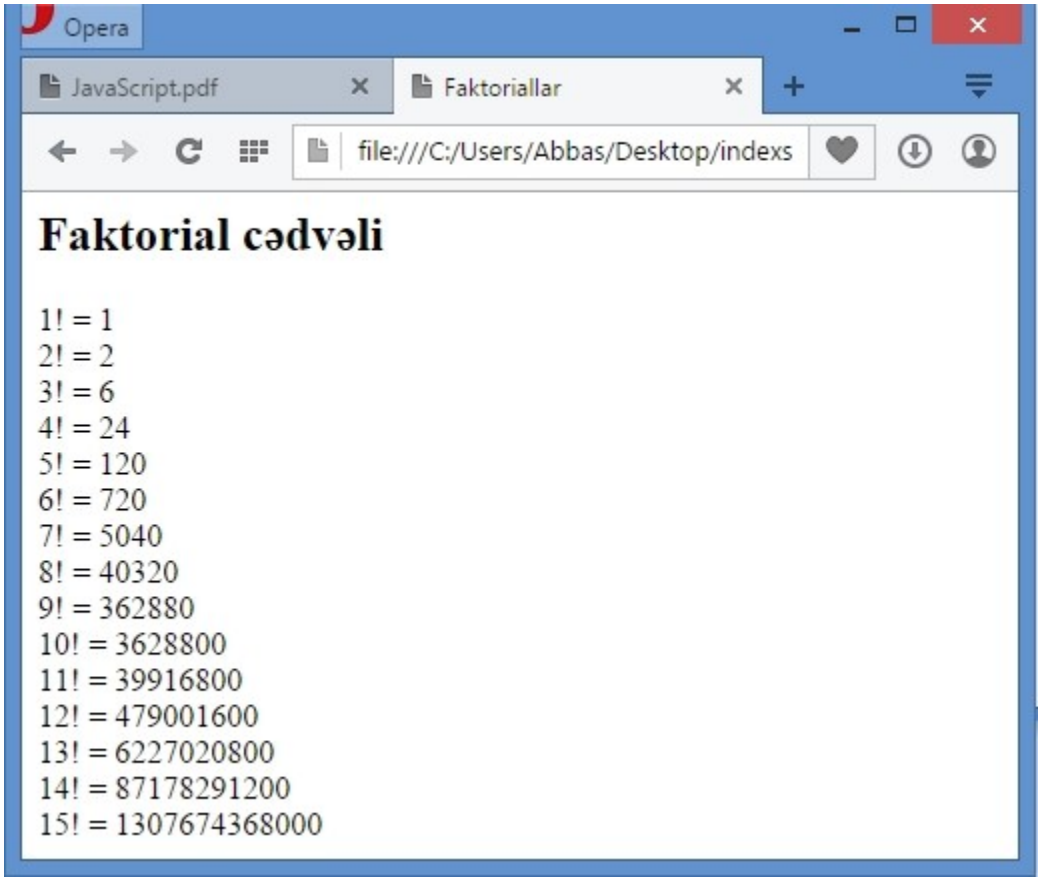
Sənədlər JavaScript-ssenariləri özündə saxlaya bilər. Bu ssenarilər, öz növbəsində DOM modelindən istifadə edərək, sənədin və ya idarə etmənin üsulunun modifikasiyası üçün istifadə edilir. Başqa sözlə demək olar ki, JavaScript kliyenti veb səhifələrin statik tərkibin davranışını müəyyən etməyə imkan verir. JavaScript kliyenti veb-proqramların hazırlamasının texnologiyalarının əsasıdır, DHTML (16-cı fəsil), AJAX (20-ci fəsil) arxitekturaların). ECMA-262 spesifikasiyası JavaScript-in baza dilinin standart versiyasını müəyyən edir və World Wide Web Consortium (W3C) təşkilatı tərəfindən standartlaşdıran DOM spesifikasiyasını hazırlamışdır hansı ki, brauzer bu standartı öz obyekt modelində dəstəkləməlidir. W3C DOM standartı ən məşhur brauzerlərlə tam dəstəklənir. Yalnız – Microsoft Internet Explorer tərəfindən dəstəklənmir; bu brauzerdə hadisələrin emalı mexanizminin dəstəyi yoxdur.

1.2.1. JavaScript kliyentindən istifadə nümunələri

JavaScript interpretatoru ilə təchiz edilmiş veb-brauzer İnternet vasitəsilə JavaScript-ssenarilər şəklində icra edilən tərkibi göstərə bilir. Nümunə 1.1-də JavaScript dilində veb-səhidəyə inteqrasiya edilmiş sadə proqram göstərilmişdir.

Nümunə1.1.JavaScriptdilindəsadəproqram

```
<html>
 <head>
   <meta charset = "utf-8">
   <title>Faktoriallar</title>
 </head>
 <body>
   <h2>Faktorial cədvəli</h2>
   <script>
    var fact = 1;
    for(i = 1; i < 16; i++) {
      fact = fact*i;
      document.write(i + "! = " + fact + "<br>");
    }
   </script>
 </body>
</html>
```



Şəkil 1.2. *JavaScript-də faktorial siyahısı alqoritmi*

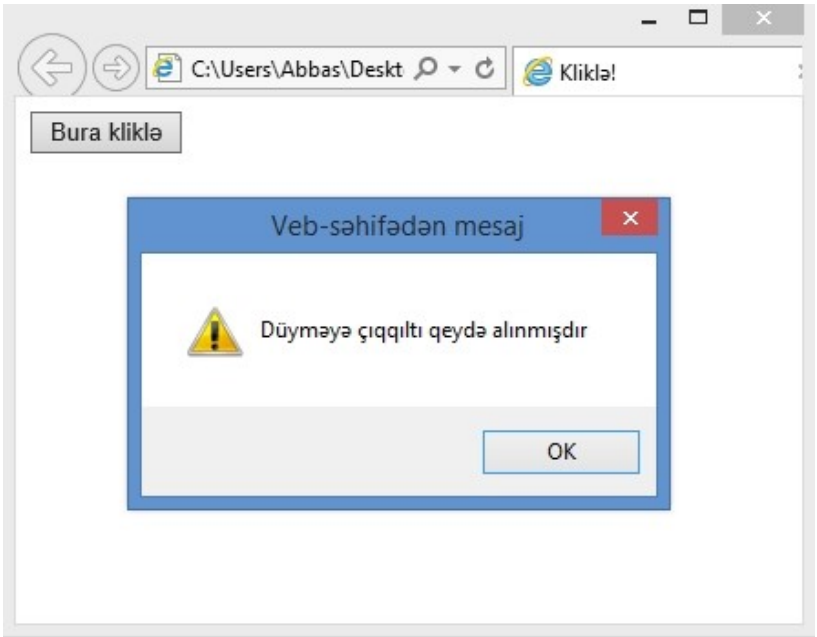
Sözügedən proqram JavaScript-i dəstəkləyən brauzer vasitəsilə icra edildikdə, şəkildəki nəticəni verəcək.

Bu nümunədən göründüyü kimi, JavaScript-kodunun HTML-faylına yerləşdirilməsi üçün `<script>` və `</script>` teqlərindən istifadə edilmişdir. Burada əsas – `document.write()` metodundan istifadə edilir. Bu metod, sənədin veb-brauzerlə yüklənməsi anında HTML-sənədin daxilində dinamik HTML-mətn istehsal etməyə imkan verir.

JavaScript yalnız HTML-sənədin tərkibini deyil, həm də onların davranışının idarəetməsini təmin edir. Başqa sözlə, JavaScript-proqram istifadəçisinin hərəkətlərinə reaksiya verə bilər (məs.: mətn sahəsinə qiymətin daxil edilməsi və ya sənəddə təsvir sahəsində siçandan icra olunan çıqqıltıya cavab). Bu sənəd üçün hadisələrin emalçılarının təyini yolu ilə əldə edilir. Məsələn, müəyyən hadisənin yaranması anında icra edilən JavaScript-kodlarını nümunə göstərmək olar. ***Nümunə 1.2***-də HTML-kodun sadə fraqmenti göstərilmişdir. Burada emalçı çıqqıltıya cavab olaraq müəyyən hadisəni (burada xəbərdarlıq) icra edir.

Nümunə1.2. *JavaScript dilində hadisə emalçısı vasitəsilə HTML-düymə*

```
<button onclick="alert('Düyməyə çıqqıltı qeydə alınmışdır');">Bura
kliklə</button>
```



Şəkil 1.3. *JavaScript-də hadisə emalçısı*

Şəkil 1.3-də düyməyə çıqqıltının nəticəsi göstərilmişdir. Nümunə 1.2-dəki onclick atributu – icra edilən JavaScript-kodunun sətridir. Göründüyü kimi bu atribut kliklәməyə cavab verir. Bu nümunədə `onclick` hadisəsinin emalçısı `alert()` funksiyasına səbəb olur. Şəkildən göründüyü kimi.

Nümunə 1.2-dəki, `alert()` funksiyası göstərilən mesajla birlikdə dialoq pəncərəsini vasitəsilə təsvir edilir. ***1.1.*** və ***1.2.*** nümunələri JavaScript kliyentinin ən sadə imkanlarıdır. Bu dilin real gücü ondan ibarətdir ki, ssenarilər HTML-sənədin tərkibində yerləşir.

1.3. JavaScript-in başqa sahələrdə istifadəsi

JavaScript – ümumi təyinatlı proqramlaşdırma dilidir və onun istifadəsi tək veb-brauzerlər ilə məhdudlaşmır. Əvvəllər JavaScript, istənilən proqrama ssenarilər ilə yerləşdirilir və icra edilir. İlk günlərdən Netscape şirkətinin veb-serverləri JavaScript ssenarilərini icra etməyi bacaran JavaScript interpretatoru dəstəkləyirdi. Oxşar üsulla Microsoft korporasiyası Internet Explorer-ə əlavə olaraq öz veb serverlərində IIS və Windows Scripting Host məhsulu üçün JScript interpretatoru istifadə edir. Adobe şirkəti öz Flash- fayllarının oxuyucusunun idarə etməsi üçün JavaScript-dən törəyən dili cəlb etdi. Həmçinin Sun şirkəti Java 6.0 distributivinə JavaScript interpretatoru quraşdırdı və bunun sayəsində istənilən Java-proqramına ssenarilərin yerləşdirilməsi imkanı əhəmiyyətli dərəcədə yüngülləşdi.

Netscape və Microsoft öz JavaScript interpretatorlarının reallaşdırmalarını, öz proqramlarını əlavə etmək istəyən şirkətlər və proqramçılar üçün əlçatan etdi. Netscape şirkəti tərəfindən yaradılmış interpretator açıq mənbəli və azad yayılan PT olub, hal-hazırda Mozilla (***http://www.mozilla.org/js/***) təşkilatı vasitəsilə yayılır. Mozilla faktiki olaraq JavaScript 1.5 interpretatorunin iki müxtəlif versiyasını yayır: biri C dilində yazılmışdır və SpiderMonkey adlanır, o biri isə Java dilində yazılmışdır və kitabın müəllifinin fikrincə çox təkmil şəkildə hazırlanmış və Rhino (kərgədan) adı verilmiş interpretatordur.

1.4. JavaScript öyrənilməsi

İstənilən yeni proqramlaşdırma dilinin metodikası zamanı praktikadan istifadə etmək lazımdır. Bu kitabı oxuduğunuzda, sizə JavaScript imkanlarını yoxlamağı məsləhət görürəm. Məsələn, sadə funksiyalardan ibarət kodlardan başlaya bilərsiniz. Proqram kodunu oxumağa və anlamağa çalışın. JavaScript öyrənilməsinə ən aşkar yanaşma – sadə ssenarilərin yazılışıdır. JavaScript kliyentinin üstünlüklərindən biri ondan ibarətdir ki, işləmə mühiti hər hansı, veb-brauzer və ən sadə mətn redaktoru ilə qurulmuşdur. JavaScript-də proqramlar yazmaq üçün, xüsusi proqram təminatının yüklənməsinə ehtiyac yoxdur. Məsələn, fatorialların yerinə ***Fibonaççi ədədlərinin*** ardıcıllığını göstərmək üçün, aşağıdakı nümunə 1.1 kopyalamaq olar:

```
<script>
  document.write("<h2>Fibonaççi ədədləri</h2>");
  for(i=0, j=1, k=0, fib =0; i<50; i++, fib=j+k, j=k, k=fib){
     document.write("Fibonacci ("+i+") ="+ fib);
     document.write("</br>");
  }
</script>
```

Bu kod sizə qəliz görünə bilər (əgər kodu anlaya bilmədizsə, narahat olmayın), amma belə qısa proqramlarla təcrübə etmək üçün, kodu olduğu kimi kopyalamaq və lokal URL-ünvanlı fayl kimi veb- brauzerdə onu icra etmək kifayətdir. Nəzərə alın ki, hesablamaların nəticəsini göstərmək üçün *document.write()* metodundan istifadə olunur. Bu metoddan istifadə, JavaScript ilə tanışlıq zamanı faydalıdır. Alternativ olaraq dialoq pəncərəsində sadə mətn nəticəsini göstərmək üçün *alert()* metodunu tətbiq etmək olar:

```
alert("Fibonaççi(" + i + ") = " + fib);
```

Qeyd edək ki, JavaScript ilə belə sadə sınaq kodlarını HTML faylın daxilində <html>, <head> və <body> teqlərinin içərisində yerləşdirmək olar.
JavaScript ilə sınaqların daha da sadələşdirməsi üçün URL-ünvana mənimsədilə bilən javascript: psevdoprotokol spesifikatoru yaradılmışdır. Bu üsul ayəsində JavaScript-də ifadənin və qiymətin nəticənin hesablamaq olar. Belə URL-ünvan psevdoprotokol spesifikatorundan (javascript:) ibarətdir, hansı ki, burada sərbəst olaraq JavaScript-kodu (təlimatlar biri-birindən nöqtəli vergül ilə ayrılır) göstərilir. URL-ünvanı

psevdoprotokol vasitəsilə yükləndikdə, brauzer sadəcə JavaScriptkodunu icra edir. Belə URL- ünvanda ifadənin son qiyməti sətir tipinə dəyişdiriləcək və bu sətir yeni sənəd kimi veb-brauzer vasitəsilə göstəriləcək. Məsələn, bəzi operatorların və JavaScript dilinin təlimatlarını yoxlamaq üçün, veb-brauzerin ünvan sahəsində aşağıdakı URL-ünvanları yığmaq olar:

```
javascript:5%2
javascript:x = 3; (x<5) ? "x qiyməti kiçikdir": "x qiyməti daha
böyükdür" javascript:d = new Date(); typeof d; javascript:for
(i=0, j=1, k=0, fib=1; i<5; i++, fib=j +k, k=j, j=fib)
alert(fib); javascript:s=""; for (i in navigator) s+=i+":"
+navigator[i] +" \n"; alert(s);
```

Firefox təksətirli veb- brauzerində ssenarilər JavaScript-konsollarını ehtiva edir. Bu konsolu Alətlər menyusundan işə salmaq olar. Burada sadəcə yoxlamaq istədiyiniz ifadəni və ya təlimatı daxil etmək lazımdır. JavaScript- konsoldan istifadə zamanı psevdoprotokol spesifikatorunu (javascript:) işə salmaq olar.

JavaScript- kodun öyrənilməsinin baza metodikası, başqa dillərin metodikası ilə uyğun gəlir.

Əgər siz JavaScript- ssenarilərində tez-tez xətalarla rastlaşırsınızsa, ehtimal ki, JavaScript-in cari sazlayıcısı ilə maraqlanacaqsınız. Internet Explorer-də Microsoft Script Debugger, Firefox-da Venkman adlı məlum genişlənmənin modul sazlayıcısından istifadə etmək olar. Bu alətlərin təsviri kitabın əhatə mövzundan kənar olmasına baxmayaraq, siz asanlıqla İnternetdə, hər hansı bir axtarış sistemindən istifadə edib bu barədə məlumat toplaya bilərsiniz. Daha bir alət jslint-dir. Ciddi desək bu sazlayıcı deyil; Bu alət JavaScript-proqram kodunda olan xətaları axtarmağa yönəlmişdir (***http://jslint.com***).

2

Leksik struktur

Proqramlaşdırma dilinin leksik strukturu – proqramların yazılma qaydalarını müəyyən elementarların dəstidir. Dilin aşağı səviyyəli sintaksisi; dəyişənlərin adları, şərhlər üçün istifadə edilən simvollar, bir təlimatı digərindən ayırmaq və s-dir. Bu qısa fəsil JavaScript-in leksik strukturunu sənədlərlə əsaslandırır.

2.1. Simvol yığımı

JavaScript-də proqramların yazılışı zamanı Unicode simvol yığımından istifadə olunur. Yalnız ingilis dili üçün ASCII-yə və ISO Latin-1-ə yaxın kodlaşdırmadan və əsas qərbi avropa dillərinin 8-dərəcəli kodlaşdırmasının 7- dərəcəli kodlaşdırmasından fərqli olaraq, Unicode-un 16-dərəcəli kodlaşdırması praktik olaraq istənilən yazı dilinin tətbiqini təmin edir. Bu imkan beynəlmiləlləşdirmə üçün və xüsusilə "ingilis olmayan" proqramçılar üçün əhəmiyyətlidir. Amerikanlar və digər ingilis dilində danışan proqramçılar proqramları, adətən yalnız ASCII və ya Latin-1 kodlaşdırmasını dəstəkləyən mətn redaktorunun köməyi ilə yazırlar və buna görə onlarda Unicode tam simvolların dəsti əlçatan deyil. Ancaq əks tərəfin bu barədə heç bir çətinliyi, bir halda ki, çünki ASCII və Latin-1 kodlaşdırmaları Unicode kodlaşdırmasının alt çoxluqlarını təşkil edir və hər hansı bir simvol dəstinin köməyi ilə yazılmış JavaScript-proqramı tamamilə düzgündür. ECMAScript v3 standartı JavaScript-proqramlarında istənilən yerdə Unicode-simvolların mövcudluğuna imkan edir.

2.2. Registrə həssaslıq

JavaScript – registrə həssas dildir. Bu isə o deməkdir ki, əsas sözlər, dəyişənlər, funksiya adları və dilin ixtiyari bir identifikatoru həmişə böyük və kiçik hərflərin eyni cür yığımını ehtiva etməlidir. Məsələn, while açar sözü "While" və ya "WHILE" kimi deyil "while" kimi yazılmalıdır. Online, Online, OnLine və ONLINE – bu dörd müxtəlifin dəyişən adları da analojidir. Qeyd edək, HTML dili, JavaScript-dən fərqli olaraq, registrə həssas deyil. HTML-in və JavaScript-kliyentinin yaxın əlaqəsini nəzərə alaraq bu fərq qarışıqlığa gətirib çıxara bilər. Nəzərə alın ki, JavaScript-obyektləri və onların xüsusiyyətləri ilə HTML dilinin teq və atributları eynidir. Əgər HTML-dilində bu teqlər və atributlar istənilən registrdə yazıla bildiyi halda, JavaScript-də adətən kiçik hərflərlə yazılmalıdır. Məsələn, HTML-də onclick hadisəsinin emalçısının atributu əksər hallarda onClick kimi, ancaq JavaScriptkodunda (və ya XHTML-sənədində) onclick kimi göstərilməlidir.

2.3. Simvol-ayırıcılar və sətir keçidləri

JavaScript – proqramın daxilində leksemləri arasında boşluqlara, tabulyasiyalara və sətir keçidlərinə məhəlqoymur. Buna görə də boşluq, tabulyasiya və yeni sətir simvolları proqram mətnində xüsusi simvolların köməyilə məhdudiyyətsiz istifadə edilə bilər. Ancaq bu barədə, növbəti bölmədə danışacağıq.

2.4. Vacib olmayan nöqtəli vergüllər

Sadə JavaScript-də təlimatları C, C++, və Java PD-ləri kimi adətən nöqtəli vergül (;) simvolları ilə qurtarır. Nöqtəli vergül təlimatlar bir-birindən ayırmasına xidmət edir. Ancaq JavaScript-də əgər hər təlimat ayrı sətirdə yerləşirsə, nöqtəli vergülü qoymamaq olar. Məsələn, aşağıdakı fraqment nöqtəli vergüllərsiz ola bilər:

```
a = 3;
b = 4;
```

Ancaq əgər hər iki təlimat bir sətirdə yerləşdirilmişsə, onda məcburi olaraq nöqtəli vergül qoyulmalıdır:

```
a = 3; b = 4;
```

Proqramlaşdırmada nöqtəli vergüllərin qoyulma kriteriyalarını düzgün anlamaq çətindir və buna görə istənilən məqamda nöqtəli vergüllərdən istifadə etməyi özünüzə vərdiş edin. Nəzəri alın ki, JavaScript, istənilən iki leksem arasında sətir boşluğu güman edir, amma JavaScript-in sintaktis analizatorunun avtomatik nöqtəli vergülləri qoyma vərdişinin bəzi istisnaları mövcuddur. Əgər proqram kodunun sətirində bitirilmiş təlimatdan sonar yeni sətirə keçilibsə, JavaScript-in sintaktis analizatoru nöqtəli vergülü avtomatik qoyur. Məsələn, aşağıdakı fraqmentə baxaq:

```
return
true;
```

JavaScript-in sintaktis analizatoru güman edir ki, proqramçının yazdığı kod aşağıdakı kimidir:

```
return;
true;
```

Hərçənd ki əslində proqramçı, return true; yazmaq istəyirdi. Gördüyünüz kimi, diqqətli olmaq lazımdır – bu kod sintaktis cəhətdən səhv deyil, amma kodda aşkar olmayan nasazlığı mövcuddur. Oxşar xoşagəlməz hadisə: break outerloop; yazanda da yarana bilər.

JavaScript break açar sözündən sonra nöqtəli vergülü avtomatik qoyur və növbəti sətiri icra edildikdə xəta ilə üzləşir. Analoji səbəblərə görə postfiks operatorlarını (++ və --) (*5-ci fəsilə* baxmaq), aid olduğu ifadənin sətirində yerləşdirmək məcburidir.

2.5. Şərhlər

JavaScript, həmçinin Java və C++, C stilində olan şərhləri dəstəkləyir. Sətir sonunda // simvolları arasında daxil edilən istənilən mətnə, şərh kimi baxılır. Şərhlərin icraedici funksiyası yoxdur. Həmçinin / * və */ simvolları arasında daxil edilən istənilən mətnə şərh kimi baxılır. C stilində sözügedən şərhlər bir neçə sətirdən ibarət ola bilər və qoyulmuş ola bilmir. Aşağıdakı kod sətirlərində düzgün JavaScript-şərhləri göstərilmişdir:

```
// Bu təksətirli şərhdir.
/* Bu da həmçinin şərhdir */ // və bu başqa cür şərh formasıdır.
/*
*Bu da daha bir şərh formasıdır.
*Bu formada olan şərhlər bir neçə sətirdə yerləşdirilə bilər.
*/
```

2.6. Literallar

Literal – proqramın mətnində bilavasitə göstərilmiş qiymətdir. Aşağıda bəzi literal nümunələri göstərilmişdir:

```
12              // On iki ədədi
1.2             // Bir tam onda iki onluq ədədi
"hello world"       // Mətn sətiri
'Hi'                // Başqa cür sətir true
true               // Məntiqi qiymət
false               // Digər bir məntiqi qiymət
/javascript/gi       // Requlyar ifadə (şablon üzrə axtarış üçün)
null                // Obyektin yoxluğu
```

ECMAScript v3-də ifadələrdə həmçinin massiv literalları və obyekt literalları dəstəklənir. Məsələn:

```
{ x:1, y:2}            // Obyektin inisializatoru
[1,2,3,4,5]            // Massivin inisializatoru
```

Literallar – istənilən proqramlaşdırma dilinin mühüm hissəsidir, çünki, literalsız proqram yazmaq mümkün deyil. JavaScript-də olan müxtəlif literallar 3-cü fəsildə təsvir edilmişdir.

2.7. İdentifikatorlar

İdentifikator – sadəcə verilən addır. JavaScript-də identifikatorlar, dəyişənlərin və funksiyaların adları kimi, həmçinin bəzi dövrlərin nişanları kimi çıxış edir. Mümkün identifikatorların formalaşması qaydaları Java və bir çox başqa proqramlaşdırma dillərinin qaydaları ilə eynidir. Birinci simvol hərf, sətir xətti_(_) simvolu və ya dollar ($) işarəsi[1] olmalıdır. Birinci simvoldan sonra istənilən hərf, rəqəm, sətir xətti simvolu (_) və ya dollar işarəsi ola bilər. (Birinci simvol heç bir halda rəqəm ola bilməz, çünki

bu halda interpretator ədədləri identifikatorlardan ayırmağa çətinlik çəkir.) Mümkün identifikator nümunələri:

```
i
my_variable_name
v13 _dummy
$str
```

ECMAScript v3-də identifikatorlar Unicode simvol dəstində olan bütün hərfləri və rəqəmləri ehtiva edə bilər. Standartın bu versiyasına qədər JavaScript- identifikatorları ASCII dəsti ilə məhdudlaşmışdı. Bu işarə yalnız kodun generasiyası vasitələri üçün nəzərdə tutulmuşdur, buna görə də identifikatorlarda bu işarənin istifadəsindən çəkinmək lazımdır. 2.8. Ehtiyata saxlanılan sözlər Unicode escape- ardıcıllıqları – identifaktorlarda 16 dərəcəli 4 onaltılıq rəqəmdən ibarət simvol kodu olan \u simvol birləşməsi vasitəsilə icra edilir. Məsələn, π identifikatoru üçün \u03c0 kimi yazmaq olar. Bu sintaksis bu cür narahat görünə bilər, amma mətnlər ilə iş zamanı tam Unicode simvol dəstini dəstəkləməyən redaktorlarda və ya digər vasitələrdə bu Unicodesimvollarının JavaScript- proqramların transliterasiyasını təmin edir.
Nəhayət, identifikatorlar ilə JavaScript-də digər məqsədlər üçün istifadə edilən açar sözləri birbirilə uyğun gəlmir. Aşağıdakı bölmədə JavaScript-in xüsusi ehtiyacları üçün ehtiyata saxlanmış açar sözlər göstərilmişdir.

2.8. Ehtiyata saxlanılan sözlər

JavaScript-də bir neçə ehtiyata saxlanılan söz mövcuddur. Bu sözlər JavaScript-proqramlarında identifikator kimi (dəyişən, funksiya və dövrlərin nişan adları kimi) çıxış edə bilməz. Cədvəl 2.1-də ECMAScript v3-də standartlaşdırılmış açar sözləri göstərilmişdir. JavaScript interpretatoruna görə bu sözlər xüsusi qiymətə malikdir və dilin sintaksis hissəsini təşkil edir.

Cədvəl 2.1. *JavaScript-in ehtiyata saxlanılan açar sözləri*

break	*do*	*if*	*switch*	*typeof*
case	*else*	*in*	*this*	*var*
catch	*false*	*instanceof*	*throw*	*void*
continue	*finally*	*new*	*true*	*while*
default	*for*	*null*	*try*	*with*
delete	*function*	*return*		

Cədvəl. 2.2 başqa açar sözləri göstərilmişdir. Hal-hazırda bu sözlər JavaScript-də istifadə olunmur, amma dilin gələcək inkişaf etdirmələri üçün ECMAScript v3-də ehtiyatda saxlanmışdır.

Cədvəl2.2. *ECMA genişlənilmələri üçün ehtiyata saxlanmış sözlər*

Abstract	*double*	*goto*	*native*	*static*
Boolean	*enum*	*implements*	*package*	*super*
byte	*export*	*import*	*private*	*synchronized*
char	*extends*	*int*	*protected*	*throws*
class	*final*	*interface*	*public*	*transient*
const	*float*	*long*	*short*	*volatile*
debugger				

ECMAScript v4 standartının cari layihələri as, is, namespace və use açar sözlərinin tətbiqinə imkan verir. Hərçənd ki, JavaScript cari interpretatorları bu dörd açar sözündən identifikator kimi istifadə edilməsini qadağan etmir, ancaq identifikatorlarda bu sözlərin istifadəsindən çəkinmək lazımdır.

Bundan başqa, JavaScript dilində qabaqcadan müəyyən edilmiş qlobal dəyişən və funksiya identifikatorlarının istifadədən çəkinmək lazımdır. Cədvəl. 2.3-də ECMAScript v 3 standartı ilə müəyyən edilən qlobal dəyişənlər və funksiyalar göstərilmişdir. Konkret reallaşdırmalar öz qlobal görünmə sahəsi vasitəsilə qabaqcadan müəyyən edilmiş elementləri ehtiva edə bilər. Bundan başqa, JavaScript-in konkret platforması üzrə (kliyent, server və başqaları) bu siyahını genişləndirə bilərik.

Cədvəl 2.3. *Çəkinməli olduğumuz digəri dentifikatorlar*

Argument	*sencodeURI*	*Infinity*	*Object*	*String*
Array	*Error*	*isFinite*	*parseFloat*	*SyntaxError*
Boolean	*escape*	*isNaN*	*parseInt*	*TypeError*
Date	*eval*		*Math*	*RangeError*
decodeURI	*undefined*		*EvalError*	*NaN*
ReferenceError	*unescape*			
decodeURIcomponent	*Function*	*Number*	*RegExp*	*URIError*

3

Məlumat tipləri və qiymətlər

Kompüter proqramları *qiymətlərin manipulyasiya* nəticəsində işləyir. Proqramlaşdırma dilində təqdim edilmiş və emal edilə bilən, qiymətlər ***məlumatlar tipindən (data types)*** və proqramlaşdırma dilinin ən fundamental xarakteristikalarından biri olan bu tipi dəstəkləyən məlumat tiplərinin dəstindən ibarətdir. JavaScript üç elementar məlumat tipi ilə işləməyə imkan verir:

- *ədədlər*
- *mətn sətirləri (və ya sadəcə sətirlər)*
- *məntiqi doğruluq qiymətləri (və ya sadəcə məntiqi qiymətlər)*

JavaScript-də həmçinin iki bayağı məlumat tipi təyin edilir: `null` və `undefined`.
Bu qiymətlərin hər biri yalnız bir qiymətə malikdir. JavaScript bu elementar məlumat tiplərindən savayı ***obyekt (object)*** kimi bilinən tərkib məlumat tipini də dəstəkləyir.
Obyekt (yəni, obyekt məlumat tipinin üzvü) qiymətlərin (ədədlər sətirlər və ya elementlər kimi və ya daha mürəkkəb, məsələn başqa obyektlər) kolleksiyasını təşkil edir. Obyektlər JavaScript-də ikili təbiətə malikdir: *obyekt* adlandırılmış qiymətlərin qaydaya salınmamış kolleksiyası kimi və ya nömrələnmiş qiymətlərin qaydaya salınmış kolleksiyası kimi.
Obyektin sonuncu vəziyyətində obyekt ***massiv (array)*** adlanır. Hərçənd ki, JavaScript-də obyektlər və massivlər ayrı-ayrı məlumat tipidir və bu kitabda ayrı tiplər kimi baxılır. JavaScript-də obyektin daha bir xüsusi tipi müəyyən edən ***funksiyadır (****Function****).***
Funksiya – icra edilən kodun bağlandığı obyektdir. *Funksiya (invoked)*, müəyyən əməliyyatın icra edilməsi üçün çağırıla bilər. Massivlər kimi, funksiyalar da, digər obyekt növlərindən fərqlər və JavaScript-də funksiyalar ilə işləmək üçün xüsusi sintaksis müəyyən edilmişdir. Buna görə biz obyektlərdən və massivlərdən asılı olmayaraq funksiyalarla tanış olacağıq. JavaScript-in baza dilində funksiyalardan və massivlərdən başqa obyektlərin bir qədər xüsusi növləri də vardır. Bu obyektlər yeni olmayan məlumat tiplərini və yalnız obyektlərin yeni siniflərini (classes) dəstəkləyir.
Fəsil başlanğıc səviyyə üçün bir qədər dar ixtisaslaşdırılmış təfərrüatları özündə ehtiva edir.

3.1. Ədədlər

Ədədlər – xüsusi izah tələb etməyən əsas məlumat tipidir. JavaScript , C və Java kimi proqramlaşdırma dillərindən fərqli olaraq, tam və həqiqi ədədlər arasında fərq qoymur. JavaScript-də bütün ədədlər 64-dərəcəli həqiqi ədədlərlə (üzən nöqtə (kəsr ədədlər) daxil olmaqla) təqdim edilirlər ki, belə format IEEE 754.1[2] standartına əsasən aparılır. Bu format vasitəsilə ±1,7976931348623157 × 10308 dən ±5 × 10-324 qədər ədədlərdən istifadə etmək olar. Bilavasitə JavaScript- proqramının kodunda olan ədəd, *ədəd literalı* adlanır. JavaScript

bir neçə ədəd literalını dəstəkləyir. Bu haqda sonrakı bölmələrdə bəhs olun. Diqqətli olun: istənilən mənfi ədəd literalı əvvəlində "mənfi" işarəsi qoyulur. Ancaq faktiki olaraq mənfi (minus) işarəsinin dəyişməsi ədəd literallarının sintaksis hissəsi olmayan unar operatorunu təşkil edir (5-ci fəsilə baxmaq) .

3.1.1. Tam literallar

JavaScript-də tam onluq ədədlər rəqəm ardıcıllığı ilə yazılır. Məsələn:

```
0
3
10000000
```

JavaScript-in ədəd formatı bu diapazonda olan bütün tam ədədləri dəqiq təqdim etməyə imkan verir: 9007199254740992 (– 253) 9007199254740992 (253) qədər daxil olaraq. Bu diapazondan kənar tam qiymətlərin kiçik dərəcədə dəqiqliyi itə bilər. Qeyd etmək lazımdır ki, JavaScript-də (xüsusən bit operatorları, 5-ci fəsildə təsvir edilmişlər) bəzi tam əməliyyatlar qiymətləri (mənaları) qəbul edən 32-dərəcəli – 2147483648 (– 2^{31}) 2147483647-ə qədər (2^{31} – 1) ədədlər ilə yerinə yetirilirlər.

3.1.2. Onaltılıq və səkkizlik

JavaScript-in onluq tam literallarında başqa onaltılıq say sistemində olan qiymətləri də tanıyır. Onaltılıq ədəd 0x və 0X ardıcıllığı ilə başlayır və həm rəqəm həm də bəzi hərflər vasitəsilə yazılır. Onaltılıq rəqəm – 0- dan 9-a qədər və ya a-dan (və ya A) *f*-ə qədər (və ya *F*) qədər hərflərdən ibarət rəqəmdir. Onaltılıq sistemin tam ardıcıllığı (0-*F*) onluq say sisteminin 0-15 ədədlərinə uyğundur. Aşağıda onaltılıq tam ədəd literallarının nümunələri göstərilib:

```
0xff             // 15*16 + 15 = 255 (10 əsaslı)
0xCAFE911
```

Hərçənd ECMAScript standartı səkkizlik say sistemində tam ədəd literallarını dəstəkləmir, yalnız JavaScript-in bəzi reallaşdırmaları bu tip məsələləri həll etməyə icazə verir. Səkkizlik literallar 0-7 aralığında olur və digər ədədlər bu rəqəmlərin vasitəsilə düzəlir. Məsələn:

```
0377             // 3*64 + 7*8 + 7 = 255 (10)
```

çünki, bəzi reallaşdırmalar səkkizlik say sisteminin literallarını dəstəklədiyi halda, bəziləri dəstəkləmir. Heç vaxt aparıcı tək sıfır rəqəminin tam literalının hansı say sistemində səkkizüzlü say kimi və ya onluq) olduğunu bilmək olmur.

3.1.3. Həqiqi ədəd literalları

Həqiqi ədəd literalları onluq say sisteminə malik olmalıdır; bu literallar həqiqi ədədlərin ənənəvi sintaksisindən ibarətdir.
Tam qiymət (tam ədədin, onluq kəsrini və ya qalıqlı ədədin tam hissəsi müəyyən edir. Həqiqi ədədlərin literalları həmçinin səciyyəvi nəsihətdə təqdim edilə bilər: həqiqi ədəd,

e ədədini (və ya E), plyus və ya minus və tam eksponenti dəstəkləyir. Bu nəsihət 10-da qiymətlə (mənayla) müəyyən edilən dərəcələrə vurulmuş (artırılmış) həqiqi ədədi ifadə edir eksponentlər. Belə sintaksisin daha yığcam təyini:

[rəqəmlər] [.rəqəmlər] [(E|e) [(+|-)] rəqəmlər]
Məsələn:

```
3.14
2345.789
.33333333333333333
6.02e23             // 6.02 X 10^23
1.4738223E-32       // 1.4738223 X 10^-32
```

Diqqətli olun: həqiqi ədədlər sonsuzdur, amma JavaScript-də həqiqi ədədlər məhdudlaşdırılmışdır (daha dəqiq 18437736874454810627) və yalnız məhdudlaşdırılmış həqiqi ədədlər dəqiq ifadə edilir. Deməli, JavaScript-də həqiqi ədədlərlə işləyərkən ədədin təqdim edilməsi zamanı həqiqi ədədləri yuvarlaqlaşdırılması ola bilər. Yuvarlaqlaşdırma dəqiqliyi, kifayət qədərdir və təcrübələrdə nadir hallarda xətalara gətirib çıxarır.

3.1.4. Ədədlərlə iş

JavaScript-dilində proqramlarda ədədlərlə işləmək üçün hesab operatorlarından istifadə edilir. Əsas hesab operatorları, toplama, çıxma, vurma, bölmə operatorlarından ibarətdir. Bu və digər hesab operatorlarının ətraflı təsviri ***5-ci fəsildə*** verilib. JavaScript-in göstərilmiş əsas hesab operatorlarından başqa daha mürəkkəb riyazi əməliyyatların icra edilməsinə kömək edən və dilin baza hissəsinə aid olan böyük miqdarda riyazi funksiya aiddir.
Rahatlıq üçün bu funksiyalar Math obyektinin xüsusiyyətləri şəklində saxlanılır və funksiyalara giriş üçün həmişə Math hərfi adından istifadə olunur. Məsələn, sinus x dəyişəninin ədədi qiymətini aşağıdakı qaydada hesablamaq olar:

```
sin_of_x = Math.sin(x);
```

Ədədin kvadrat kökü isə belə hesablanır:

```
hypot = Math.sqrt(x*x + y*y);
```

JavaScript-i dəstəklənən bütün riyazi funksiyalar haqqında ətraflı məlumatlar Math və kitabın üçüncü hissəsinin uyğun olan listinqləri obyektinin təsvirində gətirib çıxarılmışdır

3.1.5. Ədəd dəyişiklikləri

JavaScript dilində ədədləri sətirlər şəklində təqdim etmək və ədədləri sətirlərə dəyişdirmək olar. Bu dəyişikliklərin sırası 3.2 paraqrafında təsvir edilir.

3.1.6. Xüsusi ədəd qiymətləri

JavaScript-də bir neçə xüsusi ədəd qiyməti müəyyən edilmişdir. Məsələn, təsəvvür edilən ən böyük həqiqi ədəd, qiymət diapazonunu aşır və nəticəyə JavaScript-də `Infinity` kimi

bilinən xüsusi sonsuzluq qiyməti mənimsənilir. Müsbət və mənfi sonsuzluq müvafiq olaraq `Infinity` və -in`finity` kimi təqdim edilir.
JavaScriptin aldığı daha bir xüsusi ədəd qiyməti `NaN` qiymətidir. Bu qiymət riyazi əməliyyat qeyri-müəyyən nəticə aldıqda və xətalı olduqda (məsələn, sıfırın sıfıra bölünməsi, tangens 90 dərəcənin qiyməti) nəticəyə mənimsədilir. `NaN` qiyməti *"ədəd-deyil"* xüsusi qiymətinin nəticədir. *"Ədəd-deyil"* (*Not-a-Number*) qeyri-adi xüsusiyyətlərə malikdir: o heç bir ədədə hətta özünə belə bərabər deyil! Bu qiymətın yoxlanılması xüsusi `isNaN()` funksiyası mövcuddur. Oxşar funksiya olan, `isFinite()`, ədədi, `NaN` və ya müsbət/mənfi sonsuzluq bərabərsizliyində yoxlamağa imkan verir. Cədvəl. 3.1 JavaScript-də xüsusi ədəd qiymətləri üçün müəyyən edilmiş bir neçə işarə göstərilmişdir.

Cədvəl 3.1. Xüsusi ədəd sabitləri

Sabit	İzahı
`Infinity`	Sonsuzluğu ifadə edən xüsusi qiymət
`NaN`	"Ədəd-deyil" xüsusi qiyməti
`Number.MAX_VALUE`	Təsəvvür edilən maksimal qiymət
`Number.MIN_VALUE`	Təsəvvür edilən minimal qiymət
`Number.NaN`	"Ədəd-deyil" xüsusi qiyməti
`Number.POSITIVE_INFINITY`	Müsbət sonsuzluğu ifadə edən xüsusi qiymət
`Number.NEGATIVE_INFINITY`	Mənfi sonsuzluğu ifadə edən xüsusi qiymət

ECMAScript v1-də müəyyən edilmiş JavaScript 1.3 versiyasına qədər Infinity və NaN sabitləri yoxdur. Ancaq Number tipli müxtəlif sabitləri JavaScript 1.1-dən başlayaraq mövcuddur.

3.2. Sətirlər

Sətir, JavaScript-də hərf, rəqəm, punktuasiya nişanları və digər Unicode- simvollar ardıcıllıqlarını və həmçinin mətnin daxiledilməsinə imkan verən məlumat tipidir. Siz növbəti dərslərdə görəcəksiniz ki, sətir literalları cüt və ya tək dırnaq (apostrof) simvolları arasında olur. Diqqətli olun: C, C++ və Java-da simvolik məlumat tipi kimi bilinən char tipi JavaScript-də yoxdur. Tək simvol tək uzunluğa malik sətir ilə təqdim edilə bilər.

3.2.1. Sətir literalları

Sətir literalı – tək və ya ikiqat dırnaqlarla (' və ya ") əhatə olunmuş Unicode- simvolları ardıcıllığıdır. Əgər mətndə ikiqat dırnaqlardan istifadə etmək lazımdırsa literalı tək

dırnaqlarla (') əhatələmək (içərisində yazmaq) lazımdır. Eyni qaydanın əksi olaraq, əgər mətndə tək dırnaqlardan (apostrof) istifadə etmək lazımdırsa literalı cüt dırnaqlarla (") əhatələmək (içərisində yazmaq) lazımdır. Sətir literalı proqramın bir sətirində yazılmalıdır və ikinci sətirə keçirilmir. Sətir literalında yeni sətir daxil etmək üçün \n simvol ardıcıllığından istifadə etmək lazımdır. Sətir literallarına aid nümunələr:

```
"" // Boş sətir: simvolların sayı sıfır
'testing'
"3.14"
'name=" myform"'
"Siz O'Reilly nəşriyyatının kitablarına üstünlük verirsiniz, doğru deyilmi?"
"Bu sətir literalı 2 cərgə\nehtiva edir"
"π – çevrənin uzunluğun onun diametrinə olan nisbətidir"
```

Sonuncu sətir nümunəsində göründüyü kimi ECMAScript v1 standartı sətir literallarında Unicodesimvollarından istifadə etməyə imkan verir. Ancaq bu JavaScript 1.3-dən erkən versiyalarda dəstəklənmir və bu versiyalarda sətirlər adətən yalnız ASCII və ya Latin-1 dəstində olan simvolları dəstəkləyir.
Nəzərə alın ki, tək dırnaq ilə məhdudlaşdırmiş sətir ilə ehtiyatlı davranmaq lazımdır. Kiçik bir yiyəlik halı şəkilçisi (can’t, O’Reilly kimi) olan apostrofun düzgün daxil edilməməsi nəticəsində xəta baş verə bilər. Çünki apostrof ilə tək dırnaq işarəsi eyni bir simvoldur və istisna halı metodu ilə əks sleş simvolu vasitəsilə istifadə edilməlidir.
JavaScript kliyentində proqramlar adətən HTML-koddan ibarət ola bilər və öz növbəsində HTML-kodda, JavaScript-kodunun sətirlərini ehtiva edə bilər. Buna görə də JavaScript-i HTML teqlərə və hadisələrə tətbiq edərkən JavaScript kodu dırnaq içərisində yazılır. Aşağıdakı nümunədə JavaScript-ifadəsi kimi "Təşəkkür" sətiri tək dırnaqlarla əhatələnmişdir. Kodun özü isə HTML atributunun hadisə emalçısına mənimsədildiyinə görə cüt dırnaqlar ilə əhatələnmişdir:

```
<a href="" onclick="alert ('Təşəkkürlər')">Məni klikləyin!</a>
```

3.2.2. Sətir literallarında idarəedici ardıcıllıqlar

Əks sleş (\) simvolu JavaScript-sətirlərində xüsusi təyinata malikdir. Onu yanında gələn simvollarla birlikdə, müvafiq simvol ifadə edir. Adətən sətirin daxilində yerləşdirilməsi mümkün olmayan simvolların sətirə mənimsədilməsi üçün istifadə edilir. Məsələn, \n – yeni sətirə keçmək üçün istifadə olunur *(escape sequence*).
Əvvəlki bölmədə qeyd olunmuş başqa nümunə – tək dırnaq simvolunu ifadə edən \' simvol birləşməsidir. Bu idarəedici sətir ardıcıllığı tək dırnaq ilə əhatələnmiş sətir sahəsinə tək dırnağın simvolunun əlavə edilməsi üçündür. İndi bizə aydın olur ki, niyə bu ardıcıllıqları idarə edici adlandırırıq. Burada əks sleş simvolu tək dırnaq simvolunun interpretasiyasını idarə etməyə imkan verir. Aşağıdakı nümunədə biz tək dırnaq işarəsinin əks sləş simvolu ilə tətbiqi nəticəsində biz bu işarəni sətir sonluğu kimi deyil, apostrof kimi veririk:

```
'You\'re right, it can\'t be a quote'
```

Cədvəl. 3.2-də idarəedici ardıcıllıqlar və onların tərəfindən ifadə edilən simvollar göstərilmişdir. İki idarə edici ardıcıllıq ümumiləşdirilmişdir; onlar əks sləşin yanında onaltılıq say sistemində olan kodu göstərməklə Latin-1 və ya Unicode simvol dəstinin tərkibində olan

istənilən simvolun göstərilməsi üçün tətbiq edilə bilər. Məsələn, \xA9 ardıcıllığı müəllif hüquqları simvolunu ifadə edir. Bu simvol, Latin-1 kodlaşdırmasında A9 onaltılıq koduna malikdir. \u simvolları da analoji xüsusiyyətə malikdir və dörd onaltılıq rəqəmlə göstərilmiş sərbəst Unicode simvolunu ifadə edir. Məsələn, \u03c0 simvolu π ifadə edir. Qeyd etmək lazımdır ki, idarəedici ardıcıllıqlarında ECMAScript v1 standartı üzrə Unicode-simvol dəsti tələb olunur. JavaScript 1.3-dən daha əvvəl çıxmış versiyalarda bu xüsusiyyət dəstəklənmirdi.

JavaScript-in bəzi realizasiyaları həmçinin Latin-1 simvolunun tapşırığını əks sleş simvolundan sonra göstərilmiş üç səkkizlik simvolla tətbiq edilə bilər, amma belə idarəedici ardıcıllıqlar ECMAScript v3 standartında dəstəklənmir və istifadə olunmamalıdır.

Cədvəl 3.2. *JavaScript-də idarəedici ardıcıllıqlar*

Sabit	İzahı
\0	NUL simvolu
\b	“Bəkspeys” simvolu
\t	Üfüqi tabulyasiya (\u0009)Funksiya, arqumentlər
\n	Yeni sətirə keçid (\u000A)Konstruktor çağırışı
\v	Şaquli tabulyasiya (\u000B)
\f	Yeni səhifəyə keçid (\u000C)
\r	Karetin qaytarılması (\u000D)
\"	İkiqat dırnaq (\u0022)
\'	Tək dırnaq (\u0027)
\\	Əks sleş (\u005C)
\xXX	Latin-1 simvol dəstinə aid edilən, XX, onaltılıq say sistemində verilən iki ədəddir
\uxXXXX	Unicode simvol dəstinə aid edilən, onaltılıq say sistemində verilən dörd XXXX ədədi Latin-1 simvol dəstinə aid edilən, XXX, səkkizlik say sistemində verilən üç ədəddir.
\XXX	1-dən 377-ə qədər kod diapozununa malikdir. ECMAScript v3 dəstəklənmir; burada belə yazı

Əgər "\" simvolu ***cədvəl 3.2***-də olan sabitlər ilə təqdim edilmədikdə, əks sləş simvoluna

üsulu istifadə olunmamalıdır. önəm verilmi

r. Məsələn, \# – #-in özü alınır.

3.2.3. Sətirlərlə iş

JavaScript-in inteqrasiya edilmiş imkanlarından biri də sətirləri birləşdirməkdir. Əgər + operatoru ədədlərə tətbiq edilirsə, bu ədədlər toplanır, əgər sətirlərə tətbiq edilirsə, bu sətirlər bitişdirilir, bu halda ikinci birinci sətirin sonuna əlavə edilir. Məsələn:

```
msg = "Hello, "+ "world";   // Alınır: "Hello, world"
greeting = "Hörmətli, "+name+". Mənim ana səhifəmə xoş gəlmisiniz";
```

Sətir uzunluğunun (sətirdə olan simvolların miqdarı) təyini üçün length xüsusiyyətindən istifadə olunur. Məsələn, əgər s dəyişəni sətiri tipindədirsə, bu sətirin uzunluğunu aşağıdakı qaydada göstərmək olar:

```
s.length
```

Sətirlərlə işləmək üçün bir neçə metod mövcuddur. s sətirinin son simvolu belə göstərmək olar:

```
last_char = s.charAt(s.length – 1)
```

İkinci simvolu çıxartmaq üçün, s sətirinin üçüncü və dördüncü simvolları, təlimata tətbiq edilir:

```
sub = s.substring(1,4);
```

S sətirində birinci simvol olan "a" hərfinin mövqeyini aşağıdakı qaydada müəyyən etmək olar:

```
i = s.indexOf('a');
```

Sətirlərlə işləyən digər metodlar da vardır. Bu metodların təffərrüatı String obyektinin təsvirində və kitabın üçüncü hissəsinin listinqlərində sənədlərlə əsaslandırılmışdır.
Əvvəlki nümunələrdən anlamaq olar ki, JavaScript-sətirləri (və gələcəkdə tanış olacağımız JavaScript massivləri) 0-dan başlayaraq indeksləşdirilir. Başqa sözlə, sətirin birinci simvolunun sıra nömrəsi sıfıra bərabərdir. C, C++ və Java-da işləmiş proqramçılara artıq bilirlər ki, bu dillərdə sətirlərin və massivlərin numerasiyası vahiddən başlanır.
JavaScript-in bəzi realizasiyalarında sətirlər, massivlər kimi ayrı-ayrı simvollar şəklində götürülə bilər (amma bu sətirlərin yazılması prosesinə şamil edilmir) və buna görə charAt() metodunun çağırışı aşağıdakı qaydada yazıla bilər:

```
last_char = s [s.length – 1];
```

Ancaq bu sintaksis ECMAScript v3-də standartlaşdırılmamışdır, daşınan deyil və ondan çəkinmək lazımdır. Biz obyekt məlumat tipi müzakirə edəndə, xüsusiyyətin əvvəlki nümunələri və sətirlər metodları obyektlər metodları kimi istifadə olunur. Bu o demək deyil ki, sətirlər – obyektlərin tipidir. Əslində sətirlər JavaScript-in ayrı bir məlumat tipidir.

Onların xüsusiyyətlərinə və metodlarına giriş üçün obyekt sintaksisi istifadə olunur, amma sətir özü heç vaxt obyekt olmur! Bunun niyə belə olduğunu, biz bu fəsilin sonunda biləcəyik.

3.2.5. Ədəd-sətir dəyişikliyi

Sətir ədəd kontekstində istifadə olunduqda, avtomatik olaraq ədədə dəyişdiriləcək. Məsələn, aşağıdakı ifadə tamamilə mümkündür:

```
var product = "21" * "2";       // nəticədə 42 ədədi alınacaq.
```

Bu şəraitin əksi zəruri olduqda ədədi sətiri dəyişdirmək olar; bunun üçün kifayət qədər sadə metod, sətirdən 0 qiymətini çıxmaq lazımdır:

```
var number = string_value - 0;
```

(Diqqətli olun: bu vəziyyətdə toplama əməliyyatı sətirlərin bitişdirilməsi əməliyyatı kimi yerinə yetiriləcək.)

Ədəd sətir dəyişikliyinin daha az inkişaf etmiş və daha düzxətli üsulu Number() funksiya konstruktoruna müraciət ilə icra edilir:

```
var number = Number(string_value);
```

Ədəd-sətir dəyişikliyinin belə üsulunun çatışmazlığı ondan ibarətdir ki, bu qədər sadə əməliyyatı həddən artıq ciddi üsulla yerinə yetirir. Bu üsul yalnız onluq say sistemində oluna bilər və bu üsul yalnız rəqəm mövcudluğunu güman edərək, boşluq simvolları, sətirdə ədəddən sonra gələn başqa qeyri-rəqəmsal simvolların yaranmasına icazə verir.
Dəyişikliyin daha elastik üsulu parseInt () və parseFloat() funksiyalarının köməyilə təmin olunur. Bu funksiyalar istənilən qeyri-rəqəmsal simvollara məhəl qoymadan sətirin başlanğıcında duran sərbəst ədədləri dəyişdirəcək və ədədin ardınca yerləşdirilmiş simvollar qaytarılacaq. ParseInt() funksiyası yalnız tam ədəd dəyişikliyini yerinə yetirir. parseFloat() funksiyası isə həm tam, həm də həqiqi ədədlər ilə ədəd dəyişikliyini yerinə yetirə bilər. Əgər sətir "0x" və ya "0x" simvollarından başlayırsa, parseInt() funksiyası sətiri onaltılıq ədəd kimi göstərir.[3] Məsələn:

```
parseInt("3 dovşan");            // 3 qiymətini alır
parseFloat("3.14 metr") ;         // 3.14 qiymətini alır
parseInt("12.34");                  // 12 qiymətini alır parseInt
("0xFF");                           // 255 qiymətini alır
```

İkinci arqument kimi parseInt() funksiyası hesablama sistemlərini də qəbul edə bilər. 2-dən 36-a qədər olan ədəd diapozunda düzgün alınır, məsələn:

```
parseInt("11", 2);        // 3 (1*2 + 1) qiymətini alır
parseInt("ff", 16);           // 255 qiymətini alır (15*16 + 15)
parseInt("zz", 36);           // 1295 qiymətini alır (35*36 + 35)
parseInt("077", 8);           // 63 (7*8 + 7) Qaytaracaq
parseInt("077", 10);          // 77 (7*10 + 7) Qaytaracaq
```

Əgər parseInt() və parseFloat() metodlarında dəyişliyi yerinə yetirmək mümkün deyilsə, onlar NaN qiymətini alır:

```
parseInt("eleven");       // NaN qiymətini alacaq.
parseFloat("$72.47");       // NaN qiymətini
```

3.3. Məntiqi qiymətlər

Ədəd və sətir məlumat tipləri böyük və ya sonsuz mümkün qiymətlər miqdarına malikdir. Məntiqi məlumat tipi isə, əksinə, yalnız iki *true* və *false* literalları ilə təqdim edilmiş mümkün məntiqi qiymətlərdən ibarətdir. Məntiqi qiymət, doğruluq ətrafında icra edilir. JavaScript-proqramlarda yerinə yetirilən müqayisələrin nəticələri adətən məntiqi qiymətlərdir. Məsələn:

```
a == 4
```

Bu ifadə a dəyişəninin 4 ədədinə bərabər olmasını yoxluyur. Əgər a dəyişəni həqiqətən də dördə bərabərdirsə, *true* məntiqi qiymətinin şərti ödənilir. Əgər a dəyişəni dördə bərabər deyilsə, müqayisənin nəticəsi *false* olacaq.
JavaScript-də məntiqi qiymətlər adətən idarəedici konstruksiyalarda istifadə olunur. Məsələn, JavaScript-də `if/else` təlimatı hər hansı bir arqumenti yerinə yetirir, əgər məntiqi qiymət *true*-a bərabər olarsa, əməliyyat yerinə yetirilir, əks halda *false* qiyməti alınır. Adətən məntiqi qiyməti yaradan müqayisə bilavasitə istifadə olunan təlimatla birləşir. Bu sözü kodla ifadə edək:

```
if(a == 4) b = b + 1;
else a = a + 1;
```

Burada yoxlama yerinə yetirilir, əgər a dəyişəni 4 ədədinə bərabərdirsə b dəyişəninin qiymətinə bir vahid əlavə edilir; əks təqdirdə a dəyişəninin qiymətinə bir vahid əlavə edilir. İki mümkün *true* və *false* məntiqi qiymətini, bəzən "düzdür" (*true*) və ya "səhvdir" (*false*) və ya "bəli" (*true*) və "xeyr" (*false*) kimi baxılır.

3.3.1. Məntiqi qiymətlərin dəyişikliyi

Məntiqi qiymətlər başqa tiplərin qiymətlərinə asan dəyişdirilir, həm də bu tip dəyişikliklər əksər hallarda avtomatik yerinə yetirilir.[4] Əgər məntiqi qiymət ədəd kontekstində istifadə olunursa onda, *true* qiyməti `1`-ədədinə, *false* qiyməti isə `0` ədədinə dəyişdiriləcək. Əgər məntiqi qiymət sətir kontekstində istifadə olunursa, onda *true* qiyməti "*true*" sətirinə , *false* qiyməti isə "*false*" sətirinə dəyişdiriləcək.
Məntiqi qiymət birdən yuxarı ədəd kimi istifadə olunduqda, *true* qiymətinə dəyişdiriləcək. Əgər qiymətlər `0`-a və ya `NaN`-a bərabərdirsə, *false* məntiqi qiymətinə dəyişdiriləcək. Məntiqi qiymətlərdə sətirlərdə istifadə edildikdə, əgər sətir boş sətir deyilsə, *true* qiymətinə dəyişdiriləcək, əks təqdirdə dəyişiklik nəticəsində *false*

qiyməti alınacaq. *null* və *undefined* xüsusi qiymətləri *false* qiymətinə dəyişdiriləcək və istənilən funksiya, obyekt və ya massivin qiymətləri *null*-dan böyükdür və *true* qiymətinə dəyişdiriləcəklər.
Əgər siz dəyişikliyi açıq- aydın yerinə yetirmək istəyirsinizsə, `Boolean()` funksiyasından istifadə etmək olar:

```
var x_as_boolean = Boolean(x);
```

Açıq dəyişikliyin başqa üsulu ikiqat məntiqi inkar operatorunun istifadəsi ilə mümkündür:

```
var x_as_boolean = !!x;
```

3.4. Funksiyalar

Funksiya – icra edilən kodun fraqmentidir, hansı ki, JavaScript-proqramında və ya JavaScript reallaşdırmasında qabaqcadan müəyyən edilmişdir. Funksiya JavaScript- proqramında yalnız bir dəfə təyin edilir, lakin istənilən icra edilə və ya səbəb çağırıla bilər Funksiyalar arqumentləri, qiyməti və ya qiymətləri müəyyən edən və hesablamaları yerinə yetirməli olan parametrləri ötürə bilir; həmçinin funksiya bu hesablamaların nəticəsini təşkil edən qiyməti qaytara bilər. JavaScript reallaşdırmaları bir çox qabaqcadan müəyyən edilmiş funksiya təklif edir. Bunlara misal olaraq, bucağın sinusunu hesablayan Math.sin() funksiyasını göstərmək olar.

JavaScript, proqramların ehtiva etdiyi şəxsi funksiyaları da müəyyən edə bilər məsələn, belə bir kod:

```
function square(x)        // Funksiya square adlanır. O bir x arqumentini qəbul
                       // edir.
{                           // Burada funksiyanın əsası başlanır.
  return x*x;               // Funksiya öz arqumentini kvadrata yüksəldir və
                       // alınmış qiymətə qayıdır
}
                              // Burada funksiya bitir.
```

Funksiyanı müəyyən etdikdən sonra, funksiya təyin etdiyimiz ad vasitəsilə funksiyanı çağırmaq olar. Əgər funksiyada arqument təyin etmişiksə onda funksiyanın adının qarşısında mötərizə daxilində həmin arqumentə uyğun qiymət daxil edirik, əks halda funksiyanın adının qarşısında boş mötərizələr *qoyulmalıdır*. Bir çox dillərdə, həmçinin Java-da funksiyalar – dilin məlumat tipi qismində deyil, yalnız sintaktis elementləri qismində iştirak edir: buna görə də funksiyaları müəyyən etmək və çağırmaq mümkündür. JavaScript-də funksiyalar bütün həqiqi qiymətlərdən təşkil oluna bilər ki, bu da proqramlaşdırma dilinin elastikliyindən xəbər verir. Daha sadə dildə desək, funksiyalar dəyişənlərdə, massivlərdə və obyektlər saxlanıla, həmçinin arqumentlər kimi başqa funksiyalara ötürülə bilər. Funksiyaların çağırışı və onlardan istifadə etmək üsullarında 8-ci danışılır. Bir halda ki, funksiyalar ədəd və sətirlər kimi qiymətlərdən təşkil olunub, onlara obyekt xüsusiyyətlərinə mənimsədilə bilər. Funksiyaya obyekt xüsusiyyətinə mənimsədildikdə (obyekt məlumat tipi və obyektin xüsusiyyətləri ***3.5 saylı paraqrafda*** təsvir edilmişdir), bu obyekt metodu adlanır. Metodlar –

obyekt yönümlü proqramlaşdırmanın mühüm hissəsidir. Məhz, 7-ci fəsil *OYP*-yə həsr edilmişdir.

3.4.1. Funksional literallar

Əvvəlki bölmədə biz `square()` funksiyasının təyinini gördük. Adətən JavaScript-proqramlarında funksiyaların əksəriyyəti bu sintaksis ilə təsvir edilir. Ancaq ECMAScript v3 standartı funksional literalların təyini üçün sintaksisə (JavaScript 1.2-də və daha gec versiyalarda reallaşdırılmış) malikdir. Funksional literal ***`Function`*** açar sözünün köməyi ilə, funksiyanın adı ilə birlikdə verilir. Funksiyanın adının qarşısında duran mötərizələrin içərisində bu funksiyanın malik olduğu arqumentlərin siyahısı göstərilir. Funksiya fiqurlu mötərizələr ilə başlayıb, fiqurlu mötərizələr ilə bitir. Funksional literal ilə təyin olun funksiyalara ad verilməyə bilər. Ən böyük fərq ondan ibarətdir ki, funksional literallar digər JavaScript- ifadələri kimi yazıla bilər. Yəni, `square()` bu şəkildə funksiyasını təyin etmək vacib deyil:

```
function square(x)
{
  return x*x;
}
```

Bunu literalların köməyi ilə aşağıdakı kimi də etmək olar:

```
var square = function (x)
{
  return x*x;
}
```

Belə, müəyyən edilmiş funksiyalar adətən lyambda-funksiya adlandırılır. Bu, üsul ilk dəfə LISP proqramlaşdırma dilində istifadə edilmişdir.
Elementar səviyyədə siz bu literalların xeyirini görməsənizdə, mürəkkəb ssenarilərdə siz görəcəksiniz ki, onlar kifayət qədər rahat və faydalıdır. Funksiyanın təyininin daha bir üsulu mövcuddur: arqumentlərin siyahısını və funksiyanın əsasını `Function()`-konstrukturunda sətirlər şəklində vermək olar. Məsələn:

```
var square = new Function(" x", "return x*x;");
```

Funksiyaların bu cür təyindən nadir hallarda istifadə olunur. Bu üsulda funksiyanın əsas hissəsini sətir şəklində vermək narahatdır və bu üsul yuxarıda sadalanan oxşar tərzdə müəyyən edilmiş funksiyalara nisbətən daha az effektivdir.

3.5. Obyektlər

Obyekt – adlandırılmış qiymətlər kolleksiyasıdır. Bunlar, adətən obyektin *xüsusiyyəti (properties)* adlandırılır. Obyektin xüsusiyyətinə istinad etmək üçün, `obyektin_adı.xüsusiyyət` şəklində göstərmək lazımdır. Məsələn, əgər image adlı obyekt ***`width`*** və ***`height`*** xüsusiyyətlərinə malikdirsə, biz bu xüsusiyyətlərə aşağıdakı qaydada istinad edə bilərik:

```
image.width
```

```
image.height
```

Obyektlərin xüsusiyyətləri JavaScript-də dəyişənlərin xüsusiyyətinə oxşardır . Obyektlər, massivlər, funksiyalar və başqa obyektlər də daxil olmaqla istənilən məlumat tipi özündə ehtiva edə bilər. Buna görə də bu cür JavaScript-kodu göstərmək olar:

```
document.myform.button
```

Bu fraqment obyektin button xüsusiyyətinə istinad edir, hansı ki, bu xüsusiyyət, document adlı obyektin myform xüsusiyyətində saxlanılır.
Əvvəlki bölmələrdə deyildiyi kimi, obyekt xüsusiyyətində saxlanılan funksiya adətən metod adlanır və xüsusiyyətin adı metodun adı olur. Metodun çağırılması zamanı əvvəlcə funksiyanın göstərişi üçün obyekt ilə birlikdə "nöqtə" operatorundan istifadə olunur və sonra funksi. Məsələn, *`document`* adlı obyektin *`write()`* metodunu belə çağırmaq olar:

```
document.write("bu sadəcə yoxlamadır");
```

JavaScript-də, obyektlər assosiativ massivlər rolunda çıxış edə bilər, yəni sərbəst sətirlərlə sərbəst qiymətləri assosiasiya edə bilər. Assosativ massiv tətbiq edildikdə, başqa sintaksis obyektlə belə işə tələb olunur xüsusiyyətlərə: tələb edilən xüsusiyyətin adını ehtiva edən sətir kvadrat mötərizələrdən ibarət olmalıdır. Bu üsulda, yuxarı qeyd etdiyimiz image obyektinin xüsusiyyətlərinə aşağıdakı kod vasitəsi ilə müraciət etmək olar:

```
image["width"]
image["height"]
```

Assosiativ massivlər – güclü məlumat tiplərinə malikdir; onlar bir sıra proqramlaşdırma texnologiyalarının reallaşdırması zamanı faydalıdır. Assosiativ massivlər kimi obyektlərdən və onların ənənəvi tətbiqindən 7-ci fəsildə bəhs edilir.

3.5.1. Obyektlərin yaradılması

Biz 7-ci fəsildə görəcəyik ki, obyektlər xüsusi funksiya- konstruktorların çağırılması ilə yaradılır. Aşağıdakı nümunələrdə yeni obyektləri yaradılır:

```
var o = new Object();
var now = new Date();
var pattern = new RegExp(" \\sjava\\s", "i");
```

Şəxsi obyekti yaradıb, onun xüsusiyyətlərini istəyə nizamlamaq olar:

```
var point = new Object();
point.x = 2.3;
point.y = - 1.2;
```

3.5.2. Obyekt literalları

JavaScript-də obyektləri yaratmaq və onların xüsusiyyətlərini göstərməyə imkan verən obyekt literal sintaksisi mövcuddur.

Obyekt literalı (həmçinin obyektin inisializatoru adlandılır) fiqurlu mötərizələr daxilində "xüsusiyyət/qiymət" cütlüklərinin vergülləri ilə bölünmüş siyahısını təşkil edir. Cütlüklərin daxilində iki nöqtə ayırıcı rola malikdir. Beləliklə, əvvəlki nümunədə yaradılmış point obyekti, aşağıdakı sətirlə inisializasiya edilmiş ola bilər:

```
var point ={ x:2.3, y: -1.2 };
```

Obyekt literalı əvvəlcədən qoyulmuş ola bilər. Məsələn:

```
var rectangle ={
    upperLeft: { x: 2, y: 2},
    lowerRight:{ x: 4, y: 4}
    };
```

Nəhayət, mütləq obyekt literalların xüsusiyyət qiymətlərinin sabit olması məcburi deyil – bu sərbəst JavaScript- ifadələri də ola bilərlər. Bundan başqa obyekt literalının xüsusiyyət adlarını sətir qiymətləri ilə də ifadə etmək olar:

```
var square ={ "upperLeft": { x:point.x, y:point.y},
              'lowerRight':{ x: (point.x + side), y: (point.y+side)}
            };
```

3.5.3. Obyektlərin dəyişikliyi

Obyekt məntiqi kontekstdə istifadə olunduqda, dəyişikliyin nəticəsi `true` qiymətidir. Obyekt sətir kontekstində istifadə olunduqda, obyektin dəyişikliyi `toString()` metodu ilə yerinə yetirilir və bu metodla qaytarılan sətir hesablamalarda iştirak edir. Obyekt ədəd kontekstində istifadə olunduqda, obyektə əvvəlcə `valueOf()` metodu çağrılır. Əgər bu metod primitiv tipdə olan ədəd qiymətini qaytarırsa, sonrakı hesablamalarda bu qiymət iştirak edir. Ancaq çox zaman `valueOf()` metodu obyektin özünü qaytarır. Belə vəziyyətdə obyekt əvvəlcə `toString()` metodu vasitəsilə sətirə dəyişdiriləcək, sonra isə ədədə dəyişdirmək mümkün olarsa, ədədə dəyişdiriləcək.
Obyektlər primitiv tipdə olan qiymətlərə dəyişdirərkən bir neçə həssas məqamlara diqqət yetirməlisiniz ki, problemlərlə qarşılaşmayasınız. Mövzunun davamınına fəsilin sonunda qayıdacağıq.

3.6. Massivlər

Massiv (array), obyekt kimi, qiymətlər kolleksiyasını təşkil edir. Obyektdə olan hər bir qiymət ad malikdirsə, massivdə olan hər bir qiymət nömrəyə və ya indeksə malikdir. JavaScript-də massiv adının qarşısında kvadrat mötərizələr vasitəsilə massivin indeksini göstərib, massivdən həmin qiymətləri çıxartmaq olar. Məsələn, tutaq ki, a massivin adı, i isə, mənfi olmayan tam ədəddir, onda a[i] massivin elementidir. Massivin indeksləri sıfırdan başlanır, yəni a[2] a massivinin üçüncü elementinə istinad edir.
Massivlər JavaScript-in istənilən məlumat tipini ehtiva edə bilər. Bundan başqa, həmçinin başqa massivlərə, obyektlərə və ya funksiyalara istinad edə bilər. Məsələn:

```
document.images[1].width
```

Bu kod obyektin ikinci elementində saxlanılan *width* xüsusiyyətinə, həmçinin document obyektinin images xüsusiyyətində saxlanılan massivə istinad edir. Diqqətli olun: burada təsvir edilən massivlər assosiativ massivlərdən (bölmə 3.5 baxmaq) fərqlənir. Bu bölmədə mənfi olmayan tam ədədlərlə indeksləşən "əsl" massivlərdən bəhs edilir.
Assosiativ massivlər ***sətirlərlə*** indeksləşdirilir. Həmçinin qeyd etmək lazımdır ki, JavaScript-də *çoxölçülü massivlər* ***dəstəklənmir*** (hərçənd ki, massivlərin daxilində massivlərin mövcudluğuna icazə verilir).
Və nəhayət, bir halda ki, JavaScript tipləşdirilməmiş dildir və tipləşdirilmiş dillərdən bir fərqi də ondan ibarətdir ki, massivin elementləri mütləq surətdə eyni tipə malik olmaya bilər. Massivlər haqqında daha ətraflı 7- ci fəsildə bəhs edilir.

3.6.1. Massivlərin yaradılması

Massiv `Array()` funksiya- konstruktorunun köməyi ilə yaradıla bilər. Yaradılmış massivə istənilən miqdarda indeksləşdirilmiş element mənimsədilə bilər:

```
var a = new Array();
a[0] = 1.2;
a[1] = "JavaScript";
a[2] = true;
a[3] ={ x:1, y:3};
```

Massivlər, `Array()` konstruktorunun köməyilə də, elementlərinin ötürülməsi yolu ilə inisializasiya edilmiş ola bilər. Beləliklə, massivin yaradılması və massivin inisializasiyasının əvvəlki nümunəsini belə yazmaq olar:

```
var a = new Array(1.2, "JavaScript", true,{ x:1, y:3} );
```

Əgər `Array()` konstruktoru yalnız bir ədəd mənimsədilərsə, o zaman konstruktor massivin uzunluğunu müəyyən edəcək. Beləliklə, aşağıdakı ifadə qeyri-müəyyən 10 elementli yeni massiv yaradır:

```
var a = new Array(10);
```

3.6.2. Massiv literalları

JavaScript-də massivlərin yaradılması və inisializasiya üçün massivin literal sintaksisi təyin edilir. Massivin literalı və ya inisializatoru – kvadrat mötərizələr ilə əhatələnmiş və vergüllər ilə bölünmüş qiymətlərin siyahısıdır. Mötərizələrdəki qiymətlər massivin elementi kimi sıfırdan başlayaraq ardıcıllıqla indekslərləşir. Məsələn, əvvəlki bölmədəki nümunədə yaradılan və inisializasiya edilən massivin proqram kodunu aşağıdakı qaydada yazmaq olar:

```
var a = [1.2, "JavaScript", true,{ x:1, y:3}];
```

Obyekt literalı kimi, massiv literalının səhv yazılışları ola bilər:

```
var matrix = [[1,2,3], [4,5,6], [7,8,9]];
```

Obyekt literallarındakı kimi, massiv literalında da elementlər sərbəst ifadə oluna və mütləq surətdə sabit olmaya bilər:

```
var base = 1024;
var table = [base, base+1, base+2, base+3];
```

Massiv literalına qeyri-müəyyən element daxil etmək üçün, vergüllər arasında qiymət yazmamaq kifayətdir. Aşağıdakı massiv özündə beş element ehtiva edir. Bu qiymətlərdə üçü qeyri-müəyyəndir:

```
var sparseArray = [1, 5];
```

3.7. null qiyməti

JavaScript-də *null* açar sözü xüsusi mənaya malikdir. Adətən, *null* qiyməti obyekt tipinin və həmin obyektin yoxluğunu bildirir. *null* qiyməti unikaldır və digər qiymətlərdən fərqlənir. Əgər dəyişən *null*-a bərabərdirsə, onda mümkün obyekt - massiv, ədəd, sətir və ya məntiqi qiymət olmur.
null qiyməti məntiqi kontekstdə istifadə olunduqda, bu qiymət *false* qiymətinə, ədəd kontekstində olduqda, bu qiymət 0-a, sətir kontekstində istifadə olunduqda isə "*null*" sətirinə dəyişdiriləcək.[5]

3.8. undefined qiyməti

JavaScript-də bəzi hallarda istifadə edilən daha bir xüsusi qiymət *undefined* qiymətidir. Bu qiymətə müraciət etdiki ya elan edildiyi və qiyməti olmayan dəyişənə, ya da ki, mövcud olmayan obyektin xüsusiyyətinə qayıdır. Qeyd edək ki, *undefined* xüsusi qiyməti – *null* qiymətindən fərqlidir.
Hərçənd ki, *null* və *undefined* qiymətləri bir-birinə ekvivalent deyil, bərabərlik operatoru (==) onları bərabər hesab edir. Aşağıdakı ifadəyə baxaq:

```
my.prop == null
```

my.prop xüsusiyyətinin mövcudluğundan asılı olmayaraq (yəni əgər my.prop xüsusiyyəti olsa belə, onun qiyməti *null* olur) bu müqayisə həqiqidir. Bir halda ki, *null* və *undefined* qiyməti qiymətin yoxluğunu ifadə edir, bu bərabərliyin tez-tez şahidi olacağıq. *undefined* və *null* qiymətini bir-birindən fərqləndirmək tələb olunda eynilik (===) və ya *typeof* operatorundan istifadə etmək lazımdır.
Null-qiymətindən fərqli olaraq, *undefined* qiyməti JavaScript-də ehtiyata saxlanılan söz deyil. ECMAScript v3 standartı göstərir ki, həmişə *undefined* adı ilə birlikdə qlobal dəyişən mövcuddur, hansı ki, bu dəyişən *undefined* qiymətinin ilk mənasıdır. Beləliklə, standarta uyğun olan reallaşdırmada əgər qlobal dəyişə başqa qiymətə mənimsədilməmişsə, *undefined*-ə açar sözü kimi baxmaq olar.

Əgər reallaşdırmada *undefined* dəyişəninin olduğunu bilmirsinizsə, onda öz şəxsi *undefined* dəyişəninizi elan etmək olar:

```
var undefined;
```

Elan edilmiş lakin initializasiya edilməmiş (qiymət mənimsədilməmiş) dəyişən yaratdıqda siz bu dəyişənin qiymətini *undefined* təyin etmiş olursunuz. Void (5-ci fəsilə baxmaq) operatoru *undefined* qiymətinin alınmasının daha bir üsuludur. *undefined* qiyməti məntiqi kontekstdə istifadə olunduqda, *false* qiymətinə, ədəd kontekstində, NaN qiymətinə, sətir konteksində isə "*undefined*" sətirinə dəyişdiriləcək.

3.9. Date obyekti

Əvvəlki bölmələrdə biz JavaScript-in dəstəklədiyi bütün fundamental məlumat tiplərini təsvir etdik. Tarix dəyəri və vaxt bu fundamental tiplərə aid deyil, ancaq JavaScript- də tarixi təqdim etmək üçün obyekt sinifi mövcuddur və bu sinifdən istifadə edərək vaxt və digər bu tip məlumatlar ilə işləmək olar. JavaScript-də Date obyekti new operatorunun köməyi ilə yaradılır və Date() konstruktorları (new operatoru 5-ci fəsildə təsvir ediləcək, amma 7-ci fəsildə siz bu operator ilə daha yaxından tanış olacaqsınız (obyektlərin yaradılmasında istifadə edildikdə):

```
var now = new Date();       // Vaxt və tarix tipində olan məlumatların
                    // saxlanıldığı obyektin yaradılması.
                            // Yaradılan obyekt, Miladi təqviminə əsasən
                            // saxlanılır.
                            // Diqqətli olun: ayların nömrələri sıfırdan başlanır,
                            // buna görə də məsələn, dekabr ayı "11-ci aydır"!
var xmas = new Date(2000, 11, 25);
```

Date obyekti metodlar almaq və müxtəlif qiymətlər qurmağa imkan verir. Tarixlər və vaxtı sətir formatında yerli vaxt və ya Qrinviç (GMT) vaxtı üzrə dəyişmək olar. Məsələn:

```
xmas.setFullYear(xmas.getFullYear() + 1) ;        // tarixi Miladi tarixi ilə
                                                  // əvəz edirik.
var weekday = xmas.getDay();                        // Miladi təqvimində 2007-ci
                                                 // ilin ilk günü çərşənbə
                                                 // axşamına təsadüf edir.
document.write ("Bu gün: "+ now.toLocaleString()); //tarix və vaxt məlumatı.
```

Date obyektində funksiyalar təyin edilə bilməz (metodlar, çünki, funksiya Date obyekti vasitəsilə çağırıla bilmir) cərgə ilədə verilmiş tarixin dəyişikliyi üçün və ya say formasına, millisaniyələrdə daxili təqdim etməyə (təsəvvürə), faydalı üçün tarixlərlə əməliyyatların bəzi (bir qədər) növləri. Date obyekti və onun metodlarının tam təsvirini siz kitabın üçüncü hissəsində görəcəksiniz.

3.10. Requlyar ifadələr

Requlyar ifadələr mətndə şablonların təsviri ilə zəngin və güclü sintaksisini malikdir. Bu ifadələr vasitəsilə verilmiş şablona uyğun sözün axtarışı və tapılmış sözün əvəzedilməsi əməliyyatları yerinə yetirilə bilər. Requlyar ifadələrin formalaşması üçün JavaScript, Perl dilinin sintaksisi qəbul etmişdir. Requlyar ifadələr JavaScript-də RegExp obyekti ilə təqdim edilir və RegExp() konstruktorunun köməyi ilə yaradılır. Date obyekti kimi, RegExp obyekti də JavaScript-in fundamental məlumat tiplərinə aid deyil; bu yalnız JavaScript reallaşdırmalarında verilən obyektlərin standartlaşdırılmış tipidir. Ancaq Date obyektindən fərqli olaraq, RegExp obyektləri literal sintaksisinə malikdir və bilavasitə JavaScript-proqramın koduna əlavə edilə bilər. Mətn arasında müntəzəm ifadənin literalı iki sleşin simvoluyla yaradır. Cütlükdəki ikinci simvolunda həmçinin şablona uyğun bir və bir neçə hərf qeyd etmək olar. Məsələn:

```
/^HTML/
/[ 1-9] [0-9] */
/ \bjavascript\b/i
```

3.11. Error obyektləri

ECMAScript v3-də xətaların tənzimlənməsi üçün bir neçə tipdə sinif təyin edilir. Yerinə yetirilmə zamanı xəta yaranarkən, JavaScript interpretatoru bu tiplərdən birinin obyektini "yaradır". (Xətaların yaradılması və tənzimlənməsi 6-cı fəsildə `throw` və `try` təlimatlarının təsviri zamanı izah olunur.) Hər bir xəta obyekti realizasiyadan asılı olaraq özündə xəta barədə məlumatı ehtiva edən message xüsusiyyəti malikdir. Qabaqcadan müəyyən edilmiş xəta obyektlərinə – `Error`, `EvalError`, `RangeError`, `ReferenceError`, `SyntaxError`, `TypeError` və `URIError` misal göstərmək olar.

3.12. Tiplərin dəyişikliyi

Artıq bütün məlumat tipləri əvvəlki bölmələrdə müzakirə edilib. Burada biz mövcud tipdə olan qiymətləri başqa tipin qiymətlərinə necə dəyişdiriləcəyini öyrənəcəyik. Əsas qayda bundan ibarətdir: əgər bir tipin qiyməti hansısa başqa bir tipin qiymətini tələb edən kontekstdə istifadə olunursa, JavaScript interpretatoru bu qiyməti avtomatik dəyişdirməyə çalışır. Məsələn, əgər məntiqi kontekstdə ədəd tipindən istifadə olunursa, bu ədəd məntiqi tipin qiymətinə uyğun dəyişdiriləcək. Əgər obyekt sətir kontekstində istifadə olunursa, bu obyekt sətir qiymətinə dəyişdiriləcək. Əgər sətir ədəd kontekstində istifadə olunursa, JavaScript interpretatoru bu sətiri ədədə dəyişdirməyə çalışır. Cədvəl 3.4-də tiplərin avtomatik dəyişikliyinin siyahısı göstərilmişdir.

Cədvəl 3.4 *Tiplərin avtomatik dəyişikliyi*

Qiymətin tipi	***İstifadə olunduğu kontekst***			
	Sətir	**Ədəd**	**Məntiqi**	**Obyekt**
Qeyri-	`"undefined"`	`NaN`	`false`	`Error`

müəyyən qiymət				
null	“*null*”	0	*false*	*Error*
Boş olmayan sətir	Olduğu kimi	Əgər ədədə dəyişdirilə bilmirsə *NaN*	*true*	*Object.String*
Boş sətir	Olduğu kimi	0	*false*	*Object.String*
0	“0”	Olduğu kimi	*false*	*Object.Number*
NaN	“NaN”	Olduğu kimi	*false*	*Object.Number*
Infinity	“Infinity”	Olduğu kimi	*true*	*Object.Number*
-Infinity	“-Infinity”	Olduğu kimi	*true*	*Object.Number*
İstənilən digər ədəd	Ədədin sətirlə ifadəsi	Olduğu kimi	*true*	*Object.Number*
true	“true”	1	Olduğu kimi	*Object.Bollean*
false	“false”	0	Olduğu kimi	*Object.Bollean*
Obyekt	*toString()*	*valueOf()*, *toString()* yaxud *NaN*	*true*	Olduğu kimi

3.13. Elementarlar məlumat tipləri üçün obyekt-üzlüklər

Daha əvvəlki, fəsildə biz sətirləri müzakirə etdik və onda mən sizin diqqətiniz çatdırdım ki, bu məlumat tipinin qəribə xüsusiyyəti mövcuddur: sətirlərlə işləmək üçün obyekt nəsihətlərindən istifadə olunur.[6] Məsələn, sətirlərlə tipik əməliyyat bu cür ola bilər:

```
var s = "These are the times that try people's souls.";
var last_word = s.substring (s.lastIndexOf (" ") +1, s.length);
```

Əgər, məlumatınız olmasa, onda elə biləcəksiniz ki, s obyektdir və siz və bu obyektin xüsusiyyətlərinin qiymətlərini metodlar çağırmaqla oxuyursunuz.
Bu necə olur? Sətirlərdə olan məlumatlar obyektdir yaxud elementar tipdir? *Typeof* (5-ci fəsilə baxmaq) operatoru, obyekt tipində sətirləri sətir məlumat tipindən ayıra bilir. Bəs,

niyə görə sətirlərin manipulyasiyası üçün obyekt xüsusiyyətindən istifadə olunur?
Məsələ orasındadır ki, üç baza məlumat tipinin hər biri üçün uyğun obyekt sinifi müəyyən edilmişdir. Yəni JavaScript, ədəd, sətir və məntiqi məlumat tipləri dəstindən əlavə Number, String və Boolean siniflərini dəstəkləyir. Bu siniflər baza məlumat tipləri üçün *"üzlük"* təqdim edir. *Üzlük (wrapper)* baza tipi ilə eyni qiymət ehtiva edir, lakin bundan başqa da özündəqiymətin manipulyasiyasında istifadə edilən xüsusiyyətlər və metodları müəyyən edir.
JavaScript ustalıqla bir tipi başqa bir tipə dəyişdirə bilər. Biz obyekt kontekstində sətirdən istifadə etdikdə, yəni sətir xüsusiyyətinə və ya metoduna müraciət etməyə çalışdıqda, JavaScript-in daxilində sətir qiymətləri üçün obyekt-üzlük yaradılır. String obyekti baza sətir qiymətinin yerinə istifadə olunur. Obyekt üçün xüsusiyyətlər və metodlar müəyyən edilmişdir, buna görə də baza tipindəki obyekt konteksində istifadə olunur
elementar məlumat tipləri üçün üzlük obyekt kontekstində baza tipinin çeniesi. Əlbəttə ki, bu və başqa baza tipləri və onlara uyğun olan obyekt-üzlüklər bu qayda ilə işləyir; biz sadəcə obyekt kontekstində sətir tipi qədər digər tiplərlə işləmirik.
Qeyd etmək lazımdır ki, obyekt kontekstində String sətirdən istifadə zamanı, xüsusiyyətə və ya metoda müvəqqəti girişi təmin etmək üçün yaradılmışdır, təmin olunmadan sonra bu obyektə ehtiyac duyulmur və buna görə yalnız sistem tərəfindən istifadə edilir. Fərz edək ki, s – sətirdir və biz aşağıdakı təklifin köməyilə sətirin uzunluğunu müəyyən edirik:

```
var len = s.length;
```

Burada s sətiri qalır və onun ilkin mənası dəyişmir. length xüsusiyyətinə müraciət etməyə imkan verən müvəqqəti String obyekti yaradılır sonra isə, s dəyişəninin ilkin mənasını dəyişmədən bu silinir. Bu sxem sizə zərif və qeyri-təbii, eyni zamanda da mürəkkəb görünə bilər. Ancaq JavaScript reallaşdırmaları adətət daxili dəyişikliyi çox effektiv yerinə yetirir və bundan narahat olmağınıza dəyməz.
Öz proqramında String obyektindən istifadə etməklə sistem tərəfindən avtomatik silinməyəcək daimi obyekti yaratmaq olar. String obyektləri, digər obyektlər kimi yəni `new` operatorunun köməyi ilə yaradılır. Məsələn:

```
var s = "hello world";                  // Sətir tipində olan qiymət
var S = new String("Hello World");      // String obyekti
```

String tipinin S-də yaradılmış obyekt ilə nə etmək olar? Heç nə belə ki, baza tipinə uyğun olan qiyməti ilə etmək olmaz. Əgər biz `typeof` operatorundan istifadə edəcəyiksə, o bizə bildirəcək ki, S – obyektdir və burada sətir qiyməti mövcud deyil, lakin, biz baza sətir qiyməti ilə String obyektinin qiyməti arasında fərq görməyəcəyik, çünki, sətirlər tələb olunduqda avtomatik olaraq String obyektlərinə dəyişdirilir.[7] Məlum olur ki, bu əməliyyatın əksi də düzgündür, yəni biz String obyektindən baza sətir tipinin qiyməti güman edilən yerdə istifadə etdikdə, JavaScript String obyektini sətirə avtomatik olaraq dəyişdirəcək. Buna görə də, əgər biz String obyektində + operatorunda istifadə ediriksə, sətirlər bitişdirilməsi əməliyyatı icra etmək üçün müvəqqəti olaraq, bu obyektin qiyməti baza sətir tipinin qiyməti olaraq yaradılır:

```
msg = S + '!';
```

Nəzərə alın ki, bu üsul təkcə sətir qiymətləri və String obyektləri ilə məhdudlaşmır, həmçinin ədəd və məntiqi məlumat tiplərinə müvafiq olaraq, Number və Boolean obyektlərindən istifadə olunur. Bunlar haqqında daha ətraflı məlumat kitabın III hissəsində siniflərə aid məqalələrdə yerləşdirilmişdir. Nəhayət, qeyd etmək lazımdır ki, istənilən sətir, ədəd və ya məntiqi qiymətlər Object() funksiyasının köməyi ilə uyğun olan obyekt-üzlüyə dəyişdirilə bilər:

```
var number_wrapper = Object(3);
```

3.14. Obyektlərin elementar tiplərdə olan qiymətlərə dəyişikliyi

3.5.3. bölməsində qeyd edildiyi kimi obyektlər adətən elementar tipdə olan qiymətlərə kifayət qədər dəyişdiriləcək düzxətli üsul ilə dəyişdiriləcək. Lakin obyektlərin dəyişikliyi mürəkkəbdir.[8]

Hər şeydən əvvəl qeyd etmək lazımdır ki, tam obyektləri məntiqi qiymətlərə dəyişiklirdikdə, *true* qiyməti alır. Bu istənilən obyektlər (massivlər və funksiyalar da daxil olmaqla) hətta obyekt-üzlüklər üçün də ədalətlidir. Obyekt-üzlüklər elementar tiplər ilə, dəyişildikdə, *false* qiymətini alır. Məsələn, aşağıda göstərilən bütün obyektləri məntiqi kontekstə istifadə edərkən *true* qiymətinə malik olur:

```
new Boolean(false) //Daxili qiymət – false, amma obyekt true-a dəyişdiriləcək
new Number(0)
new String("")
new Array()
```

Cədvəldə. 3.3-də obyektlərdə valueOf() metodu çağırılarkən obyektin ədəd dəyişikliyi sırası göstərilmişdir. Obyektlərin əksəriyyəti gizli olaraq Object sinifinin baza obyekti olan və qiymət olaraq obyektin özünü qaytaran *valueOf()* metoduna varis olur. Bir halda ki, susmaya görə olan *valueOf()* metodu elementar tipdə olan qiyməti qaytamırsa, onda JavaScript interpretatoru bu obyekt-ədəd dəyişiklikliyini *toString()* metodunun köməyilə etməyə çalışır və sonda sətir ədəd dəyişikliyi baş verir.

Massivlər halında bu olduqca maraqlı nəticələrə gətirir. Massivlərdə *toString()* metodu massivin elementlərini sətirlərə dəyişdirəcək və massivin ayrı-ayrı elementlərini vergüllərlə ayırır. Son nəticədə sətirlərin bitişdirilməsi əməliyyatı alınır. Əgər, boş massiv boş sətirə dəyişdiriləcəksə, nəticədə 0 ədədi alınır! Bundan başqa, əgər element olaraq n ədədini ehtiva edən massiv təkelementlidirsə, onda bütün massiv n ədədini ehtiva edən sətirə dəyişdiriləcək və bundan sonra sətir-ədəd dəyişikliyi aparılacaq. Əgər massiv birdən çox təkelementlidirsə və ya massivin tək elementi ədəd deyilsə, dəyişikliyin nəticəsi NaN qiyməti olacaq.

Dəyişikliyin tipi dəyişikliyin edildiyi kontekstdən asılıdır. Elə vəziyyətlər də olur ki, birmənalı olaraq konteksti müəyyən etmək mümkün olmur. + operatoru və müqayisə (<=, > və >=) operatorları eyni zamanda həm ədəd, həm sətir, həm də obyekt arasında əməliyyat apararsa, çoxmənalılıq yaranır və belə bir təzadlı sual meydana çıxır: obyektin qiymətini hansı tipdə olan qiymətə – sətirə və ya ədədə dəyişmək lazımdır? Çox zaman JavaScript əvvəlcə

obyekti *valueOf()* metodunun köməyi ilə dəyişdirməyə çalışır. Əgər bu metod elementar tipin (bir qayda olaraq, ədədin) qiymətini qaytarırsa, onda bu qiymət istifadə olunur. Ancaq əksər hallarda *valueOf()* metodu dəyişdirilməyən obyektin özünü qaytarır. Bu halda JavaScript interpretatoru obyekti sətirə *toString()* metodunun çağırılması ilə dəyişdirməyə çalışır.
Ancaq burada bir istisna var: + operatoru *Date* obyekti ilə istifadə olunduqda, dəyişiklik birbaşa *toString()* metodunun çağırışı ilə başlanır. Bu istisna ona görədir ki, *Date* obyekti *toString()* və *valueOf()* metodlarının şəxsi reallaşdırmalarına malikdir. Ancaq, *Date* obyekti + operator ilə göstərildikdə, əksər hallarda sətirlərin bitişdirilməsi əməliyyatı nəzərdə tutulur və müqayisə əməliyyatının icra edilməsi zamanı praktik olaraq həmişə iki tarixdən hansının daha erkən olduğunu müəyyən etmək tələb olunur.
Obyektlərin əksəriyyəti *valueOf()* metoduna malik deyil və ya bu metod tələb edilən elementar tipin qiymәti qaytarmır. Obyekt + operator ilə istifadə olunduqda, adətən sətirlərin bitişdirilməsi, ədədlərin isə toplanması əməliyyatı yerinə yerinə yetirilir. Analoji olaraq, obyekt müqayisə əməliyyatlarında iştirak etdikdə, adətən ədəd qiymətlərinin deyil, sətir qiymətlərinin müqayisəsi aparılır. Şəxsi reallaşdırmasında *ValueOf()* metodu olan obyektlər bu vəziyyətdə özünü başqa cür apara bilər. Əgər siz *valueOf()* metodunu yenidən təyin edəcəksinizsə ki, o sayı qaytarsın, obyektin üstündə hesabları yerinə yetirmək mümkün olacaq və başqa say əməliyyatları, amma sətirlə obyektin toplanmasının əməliyyatı arzu oluna
Nəhayət, yadda saxlamaq lazımdır ki, *valueOf()* metodu *toNumber()* metodunu çağırmır. Daha dəqiq desək, bu metodun təyinatı ondan ibarətdir ki, obyekti elementar tipin səmərəli qiymətinə dəyişdirsin; bu səbəbdən də bəzu obyektlərdə *valueOf()* metodları sətir qiymətini qaytarır.

3.15. Qiymət və ya istinad üzrə

JavaScript-də, başqa proqramlaşdırma dillərində olduğu kimi məlumatı üç üsul ilə manipulyasiya etmək mümkündür, Birinci üsul – məlumatların kopyalamasıdır. Məsələn, qiymətə yeni dəyişən mənimsəmək olar. İkinci üsul – funksiyaya və ya metoda qiymət təyin etmək. Üçüncü – bu qiymətləri digər qiymətləri digər qiymətlər ilə müqayisə edərək, onların bərabərliyinin yoxlanılması. Proqramlaşdırma dilin tamamilə başa düşmək üçün bu üç təsirin yerinə yetirilmə necəliyini tam başa düşmək lazımdır.
Məlumatlar manipulyasiya etməyin iki fundamental üsulu mövcuddur: *qiymət üzrə, istinad üzrə.* Qiymət ölçüsü üzrə yerinə yetirilən manipulyasi onu bildirir ki, əməliyyatda məhz bu qiymət ölçüsü iştirak edir. Mənimsəmə əməliyyatında faktiki qiymətin surəti yaradılır, sonra bu surət dəyişlikliyə məruz qalarsa, obyektin xüsusiyyətində və ya massivin elementində saxlanılır. Surət və orijinal – iki tamamilə müstəqil, ayrı-ayrı saxlanılan dost qiymətdir. Funksiyaya bəzi qiymət ölçüsü üzrə ötürülmə baş verərsə, bu onu bildirir ki, funksiyalara surət ötürülür. Əgər funksiya alınmış qiyməti dəyişdirəcəksə, bu dəyişikliklər yalnız surətə təsir edəcək və heç vaxt orijinal toxunulmayacaqdır. Nəhayət, bir qiymət ölçüsünün, digər bir qiymət ölçüsü ilə müqayisəsi aparıldıqda, iki müxtəlif məlumat dəstindən birini özündə

saxlamalıdır (bu adətən ölçülərin ekvivalentliyinin yoxlanılması üçün onların baytlarının müqayisəsi edilməsini nəzərdə tutur).

Manipulyasiya etmənin başqa bir üsulu – istinad üzrə. Bu halda qiymətin yalnız bir faktiki surəti mövcuddur və manipulyasiya etmə bunu istinadlar vasitəsi ilə qiymətlərə ötürülür.[9] Qiymətlərlə hər hansı təsir istinad üzrə edildikdə, dəyişənlər qiymət ehtiva etmir və dəyişənlərə qiymət yalnız istinadlar vasitəsilə ötürülür. Məhz bu istinad informasiyası müqayisə əməliyyatlarında yamsılanır, ötürülür və iştirak edir. Beləliklə, istinad üzrə mənimsəmə əməliyyatında qiymət və ya qiymətin surəti deyil istinadın özü iştirak edir. İstənilən qiymət mənimsədilməsi qiymətə istinad edən dəyişən və qiymətin mənimsədildiyi orijinal dəyişənə şamil olacaq.

Hər iki istinad tamamilə bərabər hüquqlu hesab edilir və qiymətləri eyni dərəcədə manipulyasiya etməyi bacarır. Əgər bir istinadın köməyi ilə qiymət dəyişirsə, bu dəyişikliklər başqa istinadlar köməyi ilə də müşahidə olunur. Funksiyaya istinad üzrə ötürülmə qiymət ölçüsü üzrə ötürülmə analojidir. Qiymət funksiyaya qiymətə istinad üzrə ötürüldükdə, funksiya bu qiymətdən özünün dəyişikliklərində istifadə edə bilər. İstənilən bu cür dəyişikliklər üçün funksiyanın hüdudları görülür. Nəhayət, istinad üzrə müqayisə əməliyyatı yerinə yetirildikdə, iki istinadın eyni qiymətə istinadını yoxlamaq üçün bu istinadların müqayisəsi aparılır. İki müxtəlif qiymətə istinad edən ekvivalent istinadlar belə, bərabər hesab edilə bilmirlər. Bunlar dəyişənə qiymət manipulyasiyanın tamamilə müxtəlif üsullarıdır və bunları mükəmməl anlamaq lazımdır. Cədvəldə. 3.4-də yuxarıda göstərilənlərin qısa təsviri təqdim edilmişdir. Bu bölmədə manipulyasiya barədə olan məlumatlar proqramlaşdırma dilləri üçün ümumi xarakter daşıyır. Növbəti bölmələrdə JavaScript-də məxsus xüsusi fərqlərə baxılacaq. Yəni, hansı məlumat tipində qiymət üzrə, hansında isə, istinad üzrə manipulyasiya etmək lazım olduğu göstəriləcək.

Cədvəl 3.4. İstinad üzrə və qiymət üzrə

	Qiymət üzrə	**İstinad üzrə**
Kopyalama	`Infinity`	Yalnız qiymət istinad yamsılanır. Əgər qiymət istinad surətinin köməyi ilə dəyişdiriləcəksə, bu dəyişikliklər orijinal istinaddan istifadə zamanı müşahidə olunacaq.
Ötürülmə	Funksiyanın ötürülməsi qiy-mətin ayrı-ayrı surətlərinin ötürülməsi hesabına baş verir. Bu surət dəyişikliyi	Funksiyalar qiymətə istinad vasitəsilə ötürü-lür. Əgər funksiyanın daxilində qiymət, alınmış istinadın köməyi ilə dəyişdiriləcəksə, bu

	funksiya xaricində heç bir qiymətə təsir göstərmir.	dəyişikliklər funksiyanın hüdudları da-xilində müşahidə olunacaq.
Müqayisə	İki müxtəlif müqayisə. Qiymətlərin (adətən baytlı), bərabər olduğu müəyyən etmək üçün müqayisə aparılır	İki istinad, müqayisə edilir, eyni qiymətə istinad etdiyi müəyyən etmək üçün müqayisə edilir. Müxtəlif qiymətlərə istinad edənlərin bərabər olmaması aydın məsələdir lakin, hətta qiymətlər tamamilə eynidirsə belə bu istinadlar

3.15.1. Elementar və sitat tipləri

JavaScript-in əsas qaydası aşağıdakı qaydadır: elementar tiplər üzərində aparılan əməliyyatlar qiymət üzrə yaradılır, lakin sitat tiplərində aparılan əməliyyatlar adına uyğun olaraq, istinad üzrə aparılır.

Ədədlər və məntiqi ölçülər – JavaScript-də elementar tiplərdir. Bunlar ona görə elementar tiplərə aid edilir ki, bu tiplərin malik olduqları baytların sayı kiçikdir, buna görə də, bu tiplər ilə yüngül əməliyyatlar aparmaq mümkündür bu əməliyyatlar JavaScript-in aşağı səviyyəli interpretatoru tərəfindən yerinə yetirilir. Obyektlər sitat tiplərinin bir nümayəndəsidir. Həmçinin, massivlər və funksiyalar – ixtisaslaşdırılmış obyekt tipləri olduğuna görə onlar da sitat tipindədir. Bu məlumat tipləri xüsusiyyətlərin və ya elementlərin sərbəst sayından ibarət ola bilər, buna görə sadəcə belə onlarla əməliyyat etməyə bilərlər, bərkidilmiş ölçülərə malik olan elementar tiplərin qiymətləri (mənaları) kimi. Bir halda ki, massivlərin və obyektlərin ölçüləri fövqəladə böyük ola bilər, onların üzərində qiymət üzrə aparılan əməliyyatlar sübutsuz kopyalanmaya və nəhəng yaddaşın həcminin müqayisəsinə gətirib çıxara bilər.

Bəs sətirlərdə? Sətirlər sərbəst uzunluğa malik ola bilər, buna görə onlara sitat tipi kimi baxmaq olar. Bununla belə sətirin JavaScript-də adətən sətirlərə elementar tip kimi baxılır, ona görə ki, onlar obyekt deyil. Həqiqətdə sətirlər ikipolyar elementar yolla-sitat tipinə aid edilmir.

Məlumatların arasında fərqi anlamaq üçün ən yaxşı üsul, istinad üzrə və qiymət üzrə aparılan əməliyyatların, praktik nümunələrinin öyrənilməsindən ibarətdir. Aşağıdakı nümunəyə nəzər yetirin və xüsusilə şərhlərə diqqət yetirin. Nümunə 3.1-də kopyalama, ötürülmə və ədədlərin müqayisəsi əməliyyatları yerinə yetirilir. Bir halda ki, ədədlər elementar tipdədir, verilmiş nümunə qiymət üzrə yerinə yetirilən əməliyyatların illüstrasiyasıdır.

Nümunə 3.1. *Kopyalama, ötürülmə və qiymət üzrə ölçülərin müqayisəsi*

```
// Qiymət üzrə kopyalanma əməliyyatında birinci mərhələ


 var n = 1;  // n dəyişəni 1-qiymətini ehtiva edir
 var m = n;  // Qiymət üzrə kopyalama: başqa bir dəyişən, m dəyişəni də 1
           // qiymətini ehtiva edir.
// Bu funksiyadan qiymət üzrə ölçünün ötürülməsi əməliyyatının
// illüstrasiyası üçün istifadə olunur.
// Şahidi olacaqsınız ki, bu tip funksiya istənilən qaydada
// işləmir

 function add_to_total(total, x)
 {
  total = total + x; // Bu sətir yalnız total-ın daxili surətini dəyişdirir
 }

 // İndi n və m dəyişənlərində olan ədədlər qiyməti üzrə ötürülən funksiyaya
 // daxil edilir
 // Funksiyanın daxilində olan n dəyişəninin qiymətinin surəti total adında da
 // əlçatandır. Funksiya m və n dəyişənlərinin qiymətlərinin surətlərini
 // yaradır və nəticəni n dəyişəninin qiymətinin surətinə yazır. Ancaq bu
 // funksiya xaricində olan n dəyişəninin orijinal qiymətinə heç bir təsir
 // göstərmir.Beləliklə, bu funksiyanın çağırışı nəticəsində heç bir dəyişiklik
 // hasil olunmur.
 add_to_total(n, m);

 // İndi biz qiymət üzrə müqayisə əməliyyatını yoxlayacağıq.
 // Proqramın aşağıdakı sətirində 1 literalı proqramın mətninə mükəmməl
 // "qurulmuş" müstəqil ədəd qiymətidir
 // Biz indi n dəyişəninin ehtiva etdiyi qiyməti müqayisə edirik
 // İndiki halda, iki ədədin bərabərliyindən əmin olmaq, baytlar üzrə müqayisə
 // əməliyyatı yerinə yetirilir.
 if (n == 1) m = 2;        // n dəyişəni həminki 1 literalını ehtiva edir;
                         // beləliklə m dəyişəninə isə 2 qiyməti yazılacaq
```

İndi nümunə 3.2-ə baxacağıq. Kopyalanma, ötürülmə və müqayisə əməliyyatlarının bu nümunəsi obyektlərin üzərində həyata keçirilir. Çünki, obyektlər sitat tiplərinə aiddir və onların üzərində aparılan bütün əməliyyatlar istinad üzrə edilir. Bu nümunədə Date obyektindən istifadə edilmişdir, bu haqda daha ətraflı kitabın üçüncü hissəsində oxumaq olar.

Nümunə 3.2. *Kopyalanma, ötürülmə və istinad üzrə ölçülərin müqayisəsi*

```
 // Burada Miladi təqviminə uyğun 2007-ci il tarixli obyekt yaradılır
 // xmas dəyişəni obyektə istinad əlavə edir və lakin özü obyekt deyil

 var xmas = new Date(2007, 11, 25);
 // Sonra bu istinadın  kopyalanması əməliyyatı yerinə yetirilir, yəni orijinal
 // obyektə ikinci istinad alınır

 var solstice = xmas;

 // Hər iki dəyişən ən birinci obyektə istinad edir
 // Burada yeni istinadın köməyilə obyekt dəyişikliyi yerinə yetirilir

 solstice.setDate(21);         // Dəyişiklikləri birinci istinaddan istifadə
 zamanı
                         // müşahidə
```

```
xmas.getDate();              // ilkin qiyməti (25) deyil, 21 qiyməti qaytarılır

// Həminki funksiya obyektlərin və massivlərin ötürülməsi zamanı baş verir.
// Aşağıdakı funksiya massivin bütün elementlərinin qiymətləri ilə işləyir.
// Funksiyalar massivə istinad vasitəsilə ötürülür, lakin massivin surəti
// olmur. Bunun sayəsində funksiya verilmiş istinad üzrə massivin tərkibini
// dəyişdirə bilər. Bu dəyişiklikləri funksiyadan qaytarılmadan sonra
// görüləcək.

function add_to_totals (totals, x)
{
  totals [0] = totals [0] + x;
  totals [1] = totals [1] + x;
  totals [2] = totals [2] + x;
}
// Nəhayət, sonra istinad üzrə müqayisə əməliyyatı yerinə yetirilir.
// Yaradılmış iki dəyişənin, müqayisəsi zamanı ilk öncə onların
// ekvivalentliliyi yoxlanılır, sonra isə baytlar üzrə müqayisə aparılır
// müəyyən olur, bununla belə, tarix dəyişikliyi onlardan yalnız birinə görə
// aparılır:

(xmas == solstice) // true qiymətini qaytarır

// Bir-birindən daha gec yaradılmış iki dəyişən eyni tarixi ehtiva etsə belə,
// müxtəlif obyektlərə istinad edir.

var xmas = new Date(2007, 11, 25);
var solstice_plus_4 = new Date(2007, 11, 25);
// Amma müxtəlif obyektlərə istinad, müəyyən olunmuş "istinad üzrə
// müqayisələr"qaydasına əsasən ekvivalent hesab edilmir!

(xmas! = solstice_plus_4) // true qiymətini qaytarır
```

Obyektlərin üzərində əməliyyatların icra edilməsi mövzusunun müzakirəsini bitirməzdən əvvəl, istinad üzrə massivlərə mövzusuna bir az aydınlıq gətirək. "İstinad üzrə ötürülmə" ifadəsi bir neçə mənaya malik ola bilər. Bəzilərinizüçün bu ifadə funksiya çağırışının bu cür üsulunu (funksiyanın daxilində bu qiymətləri dəyişdirməyə imkan verən və onun hüdudları xaricində bu dəyişiklikləri müşahidə edən) bildirir. Ancaq bu termin bu kitabda bir qədər başqa mənada izah edilir. Burada sadəcə, funksiyaların massivə və ya obyektə istinad üzrə ötürüldüyü, lakin özünün obyekt olmadığı nəzərdə tutulur. Funksiya bu istinadın köməyilə obyektin xüsusiyyətlərini və ya massivin elementlərini dəyişdirməyə imkan verir və bu dəyişikliklər funksiyadan çıxışında saxlanılır. Bu terminin başqa izahları ilə tanış olanlar, deyə bilər ki, obyektlər və massivlər qiymət üzrə ötürülür, düzdür, bu qiymət ilə faktiki olaraq obyektə istinad olunur və istinadın özü obyekt olmur. Nümunə 3.3 bu problemi əyani təsvir edir.

Nümunə 3.3. *İstinadlar qiymət üzrə ötürülür*

```
// Burada add_to_totals() funksiyasının başqa versiyası işlədilir. Hərçənd o
// işləmir,çünki massivin özünün dəyişikliyinin yerinə o bu massivə istinadı
// dəyişdirir.
function add_to_totals2(totals, x)
{
  newtotals = new Array(3);
  newtotals[0] = totals[0] + x;
  newtotals[1] = totals[1] + x;
```

```
    newtotals[2] = totals[2] + x;
    totals = newtotals;       // Bu sətir funksiya xaricində olan massivə təsir
                    // göstərmir
}
```

3.15.2. Kopyalanma və sətirlərin ötürülməsi

Daha əvvəl deyildiyi kimi, sətirlər ikiqatpolyar elementar tipdir lakin istinad tipinə aid edilmir. Bir halda ki, sətirlər obyekt deyil, onları elementar tipə aid olduğunu fərz etmək tamamilə təbiidir. Əgər sətirlərə elementar məlumat tipi kimi baxılacaqsa, əvvəlki bölmədə deyilənlərə uyğun olaraq onların üzərində əməliyyatlar qiymət üzrə aparılmalıdır. Amma bir halda ki, sətirlər ixtiyari uzunluğa malik ola bilər, bu kopyalama əməliyyatında və baytlar üzrə müqayisədə sistem resurslarının qeyri-məhsuldar xərcləməsinə gətirə bilər. Beləliklə, tamamilə olmasa da sətirlərin sitat məlumat tipi kimi realizasiyasını fərz etmək olar.
Fərziyyələr qurmaq əvəzinə, JavaScript dilində kiçik fraqment yazaraq, problemi bu yolla eksperimental həll etməyə cəhd etmək olar. Əgər sətirlər istinad üzrə kopyalanırsa və ötürülürsə, başqa dəyişəndə saxlanılan istinadın köməyilə və ya funksiyaya sətirin ötürülməsi nəticəsində onların tərkibini dəyişdirmək mümkün olmalıdır. Ancaq siz bunu təcrübədən kecirmək üçün kod yazdığınızda ciddi problemlə rastlaşacaqsınız: JavaScript-də sətirin tərkibini dəyişdirmək mümkün deyil. Verilmiş simvolun sətirdəki mövqesini qaytaran CharAt() metodu mövcuddur, amma bu mövqeyə başqa simvolu daxil etməyə imkan verən setCharAt() metodu yoxdur. Bu səhv deyil. JavaScript-də sətirlər dəyişilməzdir. Çünki JavaScript-in sətir simvollarını dəyişdirmək imkanı yaradan elementləri yoxdur.
Bir halda ki, sətirlər dəyişilməzdir, onda belə nəticə çıxır ki, sətirlərin qiymət üzrə və ya istinad üzrə ötürüldüyünü yoxlamaq mümkün deyil. Fərz etmək olar ki, effektivliyin artırılması məqsədi ilə JavaScript interpretatoru belə reallaşdırılmışdır ki, sətirlər istinad üzrə ötürülsün, amma bu həm də yalnız fərziyyə olaraq qalır, çünki, JavaScript-in bu sahədə imkanları məhduddur.

3.15.3. Sətirlərin müqayisəsi

Sətirlərin qiymət üzrə, yaxud istinad üzrə kopyalanmasını müəyyən etmək mümkün olmasa da , aşağıdakı fraqmenti yazaraq, JavaScript-də sətirlərin qiymət üzrə, yaxud istinad üzrə müqayisə edildiyini müəyyən etmək olar. ***Nümunə 3.4***-də bu cür yoxlamanı yerinə yetirən fraqment göstərilib.
Nümunə 3.4. Görəsən, sətirlər istinad üzrə yaxud qiymət üzrə müqayisə edilir?

```
// Sətirlərin istinad yaxuddakı qiymət üzrə müqayisə ediləcəyini müəyyən
// etmək, kifayət qədər sadədir. Burada eyni simvolların ardıcıllıqlarını
// ehtiva edən tamamilə müxtəlif sətirlər müqayisə edilir
// Əgər qiymət üzrə müqayisə yerinə yetirilirsə
// onlar ekvivalent kimi təfsir edilməlidir, əgər
// istinad üzrə müqayisə yerinə yetirilərsə, nəticə əksi olmalıdır:

var s1 = "hello";
var s2 = "hell" + "o";
if (s1 == s2) document.write ("Sətirlər qiymət üzrə müqayisə edilir")
```

Bu sınaq sübut edir ki, sətirlər qiymət üzrə müqayisə edilir. Bu, C, C++ və Java ilə işləyən bəzi proqramçılar üçün sürpriz ola bilər, yəni, sətirlər sitat tiplərinə aiddir və istinad üzrə müqayisə edilir. Zəruri olduqda bu dillərdə sətirlərin faktiki tərkibi müqayisə etmək üçün xüsusi metodlardan və ya funksiyalardan istifadə etmək lazım gəlir. JavaScript dili yüksək səviyyəli dillərə aiddir və buna görə güman edilir ki, sətirlərin müqayisəsi yerinə yetirildikdə, ehtimal ki, qiymət üzrə müqayisə nəzərdə tutulur. Baxmayaraq ki, JavaScript-də effektivliyinin artırılması məqsədilə, sətirlər istinad üzrə yamsılanırlar və ötürülürlər, bununla belə müqayisə əməliyyatı qiymət üzrə yerinə yetirilir.

3.15.4. İstinad üzrə və ya qiymət üzrə: yekunlaşdırma

Cədvəl 3.5-də JavaScript-də müxtəlif məlumat tiplərinin üzərində əməliyyatlar yerinə yetirməsinin qısa təsviri verilmişdir.

Cədvəl 3.5. *JavaScript-də məlumat tiplərinin üzərində əməliyyatlar*

Tip	***Kopyalanma***	***Ötürülmə***	***Müqayisə***
Ədəd	Qiymət üzrə	Qiymət üzrə	Qiymət üzrə
Məntiqi qiymət	Qiymət üzrə	Qiymət üzrə	Qiymət üzrə
`Obyekt`	İstinad üzrə	İstinad üzrə	İstinad üzrə
`Sətir`	İstinad üzrə	İstinad üzrə	İstinad üzrə

4

Dəyişənlər

Dəyişən – qiymətlə ifadə olunan addır. Yəni, dəyişən hər hansı bir qiymət ehtiva edir. Dəyişənlər məlumatları proqramda saxlamağa və bu məlumatlarla işləməyə imkan verir. Məsələn, aşağıdakı JavaScript-kodunda i adında olan dəyişənə 2 qiyməti mənimsədilir:

```
i = 2;
```

Aşağıdakı kodda isə, növbəti i dəyişəninə 3 qiyməti əlavə edilir və yeni dəyişənə sum nəticəsini mənimsədilir:

```
var sum = i + 3;
```

Bu demək olar ki, bütün dəyişənlər haqqında ilkin təsəvvürü formalaşdırır. Amma JavaScript-də dəyişənlərin işinin mexanizmini tam anlamaq üçün bu iki sətir kod ilə kifayətlənməmək, başqa anlayışları da öyrənmək lazımdır. Bu fəsildə tipləşdirmə, elanlar, görünmə sahələri, tərkib və dəyişən adlarının təyini, həmçinin tullantı tənzimləmələri və "dəyişən/xüsusiyyət"[10] anlayışının müxtəliflikləri məsələlərinə toxunalacaq.

4.1. Dəyişənlərin tipləşdirməsi

JavaScript-in Java və C kimi və digər bu tip proqramlaşdırma dillərindən əsas fərqi ondan ibarətdir ki, JavaScript – ***tipləşdirilməmiş (untyped)*** dildir. Xüsusi halda, bu o deməkdir ki, JavaScript, Java və C ailəsinə yaxın PD-lərdər fərqli olaraq, dəyişənin elanı zamanı dəyişənin ehtiva edəcəyi məlumat tipi göstərilmədən, bu dəyişən istənilən tipdə olan qiyməti ehtiva edə bilər. Məsələn, JavaScript-də məlumat tipi göstərilmədən dəyişənə əvvəlcə ədəd tipində sonra isə, sətir tipində olan qiymət mənimsətmək olar:

```
i = 10;
i = "on";
```

Java, C, C++ və digər istənilən ciddi tipləşdirilmiş dildə belə kod yolverilməzdir.

Tipləşdirmənin yoxluğu JavaScript-ə axıcılıq xüsusiyyəti yaradır və bu dil ehtiyac olduğu halda qiymətləri bir tipdən digər bir tipə avtomatik dəyişdirəcək qədər sadədir. Məsələn, əgər siz sətirə tipinə ədəd tipində olan qiymət yazmaq istəyirsinizsə, JavaScript uyğun olan sətiri ədədə avtomatik dəyişdirəcək. Tiplərin dəyişiklikləri barədə daha ətraflı 3-cü fəsildə danışılır. JavaScript dilinin tipləşdirilməmiş xüsusiyyəti, tipləşdirilmiş dillər (C, C++, Java və s.) ilə müqayisədə bu dilinin sadəliyini bildirir. Eyni zamanda bir çox JavaScript- proqramları qısa ssenariləri təşkil olunur, buna görə də belə ciddi tip xüsusiyyətinə ehtiyac yoxdur və adətən bu dildə işləyən proqramçılar daha sadə sintaksisin üstünlük verirlər.

4.2. Dəyişənlərin elan edilməsi

JavaScript-də dəyişəndən istifadə etməzdən əvvəl, onu *elan etmək*[11] lazımdır. Dəyişənlər aşağıdakı qaydada var açar sözünün köməyilə elan edilir:

```
var i;
var sum;
```

Bir neçə dəyişən elan etmək olar:

```
var i, sum;
```

Bundan başqa, dəyişənin elanını onun inisializasiyası ilə birləşdirmək olar:

```
var message = "hello";
var i = 0, j = 0, k = 0;
```

Əgər var təlimatında ilk məna verilməmişdirsə, onda dəyişən elan edilir, amma onun ilk mənası qeyri- müəyyən (*undefined*) qalır. Nəzərinizə çatdıraq ki, var təlimatı həmçinin *for* və *for/in* dövrlərinə (bu barədə 6-cı fəsildə daha ətraflı danışılır) qoşula bilər, yəni bilavasitə dövrün özündə dəyişənini dövr elan etməyə imkan verir. Məsələn:

```
for (var i = 0; i<10; i++)  document.write(i, "</br>");
for (var i = 0, j = 10; i<10; i++, j--) document.write(i*j, "
</br>");
for (var i in o) document.write(i, "</br>");
```

var təlimatının köməyilə elan edilmiş dəyişənlər, “uzunömürlü” (permanent) olur: (onları delete operatorunun köməyilə silmək cəhdi səhvə gətirib çıxaracaq. (delete operatoru ilə 5-ci fəsildə tanış olacaqsınız.)

4.2.1. Təkrar və "unudulmuş" elanlar

var təlimatının köməyilə eyni anda bir neçə dəyişəni elan etmək olar. Əgər təkrar elan edilmiş dəyişən, inisializator ehtiva edirsə, onda o adi mənimsəmə təlimatı kimi davranır.
Əgər elan edilməyən dəyişənin qiymətini oxumağa çalışsaq, JavaScript xəta barədə məlumat yaradacaq. Əgər var təlimatının köməyilə elan edilməyən dəyişənə qiymət mənimsətsək, JavaScript sizin üçün bu dəyişəni gizli olaraq özü elan edəcək. Ancaq belə dəyişənlər, hər bir halda, hətta funksiyanın gövdəsində belə, qlobal dəyişən kimi elan edilir. Lakin funksiya üçün qlobal dəyişənlərdən istifadə məsləhət görülmür, çünki bu proqramda ad fəzasının dolmasına gətirib çıxarır. Ona görə də, funksiyanın gövdəsində var təlimatının köməyilə lokal dəyişənlərin elan edilməsi birmənalı olaraq məsləhət görülür. Ümumiyyətlə isə, görünmə sahəsindən asılı olmayaraq bütün dəyişənləri var açar sözü ilə elan etmək qəti şəkildə məsləhət görülür.

4.3. Dəyişənin görünmə sahəsi

Dəyişənin görünmə sahəsi (scope) – dəyişən müəyyən edilmiş proqram hissəsidir.

Qlobal dəyişən bütün JavaScript-proqramı üçün müəyyən edilmiş qlobal görünmə sahəsinə malikdir. Funksiyanın daxilində elan edilmiş dəyişənlər, yalnız funksiyanın gövdəsində müəyyən edilmişdir. Bu tip dəyişənlər lokal dəyişənlər adlandırılır və lokal görünmə sahəsinə malikdirlər.

Həmçinin, funksiyaların parametrləri yalnız bu funksiyanın gövdəsi daxilində müəyyən edilmiş ***lokal dəyişən*** hesab edilir. Funksiyanın gövdəsi daxilində lokal dəyişən eyni adlı qlobal dəyişəndən prioritet cəhətdən üstündür. Əgər lokal dəyişənin adı, qlobal dəyişənin adı ilə eyni elan edilibsə, faktiki olaraq qlobal dəyişən gizlədiləcək. Bunu aşağıdakı nümunə kod ilə belə göstərmək olar:

```
var scope = "qlobal";        // Qlobal dəyişənin elan edilməsi
function checkscope()
{
  var scope = "lokal";         // Eyni adlı lokal dəyişənin
elan edilməsi
  document.write(scope) ;      // Lokal dəyişəndən istifadə
olunur.
}
checkscope();           // ekranda "lokal" sözü yazılır
```

Qlobal görünmə sahəsi ilə dəyişənləri elan edəndə, var təlimatını ixtisar etmək olar, lakin lokal dəyişənlərin elanı zamanı var təlimatından istifadə etmək mütləqdir. Aşağıdakı nümunəyə baxaq:

```
scope = "qlobal";              // hətta "var"-sız qlobal
dəyişənin elan
                          // edilməsi
function checkscope()
{
   scope = "lokal";              // Oy! Biz indi qlobal
dəyişəni dəyişdirdik
   document.write(scope);        // Qlobal dəyişən istifadə
olunur
   myscope = "lokal";            // Burada biz gizli yeni
qlobal dəyişəni elan
                             // edirik
   document.write(myscope) ;   // Yeni qlobal dəyişən istifadə
olunur
```

```
}

checkscope();                          // Ekranda "lokallokal"
birləşməsi yazılır
document.write(scope) ;              // Ekranda "lokal" sözü
yazılır
document.write(myscope) ;            // Ekranda "lokal" sözü
yazılır
```

Funksiyalar, bir qayda olaraq, dəyişənlərin hansı görünmə sahələrində elan edildiyini və ya dəyişənin hansı məqsədlə istifadə edildiyini bilmir. Buna görə də funksiya lokal və qlobal dəyişəndən birlikdə istifadə edir və proqramın funksiyadan kənar digər hissəsində lazım olan dəyişənin qiymətini dəyişdirməyə risk edir. Xoşbəxtlikdən, bu xoşagəlməz hadisədən qaçmaq asandır: bunun üçün sadəcə olaraq, bütün dəyişənləri var təlimatının köməyi ilə elan edin.
Funksiyaların daxilində də funksiya təyin edilə bilər. Hər bir funksiya şəxsi lokal görünmə sahəsinə malikdir, buna görə də lokal görünmə sahələrinin bir neçə daxili səviyyəsi ola bilər. Məsələn:

```
var scope = "qlobal görünmə sahəsi";          // Qlobal
dəyişən
function checkscope()
{
  var scope = "lokal görünmə sahəsi";           // Lokal
dəyişən
  function nested()
  {
     var scope = "daxili görünmə sahəsi";          // Daxili
görünmə sahəsində
                                              // lokal dəyişənlər
     document.write(scope) ;                      // "daxili
görünmə sahəsi"
                                              // yazılır
  }
  nested();
}
checkscope();
```

4.3.1. Blok görünmə sahəsinin yoxluğu

Nəzərə alın ki, C, C ++ və Java-dan fərqli olaraq, JavaScript-də bloklar səviyyəsində görünmə sahələri mövcud deyil. Funksiyanın daxilində

olan və elan edilmə necəliyindən asılı olmayaraq bütün dəyişənlər funksiyanın daxilində müəyyən edilir. Aşağıdakı kodda i, j və k dəyişənləri eyni görünmə sahələrinə malikdir: üç dəyişənın hamısı bütün funksiya gövdəsində müəyyən edilmişdir. Bu cür kod C, C++ və ya Java yazıla bilməz:

```
function test(o)
{
   var i = 0;       //i bütün funksiyada müəyyən edilmişdir
   if (typeof o == "object")
   {
     var j = 0;      //j tək blokda deyil, hər yerdə müəyyən
edilmişdir
     for (var k = 0; k > 10; k ++)
     {                      // k tək dövrdə deyil hər yerdə
müəyyən edilmişdir
        document.write(k);
     }
     document.write(k);         //k dəyişənindən hələ də
istifadə etmək olar:
                               //10 yazılır
   }
   document.write(j);  //j müəyyən edilmişdir, lakin
inisializasiya
                          //edilməmiş ola bilər
}
```

Funksiyada elan edilmiş dəyişənlərin belə davranışı təəccüblü nəticələrə gətirib çıxara bilər. Məsələn:

```
var scope = "qlobal";
function f()
{
  alert(scope);              // "undefined" və "qlobal olmayan
" yazılır.
  var scope = "lokal";       // Dəyişən burada inisializasiya
olunur, amma yalnız
                     // funksiyanın daxilində müəyyən
edilmişdir.
  alert(scope);              // "lokal" göstərir
}
f();
```

Kimsə düşünə bilər ki, *alert()*-in birinci çağırışı nəticəsində ekranda "qlobal" sözü yazılacaq, çünki lokal dəyişənini elan edən var təlimatı,

hələ yerinə yetirilməmişdir. Ancaq sahələrin təyini qaydasına əsasən görünüşdə bu belə olmur. Lokal dəyişən funksiyanın bütün gövdəsində müəyyən edilmişdir, deməli, həminki eyni qlobal dəyişən funksiyanın bütün gövdəsində gizlədilmişdir. Hərçənd ki, lokal dəyişən var təlimatının icrasına qədər hər yerdə müəyyən edilmişdir, lakin o inisializasiya edilməmişdir. Buna görə də əvvəlki nümunədə *f* funksiyası aşağıdakı koda ekvivalentdir:

```
function f()
{
  var scope;                // Lokal dəyişən funksiyanın
başlanğıcında təyin
                    // edilir
  alert(scope);            // Burada dəyişən mövcuddur, lakin
undefined
                         // qiymətinə malikdir
  scope = "lokal";          // Burada biz dəyişəni
inisializasiya edirik və ona
                    // qiymət mənimsədirik
  alert(scope);             // Burada artıq dəyişən qiymətə
malikdir
}
```

4.3.2. Qeyri-müəyyən və inisializasiya edilməyən dəyişənlər

Əvvəlki bölmədəki nümunələr JavaScript-in incə proqramlaşdırma vəziyyətini təsvir edir: qeyri-müəyyən dəyişənlərin iki növü var. Birincisi – heç bir yerdə elan edilməyən dəyişəndir. Elan edilməmiş dəyişənin qiymətini oxumaq cəhdi yerinə yetirilmə zamanının səhvə gətirib çıxaracaq. *Elan edilməyən dəyişənlər* müəyyən edilmir, çünki onlar sadəcə olaraq mövcud deyil. Artıq deyildiyi kimi, elan edilməyən dəyişənə qiymət mənimsədilməsi səhv deyil – sadəcə bu dəyişən mənimsədilmə zamanı gizli qlobal dəyişən kimi elan edilir.
Qeyri-müəyyən dəyişənlərin ikinci növü – elan edilmiş dəyişənlərdir, lakin bu dəyişənlərə heç yerdə qiymət mənimsədilməmişdir. Əgər bu tip dəyişənlərin qiyməti oxunarsa, susmaya görə bu dəyişənin qiyməti `undefined` olacaq. Belə dəyişənləri digər dəyişənlərdən

fərqləndirmək üçün bu dəyişənlər *inisializasiya edilməmiş (unassigned)* adlandırılır. Aşağıdakı kodda qeyri-müəyyən və inisializasiya edilməyən dəyişənlər arasında bəzi fərqlər təsvir edilir:

```
var x;          // İnisializasiya edilməyən dəyişən elan
edirik. Dəyişən
                // undefined qiymətinə malikdir.
alert(u); // Elan edilməyən dəyişəndən istifadə səhvə gətirib
çıxaracaq.
u = 3;          // Elan edilməyən dəyişənə qiymət
mənimsədilərkən, həm də bu
                // dəyişən yaradılmış olur.
```

4.4. Elementar və sitat tipləri

Növbəti mövzuda biz dəyişənlərin tərkibi haqqında bəhs edəcəyik. Biz tez-tez deyirik ki, dəyişənlər qiymətləri özündə saxlayır. Həqiqətəndə bu belədir? Bu suala cavab vermək üçün, biz məcburuq JavaScript-də dəstəklənən məlumat tiplərinə yenidən baxaq. Bu tiplər iki qrupa bölünür: elementarlar və sitatlar.

Ədədlər, məntiqi qiymətlər, həmçinin `null` və `undefined` qiymətləri elementar tiplərdir. Obyektlər, massivlər və funksiyalar sitat tipləridir.

Elementar tip təsbit edilmiş ölçüyə malikdir. Məsələn, ədəd səkkiz bayt tutur, məntiqi qiymət isə cəmi bir bit ilə təqdim edilə bilər.

Ədəd tipi – elementar tiplərin ən böyüyüdür. JavaScript-də hər bir dəyişən üçün yaddaşda səkkiz bayt ehtiyat saxlanıldığına, dəyişən istənilən elementar tipdə olan qiymətini bilavasitə ehtiva edə bilər.[12]

Ancaq sitat tiplərində başqa cürdür. Məsələn, obyektlər hər hansı uzunluqda ola bilər – onlar təsbit edilmiş ölçüyə malik deyillər. Bu fikiri massivlərə də aid etmək olar: massiv, istənilən qədər elementə malik ola bilər. Bu fikir funksiya üçün də analojidir, yəni funksiya istənilən həcmdə olan JavaScript-kodu ehtiva edə bilər. Bir halda ki, bu tiplər təsbit edilmiş ölçüyə malik deyil, onda onların qiymətləri bilavasitə hər bir dəyişənin yaddaşında təyin olunmuş səkkiz baytda saxlanıla bilməz.

Buna görə də dəyişəndə belə qiymətlər istinad üzrə saxlanılır. Adətən bu istinad yaddaşda hər hansı göstəricidən və ya ünvandan təşkil olunur.
İstinad *– qiymət deyil*. Lakin dəyişənə *qiyməti harada tapmaq olduğunu* bildirir. Elementar və sitat tipləri arasında əsas fərq onların davranışındadır.
Ədədlər ilə əməliyyat edən aparan (elementar tip) aşağıdakı koda baxaq:

```
var a = 3.14;          // Dəyişənin elan edilməsi və
inisializasiyası
var b = a;             // Yeni dəyişənə ilk dəyişənin
qiymətinin kopyalanması
a = 4;              // İlk dəyişənin qiymətinin modifikasiyası
alert(b)               // 3.14 göstərilir; surət dəyişmədi
```

Bu kodda qeyri-adi heç nə yoxdur. Bəs əgər ədədləri massivlərlə (sitat tipi) əvəz edib kodu azca dəyişdirsək onda necə olar:

```
var a = [1,2,3]; // Dəyişəni massivə istinadla inisializasiya
edirik
var b = a;             // Yeni dəyişənə də bu istinada
köçürürük
a [0] = 99;          // İlkin istinaddan istifadə edərək
massivi dəyişdiririk
alert(b);              // yeni istinaddan istifadə edərək
dəyişdirilmiş massivi,
                  // yəni [99,2,3] göstəririk
```

Bu nəticə kimə qaranlıq qalmadısa, demək həmin şəxs artıq elementar və sitat tiplərinin arasında fərqi anlamışdır. Bu tiplər arasında fərqi anlamayanlar isə kodun ikinci sətirini diqqətlə nəzərdən keçirsinlər. Nəzərə alın ki, bu təklifdə massivin özünə deyil, "massiv" tipinin qiymətinə istinadın mənimsədilməsi yerinə yetirilir. Kodun ikinci sətirindən sonra biz hamımız hələ bir obyektə malikik massiv; yalnız biz ona iki istinad almağı bacarırıq. Dəyişənlər elementar tiplərin faktiki qiymətlərini, sitat tiplərinin istinadlarını özündə saxlayır. Baza və sitat tiplərinin müxtəlif davranışı daha ətraflı bölmə 3.15-də öyrənilir.
Siz qeyd edə bilirdiniz ki, bəs JavaScript-də sətirlərin elementar və ya sitat tiplərinə aid olduğu barədə bəhs olunmadı.
Sətirlər – qeyri-adi davranışa malikdir. Onlar dəyişən uzunluğuna malikdir və buna görə, görünür ki, bilavasitə bərkidilmiş ölçülü

dəyişənlərində saxlanıla bilmir. Effektivlik nöqteyi-nəzərində gözləmək olar ki, JavaScript interpretatoru sətirləri istinadlara köçürəcək və onlar faktiki olaraq tərkib sayılmır. Eyni zamanda bir çox münasibətlərdə sətirlər özünü elementar tiplər kimi aparır. Bu cür sadələşməyə, JavaScript-in faktiki realizasiyasının təsviri kimi baxmağa lüzum yoxdur. Burada sual oluna bilər ki, bəs sətirlər konkret olaraq hansı tipə aiddir? Bu haqda fikir söyləmək çətindir və bu biraz mübahisəli məsələdir, çünki əslində sətirlər dəyişməzdir: bu dilin, sətir qiymətinin daxilində sətirin tərkibini dəyişdirmək imkanı yoxdur. Bu o deməkdir ki, əvvəlki oxşar nümunəni, yəni massivlərin istinad üzrə yamsılanmasını bu tipdə təşkil etmək olmaz. Son olaraq, sətirlərə, özünü elementar tip qismində aparan dəyişməz sitat tipi kimi və ya sitat tipinin mexanizmindən istifadəyə realizasiya edilmiş elementar tip kimi baxmaq olmaz.

4.5. Tullantılar dəsti

Sitat tipləri təsbit edilmiş ölçüyə malik deyil; buna görə, onlar çox böyük ölçüyə malik ola bilər. Qeyd edildiyi kimi, dəyişənlər sitat tipində olan qiymətləri bilavasitə özündə saxlamır. Qiymətlər hər hansı bir yerdə saxlanılır, dəyişənlərdə isə yalnız bu yerə olan istinad saxlanılır. İndi isə, qiymətlərin real saxlamasının qısa icmalını verəcəyik.

Bir halda ki, sətirlər, obyektlər və massivlər təsbit edilmiş ölçüyə malik deyil, onların məlum ölçüsünün saxlanması üçün dinamik yer seçilməlidir. JavaScript-proqramında sətir, massiv və ya obyekt yaradıldıqda, interpretator bu məzmunun saxlanılması üçün yaddaş ayırmalıdır. Ayırılan yaddaş boş olmalıdır, belə olmadıqd JavaScript interpretatoru bütün mümkün yaddaşdan istifadə edəcək, yaddaş bitdikdə isə, sistemin imtinasına gətirib çıxaracaq.

C və C ++ kimi dillərdə, yaddaşı əl ilə boşaltmaq lazımdır. Məhz proqramçı bütün yaradılan obyektlərin izlənilməsinə və bu obyektlər tələb olunmadığı halda onların silinməsinə (yaddaşın azad olmasına) cavabdeh olmalıdır. Bu kifayət qədər ağır prosesdir və tez-tez səhvlərə yol açır. JavaScript-də isə ***tullantı kolleksiyası*** *(garbage collection)* adlandırılan texnologiya sayəsində, yaddaşı əl ilə boşaltmağa ehtiyac

yoxdur. JavaScript interpretatoru proqramda istifadə olunmayacaq hər hansı bir obyekti aşkar edə bilər. İnterpretator müəyyən edə bilər ki, obyekt əlçatmazdır (yəni, obyektə istinad oluna bilməz), interpretator müəyyənləşdirə bilər ki, obyekt daha lazım deyil və bu obyektin tutduğu yaddaş boşaldılacaq. Məsələn aşağıdakı koda baxaq:

```
var s = "hello";                    // Sətir üçün yaddaş
ayırırıq
var u = s.toUpperCase();            // Yeni sətir yaradırıq
s = u;                             // İlkin sətirə olan istinadı
köçürdürük
```

Bu kodun icrasından sonra ilk "hello" sətiri daha əlçatmazdır, yəni proqramda bir daha bu dəyişənə istinad olunmur. Sistem bu faktı müəyyən edir və yaddaşı boşaldır.
Tullantıların tənzimlənməsi avtomatik yerinə yetirilir və proqramçı üçün bu proses gizlidir. Kod düzgün yazılmalıdır ki, tullantı tənzimlənməsi ehtiyacları ayırd edə bilsin və hər köhnə obyekti silməsin.

4.6. Xüsusiyyət rolunda olan dəyişənlər

Siz artıq qeyd edə bilərsiniz ki, JavaScript-də dəyişən və obyekt xüsusiyyətləri arasında çox oxşarlıq var. Buna eyni yolla qiymətin mənimsədilmə, JavaScript-ifadələrinə eyni yolla tətbiq edilmə və s. kimi nümunələr gətirmək olar. Bəs, dəyişən və o obyekt xüsusiyyətləri arasında hər hansı prinsipial fərq varmı? Cavab: heç bir fərq yoxdur. JavaScript-də dəyişənlər obyekt xüsusiyyətlərindən prinsipial olaraq fərqlənmir.

4.6.1. Qlobal obyekt

İstənilən kodun icrasından əvvəl işə salma zamanı JavaScript interpretatoru tərəfindən yerinə yetirilən ilk davranışlardan biri – qlobal obyektin yaradılmasıdır. Bu obyektin xüsusiyyətləri JavaScript-proqramlarında qlobal dəyişənlərdən təşkil olunub. JavaScript-də qlobal dəyişən elan etdikdə, faktiki olaraq siz qlobal obyekt xüsusiyyətini müəyyən edirsiniz.

JavaScript interpretatoru bir sıra qlobal obyektin xüsusiyyətlərini, istinadları, qiyməti qabaqcadan müəyyən edilmiş və funksiyalar inisializasiya edir. Belə ki, qabaqcadan müəyyən edilmiş `Infinity` ("sonsuzluq") ədədinə, `parseInt` və `Math` xüsusiyyətləri isə, qabaqcadan müəyyən edilmiş obyektə istinad edir. Bu barədə daha ətraflı kitabın üçüncü hissəsində, qlobal qiymətlər haqqında olan bəhsdən oxumaq olar. Yuxarı səviyyəli kodda (yəni, funksiya hissəsi olmayan JavaScript-kodu) qlobal obyektə `this` açar sözü vasitəsi ilə istinad etmək mümkündür. Funksiyaların daxilində isə `this` açar sözü başqa cür tətbiq olunur. Bu barədə daha ətraflı 8-ci fəsildə tanış ola bilərsiz.
JavaScript-in kliyent dilində bütün JavaScript-kodu üçün qlobal obyekt sayılan və brauzer pəncərəsinə uyğun olan `Window` obyekti mövcuddur. Özü qlobal obyektə istinad edən bu qlobal obyekt `window` xüsusiyyətinə malikdir hansı ki, qlobal obyektə istinad üçün `this` açar sözü ilə birlikdə istifadə etmək olar. `Window` obyekti `parseInt` və `Math` kimi baza qlobal xüsusiyyətlərini, həmçinin navigator və screen kimi qlobal kliyent xüsusiyyətlərini müəyyən edir.

4.6.2. Lokal dəyişənlər - çağırış obyekti

Əgər qlobal dəyişənlər – xüsusi qlobal obyektin xüsusiyyətləri olursa, onda bəs lokal dəyişənlərdə bu proses necə olur? Lokal dəyişənlər də həmçinin obyekt xüsusiyyətlərindən təşkil olunub. Bu obyekt *çağırış obyekti* (***call object***) adlanır. Funksiyanın gövdəsi yerinə yetirilən zaman, funksiyada mövcud olan funksiyalar, arqumentlər və lokal dəyişən bu obyektin xüsusiyyətləri kimi saxlanılır. JavaScript-də lokal dəyişənlər üçün tamamilə ayrı obyektdən istifadə edildiyinə görə qlobal dəyişənlərlə ad münaqişəsi yarana bilməz.

4.6.3. JavaScript-də icra kontekstləri

Funksiya icra edilməyə başlayararkən, JavaScript interpretatoru funksiya üçün *icra kontekstini* (***execution context***) yaradır, yəni burada JavaScript-kodda mövcud olan ixtiyari fraqment yerinə yetirilir. Kontekstin mühüm

hissəsi – dəyişənlərin müəyyən edildiyi obyektdir. Buna görə də, JavaScript-proqramının kodu, hər hansı funksiya hissəsi olmayan, dəyişən təyin olunduğu qlobal obyektin icra kontekstində işləyir. JavaScript-funksiyaları isə lokal dəyişənlərin müəyyən edildiyi çağırış obyektinin şəxsi unikal icra kontekstində işləyir.

Qeyd etmək lazımdır ki, bəzi JavaScript realizasiyalarında qlobal obyektlərin ayrılıqda bir neçə qlobal icra konteksti mövcud ola bilər (Hərçənd bu halda hər bir qlobal obyekt tam olar qlobal sayılmır)[13]. Məsələn – JavaScript kliyentində, brauzerin ayrılıqda hər bir pəncərəsi və ya pəncərədəki hər bir çərçivə ayrılıqda qlobal icra kontekstini müəyyən edir. JavaScript kliyent kodu hər bir çərçivədə və ya pəncərədə şəxsi icra kontekstində yerinə yetirir və şəxsi qlobal obyektə malikdir. Ancaq ayrı-ayrı olan bu kliyent qlobal obyektlərin xüsusiyyətləri bir-birilə əlaqəlidir. Başqa sözlə, JavaScript-kodu `parent.frames[1]` ifadəsinin köməyi ilə bir çərçivədən digər çərçivəyə istinad edə bilər və birinci çərçivədə olan `x` qlobal dəyişəninə `parent.frames[0].x` ifadəsinin köməyilə ikinci çərçivədən istinad etmək olar.

JavaScript-kliyentində ayrı-ayrı olan pəncərələrin və çərçivələrin icra kontekstlərinin necə əlaqələndiyini tam anlamanıza hal-hazırda ehtiyac yoxdur. Bu mövzuya kitabın II hissəsində toxunacağıq. İndi bilmək kifayətdir ki, dilin elastikliyi sayəsində, bir JavaScript interpretatoru ssenaridə müxtəlif qlobal kontekstlərində icra etməyə imkan verir və bu kontekstləri bir-birindən ayırmağa ehtiyac yoxdur – onlar bir-birinə istinad edə bilər.

Gəlin, bu son mülahizəyə daha ətraflı aydınlıq gətirək. Əgər JavaScript bir icra kontekstindən digər icra kontekstindəki kodu oxuya və xüsusiyyətlərə qiymətlər yaza və buradakı müəyyən edilmiş funksiyaları yerinə yetirə bilərsə onda təhlükəsizlik məsələləri aktuallaşır. Nümunə kimi JavaScript-kliyentini götürək. Fərz edək ki, brauzerinin A pəncərəsində ssenarini icra edilir və ya bu pəncərə lokal şəbəkənizdə olan hər hansı informasiyanı ehtiva edir, B pəncərəsində isə internet mənbəli bir neçə sərbəst saytdan ssenari icra edilir. Biz A pəncərəsindəki xüsusiyyətlərə B pəncərəsində olan koddan girişi məhdudlaşdırmalıyıq. Axı, belə olmayan halda yad koddan korporativ

əhəmiyyətlii informasiyanı oxumaq və oğurlamaq mümkün olacaq. Beləliklə, JavaScript-kodunun təhlükəsiyini təmin etmək üçün bir icra kontekstindən digərinə girişi məhdudlaşdıran xüsusi mexanizm yaradılmalıdır.

4.7. Bir daha dəyişənlərin görünmə sahəsi haqqında

Biz ilk dəfə dəyişənin görünmə sahəsi anlayışını müzakirə edərkən, görünmə sahəsini yalnız JavaScript-kodun leksik strukturu əsasında müəyyən etdik: qlobal dəyişənlər qlobal, funksiyada elan edilmiş dəyişənlər isə lokal görünmə sahəsinə malikdir. Əgər funksiyanın daxilində başqa funksiya təyin edilmişdirsə, onda bu daxili funksiyada elan edilmiş dəyişənlər, daxili lokal görünmə sahəsinə malikdir. İndi, biz bilirik ki, qlobal dəyişənlər qlobal obyektin, lokal dəyişənlər isə xüsusi çağırış obyektinin xüsusiyyətlərini müəyyən edir, ona görə də biz görünmə sahəsi anlayışına yenidən qayıda bilərik. Bu bizə dəyişənlərin bir çox kontekstlərdə mövcudluğunu və JavaScript-in necə işlədiyi daha dərin anlamağa imkan verəcək.

JavaScript-də hər bir icran konteksti siyahıdan, "zəncirdən"[14] və obyektdən təşkil olunan *görünmə sahələri zəncirinə* ***(scope chain)*** bağlıdır. obyektlər. JavaScript-koda *x* dəyişəninin qiymətini tapmaq tələb olunan zaman (bu proses *dəyişən adına icazə* adlanır), zəncirdə axtarış ilk olaraq obyektdən başlayır. Əgər obyektdə *x* adlı xüsusiyyət tapıldısa, onda bu xüsusiyyətin qiymətindən istifadə olunur. Əgər birinci obyektdə *x* adlı xüsusiyyəti tapılmasa, onda JavaScript axtarışı zəncirin növbəti obyektində davam etdirir. Əgər ikinci obyektdə də *x* adıyla xüsusiyyət tapılmasa, axtarış növbəti obyektdə davam edir və proses iyerarxik formalaşır.

JavaScript-də yuxarı səviyyəli kodun (funksiya hissəsi olmayan kod), görünmə sahələri zənciri yalnız qlobal obyektdən ibarətdir. Bütün dəyişənlər bu obyektdə axtarılıb tapılır. Əgər dəyişən mövcud deyilsə, onda qiymət `undefined`-ə bərabərdir. Funksiyada görünmə sahələrinin zənciri iki obyektdən ibarətdir. Funksiya dəyişənə istinad

edərkən, ilk növbədə çağırış obyekti (lokal görünmə sahəsi), ikinci növbədə – qlobal obyekt (qlobal görünmə sahəsi) yoxlanılır. Daxili funksiya görünmə sahələri zəncirində üç və ya daha çox obyektə malik olur. Funksiyanın görünmə sahələri zəncirində dəyişən adının axtarılması prosesi aşağıdakı şəkildə təsvir edilir.

Dəyişənin axtarışı
Görünmə sahələri zənciri
Leksik görünmə sahəsi

müəyyən edilməyib
Yox
qlobal obyekt
Hə
qiymətin alınması

y:2

```
var x=1;

function f(){
z=2;


}


function f() {
y=2;









}
```

z:3
x:1

burada müəyyən edildi?

Yox

Hə
qiymətin alınması
g() funksiyasının çağırış obyekti

burada müəyyən edildi?

Yox

Hə
qiymətin
alınması
g() funksiyasının
çağırış obyekti

burada müəyyən
edildi?

BAŞLA

Şəkil 4.1. *Dəyişən adının görünmə sahələri və icazənin zənciri*

5

İfadələr və operatorlar

Bu fəsildə biz, JavaScript-də olan ifadələr və operatorlar ilə tanış olacağıq. C, C++ və ya Java proqramlaşdırma dillərindən anlayışı olan bu fəsil ilə ötəri tanış olacaq, çünki JavaScript PD-nin operatorları sözügedən PD-lərin operatorları ilə çox oxşardır. Əgər operatorlar və ifadələr barədə anlayışınız yoxdursa, (və ya azdırsa) bu fəsillə dərindən tanış olmağınız məsləhətdir.

5.1. İfadələr

İfadə – qiymət almaq üçün interpretator tərəfindən hesablana bilən JavaScript dilinin frazasıdır. Ən sadə ifadələr – literal və ya dəyişənlərin adlarıdır, məsələn:

```
1.7                                 // Ədəd literalı
"JavaScript is fun!"              // Sətir literalı
true                                  // Məntiqi qiymət literalı
null                                  // null qiymətinin literalı
/java/                              // Requlyar ifadə literalı literalı
{ x:2, y:2}                       // Obyekt literalı
[2,3,5,7,11,13,17,19]          // Massiv literalı
function (x){ return x*x;}     // Funksional literal
i                                     // i dəyişəni
sum                                 // sum dəyişəni
```

İfadənin literal qiyməti *- sadəcə öz literalının qiymətidir*.
İfadə-dəyişənin qiyməti *– dəyişənin və ya dəyişənin istinad etdiyi qiymətdir.*
Bu ifadələr o qədər də maraqlı deyil. Sadə ifadələrin birləşməsi yolu ilə daha mürəkkəb (maraqlı) ifadələr yaradıla bilərlər ifadələr. Məsələn, biz gördük ki, 1.7 və i ifadədir. Aşağıdakı nümunə də ifadə sayılır:

```
i + 1.7
```

Bu ifadənin qiyməti iki və ya daha çox sadə ifadənin qiymətinin toplanması yolu ilə təyin edilir. Bu nümunədə + (üstəgəl) simvolu – ifadələr arasında toplama əməliyyatını yerinə yetirən toplama operatordur. Digər operator isə ifadələr arasında çıxma əməliyyatını yerinə yetirən çıxma ("minus") operatorudur. Məsələn:

```
(i + 1.7) - sum
```

Bu ifadədə "minus" operatoru əvvəlki *i + 1.7* ifadəsinin qiymətindən *sum* dəyişənin qiymətinin çıxılması üçün tətbiq edilir.
Növbəti bölmədə siz, JavaScript-in digər operatorları ilə tanış olacaqsınız.

5.2. Operatorların icmalı

Əgər siz C, C++ və ya Java proqramlaşdırma dilləri ilə tanışsızsa, onda JavaScript-operatorların əksəriyyəti artıq sizə məlumdur. Bu operatorlar cədvəl 5.1-də göstərilmişdir. Nəzərə alın ki, operatorların əksəriyyəti punktuasiya simvolları ilə (+, -,=, ...), bəziləri isə – açar sözlər (delete və instanceof) ilə təsvir edilir. Açar sözlər və punktuasiya simvolları adi operatorları ifadə edir, sadəcə birinci halda daha geniş və asan oxunan sintaksis alınır. Bu cədvəldə "P" hərfi ilə göstərilmiş sütun operatorun prioritetini bildirir, "A" hərfi ilə göstərilmiş sütun isə – operatorun assosiativliyini (L – soldan sağa və ya R – sağdan sola) bildirir. Bunlar barədə anlayışı olmayan narahat olmasın, biz, növbəti yarımfəsillərdə izah edəcəyik.

P	A	Operator	Operand tipi	Yerinə yetirilən əməliyyat
15	L	.	Obyekt, identifikator	Xüsusiyyətə Müraciət
	L	[]	Massiv, tam ədəd	Massivin indeksləşdirməsi
	L	()	Funksiya, arqumentlər	Funksiya çağırışı
	R	new	Konstruktor çağırışı	Yeni obyektin yaradılması
14	R	++	Solyönlü ifadə	Ön şəkilçi və ya postfiks artım (unar)
	R	--	Solyönlü ifadə	Ön şəkilçi və ya postfiks dekrement (unar)
	R	-	Ədəd	Unar minus (nişanın dəyişməsi)

	R	+	Ədəd	Unar üstəgər (əməliyyat yoxdur)
	R	~	Tam ədədlər	Bit-təyinatlı operator (unar)
	R	!	Məntiqi qiymət	Məntiqi operator (unar)
	R	delete	Solyönlü qiymət	Xüsusiyyətlərin silinməsi (unar)
	R	typeof	İstənilən	Məlumat tipini qaytarır (unar
	R	void	İstənilən	Qeyri-müəyyən qiymət qaytarır (unar)
13	L	*, /, %	Ədəd	Vurma, bölmə, qalıq
12	L	+, -	Tam ədəd	Toplama, çıxma
	L	+	Sətirlər	Sətirlərin bitişdirilməsi
11	L	<<	Tam ədədlər	Sola sürüşdürmə
	L	>>	Tam ədədlər	Geniş simvol dərəcəsi ilə sağa sürüşdürmə
	L	>>>	Tam ədədlər	Sıfırlar əlavə etməklə sağa sürüşdürmə
10	L	<, <=	Ədədlər və ya sətirlər	Kiçik və kiçik bərabərdir
	L	>, >=	Ədədlər və ya sətirlər	Böyük və böyük bərabərdir

	L	*instanceof*	Obyekt, konstruktor	Obyektin tipinin yoxlanılması
	L	*in*	Sətir, obyekt	Xüsusiyyətin mövcudluğun yoxlanılması
9	L	==	İstənilən	Bərabərliyə yoxlanılması
	L	*!* =	İstənilən	Bərabərsizliyin yoxlanılması
	L	===	İstənilən	Eyniliyin yoxlanılması
	L	*!*==	İstənilən	Eynilik olmamasının yoxlanılması

Cədvəl 5.1. *JavaScript operatorları*

P	**A**	**Operator**	**Operand tipi**	**Yerinə yetirilən əməliyyat**
8	L	*&*	Tam ədəd	Bit təyinatlı VƏ
7	L	*^*	Tam ədədlər	Bit təyinatlı istisnalı VƏ YA

6	L	`||`	Tam ədədlər	Bit təyinatlı VƏ YA
5	L	`&&`	Məntiqi qiymətlər	Məntiqi VƏ
4	R	`?:`	Məntiqi qiymət, istənilən	Şərti üçölçülü operator
3	R	`=`	Solyönlü qiymət, istənilən	Mənimsəmə
2	R	`*=, /=, %=, +=, -=, <<=, >>=, >>>=, &=, ^=, |=`	Solyönlü qiymət, istənilən	Əməliyyatla mənimsətmə
1	L	`,`	İstənilən	Çoxsaylı hesablama

Cədvəl 5.1. *JavaScript operatorları (davamı)*

5.2.1. Operandların miqdarı

Operatorlar onlara tələb edilən operandların miqdarı üzrə kateqoriyalara bölünmüş ola bilər. JavaScript-də olan operatorların əksəriyyəti, məsələn toplama (+) operatoru ikilikdir. Belə operatorlar iki ifadəni bir-birinə daha mürəkkəb birləşdirir. Beləliklə bu operatorlar iki operandla işləyir. JavaScript həmçinin bir neçə unar operatoru da dəstəkləyir ki, bu operatorlar ifadəni daha mürəkkəb formada dəyişdirəcəklər. -3 İfadəsindəki "minus" operatoru 3 rəqəmin işarəsinin mənfiyə (yəni -3) dəyişməsini yerinə yetirən unar operatorudur. Və nəhayət, JavaScript bir dənə də üçlük operatorunu dəstəkləyir. Şərti operator adlandırılan bu ?: operatoru, üç ifadəni bir qiymətdə birləşdirir.

5.2.2. Operandların tipi

JavaScript-də ifadələr yaratdıqda, operatorların dəstəklədiyi və qaytardığı məlumat tipini nəzərə almaq lazımdır. Çünki bəzi operatorlar, tələb edir ki, operandlar müəyyən tipdə olan qiymətləri qaytarsın. Məsələn, sətirlərin vurulmasını yerinə yetirmək olmaz, buna görə də "a" * "b" ifadəsi JavaScript-də mümkün deyil. Ancaq JavaScript interpretatoru ifadəni mümkün qədər tələb edilən tipə dəyişməyə çalışır, buna görə də "3" * "5" ifadəsi düzgündür və onun qiyməti sətir kimi ("15") deyil ədəd kimi (15) alınacaq. JavaScript-də tiplərin müxtəlifliyi haqqında 3.12 paraqraf danışılır. Bəzi operatorlar operandların tipindən asılı olaraq müxtəlif funksiyalara malikdir.

Ən açıq nümunə – + operatordur ki, ədəd operandlarının toplanmasını və sətir operandlarını bitişdirilməsini yerinə yetirir. Bundan başqa, əgər bu operatora bir sətiri və bir ədəd, ədəd sətirə dəyişdirəcək və alınmış iki sətirin bitişdirilməsi əməliyyatı yerinə yetirəcək. Məsələn, "1"+ 0 ifadəsinin nəticəsi "10" sətiri olacaq.

Nəzərə alın ki, mənimsəmə operatorları, digər solyönlü operatorlar kimi, ifadələrin sol hissəsində solyönlü qiymətin (*lvalue*) tələb edirlər. Solyönlü qiymət – tarixi termindir və "ifadə mənimsəmə operatorunun sol hissəsində ola bilər" kimi izah edilir. JavaScript-də dəyişənlər, obyekt xüsusiyyətləri və massiv elementləri solyönlü qiymətdir. ECMAScript spesifikasiyasına müvafiq olaraq inteqrasiya edilmiş funksiyalar, solyönlü qiymətləri qaytarmağa imkan verir, amma oxşar tərzdə heç bir inteqrasiya edilmiş funksiya müəyyən etməyə icazə vermir.

Və nəhayət, operatorlar həmişə malik olduqları tipin qiymətlərini qaytarmır. Müqayisənin operatorları (böyük, kiçik, bərabər, və s.) müxtəlif tipləri arqumentlər kimi qəbul edir, amma nəticəni məntiqi tipdə qaytarır. Məsələn, a<3 ifadəsi, əgər a dəyişəninin qiyməti 3-dən azdırsa *true* qiymətini alır. Gələcəkdə biz görəcəyik, müqayisə operatorları tərəfindən alınan məntiqi qiymətləri *if* təlimatlarında istifadə olunur. Həmçinin JavaScript-də müqayisə operatorları ilə ifadələrin hesablanmasının nəticələrindən asılı olaraq proqramın icrasını idarə edən *while* və *for* dövrlərində istifadə olunur.

5.2.3. Operatorların prioritetliyi

Cədvəl 5.1-də "P" hərfi qoyulmuş sütunda hər bir operatorun prioriteti göstərilmişdir. Operatorun prioritetliyi əməliyyatların yerinə yetirmə ardıcılığını ifadə edir. "P" sütununda böyük əhəmiyyətə malik olan operatorlar, ilk sıralarda, prioriteti daha az olan operatorlar isə sonuncu sıralarda göstərilmişdir. Aşağıdakı ifadəyə baxaq:

```
w = x + y * z;
```

Vurma (*) operatoru digər operatorla (+) müqayisədə böyük prioritetə malikdir, ona görə də ifadədə ilk öncə, vurma daha sonra isə toplama əməliyyatı yerinə yetirilir. Bundan başqa, mənimsəmə operatoru (=) ən kiçik prioritetə malikdir, buna görə də mənimsəmə sağ tərəfdə olan bütün əməliyyatların tamamlanmasından sonra yerinə yetirilir.
Operatorların prioriteti mötərizələrin köməyi ilə istəyinizə uyğun təlqin edilə bilər. Məsələn, əvvəlki nümunəni elə yazarıq ki, burada toplama birinci yerinə yetirilsin:

```
w = (x + y) * z;
```

Əgər siz operatorların prioritetlərindən əmin deyilsinizsə, daha sadə metod: mötərizələrin köməyi ilə hesablamaların gedişatını istəyinizə uyğun həyata keçirin. Lakin aşağıdakı qaydaları bilməyiniz məsləhətdir: vurma və bölmə, toplama və çıxma əməliyyatlarından əvvəl yetirilir və mənimsəmə operatoru isə çox aşağı prioritetə malikdir və demək olar ki həmişə sonda yerinə yetirilir.

5.2.4. Operatorların assosiativliyi

Cədvəl 5.1-də "A" hərfi qoyulmuş sütunda operatorun assosiativliyi göstərilmişdir. L qiyməti soldan sağa assosiativliyi, R qiyməti isə sağdan sola assosiativliyi bildirir. Operatorun assosiativliyi eyni prioritetə malik olan əməliyyatların icra edilməsi ardıcılığını müəyyən edir. Assotivliyi soldan sağa onu bildirir ki, burada əməliyyatlar soldan sağa yerinə yetirilir. Məsələn, toplama operatorunun assosiativliyi soldan sağadır, buna görə də aşağıdakı iki ifadəyə ekvivalentdir:

```
w = x + y + z;
w = ((x + y) + z);
```

İndi isə bu ifadələrə baxın (ifadələr praktik olaraq mənasızdır):

```
x = ~-~y;
w = x = y = z;
q = a? b:c? d:e? f:g;
```

Bu ifadələr aşağıdakı ifadələrə ekvivalentdir:

```
x = ~(-(~y));
w = (x = (y = z));
q = a?b: (c?d:(e?f:g));
```

Çünki, unar operatorları, mənimsəmə operatorları və üçlük şərt operatorları sağdan sola assosiativliyə malikdir.

5.3. Hesab operatorları

Bu bölmədə biz hesab operatorları öyrənəcəyik:

Toplama (+)

"üstəgəl" operatoru ədəd operandlarını toplanılmasını və ya sətirlərin birləşdirməsini yerinə yetirir. Əgər operandlardan biri sətirdirsə, digər operand sətirə çevriləcək və birləşdirmə yerinə yetiriləcək. Ədədə və ya sətirlərə dəyişdirilmiş obyekt-operandlarında müvafiq olaraq toplama və bitişdirilmə yerinə yetirilir. Bu dəyişiklik `valueOf()` və/və ya `toString()` metodlarının köməyi həyata keçirilir.

Çıxma (-)

"Minus" operatorunun ikili istifadə edildikdə, o birinci operanddan ikinci operandın çıxılmasını yerinə yetirir. Əgər ədəd olmayan operandlar göstərilmişdirsə, onda operator onları ədədlərə dəyişdirməyə çalışır.

Vurma (*)

* operatoru iki operandı bir-birinə vurur. Ədəd olmayan operandlar göstərdikdə, operator onları ədədlərə çevirməyə çalışır.

Bölmə (/)

/ operatoru birinci operandı ikinci operanda bölür. Ədəd olmayan operandlar göstərdikdə, operator onları ədədlərə çevirməyə çalışır. Başqa proqramlaşdırma dilləri ilə tanış olanlar, tam ədədin tam ədədə bölünməsi nəticəsində alınan həqiqi ədədin, yalnız tam hissəsinin istifadə edildiyini güman edə bilərlər. Ancaq JavaScript- də bütün ədədlər həqiqi ədədlərdir, buna görə də bölmənin nəticəsi istənilən üzən nöqtəli qiymət ola bilər. Məsələn 5/2 əməli 2 deyil, 2.5 nəticəsini verəcək. Sıfıra bölmənin nəticəsi – ya müsbət ya da mənfi sonsuzluq olur. 0/0 isə NaN qiymətinə alır.

Qalıqlı bölmə operatoru (%)

birinci operandın ikinci operanda bölünməsindən alınan qalığı hesablayır. Ədəd olmayan operandlar göstərdikdə, operator onları ədədlərə çevirməyə çalışır. Nəticənin işarəsi birinci operandın işarəsinə uyğun gəlir, məsələn 5 % 2 1 verir. Qalıqlı bölmə operatoru adətən tam ədədlərə tətbiq edilir, amma kəsr ədədlərində də bu arifmetrik əməliyyat yerinə yetirilə bilər. Məsələn, - 4.3 % 2.1 ifadəsinin nəticəsi - 0.1 alınır.

Unar minus (-)

Minus unar operator kimi istifadə olunduqda, yeganə operanddan əvvəl göstərilir və işarənin dəyişməsinin unar əməliyyatını yerinə yetirir. Başqa

sözlə, o mənfi qiyməti müsbətə və ya əksinə dəyişəcək. Ədəd olmayan operandlar göstərdikdə, operator onları ədədlərə çevirməyə çalışır.

Unar üstəgəl (+)

JavaScript-də "unar çıxma" operatoruna simmetrik "unar üstəgəl" operatoru da mövcuddur. Bu operatorun köməyilə ədəd literallarının işarəsini açıq-aydın vermək olar. Nadir hallarda istifadə edilir və bəzi proqramçılar hesab edir ki, belə olan halda proqramın mətni daha aydın olur:

```
var profit = +1000000;
```

Bu kodda "üstəgəl" operatoru heç nəyi etmir; onun arqumentinin qiyməti onun işinin nəticəsidir. Ancaq bu operator ədəd olmayan arqumentləri ədədlərə dəyişdirəcək. Əgər arqument dəyişdirilə bilmirsə, NaN qiymətini alır.

Artım (++)

Tək operantlı (bu dəyişən, massiv elementi və ya obyekt xüsusiyyəti ola bilər) operator öz operandı üzərində inkrementləşmə (yəni, bir vahid artma) əməliyyatını yerinə yetirir. Əgər bu dəyişənin qiyməti, massiv elementi və ya obyekt xüsusiyyəti ədəd deyilsə, operator əvvəlcə onu ədədə çevirməyə çalışır. Bu operatorun operandla dəqiq davranışı vəziyyətdən asılıdır. Əgər bu operatoru operanddan əvvəl qoymusuzsa, onda operanda 1 əlavə edilir və operandın artırılmış qiyməti operatorun nəticəsi olur. Əgər bu operatoru operanddan sonra qoysanız (artımın postfiks operatoru), onda operanda 1 əlavə edilir, ancaq operandın ilkin operatorun nəticəsi olur. Əgər artırılan qiymət ədəd deyilsə, o hesablama prosesində ədədə çevriləcək. Məsələn, aşağıdakı kod i və j dəyişənlərini 2-ə bərabər edir:

```
i = 1; j = ++i;
```

Lakin burada i dəyişənin qiyməti 2, j dəyişəninin qiyməti 1 olaraq təyin edilir.

```
i = 1; j = i++;
```

Bu operator (hər iki formada) əksər hallarda dövrü idarə edən sayğacın artımı üçün tətbiq edilir. Nəzərə alın ki, artımın ön şəkilçi və ya postfiks operatorunu və onun operandını sətirlərdə tətbiq etmək olmaz, çünki, JavaScript-də nöqtəli vergüllər avtomatik qoyulur. Elə buna görə, JavaScript interpretatoru operanda baxacaq və ondan sonra nöqtəli vergülü yerləşdirəcək.

Dekrement (--)

Tək operantlı (bu dəyişən, massiv elementi və ya obyekt xüsusiyyəti ola bilər) operator öz operandı üzərində dekrementləşmə (yəni, bir vahid azalma) əməliyyatını yerinə yetirir. Əgər bu dəyişənin qiyməti, massiv elementi və ya obyekt xüsusiyyəti ədəd deyilsə, operator əvvəlcə onu ədədə çevirməyə çalışır. ++ operatoru kimi bu operatorun da operandla bağlı dəqiq davranışı operatorun vəziyyətindən asılıdır. Əgər operatoru operanddan əvvəl (artımın ön şəkilçi operatoru) qoymusuzsa, onda operandı 1 vahid azaldır və operandın azaldılmış qiyməti operatorun nəticədir. Əgər operatoru operanddan sonra qoymusuzsa (artımın postfiks operatoru), o operandan 1 vahid çıxır, ancaq operandın ilkin qiyməti operatorun nəticəsidir. Əgər azaldılan qiymət ədəd deyilsə, o hesablama prosesində ədədə çevriləcək.

5.4. Bərabərlik operatorları

Bu bölmədə bərabərlik və bərabərsizlik operatorları öyrəniləcək. İki qiyməti müqayisə edərək müqayisədən asılı olaraq, nəticəsi məntiqi qiymət (`true` və ya `false`) olan operatorlardır. 6-cı fəsildə görəcəyik ki, bu operatorlar adətən proqramın icra gedişatının idarəedilməsi üçün `if` təlimatlarında və `for` dövrlərində tətbiq edilir.

5.4.1. Bərabərlik (==) və eynilik (===)

== və === operatorları iki müxtəlif üst-üstə düşmə təyinini rəhbər tutaraq iki operandın üst-üstə düşməsini yoxlayır. Hər iki operator istənilən tipdə olan operandları qəbul edir və son nəticədə əgər operandlar uyğun gələrsə (şərt ödənərsə) `true`, əks halda falsə qiymətini alır. === operatoru, eyniliyin operatorudur. Bu operator iki operandın ciddi təsadüfü təyinini rəhbər tutaraq, "eyniliyini" yoxlayır. Operator == bərabərliyin operatoru kimi məlumdur (məşhurdur), o yoxlayır, iki bərabərdirmi tiplərin dəyişikliklərini güman edən (icazə verən) təsadüfün daha az ciddi təyininə uyğun olaraq onun operandı.
Eynilik operatoru ECMAScript v3-də standartlaşmışdır və JavaScript 1.3-də və ondan yuxarı versiyalarda mövcuddur. Eynilik operatorunun tətbiqi ilə JavaScript dili =, == və === operatorlarını dəstəkləməyə başladı. Əmin olun ki, siz mənimsəmə, bərabərlik və eynilik operatorları arasında fərqi anlayırsınız. Öz proqramlarının hazırlaması zamanı diqqətli olun və düzgün operatorlar tətbiq edin! Hərçənd ki, hər üç operatoru "bərabər"dir adlandırmaq daha rahat olardı, lakin qarışıqlıq olmasın deyə bu operatorları = operatorunu "alır", və ya "mənimsənilir", == operatorunu "bərabərdir", === operatorunu isə "eynilikdir" kimi oxumaq daha məqsədə uyğundur . JavaScript-də ədəd, sətir və məntiqi qiymətlər öz qiymətləri ilə müqayisə edilir. Bu halda iki müxtəlif ölçüyə baxılır və qiymətlərin

== və ya === olduğu yoxlanılır. Bu isə o deməkdir ki, iki dəyişən şərt ödədikdə bərabərdir və ya eynilikdir. Məsələn, iki sətir, yalnız o halda bərabərdir ki, hər iki sətir özündə eyni simvolları ehtiva etsin. Eyni zamanda obyektlər, massivlər və funksiyalar istinad üzrə müqayisə edilir. Bu isə o deməkdir ki, iki dəyişən, yalnız o halda bərabərdir ki, onlar eyni obyektə istinad edilir. İki müxtəlif massiv heç vaxt bərabər və ya eynilik olmur. Hətta onlar bərabər və ya eyni elementləri özündə ehtiva edirsə belə heç vaxt bərabər və ya eynilik olmur. Obyektlərə, massivlərə və ya funksiyalara istinad edən iki dəyişən, yalnız o halda bərabərdir ki, bu dəyişənlər eyni obyekt, massiv və ya funksiyaya istinad edir. İki müxtəlif obyektin və ya iki müxtəlif massivin eynilik və bərabərlik xüsusiyyətlərini yoxlamaq üçün, onların hər bir xüsusiyyətini və ya elementinin eyniliyini və ya bərabərliyini yoxlamaq lazımdır. (Və əgər hər hansı xüsusiyyət və ya elementin özü obyektdirsə və ya massivdirsə, müqayisə daxil edilmə dərinliyindən asılı olaraq yerinə yetirilməlidir.)

İki qiymətin eyniliyinin təyini zamanı === operatoru aşağıdakı qaydaları rəhbər tutur:

- Əgər iki qiymət müxtəlif tiplərə malikdirsə, onlar eynilik ola bilməz.
- NaN qiyməti ehtiva etməyən iki eyni qiymət, yalnız onların hər ikisi ədəddən təşkil olunubsa, eynilikdir. NaN qiyməti heç bir qiymətə, hətta özünə belə eynilik deyil! Qiymətin, NaN qiyməti olmasını yoxlamaq üçün, isNaN() qlobal funksiyasından istifadə etmək lazımdır.
- Əgər hər iki qiymət sətirdirsə və bu qiymətlərin müvafiq mövqeləri eyni simvolları ehtiva edirsə bu qiymətlər eynilikdir. Əgər sətir uzunluğuna və ya tərkibinə görə fərqlənirsə bu qiymətlər eynilik deyil. Yadınızda saxlayın, bəzi hallarda Unicode standartı sətir kodlaşdırılmasının bir neçə üsulunu təqdim edir. Ancaq effektivliyin artırılması məqsədilə sətirlər ciddi simvol uyğunluğu ilə müqayisə edilir və müqayisədən əvvəl sətirlərin "normallaşdırılmış formada" olması güman edilir. Sətirlərin müqayisəsinin başqa bir üsulu String.localeCompare() metodunur.
- Əgər hər iki qiymət `true` və ya `false` məntiqi qiymətlərinə malikdirsə, bu qiymətlər eynilikdir.
- Əgər hər iki qiymət eyni obyektə, massivə və ya funksiyaya istinad edirsə, bu qiymətlər eynilikdir. Əgər onlar müxtəlif obyektlərə (massivlərə və ya funksiyalara) istinad edirlərsə, onlar eynilik deyil. Hətta hər iki qiymət eyni xüsusiyyətlərə və ya eyni elementlərə malik olduqda belə eynilik deyil.
- Əgər hər iki qiymət `null` və ya `undefined` bərabərdirsə, bu qiymətlər eynilikdir.
  - Aşağıdakı qaydalar ==operatorun köməyi ilə bərabərliyin təyini üçün tətbiq edilir:
- Əgər iki qiymət eyni tipdəndirsə, onlar eyniliyi yoxlanırlır. Əgər qiymətlər

eynilikdirsə, onlar bərabərdir; əgər onlar eynilik deyilsə, bərabərdir deyil.

- Əgər iki qiymət eyni tipdən deyilsə belə, bu qiymətlər bərabər ola bilər. Tiplərin dəyişikliyi barədə qaydalar aşağıdakılardır:
- Əgər bir qiymət `null`-a və o biri qiymәd `undefined` bərabərdirsə, onda bu qiymətlər bərabərdir.
- Əgər bir qiymət ədəddən, o biri qiymət sətirdən təşkil olunubsa, onda sətir tipi ədəd tipinə çevriləcək və çevrilmiş qiymətlə müqayisə yerinə ediləcək. Əgər hər hansı qiymət `true`-a bərabərdirsə, bu qiymət 1 qiymətinə çevriləcək və sonra müqayisə yenidən aparılacaq.
- Əgər hər hansı qiymət `false`-yə bərabərdirsə, bu qiymət 0 (sıfır) qiymətinə çevriləcək və sonra müqayisə yenidən aparılacaq.
- Əgər qiymətlərdən biri obyekt, digəri ədəddən və ya sətirdən təşkil olunubsa, öncə obyekt elementar tipə çevriləcək və sonra müqayisə yenidən aparılacaq. Obyekt elementar tip qiymətinə `toString()`-metodunun köməyi və ya `valueOf()` metodunun köməyi ilə çevriləcək. JavaScript-in baza dilinin inteqrasiya edilmiş sinifləri əvvəlcə `valueOf()` dəyişikliyini, sonra isə `toString()` çevrilməsini yerinə yetirməyə çalışır. Lakin burada Date sinifi istisnadır, hansı ki, bu sinif həmişə `toString()` çevrilməsini yerinə yetirir. JavaScript baza hissəsi olmayanlar obyektlər, elementar tip qiymətlərinə reallaşdırmada müəyyən edilmiş üsulla dəyişdirilə bilər.
- Digər istənilən qiymət kombinasiyaları bərabər deyil. Bərabərliyə yoxlamaq üçün aşağıdakı nümunəyə baxaq: "1" == `true` Bu ifadənin nəticəsi `true`-a bərabərdir, yəni. bu müxtəlif tipdən olan qiymətlər faktiki olaraq bərabərdir. `True`-un məntiqi qiyməti 1 ədədinə dəyişdiriləcək və müqayisə yenidən aparılacaq. Daha bir nümunədə isə "1" sətiri 1 ədədinə dəyişdiriləcək. Çünki, hər iki ədəd indi uyğundur, belə olduqda da müqayisə operatoru `true` qiymətini alır.

5.4.2. Bərabərsizlik (!=) və eynilik deyil (! ==) operatorları

!= və !== operatorları yoxlamanı == və === operatorlarının əksinə yerinə yetirir. != operatoru əgər iki qiymət bir-birinə bərabərdirsə `false`, əks təqdirdə `true` qiymətini alır. Eynilik deyil operatoru (!==) əgər iki qiymət eynilikdirsə `false`, əks təqdirdə `true` qiymətini alır. Bu operator ECMAScript v3-də standartlaşmışdır və JavaScript 1.3 və daha yuxarı versiyalarda reallaşdırılmışdır. Gələcəkdə biz, ! operatorunu məntiqinin əməliyyatı həyata keçirmədiyinin şahidi olacağıq. Bu operatorları != "bərabər deyil", !== isə " eynilik deyil" kimi yadda

saxlamaq məsləhətdir. Müxtəlif tiplər üçün bərabərlik və eyniliyin təyininin təfərrüatları əvvəlki bölmələrdəki kriteriyalarla uyğundur.

5.5. Əlaqə operatorları

Bu bölmədə biz, JavaScript-də olan əlaqə operatorlarını öyrəcəyik. Bu operatorlar, iki qiymət əlaqəni yoxlayanlar və nəticədən asılı olaraq `true` və ya `false` qiymətini alır. Biz 6-cı fəsildə, proqramın icrasının gedişatını idarə etmək üçün `if` təlimatlarında, `while` və digər dövr təlimatlarında geniş istifadə edildiyinin şahidi olacağıq.

5.5.1. Müqayisə operatorları

İki ölçünün nisbi sırasının təyini üçün - əlaqə operatorları arasında ən çox müqayisə operatorlarından istifadə edilir. Müqayisə operatorları aşağıdakılardır:
Kiçik (<)
< operatorun nəticəsi; əgər birinci operand ikinci operanddan kiçikdirsə, `true` qiymətinə, əks təqdirdə o `false` qiymətinə bərabərdir.
Böyük (>)
> operatorunun nəticəsi; əgər birinci operand ikinci operanddan böyükdürsə, `true` qiymətinə, əks təqdirdə o `false` qiymətinə bərabərdir.
Kiçik və ya bərabərdir (<=)
<=operatorun nəticəsi əgər birinci operand ikinci operanddan kiçikdirsə və ya bu operandlar bərabərdirsə `true` qiymətinə, əks təqdirdə o `false` qiymətinə bərabərdir.
Böyük və ya bərabər (>=)
>= operatorun nəticəsi əgər birinci operand ikinci operanddan böyükdürsə və ya bu operandlar bərabərdirsə, `true` qiymətinə, əks təqdirdə o `false` qiymətinə bərabərdir.
Bu operatorlar istənilən tipdən olan operandları müqayisə etməyə imkan verir. Ancaq müqayisə yalnız ədəd və sətirlər üçün yerinə yetirilə bilər, buna görə də, ədəd və ya sətir olmayan operandlar, dəyişdiriləcək. Müqayisə və dəyişiklik aşağıdakı qaydada yerinə yetirilir:

- Əgər hər iki operand ədəddirsə və ya ədədlərə dəyişdiriləcəksə, onlar ədəd kimi müqayisə edilirlər.
- Əgər hər iki operand sətirdirsə və ya sətirlərə dəyişdiriləcəksə, onlar sətirlər kimi müqayisə edilir.
- Əgər bir operand sətirdirsə və ya sətirə dəyişdiriləcəksə, digəri isə

ədədirsə və ya ədədə dəyişdiriləcəksə, operator ədədi sətirə dəyişdirməyə və müqayisəni ədəd kimi yerinə yetirməyə çalışır. Əgər sətirin tərkibi ədəd ehtiva etmirsə, operator NaN qiymətini alır müqayisə isə `false` olur. (JavaScript 1.1 versiyasında ədədin sətirə dəyişikliyi baş vermir və bu tip arqumentlər xətaya səbəb olur və NaN qiymətini alır.)

- Əgər obyekt həm ədədə, həm də sətirə, dəyişdirilə bilərsə, JavaScript interpretatoru ədəd dəyişikliyini yerinə yetirir. Məsələn, Date obyektləri ədəd kimi müqayisə edilir, yəni iki tarix arasında müqayisə əməliyyatını həyata keçirmək olar.
- Əgər hər iki operand ədədlərə və ya sətirlərə müvəffəqiyyətlə dəyişdirilə bilmirsə, operatorlar həmişə `false` qiymətini alır.
- Əgər operandlardan biri NaN qiymətinə bərabərdirsə və ya bu qiymətə dəyişdiriləcəksə, onda müqayisə operatorunun nəticəsi `false` qiymətidir.

Nəzərə alın ki, sətirlərin müqayisəsi zamanı,Unicode kodlaşdırmasında hər simvolun ədəd qiymətlərinin ciddi uyğunluğu mövcuddur. Bəzi hallarda Unicode standartının tətbiqi ilə ekvivalent sətirlərin müxtəlif simvol ardıcıllıqlarında eyni cür kodlaşdırılmasına icazə veririlir, amma JavaScript-in müqayisə operatorları kodlaşdırmalarda olan bu fərqləri aşkar etmir; operator güman edir ki, bütün sətirlər normallaşdırılmış formada təqdim edilmişdir. Diqqətli olun: sətirlərin müqayisəsi simvol cədvəli vasitəsilə həyata keçirilir, yəni Unicode kodlaşdırmasında (ekstremal üzrə ASCII alt çoxluğu üçün) bütün böyük hərflər sətirdəki digər hərflərdən böyükdür. Bu qayda anlaşılmayan nəticələrə gətirə bilər. Məsələn, "Zoo" sətirinin "aardvark" sətirindən böyük olduğu göstərilir. Sətirlərin müqayisəsi zamanı String.localeCompare () metodu çox yaxşıdır, hansı ki, lokal "əlifba sırasının" təyinini nəzərə alır.

Fərqli uçotlu registrləri müqayisə etmək üçün əvvəlcə String.toLowerCase() metodunun köməyilə kiçik hərfli sətirlərə və ya String.toUpperCase() metodunun köməyilə böyük hərfli sətirlərə dəyişdirmək lazımdır.

<= (kiçik və ya bərabər) və >= (böyük və ya bərabər) operatorları iki qiymətin bərabər olmasını bərabərlik və ya eynilik operatorlarının köməyi ilə müəyyən etmir. "Kiçik və y Kiçik bərabər" sadəcə "deyil daha" kimi təyin edilir bərabərdir, və (amma) operator "böyük və ya bərabər" – "deyil daha az" kimi. Tək istisna (çıxarılma) baş verir, nə vaxt ki, operandlardan biri NaN qiymətini (mənasını) təşkil edir (və ya dəyişdiriləcək ona); bu halda hamı (hər şey) müqayisənin dörd operatoru `false`-ı qaytarır.

5.5.2. in operatoru

`In` operatoru tələb edir ki, sol operand sətir və ya sətirə dəyişdirilə bilən olsun. Sağ operandla obyekt və ya massiv olmalıdır. Əgər sol qiymət sağda göstərilmiş

obyektin xüsusiyyətinin adını təşkil edərsə, operatorun nəticəsi *true* olacaq. Məsələn:

```
var point ={x:1, y:1};                  // Obyekti müəyyən edirik.
var has_x_coord = "x" in point;     // true qiymətinə bərabərdir.
var has_y_coord = "y" in point;     // true qiymətinə bərabərdir.
var has_z_coord = "z" in point;     // false qiymətinə bərabərdir,
                                // bu üçölçülü nöqtə deyil.
var ts = "toString" in point;         // Mənimsədilmiş xüsusiyyət,
                                // true qiymətinə bərabərdir.
```

5.5.3. instanceof operatoru

instanceof operatoru tələb edir ki, sol operand obyekt, sağ operand isə obyekt sinfinin adı olsun. Əgər göstərilmiş obyekt solda, sinifin nüsxəsi isə sağda göstərilmişsə nəticə *true*, əks halda *false* olacaq. Biz 9-cu fəsildə görəcəyik ki, JavaScript-də siniflər obyektlər onların funksiya-konstuktoru tərəfindən inisializasiya etmiş təyin edilirlər. Beləliklə, instanceof-un sağ operandı funksiyanın adı-konstruktorlar olmalıdır. Diqqətli olun: bütün obyekt nüsxələri *Object* sinifi ilə təşkil olunmalıdır. Məsələn:

```
var d = new Date();                 // Date() konstrukturunun köməyi ilə
yeni
                                // obyekt yaradırıq
d instanceof Date;                    // true qiymətinə bərabərdir; d
obyekti
                                // Date() funksiyasının köməyilə
yaradılmışdır
d instanceof Object;                // true qiymətinə bərabərdir; bütün
obyektlər
                                // Object sinifinin nüsxələrindən təşkil
                                // olunur
d instanceof Number;                // false qiymətinə bərabərdir
                                // d Number obyektini ehtive etmir
d Number var a = [1, 2, 3];     // Massiv literalının köməyi ilə massiv
                                // yaradırıq
a instanceof Array;                 // true qiymətinə bərabərdir a –
massivdir
a instanceof Object;                // true qiymətinə bərabərdir;
                                // bütün massivlər obyektlərdən təşkil
olunub
a instanceof RegExp;                // false qiymətinə bərabərdir;
                                // massiv requlyar ifadə deyil.
```

Əgər instanceof-un sol operandı obyekt deyilsə və ya əgər sağ operand obyektdirsə, lakin funksiya-konstuktoruna malik deyilsə, instanceof *false*

qiymətini alır. Amma əgər sağ operand da obyekt deyilsə, icra zamanı xəta baş verir

5.6. Sətir operatorları

Əvvəlki bölmələrdə qeyd edildiyi kimi elə operatorlar var ki, sətir tipində onlar müxtəlif funksiya malikdir.
+ operatoru iki sətir operandının birləşdirməsini yerinə yetirir. Başqa sözlə, birinci sətirdən ibarət olan sətirə, ikinci sətirə bərabər olan yeni sətir əlavə edir. Aşağıdakı ifadə "hello there" sətirinə bərabərdir:

```
"hello" + " "+ "there"
```

Aşağıdakı təlimatlar nəticədə "22" sətiri alınır:

```
a = "2";
b = "2";
c = a + b;
```

<, <=, > və >= operatorları iki sətiri müqayisə edir. Müqayisə əlifba sırasına əsaslanmışdır. 5.1.1-bölməsində qeyd edildiyi kimi, bu əlifba sırası JavaScript-də istifadə edilən Unicode kodlaşdırmasına əsaslanır. Bu kodlaşdırmada bütün böyük hərflər (latın əlifbasından başlayaraq), digər sətri hərflərdən böyükdür, buna görə də gözlənilməz qarşılaşa bilərsiz.
Bərabərlik operatorları (==) və bərabərsizliklər (!=) yalnız sətirlərə deyil, bütün məlumat tiplərinə tətbiq edilir və sətir tipində xüsusi bir funksiyası yoxdu.
+ operatoru xüsusidir, çünki, bu operatorun ədədlərə nisbətən sətirlərdə yüksək prioritetlidir. Artıq qeyd edildiyi kimi, əgər + operatorunun operandlarından biri sətirdisə, onda digər operand sətirə dəyişdiriləcək (və ya hər iki operand sətirə dəyişdiriləcək) və konkateniruyutsya operandları, və (amma) formalaşmırlar. Digər tərəfdən, müqayisə operatorları yalnız hər iki operand sətirlərdən təşkil olunduqda sətir müqayisəsini yerinə yetirir yetirirlər. Əgər yalnız bir operand sətirdirsə, onda JavaScript interpretatoru onu ədədə dəyişdirməyə çalışır. Aşağıda bu qaydaların illüstrasiyası aparılır:

```
1 + 2            // Toplama. Nəticə 3-ə bərabərdir.
"1" + "2"        // Bitişdirmə. Nəticə "12"-ə bərabərdir.
"1" + 2          // Bitişdirmə. 2 ədədi "2" sətirinə dəyişdiriləcək.
                 // Nəticə "12"-ə bərabər olacaq.
11 < 3             // Ədədlərin müqayisəsi. Nəticə false qiymətinə
bərabərdir.
"11" < "3"       // Sətirlərin müqayisəsi. Nəticə true qiymətinə
bərabərdir.
```

```
"11" < 3          // Ədədlərin müqayisəsi; "11" sətiri 11 ədədinə
                 // dəyişdiriləcək. Nəticə false qiymətinə bərabərdir.
"one" < 3         // Ədədlərin müqayisəsi; "one" sətiri NaN qiymətinə
                 // dəyişdiriləcək. Nəticə false qiymətinə bərabərdir.
```

Və nəhayət, qeyd etmək lazımdır ki, nə vaxt ki, + operatoru sətirlərə və saylara tətbiq edilir, o assosativ olmaya bilər. Başqa sözlə, nəticə əməliyyatları yerinə yetirmə sırasından asılı ola bilər. Bunu aşağıdakı nümunələrdə görmək olar:

```
s = 1 + 2 + "dovşan";       // Nəticə: "3 dovşan"
t = "dovşan: "+ 1 + 2;       // Nəticə: "dovşan: 12"
```

Bu təəccüblü davranışın səbəbi odur ki, + operatoru soldan sağa (əgər mötərizələr bu sıranı dəyişdirmirsə) işləyir. Beləliklə, son iki nümunə aşağıdakına ekvivalentdir:

```
s = (1 + 2) + "dovşan";       // Birinci əməliyyatın nəticəsi ədəd;
ikincinin isə
                         // sətirdir
t = ("dovşan: "+ 1) + 2;      // Hər iki əməliyyatın nəticəsi sətirdir
```

5.7. Məntiqi operatorlar

Məntiqi operatorlar adətən cəbrdə, bull əməliyyatlarının icra edilməsi üçün istifadə olunur. Onlar *if*, *while* və *for* təlimatlarında bir neçə dəyişənin iştirakıyla mürəkkəb müqayisələri həyata keçirilməsi üçün müqayisə operatorları birlikdə tətbiq edilir.

5.7.1. Məntiqi VƏ (&&)

Məntiqi operandlarla istifadə zamanı && operatoru iki qiymətin üzərində məntiqi əməliyyatı yerinə yetirir: bu operator yalnız birinci və ikinci operandlar *true* qiymətinə bərabər olarsa, o halda *true* qiymətini alır. Əgər bir və ya hər iki operand *false*-yə bərabərdirsə, operator *false* qiymətini alır.
Bu operatorun real davranışı bir qədər mürəkkəbdir. O hesablama əməliyyatına sol operanddan başlayır. Əgər alınmış qiymət *false*-yə dəyişdirilə bilərsə (əgər sol operand *null*, 0, "" və ya *undefined*-ə bərabərdirsə), operator sol ifadənin qiymətini alır. Əks təqdirdə operator sağ operandı hesablayır və ifadənin qiymətini qaytarır[15].
Qeyd etmək lazımdır ki, sol ifadənin qiymətindən asılı olaraq bu operator sağ ifadəni ya hesablayır ya da hesablamır. && operatorunun bu xüsusiyyətindən

qəsdən istifadə etmək də olar. Məsələn aşağıdakı JavaScript-kodunun iki sətiri ekvivalent nəticəyə bərabərdir:

```
if (a == b) stop();
(a == b) && stop();
```

Bəzi proqramçılar (xüsusi Perl-lə işləmiş) belə stili proqramlaşdırmada təbii və faydalı hesab edir, amma mən bu metodu sizə məsləhət görmürəm. Bu faktdır ki, sağ tərəfin hesablanmasına zəmanət verilmir, bu da tez-tez səhvlərə gətirib çıxardır. Aşağıdakı koda baxaq:

Perl - yuxarı səviyyəli, dinamik proqramlaşdırma dilidir. Perl 1987-ci ildə Lary Wall tərəfindən Unix script dili kimi yaradılmışdır.
Perl Vebdə CGİ proqramlaşdırma dili kimi istifadə olunur. Bundan başqa Perl qrafik proqramlaşdırma, sistem proqramlaşdırma, şəbəkə proqramlaşdırmasında, bioinformatikada və digər sahələrdə istifadə olunur.

```
if ((a == null) && (b++ > 10)) stop();
```

Çox ehtimal ki, bu təlimat proqramçının güman etdiyi kimi işləməyəcək, çünki sağ tərəfdə artım operatoru sol ifadə *false*-a bərabər olduğu zaman hadisələrdə hesablanmayacaq. Bu sualtı daşı keçmək üçün, əgər prosesə bələd deyilsinizsə, && operatorunun sağ tərəfinə əlavə təsirlərə malik olan (mənimsəmə, artım, dekrement və funksiya çağırışları) ifadələr yerləşdirməyin. Bu operatorun iş prinsipinin kifayət qədər dolaşıq alqoritminə baxmayaraq, operatorun davranışı Bull cəbri operatoru kimi kifayət qədər sadə və təhlükəsizdir. Əslində bu operator məntiqi qiymət qaytarmır, lakin qaytardığı qiymət, həmişə məntiqi qiymətə dəyişdirilə bilər.

5.7.2. Məntiqi VƏ (||)

Məntiqi operandlarla istifadə zamanı || operatoru iki qiymət üzərində "məntiqi və ya" əməliyyatını yerinə yetirir: bu operator əgər birinci və ya ikinci operand (və ya hər iki operand) *true* qiymətinə bərabərdirsə *true* qiymətini qaytarır. Əgər hər iki operand *false*-yə bərabərdirsə, onda operator *false* qiymətini qaytarır.
Hərçənd ki, || operatoru əksər hallarda sadəcə "məntiqi və ya" operatoru kimi tətbiq edilir, lakin bu operatorun da, && operatoru kimi iş prinsipi mürəkkəbdir. Operatorun işi sol operandın hesablanmasından başlanır. Əgər bu ifadənin qiyməti *true*-a dəyişdirilə bilərsə, sol ifadənin qiymətini qaytarır. Əks təqdirdə operator sağ operandı hesablayır və bu ifadənin qiymətini

qaytarır.[16]
&& operatorunda olduğu kimi, əgər prosesə nabələdsinizsə, əlavə təsirlərə malik olan sağ operandlardan çəkinmək lazımdır.
Hətta || operatoru məntiqsiz tipdə olan operandları ilə tətbiq edilsə belə, qaytarılan qiymət tipindən asılı olmayaraq məntiqi qiymət dəyişdirilə bilər.
Eyni zamanda bəzən || operatorunun harada məntiqi, harada isə, məntiqi qiymət qaytardığı barədə çaşğınlıq yaranır. Bu yanaşmanın mahiyyəti odur ki, || operatorü təklif edilmiş alternativlərdən `null` qiymətini ehtiva etməyən (yəni `true` qiymətinə dəyişdiriləcək) birinci qiyməti seçir. Sonra isə proseslər aşağıdakı nümunədə göstərilmiş formada cərəyan edir:

```
// Əgər max_width dəyişəni müəyyən edilmişdirsə, onun qiymətindən
istifadə
// olunur.
// Əks təqdirdə qiymət preferences obyektindən götürülür.
// Əgər obyekt (və ya obyektin max_with xüsusiyyəti) müəyyən
edilməmişdirsə,
// proqramın mətnində təyin edilmiş sabitin qiymətindən istifadə
olunur.
var max = max_width || preferences.max_width || 500;
```

5.7.3. Məntiqi İNKAR (!)

! operatoru tək operanddan əvvəl yerləşdirilən unar operatorudur. Bu operator öz operandının qiymətini dəyişir. Belə ki, əgər *a* dəyişəni `true` qiymətini (və ya `true` qiymətinə dəyişdirilə bilən) ifadə edirsə, onda `!a` ifadəsi `false` qiymətini ifadə edir. Və əgər `p && q` ifadəsi `false`-ə bərabərdirsə (və ya `false` qiymətinə dəyişdirilə bilən), onda `!(p && q)` ifadəsi `true`-a bərabərdir. Nəzərə alın ki, bu operatoru iki dəfə tətbiq edildikdə, istənilən tipdə olan qiyməti məntiqi qiymət dəyişdirmək olar:!! x.

5.8. Bit-təyinatlı operatorlar

JavaScript-də bütün ədədlər həqiqi ədədlərdir. Bit-təyinatlı operatorların operandları tam ədəd olmalıdır. Bu operatorlar 32 dərəcəli quruluş ilə təqdim edilmiş tam operandlarla (bu təqdim etməyə üzən nöqtəli operandlarlar ekvivalent deyil) işləyirlər. Bu operatorlardan dördü ***Bull cəbrinin***[?] bit-təyinatlı əməliyyatlarını yerinə yetirir ki, bu operatorlar əvvəlki bölmədə təsvir edilən məntiqi operatorlara analojidir, lakin bu operatorlarda operandın hər bitinə ayrı bir məntiqi qiymət kimi baxılır. Bit-təyinatlı digər üç operator bitlərin sola və sağa operator bitlərin yerdəyişməsi üçün tətbiq edilir.

Bul cəbri — ədədi dövrələrin analiz və dizaynını təmin edən riyazi nəzəriyyədir. Rəqəmli kompüter dövrələrinin tətbiqində, ikili dəyişənlər üzərində təyin olunan ədədi əməliyyatları göstərir. Bul cəbri ikilik say sisteminə əsaslanır.
Məntiqi cəbrin yəni Bul cəbrinin əsası ingilis riyaziyyatçısı ***Corc Bul*** tərəfindən qoyulmuşdur.

Əgər operandlar tam ədəd deyilsə və ya 32 dərəcəli tam ədəd aralığındı yerləşmirsə (yəni, operand çox böyüksə), bit-təyinatlı operatorlar operandın kəsr və 32 dərəcədən böyük ixtiyari hissəsini ixtisasa salaraq, operandları sadəcə 32 dərəcəli tam ədəddə "yerləşdirir". Yerdəyişmə operatorları tələb edirlər ki, sağ operandın qiyməti 0- 31 aralığında tam ədəd olsun. Yuxarıda təsvir edilən üsullarla operandın 32 dərəcəli tam ədədə dəyişikliyindən sonra onlar uyğun olan diapazonda ədədi alaraq ixtiyari 5-dərəcəli bitdən az hissəsi ixtisara salınır.
Əgər siz, ikilik say sistemi ilə tanış deyilsinizsə və ya onluq say sistemində göstərilən tam ədədlərin ikilik say sisteminə necə keçirildiyi barədə təsəvvürünüz yoxdursa, bu bölmədə baxılan operatorların iş prinspini anlamayacaqsınız. Buna görə də, bölmənin bu hissəsini ixtisara sala bilərsiniz. Bu operatorlar ikilik say sistemi ilə aşağı səviyyəli (dərin) manipulyasiyalar üçün tələb olunur. Bit-təyinatlı operatorlar JavaScript-də proqramlaşdırma zamanı nadir hallarda tətbiq edilir.

Aşağıdakı bit-təyinatlı operatorların siyahısı göstərilir:

Bit-təyinatlı VƏ (&)

& operatoru operantların bit cütləri arasında "məntiqi VƏ" əməliyyatını yerinə yetirir. Əgər bit cütləri hər ikisi 1 olarsa əməliyyatın nəticəsinə 1, əks halda 0 əlavə edilir. Yəni ifadə 0x1234 & 0x00FF nəticədə say 0x0034-ü verəcək.

Bit-təyinatlı VƏ YA (|)

| operatoru hər bir operand üzərində "məntiqi VƏ YA" əməliyyatını yerinə yetirir. Əgər operand cütlərindən heç olmasa biri 1 olarsa əməliyyatın nəticəsinə 1, əks halda 0 əlavə edilir. Məsələn, 9 | 10 11 bərabərdir.

Bit-təyinatlı istisnalı VƏ YA (XOR) (^)

^ operatoru hər bir operand üzərində "məntiqi VƏ YA" əməliyyatını yerinə yetirir. Əgər operand cütləri bərabərdirsə əməliyyatın nəticəsinə 0, əks halda 1 əlavə edilir. İstisnalı VƏ YA ifadə edir ki, ya birinci, ya da ikinci operand həqiqi ədəd olmalıdır.

Aşağıdakı cədvəldə 9 və 14 ədədləri üzərində **`AND, OR, XOR`** əməliyyatları aparılır. Bir daha qeyd edək ki, əməliyyatlar ikilik say sistemində həyata keçirilir və cədvəldə əməliyyat növünün qarşısında mötərizə daxilində göstərilmiş ədəd əməliyyatın nəticəsinin onluq say sistemində təqdim edilməsidir.

9	14	AND (8)	OR (15)	XOR (7)
1001	1110	1000	1111	0111

Bit-təyinatlı İNKAR (~)

~ operatoru tək tam arqumentdən əvvəl göstərilən unar operatorudur. Bu operator operandın bütün bitlərinin inversiyasını yerinə yetirir. JavaScript-də qiymətə ~ operatorunu tətbiq etməklə onun işarə dəyişikliyinə nail olmaq olar.

Sola yerdəyişmə (<<)

<< operatoru birinci operanddakı bütün bitləri ikinci operandda göstərilmiş mövqelərin miqdarında yerləşdirir. İkinci operanda [0 - 31] aralığında tam ədəd olmalıdır. Məsələn, a<<1 əməliyyatında a operandının birinci biti ikinci bitə, ikinci biti üçüncü bitə və s. yerləşdirilir. Boş qalmış birinci bit sıfır olaraq təyin edilir və əməliyyat 32-ci bitə qədər yerinə yetirilir. Qiymətin birinci mövqe ilə sola yerləşdirilməsi birinci operandın 2-yə, ikinci mövqe ilə – 4-də və s. vurulmasına ekvivalentdir. Məsələn, 7 << 1 14-ə bərabərdir.

İşarənin saxlanmasıyla sağa yerləşdirmə

>> operatoru birinci operanddakı bütün bitləri ikinci operandda göstərilmiş mövqelərin miqdarında sağa yerləşdirir. İkinci operand 0 31 aralığında tam ədəd olmalıdır. Sağ küncdəki bitlər silinir. Ən böyük bit (32-ci) nəticənin işarəsini saxlamaq üçün dəyişmir. Əgər birinci operand müsbətdirsə, böyük bit nəticəsi sıfırlarla doldurulur; əgər birinci operand mənfidirsə, böyük bit nəticəsi vahidlərlə dolur. Qiymətin bir mövqə ilə sağa yerləşdirilməsi birinci operandın (qalığın atılmasıyla) 2-yə bölünməsinə, ikinci mövqe ilə sağa yerləşməsi 4-ə bölünmə ilə və s. ekvivalentdir. Məsələn, 7 >> 1 3-ə –7 >> 1 isə 4-ə bərabərdir.

Sıfırlar doldurmaqla sağa yerləşdirmə (>>>)

>>> operatoru >> operatoruna analojidir, lakin yerləşdirmə zamanı böyük dərəcələr birinci operandın işarəsindən asılı olmayaraq sıfırlarla dolur.

Bit-təyinatlı operatorla bağlı daha təkmil informasiya https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Bitwise_Operators ünvanında mövcuddur.

5.9. Mənimsəmə operatorları

4-cü fəsildə dəyişənlərin müzakirəsi zamanı, JavaScript-də dəyişənə qiymətin mənimsədilməsi üçün = simvolundan istifadə olunduğunu qeyd etdik. Məsələn:

```
i = 0
```

Bu sətirə JavaScript-də ifadə kimi baxmaq olmaz, hansı ki, nəticəyə malikdir, amma həqiqətdə bu ifadə və rəsmi nişanı = operatoru təşkil edir. = operatorunun sol operandı dəyişən, massiv elementi və ya obyektin xüsusiyyəti olmalıdır.
Sağ operand isə istənilən tipdə olan istənilən qiymət ola bilər. Sağ operandın qiyməti mənimsəmənin operatorunun qiymətidir. = operatorunun əlavə təsiri solda göstərilmiş dəyişənə, massivin elementinə və ya xüsusiyyətə sağ operandın qiymətinin mənimsənməsindən ibarətdir ona görə də dəyişənə, massivin elementinə və ya xüsusiyyətə növbəti dəfə müraciət edən zamanı bu qiymət alınacaq.
= operatorunu təşkil edir, daha mürəkkəb ifadələrə də daxil etmək olar. Mənimsəmə və qiymətin yoxlanılması əməliyyatlarını Belə, bir ifadədə birləşdirmək olar:

```
(a = b) == 0
```

Belə olan halda, açıq-aydın anlamaq lazımdır ki, = və == operatorları arasında böyük fərq var! Əgər ifadədə bir neçə mənimsəmə operatoru varsa, bu operatorlar sağdan sola hesablanırlar. Buna görə də, bir qiyməti bir neçə dəyişənə mənimsəyən kod yazmaq olar məsələn:

```
i = j = k = 0;
```

Yenə də xatırladaq ki, hər bir mənimsəmə ifadəsi, bərabərliyin sağ tərəfin qiymətini alır. Buna görə də, göstərilən kodda birinci mənimsəmənin (ən sağ) qiyməti ikinci mənimsəmənin (orta) sağ tərəfi olur, o da öz növbəsində üçüncü sonuncu sağ tərəfi olur.

5.9.1. Əməliyyatla mənimsətmə

JavaScript-də Adi mənimsətmə operatorundan başqa bir neçə əməliyyatla mənimsətmə operatoru var ki, bu operatorlar əməliyyatla mənimsətməni birləşdirən ixtisarlar şəklində təyin. Məsələn, += operatoru toplamanı və mənimsətməni yerinə yetirir. Aşağıdakı ifadələr ekvivalentdir:

```
total += sales_tax
total = total + sales_tax
```

+= operatoru sətirlərlə də işləyir. Əgər bu operator ədəd operandlarını toplamağı və mənimsətməni yerinə yetirirdisə, sətir operandlarında bitişdirməni və mənimsətməni yerinə yetirir. Analoji olaraq, *=, &= və başqa operatorları da uyğun əməliyyatları mənimsətmə ilə həyata keçirirlər. Bütün əməliyyatla mənimsətmə operatorları cədvəl 5.2-də göstərilmişdir.
Adətən aşağıdakı ifadələr ekvivalentdir (burada op operatoru bildirir):

```
a op = b
a = a op b
```

Bu ifadələr yalnız, a dəyişəni funksiya çağırışı və artım operatorunun tətbiqi kimi əlavə təsirləri ehtiva etdikdə fərqlənir.

5.10. Digər operatorlar

JavaScript-in bir neçə əlavə operator dəstəkləyir, hansı ki, bu operatorlar növbəti bölmələrdə təsvir ediləcək.

5.10.1. Şərt operator (?:)

Şərt operator – JavaScript-də yeganə üçlük operatordur (üç operandla) və bəzən bu operatora – "üçlük operator" da deyilir. Bu operator adətən ?: kimi yazılır, hərçənd ki, bu yazılış proqram mətnlərində müxtəlif görünür. Operator üç operanda malikdir, birinci operand ? operatorundan əvvəl, ikinci – ? və : operand operatorları, üçüncü isə : operatorundan sonra yerləşir. Bu operatorlar aşağıdakı qaydada istifadə olunur:

```
x > 0 ? x*y : -x*y
```

Şərt operatorun birinci operandı məntiqi qiymət (və ya məntiqi qiymətə dəyişə bilən qiymət) olmalıdır. Bu əksər hallarda müqayisə ifadəsinin nəticəsi olur. İkinci və üçüncü operand isə istənilən qiymət ola bilər. Şərt operatoru tərəfindən qaytarılan qiymət birinci operandın məntiqi qiymətindən asılıdır. Əgər bu operand `true` qiymətinə bərabərdirsə, onda şərt ifadəsi ikinci operandın qiymətini qəbul edir. Əgər birinci operand `false` qiymətinə bərabərdirsə, onda şərt ifadəsi üçüncü operandın qiymətini qəbul edir.
Analoji nəticəni `if` təlimatının köməyilə almaq olar, amma belə hallarda ?: operatorundan daha çox istifadə rahat olunur. Aşağıdakı nümunələrdə eyni prosesin həm şərti ifadə ilə, həm də ifadə təlimatı ilə tətbiqi göstərilmişdir:

```
greeting = "hello "+ (username != null ? username : "there");
```

Bu yazı *if*-in növbəti konstruksiyaları ekvivalentdir, amma daha (daha çox) yığcamdır:

```
greeting = "hello ";
if (username! = null)
    greeting += username;
else
    greeting += "there";
```

5.10.2. typeof operatoru

typeof unar operatoru tək operanddan əvvəl yerləşir. Bu operator istənilən tip malik ola bilər. Bu operatorun qiyməti operandın məlumat tipini göstərən sətirdən ibarətdir.
Əgər operandın qiyməti ədəd, sətir və ya məntiqi tipdədirsə bu operatorun nəticəsi müvafiq olaraq "number", "string" və ya "boolean" sətiri olacaq. Obyektlər, massivlər və *null* qiyməti (qəribə olsa da) üçün nəticə olaraq "object" sətiri olacaq. Funksiya operandı üçün "function" sətiri, amma qeyri-müəyyən operand üçün – *"undefined"* sətiri nəticə olaraq qeyd olunacaq.
Operand Number, String və ya Boolean obyekt-üzlüklərindən ibarət olarsa typeof operatorunun qiyməti "object"-ə bərabər olacaq. Həmçinin Date və RegExp obyektləri üçün də typeof operatorunun qiyməti "object"-ə bərabərdir. JavaScript-in baza dilinin bir hissəsi olmayan, amma JavaScript-də qurulan və müəyyən kontekstli verilənlərə malik obyektlər üçün typeof operatorunun qaytaracağı qiymət reallaşdırmadan asılıdır. Ancaq JavaScript-in kliyent dilində typeof operatorunun qiyməti adətən bütün kliyent və baza obyektləri üçün "object"-ə bərabərdir.
typeof operatoru belə ifadələrdə tətbiq edilə bilər, məsələn,:

```
typeof i
(typeof value == "string") ? "'" + value + "'" : value
```

typeof operandını mötərizəyə almaq olar, bu zaman typeof açar sözü operator və ya açar sözü kimi deyil, funksiya adı kimi görünür:

```
typeof(i)
```

bütün obyekt və massiv tipləri üçün t typeof operatorunun nəticəsi "object" sətiri olacaq, buna görə də bu operatorun yalnız baza tipindəki obyektləri ayırmaq üçün istifadəsi faydalı ola bilər. typeof operatorunu istənilən nəticə vermədiyi obyekt tiplərini ayırmağın digər bir üsulu, instanceof operatorundan və ya *Object*.constructor xüsusiyyətlərindən istifadə etmək olar.
Typeof operatoru ECMAScript v1 spesifikasiyasında müəyyən edilmişdir və JavaScript 1.1-də və daha əvvəlki versiyalarda realizasiya edilmişdir.

5.10.3. Obyektin yaradılmasının operatoru (new)

new operatoru yeni obyekt yaradır və inisializasiya funksiya-konstuktorunun icrasına səbəb olur. Bu operator konstruktordan əvvəl çağrılan və aşağıdakı sintaksisə malik olan unar operatorudur:

```
new konstruktor(arqumentlər)
```

Burada konstruktor – funksiya-konstuktoru nəticəsində əldə ifadədir. Konstruktor-funksiyasına mötərizələrin daxilində bir-birindən vergüllərlə ayrılmaq şərtilə istənilən qədər arqument ötürmək olar. Əgər funksiya-konstuktoruna heç bir ötürülmürsə boş mötərizələrin daxil edilməsinə ehtiyac yoxdur. Aşağıda new operatorundan istifadənin bir neçə nümunəsi göstərilmişdir:

```
o = new Object;     // Burada vacib olmayan mötərizələr ixtisara
salınmışdır
d = new Date();     // Cari vaxtı ehtiva edən Date obyektini qaytarır
c = new Rectangle(3.0, 4.0, 1.5, 2.75);           // Rectangle
sinifinin
                                                  // obyektini yaradır
obj[i] = new constructors [i] ();
```

new operatoru yeni obyekti əvvəlcə qeyri-müəyyən xüsusiyyətlərlə yaradır, sonra isə müəyyən edilmiş funksiya-konstuktorunu icra edir ki, mötərizələrin daxilində arqumentləri verərək prosesləri istədiyiniz kimi tənzimləyə bilərsiniz. həmçinin *this* açar sözünün köməyilə indi yaradılmış obyektin funksiya-konstuktoruna müraciət edə bilərsiniz. Bu sözün köməyilə funksiya-konstuktoru istənilən obrazla yeni obyektə inisializiya edilə bilər obyekt istənilən lazımlı obrazla bacarar. 7-ci fəsildə new operatoru, *this* və funksiya açar sözü konstruktorlar daha ətraflı baxılmışdır.
new operatoru həmçinin new Array() sintaksisinin köməyi ilə massivlərin yaradılması üçün tətbiq edilə bilər. Daha ətraflı obyektlərin və massivlərin yaradılması və onlarla iş haqqında biz 7-ci fəsildə danışacağıq.

5.10.4. delete operatoru

delete unar operatoru göstərilmiş obyekt xüsusiyyətini, massiv elementini və ya dəyişəni silir.[17] Operator, müvəffəqiyyətlə silinmə zamanı *true*, əks təqdirdə *false* qiymətini qaytarır. Elə dəyişənlər və xüsusiyyətlər var ki, silinə bilmir, məsələn JavaScript-in baza və kliyent dillərinin bəzi inteqrasiya edilmiş xüsusiyyətlərində silinmə əməliyyatını aparmaq olmur. Bundan başqa, var təlimatının köməyilə müəyyən edilmiş dəyişənlər, istifadəçi tərəfindən silinə bilmir. Əgər delete operatoru mövcud olmayan xüsusiyyətə tətbiq edilirsə, onda o, *true* qaytarır. (Qəribə olsa da, ECMAScript standartı müəyyən edir ki, delete operatoru həmçinin əgər operandı xüsusiyyət massiv və ya dəyişən elementi olmadıqda belə *true* qiymətini qaytarır.) aşağıdakı bu operatorun tətbiqinin bir neçə nümunəsi göstərilmişdir:

```
var o = {x:1, y:2};       // Dəyişəni müəyyən edirik
delete o.x;               // Obyektin xüsusiyyətlərindən birini
silirik; true
                          // qiyməti qaytarılır
typeof o.x;               // Xüsusiyyət mövcud deyil; "undefined"
qiyməti
                          // qaytarılır
delete o.x;               // Mövcud olmayan xüsusiyyəti silirik; true
qiyməti
                          // qaytarılır
delete o;                      // Elan edilmiş dəyişəni silmək olmaz;
false qiyməti
                          // qaytarılır
delete 1;                 // Tam ədədi silmək olmaz; true qiyməti
qaytarılır
x = 1;                    // var açar sözü olmadan dəyişəni gizli olaraq
elan
                          // edirik
delete x;                      // Bu cür elan edilmiş dəyişənləri silmək
olar; true
                          // qiyməti qaytarılır
```

Nəzərə alın ki, silinmiş xüsusiyyət, dəyişən və ya deyil massiv elementi, yenidən *undefined* təyin edilmir. Xüsusiyyət silindiyi, onun mövcudluğu da itir. Bu mövzu bölmə 4.3.2-də müzakirə edilirdi.
Başa düşmək lazımdır ki, delete operatoru xüsusiyyətin istinad etdiyi obyektə deyil, yalnız xüsusiyyətlərə təsir edə bilir. Aşağıdakı fraqmentə baxın:

```
my.hire = new Date();          // my.hire Date obyektinə istinad edir
my.fire = my.hire;        // my.fire həminki obyektə istinad edir
delete my.hire;                // hire xüsusiyyəti silinir; true qiyməti
qaytarılır
document.write(my.fire);       // Amma my.fire Date obyektinə istinad
etməyə davam
                          // edir.
```

5.10.5. void operatoru

void unar operatoru tək operanddan əvvəl göstərilir. Operand istənilən tipdə ola bilər. Bu operatorun qeyri-adi təsiri var: o operandın qiymətini tullayaraq, bu qiyməti *undefined* təyin edir. Bu operatordan əksər hallarda müştəri tərəfində URL-ünvana javascript: psevdoprotokola əlaməti vasitəsilə tətbiq edilir. Bu halda brauzerdə əməliyyatın hesablanmış qiyməti əks etdirilmədən yerinə yetirilir.
Məsələn, void operatorunu HTML-teqə tətbiq etmək olar:

```
<a href="javascript:void window.open();">Yeni pəncərə aç</a>
```

void operatoru ECMAScript v1-də təyin edilir və JavaScript 1.1-də realizasiya olunur. JavaScript 1.5-də reallaşdırılmış ECMAScript v3-də *undefined* qlobal xüsusiyyəti təyin edilir. Ancaq versiyalar arasında uyğunluğun saxlanılması üçün *undefined* xüsusiyyətinə deyil, void 0 kimi ifadəyə müraciət etmək daha yaxşıdır.

5.10.6. "vergül" operatoru

"vergül" operatoru (,) çox sadə operatordur. O öz sol və sağ operandını hesablayır və sağ operandın qiymətini qaytarır. Məsələn

```
i=0, j=1, k=2;
```

Burada vergül operatoru 2-qiymətini qaytarır və praktik olaraq aşağıdakı ekvivalentdir:

```
i = 0;
j = 1;
k = 2;
```

Bu qəribə operatorun yalnız məhdud hadisələrdə tətbiqi faydalıdır; əsasən onda, nə vaxt ki, orada əlavə təsirlərlə bir neçə (bir qədər) müstəqil ifadə hesablamaq tələb olunur, harada yalnız bir ifadəni buraxılır. Təcrübədə "vergül" operatoru faktiki olaraq yalnız *for* təlimatıyla uyğunluqda istifadə olunur, hansını ki, biz 6-cı fəsildə baxacağıq.

5.10.7. Massivlərə və obyektlərə girişin operatorları

3-cü fəsildə qeyd edildiyi kimi, massivin elementlərinə kvadrat mötərizələr ([]) vasitəsilə, obyektin elementlərinə isə nöqtə (.) vasitəsilə müraciət etmək olar. JavaScript-də kvadrat mötərizələrə və nöqtəyə operatorlar kimi baxılır.
"nöqtə" operatoruna sol operand kimi obyekt, sağ operator kimi – identifikator (xüsusiyyətin adı) tələb olur. Sağ operand sətir və ya sətiri ehtiva edən dəyişən ola bilməz; bu operand hər hansı dırnaqsız xüsusiyyətin və ya metodun dəqiq adı olmalıdır. Məsələn:

```
document.lastModified
navigator.appName
frames[0].length
document.write("hello world")
```

Əgər obyektdə göstərilən xüsusiyyət yoxdursa, JavaScript interpretatoru xəta yaratmır, amma nəticəni *undefined* ifadəsinin qiyməti kimi qaytarır.
Operatorların əksəriyyəti öz operandları üçün sərbəst ifadələri icazə verir. "nöqtə" operatoru istisnalıq təşkil edir: sağ operand qeyd-şərtsiz identifikator olmalıdır.
[] operatoru massivin elementlərinə girişi təmin edir. O həmçinin "nöqtə" operatorunun sağ operandında göstərilən obyektin xüsusiyyətlərinə məhdudiyyətsiz girişi təmin edir. Əgər birinci operand (sol mötərizədən əvvəl göstərilmiş) massivə istinad edirsə, onda ikinci operand (mötərizələrin arasında göstərilmiş) tam qiymətə malik olan ifadə olmalıdır. Məsələn:

```
frames[1]
document.forms[i + j]
document.forms[i].elements [j++]
```

Əgər [] operatorunun birinci operandı obyektə istinad edirsə, onda ikinci operand obyektin xüsusiyyəti adına uyğun olan sətir olmalıdır. Nəzərə alın ki, bu halda ikinci operand identifikator deyil, sətir olur. Bu sətir, dırnaqlarla bağlanmış sabitlə, sətirə istinad edən dəyişənlə və ya ifadəylə ola bilər. Məsələn:

```
document["lastModified"]
frames[0]['length']
data["val" + i]
```

[] operatoru adətən massivin elementlərinə müraciət üçün tətbiq edilir. Obyektin xüsusiyyətlərinə giriş bu operatorun tətbiqi nöqtə operatoruna nisbətən narahatdır, çünki, xüsusiyyət adları dırnaqlarla əhatələnməlidir. Ancaq obyekt assosiativ massiv rolunda çıxış etdiyi zaman, xüsusiyyətlərin adları dinamik yaradılır, buna görə də "nöqtə" operatorundan istifadə oluna bilmir. Məhz belə hallarda [] operatoru tətbiq edilir. Belə vəziyyət ***6-cı fəsildə baxılacaq*** *for*/*in* dövründə müşahidə ediləcək. Məsələn, obyektdəki adların nəticəsi və həmin obyektin bütün xüsusiyyətlərinin qiymətləri üçün aşağıdakı fraqmentdə *for*/*in* dövründən və [] operatorundan istifadə olunur:

```
for (f in o) {
  document.write('o.' + f + ' = ' + o[f]);
  document.write('<br>');
}
```

5.10.8. Funksiya çağırışı operatoru

JavaScript-də funksiyaların çağırışı () operatoru üçün nəzərdə tutulmuşdur. Bu operatorun qeyri-adiliyi ondadır ki, bu operatorda istənilən qədər operanddan istifadə edilə bilər. Birinci operand – həmişə funksiyaya istinad edən funksiyanın adı və ya ifadədir. Onu sol mötərizə vasitəsilə, istənilən miqdarda əlavə operandlar müşahidə edirki, bu operandlar, bir-birindən vergüllərlə ayırılmış sərbəst ifadə ola bilər. Son operandı sağ mötərizə müşahidə edir. () operatoru göstərilmiş operandları hesablayır və sonra hesablanmış operandları arqument kimi birinci operandla verilmiş funksiyaya ötürərək, funksiyanın icrasına səbəb olur. Məsələn:

```
document.close()
Math.sin(x)
alert("Welcome " + name)
Date.UTC(2000, 11, 31, 23, 59, 59)
funcs[i].f(funcs[i].args[0], funcs[i].args[1])
```

6

Təlimatlar

Əvvəlki fəsildə biz ifadələrlə tanış olduq. ***İfadə*** – JavaScript dilində "frazalardır" və ifadələrin hesablanması nəticəsində qiymətlər alınır. İfadələrə daxil olan operatorlar əlavə təsirlərə malik ola bilər, amma adətən ifadənin özü heç bir təsirə malik olmur. Bu fəsildə JavaScript-təlimatlarının təyinatı və sintaksis müxtəlifliyi öyrənilir. JavaScript, proqram təlimatlar dəstini təşkil edir. Siz bu təlimatlarla tanış olduqda, proqramların yazılışına başlaya bilərsiz.
JavaScript-də təlimatlar haqqında danışmazdan əvvəl, xatırlamaq lazımdır ki, 2.4 fəsildə deyilirdi ki, JavaScript-də təlimatlar, bir- birindən nöqtəli vergüllərlə ayrılır. Ancaq, əgər hər təlimat ayrı-ayrı sətir(lər)də yerləşirsə, JavaScript interpretatoru onların yoxluğunu güman edir və nöqtəli vergülü qoyulmuş kimi qəbul edir. Amma, arzuolunandır ki, yerindən asılı olmayaraq, nöqtəli vergülləri qoymağı özünüzə vərdiş edəsiniz.

6.1. Təlimat – ifadə

JavaScript-də təlimatların hərəkəti, təlimatların ən sadə növüdür. Bu əlavə effektlərə malik olan ifadələrdir. Biz 5-ci fəsildə bu barədə öyrəndik. Təlimat-ifadələrinin əsas kateqoriyası – mənimsəmə təlimatlarıdır. Məsələn:

```
s = "Salam "+ name;
i *= 3;
```

++ və -- yəni, artımın və dekrementin operatorları, mənimsəmə operatorlarına qohumdur. Mənimsəmənin icra edilməsi zamanı qiymət dəyişikliyi onlarada təsir göstərir:

```
counter++;
```

delete operatoru əhəmiyyətli əlavə təsirə, yəni obyektin xüsusiyyətinin silinməsi xüsusiyyətinə malikdir. Buna görə o demək olar ki həmişə təlimat kimi, qısa ifadələrdə kimi deyil, daha mürəkkəb ifadələrdə tətbiq edilir:

```
delete o.x;
```

Funksiyaların çağrılması – təlimatı ifadələrinin daha bir böyük kateqoriyasıdır. Məsələn:

```
alert("Xoş gəlmisiniz, "+ name); window.close();
```

Kliyent funksiyalarının bu cür çağırışları ifadələrdən, eləcə də təlimatlardan təşkil olunur, buna görə də ancaq onlar sadəcə veb-brauzerə təsir edir.

Əgər funksiya hər hansı əlavə təsirlərə malik deyilsə və hər hansı dəyər ifadə edirsə bu funksiya hər hansı bir dəyişənə mənimsədilir. Məsələn, praktikada heç kimə sadəcə kosinusu hesablamaq və nəticəni göstərmək maraqlı deyil:

```
Math.cos(x);
```

Əksinə, qiyməti hesablamaq və alınan qiyməti istifadə etmək dəyişənə mənimsətmək lazımdır:

```
cx = Math.cos(x);
```

Yenə də fikir verin ki, nümunələrdəki hər bir sətir nöqtəli vergüllə bitir.

6.2. Tərkib təlimatlar

5-ci fəsildə gördük ki, bir neçə ifadəni "vergül" operatoru vasitəsilə bir dəyişəndə birləşdirmək olar. JavaScript-də həmçinin bir təlimata və ya təlimatlar blokuna bir neçə təlimat birləşdirmək olar. Bunu fiqurlu mötərizələrin daxilində istənilən qədər təlimatlar yerləşdirməklə etmək olar. Beləliklə, JavaScript interpretatoru tək təlimatın

mövcudluğunu tələb etdiyi yerlərdə aşağıdakı sətirlərə bir təlimat kimi baxılır və hər yerdə istifadə oluna bilər:

```
{
  x = Math.PI;
  cx = Math.cos(x);
  alert("cos(" + x + ") = " + cx);
}
```

Nəzərə alın ki, təlimatlar bloku bir təlimat kimi hesab edilsədə, nöqtəli vergüllə qurtarmır. Blokun daxilində ayrı təlimatlar nöqtəli vergüllərlə qurtarır, ancaq blokun özü nöqtəli vergüllə bitmir.
Əgər "vergül" operatorunun köməyi ilə ifadələrin birləşməsi nadir hallarda istifadə olunursa, onda kodun bloklarına təlimatların birləşməsi hər yerdə yayılmışdır. Necə biz sonrakı bölmələrdə görəcəyik, bəziləri özü JavaScript-təlimatlar başqa təlimatları özündə saxlayırlar (kimi ifadələr başqa ifadələri özündə saxlaya bilər); ***Belə təlimatlar tərkib adlanırlar.***
JavaScript-in formal sintaksisi müəyyən edir ki, bu tərkib təlimatlardan hər biri tək alt-təlimatı ehtiva edir. Təlimat bloklarında istənilən qədər təlimat yerləşdirmək olar ki, burada heç olmasa bir alt-təlimatının mövcudluğu mütləqdir.
JavaScript interpretatoru tərkib təlimatı icra edərkən, daxilində digər təlimatları ehtiva edən təlimatı sicra edir. Adətən interpretator bütün təlimatları icra edir, ancaq bəzi hallarda tərkib təlimatın icrası qəflətən kəsilə bilər. Bu o hallarda olur ki, tərkib təlimatda `break`, `continue`, `return` və ya `throw` təlimatı olsun və icra və ya funksiya çağırışı zamanı xəta və ya emal edilməyən informasiya yaransın. İşin bu cür qəfildən dayandırmaları haqqında biz sonrakı bölmələrdə daha ətraflı biləcəyik.

6.3. if təlimatı

`if` təlimatı – JavaScript İnterpretatoruna təlimatları icra etmək şəraitdən asılı olaraq qərarlar qəbul etməyə imkan idarəedici təlimatdır. Təlimat iki formaya malikdir. Birincisi: `if` təlimatının bu formasında əvvəlcə ifadə

hesablanır. Əgər nəticə *true* qiymətinə bərabərdirsə və ya *true* qiymətinə dəyişdirilibsə, onda təlimat həyata keçirilir. Əgər ifadə *false* qiymətinə bərabərdirsə və ya *false* qiymətinə dəyişdirilibsə, onda təlimat həyata keçirilmir. Məsələn:

```
if(username == null)   // Əgər username dəyişəni null-a və ya
undefined-ə
                  // bərabərdirsə
username = "John Doe"; // onu müəyyən edirik
```

Analoji olaraq:

```
     // Əgər username dəyişəni null, undefined, 0, "" və ya NaN
qiymətinə
     // bərabərdisə onda false qiymətinə dəyişdirirləcək və bu
təlimat dəyişənə
     // yeniqiymət mənimsədəcək
     if(!username) username = "John Doe";
```

İfadənin ətrafında olan mötərizələr artıq görünsə belə, bu mötərizələr *if* təlimatının sintaksisinin icbari hissəsidir. Əvvəlki bölmədə qeyd edildiyi kimi, biz həmişə tək təlimatı, təlimatlar bloku ilə əvəz edə bilərik. Buna görə də *if* təlimatı bu cür də yazıla bilər:

```
if((address == null) || (address == ""))
{
  address = "undefined";
  alert("Zəhmət olmasa, poçt ünvanını göstərin.");
}
```

Bu nümunələrdə olan boşluqlar məcburi deyil. Onsuzda, JavaScript-də əlavə boşluqlar və tabulyasiyalara icazə məhəl qoymur, ona görə, biz sonra qoyurduq hər ayrı təlimata nöqtəli vergülü, bu nümunələr bir sətiri yazılmış ola bilirdi. Burada göstərildiyi kimi sətir və boşluq simvollarının istifadəsi ilə mətnin daha rahat oxunması və kodun başa düşülməsini yüngülləşdirir.

if təlimatının ikinci forması ifadə *false* qiymətini aldıqda icra edilən *else* konstruksiyasını ehtiva edir. İkinci formanın sintaksisi belədir:

```
if(ifadə)
  təlimat1
else
```

```
    təlimat2
```

Təlimatın bu formasında əvvəlcə ifadə hesablanır, əgər ifadə *true* qiymətinə bərabərdirsə, onda təlimat1 həyata keçilir, əks təqdirdə isə təlimat2 həyata keçirilir. Məsələn:

```
if(username != null)
   alert("Salam"+ username + "\nMənim ana səhifəmə xoş
 gəlmisiniz.");
else
{
   username = prompt("Xoş gəlmisiniz!\n Sizi kim deyə
çağırırlar?");
   alert("Salam " + username);
}
```

if təlimatları *else* blokları ilə qurulduğu zaman bəzi məqamlara diqqət yetirmək lazımdır, yəni *else*-nin *if* təlimatına uyğun olmasından əmin olmaq lazımdır. Aşağıdakı sətirlərə baxaq:

```
i = j = 1;
k = 2;
if (i == j)
    if (j == k)
      document.write("i k-ya bərabərdir");
else
   document.write("i j-yə bərabər deyil"); // YANLIŞ!!
```

Bu nümunədə daxili *if* təlimatı qismində yeganə xarici *if* təlimatı iştirak edir. Təəssüf ki, *else* blokunun *if*-in hansı təlimatına aid olduğu aydın deyil. Və bu nümunədə boşluqlar yanlış qoyulmuşdur, axı JavaScript interpretatoru əvvəlki nümunəni real olaraq belə izah edir:

```
if (i == j)
{
  if (j == k)
     document.write("i k-a bərabərdir");
  else
     document.write("i j-a bərabər deyil"); // !
}
```

JavaScript qızıl qaydası (və digər proqramlaşdırma dillərinin əksəriyyəti): *else* **konstruksiyası ona ən yaxın** *if* **təlimatının hissəsidir.**

Bu nümunəni oxumaq, anlamaq, müşayiət etmək üçün daha sadə və daha yüngül üsul, fiqurlu mötərizələrdən istifadə etmək lazımdır:

```
if (i == j)
{
  if (j == k)
  {
    document.write("i k-ya bərabərdir");
  }
}
else
{
  // Bütün bu təzadlar fiqurlu mötərizələrin yerləşməsindən
yaranır!
  document.write("i j-yə bərabər deyil");
}
```

Əksər proqramçılar fiqurlu mötərizələrinin köməyilə sistemli şəkildə *if* və *else* təlimatlarının (həmçinin digər tərkib təlimatlarında, məsələn, *while* dövrlərində olduğu kimi) təlimat gövdəsini qururlar. Həmişə bu qaydaları tətbiqi etsəniz və özünü bu üsula öyrəşdirsəniz, xoşagəlməz hadisələrdən rastlaşmayacaqsınız.

6.4. else if təlimatı

Biz *if/else* təlimatının şərtin yoxlanması və yoxlamanın nəticəsindən asılı olaraq kodun iki fraqmentindən birinin icrası üçün istifadə olunduğunun şahidi olduq. Amma, əgər kodun bir neçə fraqmentindən yalnız birini yerinə yetirmək tələb olunursa nə etməli? Bunu *else if* təlimatının tətbiqi ilə etmək mümkündür. Bu təlimatı rəsmi olaraq, JavaScript-təlimatı sayılmır və bu yalnızca proqramlaşdırmada təkrarlanan *if/else* təlimatlarının tətbiqində istifadə olunan məşhur üsuldur:

```
if(n == 1)
{
   //Kod 1 blokunu icra edirik
}
else if(n == 2)
{
   //Kod 2 blokunu icra edirik
```

```
}
else if(n == 3)
{
  //Kod 3 blokunu icra edirik
}
else
{
   //Əgər bütün şərtləri yerinə yetirilmirsə, blok 4- ü icra
edirik
}
```

Bu fraqmentdə xüsusi heç nə yoxdur. Bu sadəcə *if* təlimatlarının ardıcıllığıdır, yəni *if*-in hər təlimatı əvvəlki *else* konstruksiyasının hissəsidir. *else if* stilinə ekvivalent, ondan daha üstün və daha aydın sintaksisi bu cür göstərmək olar:

6.5. switch təlimatı

if təlimatı proqramın icra gedişatında budaqlanma yaradır. Əvvəlki bölmədə göstərildiyi bir neçə *if* təlimatı vasitəsi ilə çoxölçülü budaqlanma yaratmaq olar. Ancaq bu həmişə ən optimal çıxış yolu sayılmır. Xüsusilə əgər bütün budaqlar bir dəyişənin qiymətindən asılıdırsa, hər bir *if* təlimatında eyni dəyişənin qiymətini təkrar-təkraq yoxlamaq bəyənilməyən proqramlaşdırma üsuludur. Məhz belə vəziyyətlərdə *switch* təlimatı təkrarlanan *if* təlimatlarına nisbətən daha çox effektivdir. JavaScript-də *switch* təlimatı Java və ya C-dəki *switch* təlimatına çox oxşardır. *Switch* təlimatını demək olar ki, *if* təlimatındakı kimi ifadə və kod bloku müşahidə edir:

```
switch(ifadə)
{
  təlimatlar
}
```

Lakin, *switch* təlimatının tam sintaksisi burada göstəriləndən də daha yığcamdır. Kod blokunun müxtəlif yerlərində *case* açar sözü ilə birlikdə iki nöqtə simvolu vasitəsilə tətbiq edilən nişanlar təyin edilir. *switch* təlimatı yerinə yetirilirdiyi zamanı, təlimat ifadənin qiymətini hesablayır

və sonra bu qiymətə uyğun olan *case* nişanını axtarır. Əgər uyğun nişan tapılarsa, həmin nişanı müşahidə edən birinci təlimatdan başlayaraq, kod bloku icra edilir. Əgər qiymətə uyğun *case* nişanı tapılmasa, icra prosesi xüsusi hallarda təyin edilən *default* nişanını müşahidə edən kod blokuna ötürülür. Əgər *default* nişanları təyin edilməyibsə, kod bloku tamamilə sərbəst buraxılır. *Switch* təlimatının iş prosesini sözdə izah etmək çətindir, buna görə gəlin nümunələrə keçək. Aşağıdakı *switch* təlimatı əvvəlki bölmədə göstərilmiş təkrarlanan *if/else* təlimatlarına ekvivalentdir:

```
switch(n)
{
 case 1:           // əgər n == 1 olarsa,
                  // 1-ci kod bloku yerinə yetirilir.
    // Burada isə dayandırılır.
    break;

 case 2:           // əgər n == 1 olarsa,
                  // 2-ci kod bloku yerinə yetirilir.
     // Burada isə dayandırılır.
         break;

 case 3:           // əgər n == 1 olarsa,
                   // 3-cü kod bloku yerinə yetirilir.
    // Burada isə dayandırılır.

    break;
 default:          // Əgər seçimlərin heç biri əlverişli
deyilsə,
                  // 4-cü kod bloku yerinə yetirilir.
    // Burada isə dayandırılır.
    break;
}
```

Bu fəsilin davamında təsvir edilən *break* təlimatı *switch* və ya dövr təlimatını sona çatdırır. *Switch* təlimatındakı *case* konstruksiyaları icra edilən kodun yalnız başlanğıc nöqtəsini verir. *Break* təlimatlarının yoxluğu halında *switch* təlimatı ifadənin qiymətinə uyğun olan *case* nişanındakı kod blokunun icrasını başlayır və icra blokun sonuna qədər davam edir. Nadir hallarda *break* istifadəyə lüzum yoxdur, lakin praktikada bu təlimatla yazılan kodların 99 %-ində hər bir *case* blokunda

break təlimatının tətbiqi vacibdir. (*switch* təlimatından funksiyanın daxilində istifadə edilərsə, *break* təlimatının yerinə *return* təlimatını da yerləşdirmək olar. Hər iki təlimat *switch* təlimatının işi başa çatdırmasına və növbəti *case* nişanına keçidin qarşısının alınmasına xidmət edir.)

Aşağıda *switch* təlimatından istifadənin daha praktik nümunəsi göstərilmişdir; burada sətir qiyməti göstərilən qiymət tipindən asılı olaraq dəyişdirəcək:

```
function convert(x)
{
switch(typeof x) {
case 'number':          // Ədədi onaltılıq say siteminə
dəyişdiririk
return x.toString(16);
case 'string':          // Dırnaqlarla bağlanmış sətir
qaytarılır
return '"' + x + '"';
case 'boolean':           // Böyük hərf registri ilə TRUE və
FALSE-yə
                        // dəyişdirilir
return x.toString().toUpperCase();
default:                  // İstənilən başqa tip sətirə
çevrilir
return x.toString()
}
}
```

Nəzərə alın ki, əvvəlki iki nümunədə *case* açar sözünü həm ədəd, həm də sətir literalları müşahidə edir. Praktikada *switch* təlimatı məhz bu cür istifadə olunur, ECMAScript v3 standartı *case* açar sözündən sonra sərbəst ifadələrin daxil edilməsinə icazə verir[18]. Məsələn:

```
case 60*60*24:
case Math.PI:
case n+1:
case a[0]:
```

switch təlimatı əvvəlcə switch açar sözünün qarşısındakı ifadəni hesablayır və sonra müvafiq qiymət tapılmayana qədər sıra ilə case ifadələrini hesablayır. Uyğunluq faktı bərabərlik operatoru ilə deyil,

eynilik operatoru ilə təyin edilir və buna görə də ifadələr hər hansı tip dəyişikliyi olmadan uyğun gəlməlidir.
Nəzərə alın ki, əlavə təsirlərə malik (mənimsəmə və funksiya çağırışları, artım və s. kimi) ifadələri `case` açar sözündə tətbiq etmək tövsiyə edilmir, çünki hər bir `switch` təlimatının icrası zamanı bütün `case` ifadələri hesablanmır. Əlavə təsirlərə malik ifadələrin tətbiqi yalnız bəzi hallarda işə yarayır ki, bu zaman da proqramın davranışını qabaqcadan anlamaq çətin olur. Bütün `case` ifadələrini sabitlərlə məhdudlaşdırılmaq daha təhlükəsiz üsuldur.
Əvvəl də qeyd etdiyimiz kimi, əgər heç bir `case` ifadələrindən heç biri `switch` ifadəsinə uyğun deyilsə, `switch` təlimatı icranı `default` nişanını müşahidə edən təlimatdan başlayır. Əgər `default` nişanı yoxdursa, onda `switch` təlimatı tamamilə sərbəst buraxılır. Nəzərə alın ki, əvvəlki nümunələrdə `default` nişanı, bütün `case` nişanlarından sonra, `switch` təlimatının gövdəsinin sonunda yerləşdirilmişdir. Bu nişan üçün bu yer məntiqli yerdir, amma əslində bu nişan, `switch` təlimatının daxilində istənilən yerdə yerləşdirilə bilər

6.6. while təlimatı

`if` təlimatı JavaScript interpretatoruna qərarlar qəbul etməyə imkan verən baza idarəetmə təlimatı olduğu kimi `while` təlimatı da – JavaScript interpretatoruna dövri prosesləri yerinə yetirməyə imkan verən baza təlimatıdır.
Təlimat aşağıdakı sintaksisə malikdir:

```
while(ifadə)
    təlimat
```

`while` təlimat işini ifadənin hesablanmasından başlayır. Əgər o `false`-yə bərabərdirsə, JavaScript interpretatoru proqramın aşağıdaki təlimatına keçir, əks təqdirdə, yəni `true` qiymətinə bərabərdisə, onda dövrün gövdəsində yer alan təlimat(lar) icra edilir və ifadə yenidən hesablanır. İfadənin hesablanması, təlimatda göstərilən ifadə `false`-yə bərabər

olana qədər davam edəcək. Yadda saxlayın ki, ifadə *false* olmayınca *false*-yə bərabər olmayana dövr davam edəcək. *while* (*true*) sintaksisinin köməyi ilə sonsuz silsilə yaratmaq olar.
Adətən JavaScript İnterpretatoruna eyni əməliyyatın yenidən təkrar-təkraq yerinə yetirilməsi tələb olunmur. Demək olar ki, hər dövrün, dövr iterasiyasında bir və ya bir neçə dəyişən öz qiymətini dəyişir. Çünki dəyişən, dövrdə göstərilən təlimatların icrasında iştirak edir və hər dövrdə fərqli qiymət alır. Bundan başqa, əgər dəyişən (və ya dəyişənlər) ifadədə yer alıbsa, ifadənin qiyməti hər dövrdə dəyişə bilər. *Ümumiyyətlə dövr prosesində qiymət dəyişikliyi əhəmiyyətli hadisədir, çünki əks təqdirdə, qiyməti true olan ifadənin qiyməti dəyişməyəcək və dövr heç vaxt bitməyəcək!* *While* dövrünün nümunəsi:

```
var count = 0;
while (count < 10)
{
  document.write(count + ""); count++;
}
```

Gördüyünüz kimi, nümunədə count dəyişəninə başlanğıcında 0 qiymət mənimsədilir, sonra isə onun qiyməti dövr gövdəsi icra edildiyi zaman hər dəfə böyüyür. Dövr 10 dəfə yerinə yetirildikdən sonra, ifadə *false*-yə bərabər olur (yəni. count dəyişəni artıq 10-dan kiçik olmur) və *while* təlimatı bitir. Bundan sonra JavaScript proqramın növbəti təlimatına keçir. Dövrlərin əksəriyyəti count dəyişənə analoji olaraq sayğaclara malikdirlər. Əksər hallarda dövr sayğaclarında i, j və k adlı dəyişənlərdən istifadə olunur, hərçənd ki, kodu daha aydın etmək üçün, sayğaclara daha aşkar adlar vermək lazımdır.

6.7. do/while dövrü

do/while dövrü bir çox əlamətinə *while* dövrünə bənzəyir, lakin *do/while* dövründə ifadə dövrün başlanğıcında deyil, sonunda yoxlanılır. Bu isə o deməkdir ki, dövrün gövəsi heç olmasa bir dəfə icra edilir. Bu dövrün sintaksisi belədir: *do while* (ifadə) təlimatı; *do/while* Dövrü *while* dövrünə nisbətən az istifadə olunur. Məsələ

ondadır ki, təcrübədə nadir hallarda bu dövrün iş prinsipinə uyğun vəziyyət yaranır. Məsələn:

```
function printArray(a)
{
  if (a.length == 0)
     document.write("Boş massiv");
  else
  {
     var i = 0;
     do
     {
       document.write(a [i] + "");
     }
     while(++i < a.length);
  }
}
```

do/while və *while* dövrü arasında iki böyük fərq var. Birincisi, bu dövr *do* (dövrün başlanğıc qiyməti üçün) və *while* (dövrün sonunun qiyməti və şərtinin göstərilməsi üçün) açar sözlərindən istifadə olunmalıdır. İkincisi, *while* dövründən fərqli olaraq, *do* dövrü nöqtəli vergüllə bitir. Bunun səbəb odur ki, *do* dövrü dövrü, dövrün gövdəsini əhatələyən fiqurlu mötərizələrlə deyil, dövrün sonunu ifadə ilə bitir.

6.8. for təlimatı

for təlimatından başlanan dövr *while* dövrünə nisbətən daha rahat istifadəyə malikdir. *for* təlimatı əksər dövrlər üçün ümumi şablondan istifadə edir (həmçinin daha əvvəl göstərilən *while* dövrünün nümunəsi). Dövrlərin əksəriyyəti bir neçə dəyişən-sayğacına malikdir. Bu dəyişən dövrün başlanğıcından əvvəl inisializasiya olunur və hesablanan ifadədə hər dövr iterasiyasından əvvəl yoxlanır. Və nəhayət, dəyişən-sayğacı inkrementizasiya edirsə və ya dövrün gövdəsinin sonunda hər hansı başqa obrazla dəyişikliyə məruz qalırsa, bu dəyişiklik ifadənin təkrar hesablamasından bilavasitə qabaq həyata keçirilir.
İnisializasiya, yoxlama və yeniləmə – dövr dəyişənləri ilə yerinə yetirilən üç əsas əməliyyatdır; *for* təlimatı bu üç addımı dövrün sintaksisinin açıq hissəsində edir. Bu *for* dövrü ilə yerinə yetirilən

əməliyyatların başa düşülməsini yüngülləşdirir və inisializasiya və ya dövr dəyişəninin inkrementizasiyasının unudulması kimi xətaların qarşını alır. *For* dövrünün sintaksisi:

```
for(inisializasiya; yoxlama; artım)
təlimat
```

for dövrünün iş prinsipini ona ekvivalent olan *while* dövrü ilə göstərib, izah edək[19]:

```
inisializasiya;
while (yoxlama){
      təlimat
      artım;
}
```

Başqa sözlə, inisializasiya olunan ifadə bir dəfə dövrün başlanğıcından əvvəl hesablanır. Bu ifadə, bir qayda olaraq, əlavə təsirlərə malik (adətən mənimsəmə) ifadədir, ifadə çünki yalnız bu zaman faydalı olmalıdır. 1 bizim Kimi
JavaScript həmçinin, inisializasiya olunan ifadədə var təlimatı vasitəsilə dəyişəninin elan edilməsinə icazə verir, buna görə də eyni zamanda dəyişəni elan etmək və dövr sayğacı insiallaşdırmaq olar. Yoxlama ifadəsi hər iterasiyadan əvvəl hesablanır və dövrün gövdəsinin yerinə yetirilib-yetirilməyəcəyini müəyyən edir. Əgər yoxlamanın nəticəsi *true* olarsa, dövr gövdəsində olan təlimat(lar) yerinə yetirilir. Dövrün sonunda isə, artım ifadəsi hesablanır. Və bu ifadə, hər hansı fayda vermək üçün, əlavə təsirlərlə malik olan ifadə olmalıdır. Burada adətən adi və ya mənimsəmə ifadələrində ++ və --operatordan istifadə edilir.
İndi isə gəlin, əvvəlki bölmədə *while* dövrü vasitəsilə 0-dan 9-a qədər rəqəmləri ekranlaşdıran nümunəni *for* dövrü ilə tətbiq edək:

```
for (var count = 0; count < 10; count++)
     document.write(count + "");
```

Nəzərə alın ki, bu cür sintaksis dövrün işini daha aydın edərək, dövr dəyişəni haqqında bütün vacib məlumatları bir sətirdə yerləşdirir. Bundan başqa, artım-ifadəsinin *for* təlimatında yerləşdirilməsi öz-özlüyündə

dövr gövdəsini sadələşdirir; gördüyünüz kimi, hətta bizim təlimatlar blokunun formalaşdırılması üçün fiqurlu mötərizələrə ehtiyacımız olmadı. Əlbəttə, bu nümunələrə nisbətən daha mürəkkəb dövrlər də yaradıla bilər və bu halda bəzən dövrün hər iterasiyasında bir neçə dəyişəninin dəyişdirilməsinə ehtiyac yarana bilər. Bu vəziyyətdə – JavaScript-də nadir hallarda istifadə edilən "vergül" operatorundan istifadə edilir. Sözügedən operator vasitəsilə *for* dövründə bir inisializasiya və inkrementizasiya olunan ifadəyə bir neçə ifadə birləşdirmək olur. Məsələn:

```
for (i = 0, j = 10; i < 10; i++, j)
     sum += i * j;
```

6.9. for/in təlimatı

JavaScript-də *for* açar sözü iki təlimatda istifadə edilis. Bunlardan birinin *for* dövründə şahidi olduq. İkincisi isə, *for/in* təlimatında istifadə olunur. Bu təlimat – aşağıdakı sintaksisə malik olan, dövrün bir başqa növüdür:

```
for (dəyişən in obyekt)
     təlimat
```

Burada dəyişən, var təlimatı ilə də elan edilə bilər. Dəyişən qismində, massiv elementi, obyekt xüsusiyyəti, əlqərəz mənimsəmənin sol hissəsinin yarıyan istənilən informasiya iştirak edə bilər. Obyekt parametri – obyektin adı və ya nəticəsi obyekt olan ifadədir. Və həmişəki kimi, təlimat – dövrün gövdəsini yaradan təlimatlar blokudur.
Massivin elementləri *while* və ya *for* dövrünün gövdəsinin hər icrası zamanı indeks dəyişəninin artırılması seçmək kifayət qədər asandır. *For/in* təlimatı obyektin bütün xüsusiyyətlərinin izafə vasitəsini verir. *For/in* dövrünün gövdəsi obyektin hər bir xüsusiyyəti üçün bir dəfə həyata keçirilir. Dövrün gövdəsindən icrasından əvvəl obyektin xüsusiyyətlərindən hər hansı birinin adı sətir şəklində dəyişənə mənimsədilir. Sonra isə bu dəyişəni, dövrün gövdəsində [] operatorun köməyilə obyektin xüsusiyyət qiymətinin alınması üçün istifadə etmək

olar. Məsələn, aşağıdakı *for/in* dövrü, obyektin bütün xüsusiyyətlərinin qiymətlərini və adlarını çap edir:

```
for (var prop in my_object)
{
   document.write ("ad: "+ prop + "; qiymət (məna): "+
my_object[prop], "");
}
```

Nəzərə alın ki, *for/in* dövründə dəyişən qismində mənimsəmənin sol hissəsinə yarayan istənilən ifadə ola bilər. Bu ifadə dövrün gövdəsinin hər çağırışı zamanı hesablanır, yəni bu ifadə hər dəfə müxtəlif qiymət ala bilər. Obyektin bütün xüsusiyyətlərinin adlarını bu cür massivə kopyalamaq aşağıdakı qaydada mümkündür:

```
var o = { x:1, y:2, z:3 };
var a = new Array();
var i = 0;
for (a [i++] in o); /* boş dövr gövdəsi */
```

*JavaScript-də **massivlər** – sadəcə obyektlərin xüsusi tipidir.* Beləliklə, *for/in* dövrü massivin elementləri kimi obyektin xüsusiyyətlərinin də izafəsi üçün istifadə oluna bilər. Məsələn, yuxarıda göstərilmiş koda bu sətiri əlavə etdikdə, təlimat 0, 1 və 2 massiv "xüsusiyyətlərinin" bir-bir sadalayır:

```
for (i in a) alert(i);
```

for/in dövrü obyektin xüsusiyyətlərinin mənimsədildiyi dəyişəni sıralamır. Ona görə də dəyişəndəki qiymətlərin nə cür sıralandığını qabaqcadan bilmək olmur, çünki JavaScript-in reallaşdırmaları arasında davranışlar müxtəlif ola bilər. Əgər sadalanmamış obyektin xüsusiyyəti *for/in* dövrünün gövdəsində silinərsə, onda bu xüsusiyyət sadalanmayacaq. Əgər yeni xüsusiyyətlər dövrün gövdəsində müəyyən edilərsə, onda bu xüsusiyyətlərin sadalanması reallaşdırmadan asılıdır.
for/in dövrü əslində bütün obyektlərin bütün xüsusiyyətlərini seçmir. Obyektlərin bəzi xüsusiyyətlərini yalnız oxuma və ya silinməyə qarşı sığortalı yaratmaq mümkün olduğu kimi, sadalanmayan xüsusiyyətlər yaradıla bilər. Məhz belə xüsusiyyətlər *for/in* dövrüylə sadalanmır. Əgər istifadəçi tərəfindən müəyyən edilmiş bütün xüsusiyyətlər

sadalanırsa, onda metodlar daxilinə inteqrasiya edilmiş xüsusiyyətlər sadalanmır. 7-ci fəsildə şahidi olacağıq ki, obyektə digər obyektlərdən xüsusiyyətlər varis edilə bilər. İstifadəçi tərəfindən müəyyən edilmiş varis olunmuş xüsusiyyətlər də, *for/in* dövrüylə sadalanır.

6.10. Nişanlar

Case və *default* nişanları: *switch* təlimatına uyğun olan – daha çox ümumi hadisənin xüsusi variantıdır. İstənilən təlimatın qarşısında göstərilmiş nişan identifikatorun adı və iki nöqtə ilə qoyulmuş ola bilər:

```
identifikator:təlimat
```

Burada identifikator, ixtiyari JavaScript-də ehtiyata saxlanılan söz kimi deyil, mümkün identifikator kimi ola bilər. Əgər nişanın adı dəyişənin və ya funksiyanın adı ilə eynidirsə, proqramçı bundan narahat olmamalıdır, çünki nişanların adları, dəyişənlərin və funksiyaların adlarından ayırılmışdır. *while* təlimatının nişanla nümunəsi:

```
parser:
while (token != null)
{
  // kodunuz
}
```

Biz təlimata nişan qoyduqda, ona ad verdikdə, bu təlimatı proqramın istənilən yerində istifadə edə bilərik. İstənilən təlimata nişan qoymaq olar, hərçənd ki, adətən *while*, *do/ while*, *for* və *for/in* dövrlərində nişan qoyulur. Dövrə ad verib, *break* və *continue* təlimatları vasitəsi ilə dövrdən çıxmaq və ya dövrün təkrarlanmasını icra etmək olar.

6.11. break təlimatı

break təlimatı ən daxili dövrdən və ya *switch* təlimatından təcili çıxışa səbəb olur. Sintaksisi çox sadədir:

```
break;
```

break təlimatı dövrdən və ya *switch* təlimatından çıxışa səbəb olur, buna görə də *break*-in belə forması yalnız bu təlimatların daxilində mümkündür. JavaScript *break* açar sözü üçün nişanın adının göstərişinə icazə verir:

```
break:nişan_adı;
```

Diqqətli olun: *nişan_adı* –sadəcə identifikatordur; bunun üçün təlimat nişanının təyini halında olduğu kimi iki nöqtə yazılmır.

break nişanla istifadə edildikdə, adlandırılmışın təlimatlar sonuna keçid olur və ya onun icrasının dayandırılması; adlandırılmış təlimat, *break* ilə bağlı istənilən xarici təlimat ola bilər. Adlandırılmış təlimat dövr və ya *switch* təlimatı olmağı vacib deyil; nişanla istifadə edilmiş *break* təlimatının hətta dövrün və ya *switch* təlimatının daxilində olmağı da vacib deyil. *break* təlimatlarında göstərilmiş nişana tək məhdudiyyət – verilmiş ad xarici təlimatın *break*-nə bağlı olmalıdır. Nişan *if* təlimatının adı və ya hətta yalnız fiqurlu mötərizələr vasitəsilə nişanın mənimsədilməsi üçün bu bloka bağlanmış təlimatlar bloku da ola bilər.

2-ci fəsildə deyildiyi kimi, *break* açar sözünün və nişan adı arasında yeni sətirə keçməyə icazə verilmir. Məsələ orasındadır ki, JavaScript interpretatoru buraxılmış nöqtəli vergülləri avtomatik qoyur. Əgər *break* açar sözü ilə nişan arasında sətir bölünübsə (yeni sətirə keçirilibsə), interpretator fərz edəcək ki, bu *break* təlimatının sadə, nişansız forması nəzərdə tutulur və nöqtəli vergül əlavə edəcək. Əvvəlki bölmələri *switch* təlimatında yerləşdirilmiş *break* təlimatının nümunələri nümayiş etdirilirdi. *break* dövrlərdə adətən hadisələrdən vaxtından əvvəl çıxış üçün istifadə olunur. Dövrdə mürəkkəb çıxış şəraiti olduqda, *break* təlimatın köməyi ilə bu şəraitin bəzi hissələri daha sadə yolla reallaşdırmaq və onların hamısının dövrün bir ifadəsi kimi daxil edilməsinə məhəl qoymamaq olar.

Aşağıdakı kod massiv elementləri arasında müəyyən qiymət axtarışını yerinə yetirir. Dövr sonuna kimi çatdıqda massiv təbii üsulla kəsilir; əgər axtarılan qiymət tapılmışdırsa, massiv *break* təlimatının köməyilə kəsilir:

```
for (i = 0; i<a.length; i ++)
```

```
{
  if (a[i] == target)
    break;
}
```

break təlimatının nişanlı forması yalnız qoyulmuş dövrlərdə və ya *switch* təlimatında zəruri olduqda ən daxili olmayan təlimatdan çıxmaq tələb olunduqda istifadə olunur. Aşağıdakı nümunədə nişan qoyulmuş *for* dövrünü və *break* təlimatı göstərilir:

```
outerloop:
  for (var i = 0; i<10; i++)
  {
    innerloop:
       for (var j = 0; j<10; j++)
       {
         if (j<3)
            break; //Ən daxili dövrdən çıxış
         if (i == 2)
            break innerloop; //Ən daxili dövrdən çıxış
         if (i == 4) break outerloop; // Xarici dövrdən çıxış
            document.write ("i = " + i + " j = " + j + "<br
/>");
       }
  }
document.write ("FINAL i = " + i + " j = " + j + "</br>);
```

6.12. continue təlimatı

continue təlimatı, *break* təlimatının tətbiq edildiyi dövrlərdə tətbiq edilə bilər. Ancaq *continue* təlimatı dövrdən çıxmaq yerinə dövrün yeni iterasiyasına səbəb olur. *continue* təlimatının sintaksisi break təlimatının sintaktsisi qədər sadədir:

```
continue;
```

continue təlimatı həmçinin nişanla da istifadə oluna bilər:

```
continue imya_metki;
```

continue təlimatından yalnız while, *do*/*while*, *for* və *for*/*in* dövrlərinin gövdəsində həm nişanlı, həm də nişansız istifadə oluna bilər.

Bu təlimat, göstərilən dövrlərdən kənar istifadəsi sintaktis xətadır. continue təlimatı yerinə yetirildikdə, dövrün cari iterasiyası kəsilir və növbəti iterasiyası başlayır.
Müxtəlif dövr tipləri üçün fərqli davranışlar mövcuddur:
while dövründə
• dövrün başlanğıcında göstərilmiş ifadə yenidən yoxlanır və əgər ifadə true-a bərabərdirsə, əvvəlcə dövrün gövdəsi yerinə yetirilir. `do`/`while` dövründə
• dövrün növbəti iterasiya şərti dövrün sonunda yoxlanılır
`for` dövründə
• artımın ifadəsi hesablanır və növbəti iterasıyanı yerinə yetirməyi müəyyən etmək üçün yoxlama ifadəsi yenidən yoxlanır.
`for`/`in` dövründə
• dövr göstərilmiş dəyişənin adına növbəti xüsusiyyətin mənimsədilməsiylə yenidən başlayır.
`while` və `for` dövrlərində continue təlimatının davranışında fərqlər mövcuddur – `while` dövrü bilavasitə öz şərtinə qayıdır, amma `for` dövrü əvvəlcə artım ifadəsini hesabladıqdan sonra şərtə qayıdır. Əvvəlki bölmədə mən `while` dövrünə ekvivalent `for` dövr ifdəsi barədə bəhs etmişdim. Bir halda ki, `continue` təlimatı bu iki dövrdə müxtəlif davranışlara malikdir, onda `while` dövrünün köməyilə for dövrünü dəqiq təqlid etmək mümkün deyil.

Aşağıdakı nümunədə səhv halında dövrün cari iterasıyasından çıxmaq üçün nişansız `continue` təlimatından istifadə qaydası göstərilmişdir:

```
for(i = 0; i < data.length; i++) {
 if (data[i] == null)
  continue; // Qeyri-müəyyən məlumatlarla davam mümkün deyil
 total += data[i];
}
```

`continue` açar sözünün və nişan adı arasında sətirdən sətirə keçmək olmaz.

6.13. var təlimatı

`var` təlimatı bir və ya eyni anda bir neçə dəyişəni açıq-aydın elan etməyə imkan verir. Təlimat aşağıdakı sintaksisə malikdir:

```
    var deyiseninAdi_1 [=qiymet_1]
[,...,deyiseninAdi_n [= qiymet_n]]
```

`var` açar sözündə elan edilən dəyişənlərin siyahısı vergül vasitəsilə nizamlanır; siyahıda olan hər bir dəyişən ilkin mənaya, yəni xüsusi ifadə-inisializatoruna malik ola bilər. Məsələn:

```
var i;
var j = 0;
var p, q;
var greeting = "hello" + name;
var x = 2.34,  y = Math.cos(0.75), r, theta;
```

`var` təlimatı sadalanan hər bir dəyişəni bu adla xüsusiyyətin yaradılması yolu ilə funksiyanın çağırış obyektində müəyyən edərək qlobal obyektdə *(əlbəttə, elan edilmə funksiyanın gövdəsində həyata keçirilmirsə)* saxlayır. `var` təlimatının köməyi ilə yaradılan xüsusiyyət və ya xüsusiyyətlər `delete` operatoru tərəfindən silinə bilməz. Nəzərə alın ki, `var` təlimatının `with` təlimatının (***bölmə 6.18*** baxmaq) daxilində istifadəsi onun davranışını dəyişdirmir.
Əgər var təlimatında dəyişənin ilkin mənası göstərilməmişdirsə, onda dəyişən təyin edilir, ancaq onun ilkin mənası qeyri- müəyyən (*undefined*) qalır. Bundan başqa, var təlimatı *for* və *for/in* dövrlərində bilavasitə istifadə edilə bilər. Məsələn:

```
for(var i = 0; i<10; i++)
        document.write(i, "</br>");
for(var i = 0, j=10; i<10; i++, j--)
        document.write(i*j, "</br>");
for(var i in o)
        document.write(i, "</br>");
```

JavaScript-də olan dəyişənlər və onların elanı haqqında ***4-cü fəsildə*** bəhs edilir.

6.14. function təlimatı

JavaScript-də *function* təlimatı funksiyanı müəyyən edir və aşağıdakı sintaksisə malikdir:

```
    function funksiyanin_adi([arq1 [, arq2 [..., arqn]]])
    {
      təlimatlar
    }
```

Burada *funksiyanin_adi* – müəyyən edilən funksiyanın adıdır. Bu ad sətir və ya ifadə kimi deyil, identifikator kimi olmalıdır. Funksiyanın adının qarşısında mötərizələr vasitəsilə əhatələnmiş və vergüllərlə vasitəsilə bölünmüş arqument adlarının siyahısı olur. Bu identifikatorlar, funksiyanın çağırılması zamanı göstərilmiş arqumentlərin qiymətlərinə funksiyanın gövdəsində istinad üçün istifadə oluna bilər.
Funksiyanın gövdəsi, fiqurlu mötərizələr ilə əhatələnmiş JavaScript-təlimatların sərbəst sayından ibarətdir. Bu təlimatların təyini zamanı funksiyalar həyata keçirilmir. Onlar üçün funksiyanın yeni obyekti vasitəsilə kompilyasiya olunur və funksiyanın çağırışı anında, () çağırış operatorunun köməyi ilə icra edilməyə göndərilir. Yadınızda saxlayın, fiqurlu mötərizələr – *function* təlimatının icbari hissəsidir. *while* və digər dövr konstruksiyaları bloklarından fərqli olaraq, funksiyanın gövdəsi fiqurlu mötərizələri tələb edir. Hətta, əgər funksiya yalnız bir təlimatdan ibarət olsa belə fiqurlu mötərizələri qoyulması məcburidir.
Funksiyanın təyini zamanı, funksiyanin_adi adı ilə funksiyanın yeni obyekti yaradılır və yaradılmış obyekt xüsusiyyəti saxlanılır. Aşağıda funksiyaların təyinin bir neçə nümunəsi verilmişdir:

```
function welcome()
{
  alert("Mənim ana səhifəmə xoş gəlmisiniz!");
}

function print(msg)
{
  document.write(msg, "</br>");
}

function hypotenuse(x, y)
```

```
{
  return Math.sqrt(x*x + y*y);       // return təlimatı
növbəti bölmədə təsvir
                                                // edilmişdir
}

function factorial(n)
{                 // Rekursiv funksiya
  if (n <= 1)
    return 1;
  return n * factorial(n - 1);
}
```

Funksiyaların təyini adətən yuxarı səviyyəli JavaScript-kodundadır. Funksiyalar həmçinin başqa funksiyaların daxilində təyin edilə bilərlər, lakin "yuxarı səviyyədə", yəni funksiyanın içərisində olan *if* təlimatları, *while* və ya istənilən başqa dövr konstruksiyaları içərisində funksiyalar təyin edilə bilməz.

Rəsmi olaraq *function* təlimat deyil. Təlimatlar JavaScript-proqramlarında bəzi dinamik davranışlara gətirib çıxarır, amma funksiyaların təyini proqramın statik strukturunu təsvir edir. Təlimatlar proqram icra edilən zamanı, funksiyalar isə analiz və ya JavaScript-kodun kompilyasiyası zamanı, yəni proqramın faktiki icrasına qədər təyin edilir. JavaScript-in sintaktis analizatoru funksiya təyini ilə qarşılaşdıqda, analizator funksiyanın gövdəsində təşkil olunan təlimatları təhlil edir (icrasız) və özündə saxlayır. Sonra isə funksiyanın təyinində göstərilmiş xüsusiyyət (əgər funksiya başqa funksiyanın daxilində təyin edilmişdirsə çağırış obyektində; əks təqdirdə – qlobal obyektdə) müəyyən edilir. Məhz, funksiyaların sintaktis analiz mərhələsində təyin edilməsi, bəzi maraqlı effektlərə gətirib çıxarır. Məsələn, aşağıdakı koda baxaq:

```
alert(f(4));       // 16 göstərir. f() funksiyası eyni
zamanda təyin edilərək
                          // çağrıla bilər.
var f = 0;       // Bu təlimat f xüsusiyyətinin tərkibini
təyin edir.
function f(x) {          // Bu “təlimat” yuxarıdakı sətirlər
yerinə yetirilənə
  return x*x;            // qədər f funksiyasını təyin edir.
}
```

```
alert(f);       // 0 göstərir. f() funksiyası f dəyişəni ilə
kəsilmişdir.
```

Belə qeyri-adi nəticələr ona görə yaranır ki, təyin edilən zaman funksiya təyin edilmir. Xoşbəxtlikdən, belə situasiyalara çox sıx rast gəlinmir. Funksiya barədə daha ətraflı 8-ci fəsildə bəhs olunur.

6.15. return təlimatı

Yəqin xatırlayırsınız ki, () operatorunun köməyi ilə funksiya çağırışı ifadəni təşkil edir. Bütün ifadələr qiymətlərə malikdir, və *return* təlimatı funksiya vasitəsilə qaytarılan qiyməti təyin edir. Bu qiymət funksiyanın çağırışı zamanın funksiya ifadəsinin qiyməti olur. eeturn təlimatı aşağıdakı sintaksisə malikdir:

```
return ifadə;
```

Təlimat yalnız funksiyanın gövdəsində yerləşdirilə bilər. Funksiyadan başqa istənilən yerdə istifadəsi sintaktis xətadır. *return* təlimatı yerinə yetiriləndə, ifadə hesablanır və onun qiyməti funksiyanın qiyməti kimi qayıdır. *return* təlimatı funksiyanın icrasını dayandırır (hətta əgər funksiyanın gövdəsində başqa təlimatlar qalsa belə). *return* təlimatı qiymətin qayıdışı üçün aşağıdakı qaydada istifadə olunur:

```
function square(x)
{
   return x*x;
}
```

return təlimatı həmçinin ifadəsiz istifadə oluna bilər, bu zaman təlimat heç bir qiymət qaytarmadan sadəcə funksiyanın icrasını dayandırır. Məsələn:

```
function display_object (obj)
{
  // Əvvəlcə biz arqumentimizin düzgünlüyündən əmin olmalıyıq
  // Uyğunsuzluq zamanı funksiyanın işini dayandırırıq
  if (obj == null) return;
```

```
    // Burada isə funksiyanın əsas icra hissəsi yerləşir...
}
```

Əgər funksiyada ifadəsiz *return* təlimatı yerinə yetirilirsə və ya əgər funksiyanın icrası funksiyanın gövdəsinin sona çatması ilə bitirsə, funksiya çağırışının ifadə qiyməti qeyri-müəyyən (*undefined*) olur. JavaScript nöqtəli vergülü avtomatik qoyur, buna görə də sətirdən sətirə keçərkən *return* təlimatını kəsmək olmaz.

6.16. throw təlimatı

İstisna – hər hansı xüsusi vəziyyətin və ya xətanın yaranmasını göstərən siqnaldır. İstisnanın (*throw*) generasiyası – bu cür səhv və ya xüsusi vəziyyət haqqında siqnal vermək üsuludur. İstisnanın qarşısını almaq (*catch*) dedikdə, istisna baş verdikdə onu emal etmək başa düşülür. JavaScript-də istisna yerinə yetirilmə zamanı xəta yaranırkən, proqram onu *throw* təlimatının köməyi ilə açıq-aydın yaradır. İstisnaların qarşısı *try*/*catch*/*finally* təlimatının köməyi ilə alınır ki, növbəti bölmədə bu təlimatlar təsvir edilmişdir.
Throw təlimatı aşağıdakı sintaksisə malikdir:

```
throw ifadə;
```

ifadənin Nəticəsi istənilən tipdə olan qiyməti ola bilər. Ancaq adətən bu Error obyektidir və ya bu obyektin yarımsiniflərindən birinin nüsxəsidir. Həmçinin ifadələri bəzi xəta kodunu ifadə edən xəta barədə məlumatı və ya xəta qiymətini əks etdirən sətir kimi istifadə etmək olar.
Aşağıdakı nümunədə, *throw* təlimatı istisnanın generasiyası üçün tətbiq edilir:

```
function factorial(x)
{
    // Əgər giriş arqumenti yararlı deyilsə, istisna
yaradırıq!
    if (x < 0) throw new Error("x mənfi ola bilməz");
    // Əks təqdirdə qiyməti hesablayırıq və funksiyasından
normal obrazla
    // çıxırıq
```

```
    for (var f = 1; x > 1; f *= x, x); /* boş dövr gövdəsi */
    return f;
}
```

İstisna yaradılırkən, JavaScript interpretatoru dərhal proqramın normal icrasını kəsir və ən yaxın istisnanın emalı prosesinə keçir. İstisna emalçılarında *try*/*catch*/*finally* təlimatının *catch* konstruksiyasından istifadə olunur ki, təsviri növbəti bölmədə bu mövzuya ətraflı izah verilmişdir. Əgər istisna yaradılan kodun bloku, *catch* konstruksiyasına malik deyilsə, bu zaman interpretator kodun növbəti xarici blokunu təhlil edir və istisna emalçısının blokla bağlılığını yoxlayır. Bu proses emalçı tapılmayana qədər davam edir. Əgər istisna funksiyada yaradılıbsa və funksiya bu istisnanın emalı üçün nəzərdə tutulmuş *try*/*catch*/*finally* təlimatını ehtiva etmirsə, onda istisna çağrıldığı funksiyanın kodu üzrə yayılır. Belə istisnalar JavaScript metodlarının leksik strukturuna uyğun olar yuxarı çağırış steki üzrə yayılır. Əgər istisna emalçısı tapılmasa, istisnaya xəta kimi baxılır və istifadəçiyə bu haqda xəbərdarlıq ebdilir.
throw təlimatı ECMAScript v3-də standartlaşdırılmışdır və JavaScript 1.4-də reallaşdırılmışdır. Error sinifi və həmçinin onun yarımsinifləri ECMAScript v3 standartının bir hissəsidir, amma onlar JavaScript 1.5-ə qədər reallaşdırılmamışlar.1.
JavaScript-də *throw* və *try*/*catch*/*finally* təlimatı C++ və Java-dakı müvafiq təlimatlara uyğundur.

6.17. try/catch/finally təlimatı

Try/*catch*/*finally* təlimatı JavaScript-də istisnaların emal mexanizmini reallaşdırır. Bu təlimatda *try* konstruksiyası sadəcə emal edilən istisnaların kod blokunu müəyyən edir. *Try* blokunda olan istisnalara *catch* konstruksiyasında baxılır. *Catch* konstruksiyasını təmizləmə kodunu ehtiva edən *finally* bloku davam etdirir. Bu blokda, proseslər *try* blokundan asılı olmayaraq yerinə yetirilir. *catch* və *finally* bloku məcburi hissə sayılmır, ancaq *try* blokundan sonra

onlardan heç olmasa biri mütləq daxil edilməlidir. *try*, *catch* və *finally* blokları fiqurlu mötərizələrlə əhatələnməlidir. Bu sintaksisin məcburi hissəsidir və heç bir halda ixtisara salına bilməz (hətta blokda yalnız bir təlimat varsa belə). *throw* təlimatı kimi, *try*/*catch*/*finally* təlimatı da ECMAScript v3-də standartlaşdırılmışdır və JavaScript 1.4-də reallaşdırılmışdır.

Növbəti fraqmentdə *try*/*catch*/*finally* təlimatının sintaksisi və mahiyyətini təsvir edilir. Nəzərə alın, xüsusi halda, *catch* açar sözünü mötərizələrdə identifikator müşahidə edir. Bu identifikator funksiyanın arqumentinə oxşardır. O yalnız *catch* blokunun gövdəsində mövcud olan lokal dəyişənin adını mənimsəyir. JavaScript istisnanın generasiyası zamanı göstərilmiş istisnanın və ya qiymətin obyektini bu dəyişənə mənimsəyir:

```
try
{
    // İstisana olmadıqda bu kod başlanğıcdan sona qədər
icra edilir.
    // Amma hər hansı istisna yaranarsa, bu istisna ya
bilavasitə throü
    // təlimatının ya da istisnanı emal edə bilən
funksiyanın köməyilə
    // emal edilir.
}
catch(e)
{
    // Bu blokda təlimatlar yalnız blokda istisna yarankən
icra edilir.
    // Bu təlimatlar e lokal dəyişəndən istifadə edə, Error
obyektinə
    // və ya throw təlimatlarında göstərilmiş digər
qiymətlərə istinad edə
    // bilər. Yaxud hər hansı obrazla istisnanı emal
etməmək, bu istisnanaya
    // əhəmiyyət verməmək, başqa nəsə icra etmək və ya
istisnanı throw
    // təlimatının köməyilə yenidən yaratmaq olar.
}
finally
{
    // Bu blok təlimatları try blokundan asılı olmayaraq
həmişə yerinə
```

```
    // yetirilən yəlimatları ehtiva edir. Bu blok kod try
bloku ilə
    // kəsildiyi zaman aşağıdakı hallarda istifadə edilir:
    // 1) blokun normal obrazla sona çatdırılması üçün
    // 2) break, continue və ya return təlimatları üçün
    // 3) daha əvvəl göstərilən catch blokuyla emal edilmiş
istisnalar üçün
    // 4) Öz-özünə davam edən yuxarı səviyyəli istisnalar
üçün
}
```

İndi isə gəlin *try*/*catch* təlimatının daha praktik nümunəsi ilə tanış olaq. Nümunədə əvvəlki bölmədə göstərilmiş factorial() və prompt() metodlarından, həmçinin çıxış məlumatını ekranlaşdırmaq üçün JavaScript-in kliyent dilinin *alert()* metodundan istifadə edilir:

```
try
{
  // İstifadəçidən ədəd daxil etməyi xahiş edirik
  var n = prompt("Xahiş edirik müsbət ədəd daxil edin edin",
 "");
  // Daxil edilən ədədin düzgünlüyünü güman edərək onun
faktorialını
  hesablayırıq.
  var f = factorial(n);
  // Nəticəni göstəririk
  alert(n + "! = "+ f);
}

catch (ex)
{
  // Əgər daxil edilmiş məlumatlar səhvdirsə, biz blokun
icrasına keçəcəyik
  alert(ex) ; // Xəta haqqında istifadəçiyə bildiririk
}
```

Bu nümunə *finally* konstruksiyası olmadan *try*/*catch* təlimatının nümunəsidir. Hərçənd *finally*, *catch*-a nisbətən sıx istifadə olunmur, bununla belə bəzi hallarda bu konstruksiya faydalı olur. Ancaq *finally* konstruksiyanın davranışı əlavə izahat tələb edir. *Finally* bloku prosesləri, *try* blokundan, asılı olmayaraq həyata keçirir. Bu imkandan adətən *try* təklifindəki kodun icrasından sonra təmizləmə üçün istifadə olunur.

Adi vəziyyətdə mexanizm *try* blokunun sonuna kimi çatır və sonra *finally* blokuna keçir ki, burada bütün lazımlı təmizləmələr yerinə yetirilir. Əgər idarəetmə *return*, *continue* və ya *break* təlimatları vasitəsilə *try* blokundan çıxdısa, idarəetmənin kodun digər hissəsinə ötürülməsindən əvvəl *finally* bloku həyata keçir.
Əgər *try* blokunda istisna yaranırsa və onun emalı üçün uyğun *catch*-in uyğun bloku varsa, idarəetmə əvvəlcə *catch* blokuna, sonra isə *finally* blokuna ötürülür. Əgər lokal *catch* bloku yoxdursa, onda idarəetmə - əvvəlcə *finally* blokuna, sonra isə yaranan istisnanı emal edə bilən ən yaxın xarici *catch* blokuna ötürülür.
Əgər *finally* blokunun özündə idarəetmə *return*, *continue*, *break* və ya *throw* təlimatının köməyilə (və ya istisna yaradan metodun çağırılması ilə) qurulubsa, onda bitməmiş komanda ləğv edilir və yeni komanda yerinə yetirilir. Məsələn, əgər *finally* blokunda istisna yaradılarsa, bu istisna *finally* blokundan kənarda yaradılmış istənilən istisnanı əvəz edir. Əgər *finally* blokunda *return* təlimatı varsa, blokda yaradılan emal edilməmiş istisnanın mövcudluğuna baxmayaraq metoddan normal çıxış edilir.
try və *finally* təlimatları *catch* konstruksiyası olmadan birlikdə istifadə oluna bilər. Bu halda *finally* blokundan – sadəcə təmizləmə kodu kimi istifadə edilir ki, buradakı proseslər *try* blokundakı *break*, *continue* və ya *return* təlimatlarının mövcudluğundan asılı olmayaraq yerinə yetiriləcək. Məsələn, aşağıdakı kodda *try*/*finally* təlimatı istifadə olunur. Nümunədə göstərilir ki, dövrün sayğacı hər bir təkrarında sonunda, hətta əgər təkrar *continue* təlimatının vasitəsilə kəsilsə belə, inkrementasiya (bir vahid artma) olsun:

```
var i = 0, total = 0;
while (i < a.length)
{
 try
 {
   if ((typeof a[i]! = "number") || isNaN(a[i])) // Əgər bu
ədəd deyilsə,
   continue;       // növbəti dövr iterasiyanına keçirik.
   total += a[i];  // Əks təqdirdə ümumi ədədi ümumi cəmə
əlavə edirik.
 }
```

```
 finally
 {
   i++;                    // continue təlimatının
mövcudluğundan asılı
                        // olmayaraq həmişə i dəyişənini bir
                        vahid artırırıq.
 }
}
```

6.18.with təlimatı

4-cü fəsildə biz dəyişənlərin görünmə sahəsini və görünmə sahələrinin zəncirini müzakirə etdik – obyektlərin siyahısı, hansılarda ki, dəyişən adına icazə vaxtı axtarış yerinə yetirilir. `with` təlimatı görünmə sahələri zəncirinin müvəqqəti dəyişikliyi üçün istifadə olunur. Bu təlimat aşağıdakı sintaksisə malikdir:

```
with(obyekt)
     təlimat
```

Bu təlimat, obyekti görünmə sahələri zəncirinin başlanğıcına əlavə edir, təlimatı icra edir, və sonra görünmə sahələri zəncirini ilkin vəziyyətində bərpa edir. Təcrübədə `with` təlimatı yığılan mətnin həcmini əhəmiyyətli dərəcədə azaltmağa kömək edir. JavaScript-in kliyent dilində bu təlimatdan adətən obyektlərin dərin təyin edilmiş iyerarxiyaları zamanı istifadə edilir. Məsələn, HTML-formanın elementlərinə giriş üçün sizə, belə ifadələrdən istifadə etmək lazım olacaq:

```
frames[1].document.forms[0].address.value
```

Əgər bu formaya bir neçə dəfə müraciət etmək lazımdırsa, formanın görünmə sahələri zəncirinə əlavə edilməsi üçün `with` təlimatından istifadə etmək olar:

```
with (frames[1].document.forms[0])
{
  // Burada bilavasitə formanın elementlərinə müraciət edirik,
məsələn:
  name.value = "";
  address.value = "";
  email.value = "";
```

```
}
```

Bu proqramın mətninin həcmini azaldır – bundan sonra *frames[1]* xüsusiyyətinin hər bir adından əvvəl *document.forms[0]* fraqmentini göstərmək lazım deyil. Bu obyekt görünmə sahələri zəncirinin müvəqqəti hissəsini təşkil edir və axtarışda avtomatik iştirak edir.

Bu konstruksiyanın rahatlığına baxmayaraq bəzi hallarda, onun istifadəsi dəstəklənmir. JavaScript-də with təlimatı kodu optimallaşdırmağa yönəlib və buna görə də kodunuz, with təlimatı istifadə edilməyən ekvivalent kod ilə nisbətdə daha yavaş işləyə bilər. Bundan başqa, funksiyaların təyinləri və with təlimatının bədənində dəyişənlərin inisializasiyası bu səbəblərdən rezultatam.1 anlaması üçün qəribələrə və çətinlərə with təlimatından istifadə etməyə gətirə bilər tövsiyə edilmir. Bundan başqa mövcuddurlar və yığılan mətnin həcminin azaldılmasının başqa tamamilə qanuni üsulları. Belə, əvvəlki nümunə aşağıdakı qaydada köçürtmək olar:

```
var form = frames[1].document.forms[0];
form.name.value = "";
form.address.value = "";
form.email.value = "";
```

6.19. Boş təlimat

Və nəhayət, son olaraq – boş təlimat barədə danışaq. O buna bənzəyir:

```
;
```

Boş təlimatın icrası heç bir effektə və təsir malik deyil. Bu səbəbdən düşünmək olar ki, onun tətbiqi üçün xüsusi lüzum yoxdur, ancaq bəzi hallarda boş təlimat faydalı ola bilər. Məsələn boş gövdəyə malik olan dövr yaratmaq tələb olunduqda.

```
// a  massivinin inisializasiyası
for(i=0; i < a.length; a[i++] = 0);
```

Nəzərə alın ki, *for* və *while* dövrlərində və ya *if* təlimatında sağ yumru mötərizədən sonra nöqtəli vergülün ixtiyarsız göstərişi aşkar

edilməsi çətin olan xoşagəlməz xətalara gətirib çıxara bilər. Məsələn, aşağıdakı fraqmentin xətalı olduğu söyləmək çətindir:

```
if ((a == 0) || (b == 0));        // Axı bu sətir heç nə
etmir...
o = null;                           // amma bu sətir həmişə
yerinə yetirilir.
```

Xüsusilə, boş təlimat tətbiq edilərkən, kodu multi-sətir şərhlərlə təşkil etmək arzu olunandır. Məsələn:

```
for (i=0; i < a.length; a [i++] = 0); /* boş dövr gövdəsi */
```

7

Obyektlər və massivlər

3-cü fəsildə obyektlər və massivlər barədə deyilirdi ki, obyektlər və massivlər – JavaScript-də iki fundamental və ən əhəmiyyətli məlumat tipidir. Obyektlər və massivlər ədəd və sətir tipi kimi elementar məlumat tiplərindən fərqlənir, çünki onlar özlərində istənilən qədər qiymət və qiymət dəsti ehtiva edə bilərlər. Obyekt, adlandırılmış xüsusiyyətlərin məcmusudur, massivlər isə ixtisaslaşmış obyektlərdən təşkil olunub ki, onları da nömrələnmiş qiymətlərin məcmusu kimi adlandırmaq olar. Bu fəsildə biz JavaScript dilinin obyektləri və massivləri ətraflı tanış olacağıq.

7.1. Obyektlərin yaradılması

Obyektlər – bir neçə qiyməti vahid modulda birləşdirməyə, saxlamağa və onları adları üzrə çıxartmağa imkan verən tərkib məlumat tipidir. Başqa sözlə desək,obyektlər – xüsusiyyətlərin nizamlanmamış məcmusudur ki, bu xüsusiyyətlərin hər birinin öz adı və qiyməti olur. Obyektdə saxlanılan adlandırılmış qiymətlər ədəd, sətir tipi kimi elementar məlumatı tiplərində və ya özləri də obyekt ola bilər. Obyektlərin yaradılmasının ən sadə üsulu obyektin proqrama obyekt literalının əlavə edilməsidir.

Obyekt literalı – fiqurlu mötərizələrin daxilində, vergüllərlə bölünmüş xüsusiyyətlərin ("ad – qiymət" cütlüyü) siyahısıdır. Hər bir xüsusiyyətin adı JavaScript-identifikatorı və sətirləri ilə, qiyməti isə sabit və ya JavaScript ifadəsi ola bilər. Aşağıdakı nümunədə obyektlərin yaradılmasının bir neçə üsulu göstərilmişdir:

```
var empty ={};                  // Xüsusiyyətlərisiz obyekt
var point ={ x:0, y:0 };
var circle ={ x:point.x, y:point.y+1, radius:2 };
var homer = {
     "name": "Homer Simpson",
     "age": 34,
     "married": true,
     "occupation": "plant operator",
     'email': "homer@example.com"
};
```

Obyekt literalı – yeni obyekt yaradan ifadədir və bu ifadənin hesablanması zamanı hər dəfə inisializasiya edilir. Beləliklə, əgər literalı dövr və ya çağrılan funksiya daxilində yerləşdirsək, bir obyekt literalı köməyi ilə bir çox yeni obyekt yaratmaq olar.

new operatoru da obyektlərin yaradılmasının bir başqa üsuludur. Bu operatorun tətbiqi zamanı obyektin xüsusiyyətlərinin inisializasiyasını yerinə yetirən funksiya-konstuktorunun adını da göstərilməlidir. Məsələn:

```
     var a = new Array();                    // Boş massiv yaratmaq
     var d = new Date ();                    // Cari vaxt və tarix
obyekti yaratmaq
     var r = new RegExp (" javascript", "i"); // Requlyar ifadə
obyekti yaratmaq
```

Burada nümayiş etdirilmiş Array(), Date() və RegExp() JavaScript-in baza dilindən inteqrasiya edilmiş funksiya-konstruktorlarıdır. Object() konstruktoru {} obyekt literalı kimi boş obyekt yaradır. Yeni yaradılan obyektlərin inisializasiyası üçün şəxsi konstruktorları müəyyən etmək imkanı da mövcuddur. Bu proses, 9-cu fəsildə göstərilmişdir.

7.2. Obyektlərin xüsusiyyətləri

Adətən obyektin xüsusiyyətlərinin qiymətlərinə giriş üçün nöqtə operatorundan istifadə olunur. Operatorun sol hissəsindəki qiymət, xüsusiyyətlərinə giriş ediləcəyi obyektə istinad etməlidir. Adətən bu obyektə əsaslananın dəyişənin adı olur, amma mümkün JavaScript-ifadəsi də obyekt ola bilər. Operatorun sağ tərəfindəki qiymət

xüsusiyyətin adı olmalıdır. Burada sətir və ya ifadə deyil, mütləq halda identifikator olmalıdır. Belə ki, əgər o obyektinin p-ı xüsusiyyətinə müraciət etmək istəyiriksə o.p ifadəsindən, circle obyektinin radius xüsusiyyətinə müraciət etmək istəyiriksə circle.radius ifadəsindən istifadə edə bilərik. Obyektin xüsusiyyətləri dəyişənlər kimi işləyir: onlarda qiymətlər saxlamaq və bu qiymətləri oxumaq olar. Məsələn:

```
    // Obyekt yaradırıq və dəyişən vasitəsilə bu obyektə istinad
edirik.
    var book = new Object(); // Obyektdə xüsusiyyət təyin edirik.
    book.title = "JavaScript-in əsasları"

    // Başqa xüsusiyyətlər də təyin edirik. Daxil edilmiş
obyektlərə fikir verin.
    book.chapter1 = new Object();
    book.chapter1.title = "JavaScript-ə giriş";
    book.chapter1.pages = 11;
    book.chapter2 ={
        title: "Leksik struktur",
        pages: 6
    };

    // Obyektdəki xüsusiyyətlərin qiymətlərinin
ekranlaşdırılması.
    alert ("Başlıq: "+ book.title + "\n\t" +
          "1-ci fəsil "+ book.chapter1.title + "\n\t" +
          "2-ci fəsil "+ book.chapter2.title);
```

Bu nümunədə bir məqamı nəzərə almaq əhəmiyyətlidir ki, obyektə yeni xüsusiyyəti əlavə etmək üçün sadəcə xüsusiyyətə qiymət mənimsətmək kifayətdir. Əgər dəyişənlərin var açar sözünün köməyilə elan edilməli zərurəti varsa, onda bu imkan obyektinin xüsusiyyətləri üçün (və imkanlar) yoxdur. Bundan başqa obyektin xüsusiyyətinin yaradılmasından sonra (mənimsəmə nəticəsində) aşağıdakı formada, istənilən zaman sadə mənimsəməylə onun qiymətini dəyişdirmək mümkün olacaq:

```
book.title = "JavaScript: Kərgədanlı kitab";
```

7.2.1. Xüsusiyyətlərin sadalanması

6-cı fəsildə qeyd edildiyi kimi *for*/*in* dövrü vasitəsilə obyektin xüsusiyətlərini sadalamaq olar. Bu dövri ssenarilərin yaradılması zamanı və ya obyektlərlə işləyərkən qabaqcadan adları naməlum olan ixtiyari xüsusiyyətlə istifadə etmək olar. Aşağıdakı fraqmentdə obyektdə xüsusiyyətləri sadalayan funksiya nümayiş etdirilir:

```
function DisplayPropertyNames(obj) {
   var names = "";
   for(var name in obj) names += name + "\n";
   alert(names);
}
```

Nəzər alın ki, *for*/*in* dövrü nizamsız verilmiş xüsusiyyətləri sadalamır və bu dövr istifadəçi tərəfindən müəyyən edilmiş bütün xüsusiyyətləri sadalamasına baxmayaraq, qabaqcadan müəyyən edilmiş bəzi xüsusiyyətlər və metodlar sadalanmır.

7.2.2. Obyektdə xüsusiyyətin mövcudluğunun yoxlaması

Obyektdə xüsusiyyətlərin mövcudluğunun yoxlaması üçün *in* operatorundan (5-ci fəsildə göstərilmişdir) istifadə oluna bilər. Bunun üçün operatordan sol tərəfdə sətir şəklində xüsusiyyətin adı yerləşir, sağ tərəfdə isə yoxlanan obyekt göstərilməlidir. Məsələn:

```
// Əgər o obyekti "x" adlı xüsusiyyətə malikdirsə, qiymət
təyin edilir
if (" x" in o) o.x = 1;
```

Ancaq *in* operatorundan çox sıx istifadə edilmir, çünki mövcud olmayan xüsusiyyətə müraciət zamanı *undefined* qiyməti qaytarılır. Beləliklə, göstərilən fraqment aşağıdakı qaydada yazıla bilər:

```
// Əgər x xüsusiyyəti mövcuddursa, deməli onun qiyməti
undefined-ə bərabər
// deyil və belə olan halda xüsusiyyətə qiymət təyin edirik.
if (o.x! == undefined) o.x = 1;
```

Nəzərə alın ki, bu zaman xüsusiyyət faktiki mövcudluğuna ehtimal var və sadəcə olaraq xüsusiyyət müəyyən edilməmiş ola bilər. Məsələn, əgər bu cür sətiri yazsaq:

```
o.x = undefined
```

o obyektinin *x* xüsusiyyəti mövcud olacaq, lakin qiyməti olmayacaq. Bu halda göstərilmiş fraqmentlərdən birincisində *x* xüsusiyyətinə 1 qiyməti əlavə ediləcək, ikincisində isə əlavə edilməyəcək.
Bundan başqa, nəzərə alın ki, burada != operatorun yerinə !== operatorundan istifadə edilmişdi. ! == və === operatorları *undefined* və *null* qiymətləri bir-birindən ayıra bilir, hərçənd ki, bəzən buna da ehtiyac yoxdur:

```
// Əgər doSomething xüsusiyyəti mövcuddursa və null və ya
undefined qiymətini
// ehtiva etmirsə, onda fərz etmək ki, bu funksiyadır və onu
çağırmaq
// lazımdır!
if (o.doSomething) o.doSomething();
```

7.2.3. Obyektin xüsusiyyətinin silinməsi

Obyektdəki xüsusiyyətlərin silinməsi üçün delete operatoru nəzərdə tutulmuşdur:

```
delete book.chapter2;
```

Nəzərə alın ki, xüsusiyyətin silinməsi zamanı xüsusiyyətin qiymətini *undefined* müəyyən edilməsi ilə də mümkündür, lakin delete operatoru ilə xüsusiyyət obyektdən tamamilə silinir. Bu fərqi *for*/*in* dövrü açıq-aydın göstərir: belə ki, o qiymət mənimsədilməmiş, *undefined* qiyməti ehtiva edən xüsusiyyətləri sadalayır, lakin silinmiş xüsusiyyətləri sadalamır.

7.3. Obyektler assosiativ massivlər qismində

Biz artıq bilirik ki, obyektin xüsusiyyətlərinə giriş "nöqtə" operatoru vasitəsilə həyata keçirilir. Obyektin xüsusiyyətlərinə giriş həmçinin [] operatorun köməyilə də mümkündür ki, bu operator adətən massivlərlə işləyərkən tətbiq edilir. Beləliklə, aşağıdakı iki JavaScript-ifadəsi eyni qiymətə malikdir:

```
object.property
object ["property"]
```

Nəzərə alın ki, bu iki sintaksis bir-birilə əhəmiyyətli fərqlənir. birinci variantda xüsusiyyət adı identifikatordan, ikincidə variantda isə sətirdən təşkil olunub. Bu fərqin niyə belə əhəmiyyətli olduğunu tezliklə özünüz anlayacaqsınız.
Java, C, C++ və digər viddi tipləşdirilmə olan dillərdə obyekt xüsusiyyətləri yalnız təsbit edilmiş sayda ola bilər və bu xüsusiyyətlərin adları qabaqcadan müəyyən edilmiş olmalıdır. Bir halda ki, JavaScript – zəif-tipləşdirilmiş dil sayılır, onda bu qaydalar yaranır; proqram istənilən obyektdə istənilən qədər istifadə edilməyən xüsusiyyət yarada. Ancaq obyektin xüsusiyyətinə giriş üçün "nöqtə" operatorundan istifadə edilən zaman xüsusiyyətin adı identifikatorla yazılır. İdentifikatorlar JavaScript-proqram mətninin bir hissəsi olmalıdır – identifikator məlumat tipi deyil və onlar ilə proqramdan manipulyasiya etmək olmaz.
Eyni zamanda massiv notasiyalarının köməyilə obyektin xüsusiyyətinə müraciət zamanı xüsusiyyətin adı [] daxilində sətir şəklində yazılır. JavaScript-də sətirlər – məlumat tipi olduğu üçün proqram gedişatında, yaradıla və dəyişdirilə bilər. Və buna görə də JavaScript-də aşağıdakı formada kod yazmaq olar:

```
var addr = "";
for (i = 0; i < 4; i++)
{
  addr += customer ["address" + i] + '\n';
}
```

Bu fraqmentdə customer obyektinin address0, address1, address2 və address3 xüsusiyyətləri bir sətirdə birləşdirilərək oxuna bilir.
Bu qısa nümunə, sətir ifadələrinin köməyilə obyekt xüsusiyyətlərinə müraciət edərkən massiv notasiyalarını necə elastik olduğu görünür. Biz bu nümunəni "nöqtə" operatorunun köməyilə də yaza bilərdik, amma elə

vəziyyətlər var ki, orada massiv notasiyalarının istifadəsi uyğun gəlir. Fərz edək ki, siz istifadəçinin maliyyə bazarındakı investisiyalarının cari məbləğini hesablamaq üçün şəbəkə resurslarına müraciət edən proqram yazmalısınız. Proqram istifadəçiyə istənilən mövcud səhm adını, həmçinin hər bir səhm növünün miqdarını daxil etməyə imkan verir. Hər səhm növü üçün bir xüsusiyyətə malik olan portfolio adlı obyektin köməyilə bu informasiyanın saxlanılmasını təşkil etmək olar.
Xüsusiyyətin adı – səhm adı, xüsusiyyətin qiyməti isə, bu növ səhmlərin miqdarıdır. Başqa sözlə, məsələn əgər, istifadəçidə IBM 50 səhmi varsa, onda portfolio.ibm xüsusiyyəti 50 qiymətini ehtiva edir.
Bu proqramın bir hissəsində, istifadəçidən onda olan səhmlərin adını və sonra bu növ səhmlərinin miqdarını sorğulayan dövr təşkil olunmalıdır. Dövrün daxilindəki kod aşağıdakı formada ola bilər:

```
var stock_name = get_stock_name_from_user();
var shares = get_number_of_shares();
portfolio [stock_name] = shares;
```

Bir halda ki, istifadəçi səhmlərin adını proqramın icrası zamanı daxil edir, onda xüsusiyyət adlarını qabaqcadan bilmək olmur. Bu səbəbdən, proqramın yazılışı zamanı xüsusiyyətlərin adları naməlum olur və belə olduğu halda portfolio obyektinin xüsusiyyətlərinə "nöqtə" operatorunun köməyilə giriş qeyri-mümkündür. Bu halda obyektin xüsusiyyətlərinə [] ancaq operatorunun köməyilə müraciət etmək olar, çünki bu operatorda üçün xüsusiyyət adı identifikatorla (hansı ki, bilavasitə proqram mətnində göstərilir) deyil, sətir qiyməti (hansı ki, bu qiymət icra zamanı dəyişə bilər) göstərilir.
Adətən obyektin bu formada istifadəsi, assosiativ massiv adlanır. Assosativ massiv, sərbəst qiymətlərlə, sərbəst sətirlər arasında əlaqə yaradan verilənlər strukturudur. Bu vəziyyəti izah etmək üçün adətən map terminindən istifadə olunur: JavaScript-obyektləri öz qiymətlərini (xüsusiyyətlərin adları) sətirləri kimi əks etdirir.
Obyektlərin xüsusiyyətləri giriş üçün nöqtədən (.) istifadə onları C++ və Java dillərində statik obyektlərə bənzədir və bu operator lazımi vəziyyətdə öz işini yaxşı görür. Amma onlar həmçinin sərbəst sətirlərlə qiymətlərin (mənaların) əlaqəsi üçün güclü vasitəni verirlər. Bu cəhətdən JavaScript obyektlər daha çox Perl-də massivlərə oxşardır, nə

qədər C++-da və ya Java-da obyektlərə. 6-cı fəsildə `for`/`in` dövrü daxil edilmişdi. JavaScript əsl gücü assosiativ massivlərlə istifadə zamanı aydın hiss olunur. Məsələn səhmlər portfeliylə qayıdaq. İstifadəçi tərəfindən öz portfeli üzrə məlumatların daxil edilməsindən sonra sonuncu ümumi cari qiyməti aşağıdakı kod vasitəsilə hesablamaq olar:

```
var value = 0;
for (stock in portfolio) {
  // Portfeldəki hər bir səhmin bir səhm üzrə dəyərini əldə edirik və
  // onu səhmlərin miqdarına vururuq
 value += get_share_value(stock) * portfolio[stock];
}
```

Burada `for`/`in` dövrü alternativi yoxdur, çünki, səhmlərin adları qabaqcadan məlum deyil. Bu üsul, portfolio adlı assosiativ massivdən (JavaScript-obyekti) xüsusiyyət adlarının çıxardılmasının yeganə həllidir.

7.4. Universal Object sinifinin xüsusiyyətləri və metodları

Artıq qeyd edildiyi kimi, JavaScript-dəki bütün obyektlərin metodları və xüsusiyyətləri `Object` sinifindən varis olur. Belə olan halda Date() və ya RegExp() kimi ixtisaslaşmış obyekt sinifləri, öz konstruktorları ilə yaradılmasına, daxilinə şəxsi xüsusiyyət və metodlar müəyyən edilməsinə baxmayaraq, bütün obyektlər kimi öz mənşələrini ilk olaraq `Object` sinifindən götürür və bu sinifin dəstəklədiyi xüsusiyyət və metodlar onlar üçün də əlçatandır. Onların universallığı səbəbdən bu xüsusiyyətlər və metodlar xüsusi marağı təqdim edir.

7.4.1. constructor xüsusiyyəti

JavaScript-də istənilən obyekt constructor xüsusiyyətinə malikdir ki, bu obyektin inisializasiyası üçün istifadə edilən funksiya-konstuktoruna

istinad edir. Məsələn, əgər d obyekti Date()-konstruktorunun köməyilə yaradılırsa, onda d.constructor xüsusiyyəti Date funksiyasına istinad edir:

```
var d = new Date ();
d.constructor == Date; // true-ya bərabərdir
```

Konstruktor funksiyası kateqoriya, sinif və ya obyekti müəyyən edir, buna görə də constructor xüsusiyyəti verilmiş ixtiyari obyektin tipinin təyini üçün istifadə oluna bilər. Məsələn, obyektin tipini bu yolla aydınlaşdırmaq olar:

```
if ((typeof o == "object")&&(o.constructor == Date))
     // Date obyekti ilə hər hansı davranış...
```

constructor xüsusiyyət qiymətini instanceof operatorunun köməyilə yoxlamaq olar, yəni yuxarıda göstərilən fraqmentı bu cür yazsaq:

```
if ((typeof o == "object")&&(o instanceof Date))
// Date obyekti ilə hər hansı davranış...
```

7.4.2. toString() metodu

`toString()` metodu arqument tələb etmir; o sətiri qaytarır, hər hansı obrazda təqdim edilən tipi və/və ya obyektin qiymətinin sətirə çevirir. JavaScript interpretatoru obyekti sətirə dəyişdirmək tələb olunan bütün hadisələrdə obyektin bu metodunu çağırır. Məsələn, *`alert`*() və ya document.write() kimi metodlarda + operatoru vasitəsi sətirlə obyektin birləşdirilməsi və ya obyektin ötürülməsi zamanı *`toString()`* metodu çağrılır. *`toString()`* metodu susmaya görə çox informativ deyil. Məsələn, aşağıdakı fraqment sadəcə s dəyişəninə "[object Object]" sətirini yazır:

```
var s ={
        x:1,
        y:1
       }.toString ();
```

Bu metod xüsusilə faydalı məlumatları susmaya görə əks etdirmir, buna görə də bir çox siniflər *`toString()`* metodunun şəxsi versiyalarını müəyyən edir. Məsələn, massiv sətirə dəyişdirildikdə, biz massiv

elementlərinin hər biri sətirə dəyişdirilərək siyahı şəklində təqdim olunacaq, amma funksiya sətirə dəyişdirildikdə, nəticə olaraq, biz bu funksiyanın mənbə kodunu alacağıq.
9-cu fəsildə obyektlərin öz şəxsi tipləri üçün `toString()` metodunu yenidən təyin etmək olunması qaydası göstərilir.

7.4.3. toLocaleString()

ECMAScript v3-də və JavaScript 1.5-də `Object` sinifi `toString()` metoduna əlavə olaraq toLocaleString() da metodunu müəyyən edir. Sonun təyinatı obyektin lokallaşmış sətir ilə təqdim edilməsindən ibarətdir. `Object` sinifində susmaya görə müəyyən edilən toLocaleString() metodu heç bir lokallaşdırma yerinə yetirmir; o həmişə `toString()` qaytardığı sətiri qaytarır. Ancaq yarımsiniflər toLocaleString() metodunun şəxsi versiyalarını müəyyən edə bilər. ECMAScript v3-də Array, Date və Number sinifləri lokallaşmış qiymətlər qaytaran toLocaleString() metodunun versiyalarını müəyyən edir.

7.4.4. valueOf() metodu

`valueOf()` metodu demək olar ki `toString()` metoduna bənzəyir, amma bu meod obyekti sətirdən başqa hər hansı elementar tipdə (adətən ədəd tipində) olan qiymətə dəyişdirmək tələb olunan zaman çağrılır. Əgər obyekt elementar tipdə olan qiymət kontekstində istifadə olunursa JavaScript interpretatoru bu metodu avtomatik çağırır. `valueOf()` metodu susmaya görə heç nə yerinə yetirmir, amma obyektlərin bəzi inteqrasiya edilmiş kateqoriyaları (məsələn, Date.`valueOf()`) `valueOf()` metodunu yenidən təyin edir. 9-cu fəsildə obyektlərin şəxsi tiplərində `valueOf()` metodunu yenidən təyin edilməsi qaydası təsvir edilmişdir.

7.4.5. hasOwnProperty() metodu

hasOwnProperty() metodu əgər tətbiq edildiyi obyektdə, öz tək sətir arqumentində göstərilmiş ada uyğun varis olunmuş xüsusiyyət müəyyən edilməmişsə, *true* qiymətini qaytarır. Əks təqdirdə bu metod *false* qiymətini qaytarır. Məsələn:

```
var o = {};
o.hasOwnProperty("undef");              // false: xüsusiyyət
təyin edilməyib
o.hasOwnProperty("toString");       // false: toString –
varis olunmuş
                                    // xüsusiyyətdir
Math.hasOwnProperty("cos");           // true: Math
obyektində cos xüsusiyyəti
                                    // mövcuddur
```

Xüsusiyyətlərin varisliyi barədə 9-cu fəsildə bəhs olunur.
hasOwnProperty() metodu ECMAScript v3 standartıyla təyin edilir və JavaScript 1.5-də və daha gec versiyalarda realizasiya edilmişdir.

7.4.6. propertyIsEnumerable() metodu

propertyIsEnumerable() metodu əgər obyektdə metodun tək sətir arqumentində göstərilmiş ada müvafiq xüsusiyyəti təyin edilibsə və bu xüsusiyyət *for*/*in* dövrüylə sadalana bilirsə *true* qiymətini qaytarır. Əks təqdirdə metod *false* qiymətini qaytarır. Məsələn:

```
o.propertyIsEnumerable("x");              // true: xüsusiyyət
təyin olunub və
                                    // sadalana bilir
o.propertyIsEnumerable("y");              // false: xüsusiyyət
təyin olunmayıb
o.propertyIsEnumerable("valueOf");        // false: xüsusiyyət
sadalana bilmir
```

propertyIsEnumerable() metodu ECMAScript v3 standartıyla təyin edilir və JavaScript 1.5-də və daha gec versiyalarda realizasiya edilmişdir. Yadınızda saxlayın ki, istifadəçi tərəfindən müəyyən edilən bütün obyekt xüsusiyyətləri sadalanandır. Sadalana bilməyən xüsusiyyətlər adətən varis olmuş xüsusiyyətlər (xüsusiyyətlərin varisliyi mövzusuna 9-cu

fəsildə baxılır) olur, buna görə də praktik olaraq bu metod həmişə hasOwnProperty() metodunun qaytardığı qiyməti qaytarır.

7.4.7. isPrototypeOf() metodu

`isPrototypeof()` metodu əgər arqument qismində ötürülən metodun aid olduğu obyekt, obyekt-prototipində mövcuddursa *`true`* qiymətini qaytarır. Əks təqdirdə metod *`false`* qiymətini qaytarır. Məsələn:

```
    var o = {};
    Object.prototype.isPrototypeOf(o);            // true:
o.constructor == Object
    Object.isPrototypeOf(o);                   // false
    o.isPrototypeOf(Object.prototype);            // false
    Function.prototype.isPrototypeOf(Object);      // true:
                                          // Object.constructor ==
                                          Function
```

7.5. Massivlər

Massiv – nömrələnmiş qiymətləri ehtiva edən məlumat tipidir. Hər bir nömrələnmiş qiymət massivin elementi adlanır. Bu elementə bağlı nömrə (ədəd) isə, onun indeksi adlanır. JavaScript-in tipləşdirilməmiş dil olduğunu əsas götürərək, massivin elementi istənilən tipə malik ola bilər. Həmçinin, bir massivin elementləri müxtəlif tiplərdə ola bilər. Massivin elementləri hətta başqa massivləri özündə ehtiva edə bilər ki, bu da bizə massivlərin massivlərini yaratmaq imkanı verir.
Kitab boyunca biz ayrı-ayrı məlumat tiplərini ehtiva edən obyektlərin və massivlərə şahidi olduq. Bu faydalı və səmərəli sadələşdirmə – JavaScript-də obyektlər və massivlər proqramlaşdırmanın məsələlərinin əksəriyyəti üçün müxtəlif tiplər kimi baxmaq olar. Ancaq ki, obyektlərin və massivlərin davranışını anlamaq yaxşıdır: massiv – bu deyil ki, başqa, əlavə funksionallığın incə qatıyla obyekt kimi. Bu görmək olar, *`typeof`* operatorunun köməyi ilə massivin tipi müəyyən edib – "object" sətiri alınacaq.
Kvadrat mötərizələr ilə massiv literalının köməyilə bir-birindən vergüllərlə ayrılmış sadə siyahılı massiv yaratmaq olduqca asandır.

Məsələn:

```
    var empty = [];                 // Boş massiv
    var primes = [2, 3, 5, 7, 11];  // Massiv beş ədəd
elementindən ibarətdir
    var misc = [ 1.1, true, "a" ];  // 3 müxtəlif tipdə olan
element
```

Massivin literalındakı qiymət mütləq surətdə sabit olmamalıdır və buraya istənilən ifadə yerləşdirilə bilər:

```
    var base = 1024;
    var table = [base, base+1, base+2, base+3];
```

Massiv literalları özündə obyekt literallarını və ya başqa massiv literallarını ehtiva bilər:

```
    var b = [[1,{ x:1, y:2}], [2,{ x:3, y:4}]];
```

yeni yaradılmış massivdə massiv literalının birinci qiyməti 0-indeksli elementdə, ikinci qiymət isə, 1-indeksli elementdə, və s. yerləşir. Əgər literalda elementin qiyməti qeyri-müəyyən qiymətlə təyin edilmişdirsə, onda massiv elementləri həmin qiymət aralığında yaradılacaq:

```
    var count = [1, 3];     // 3 elementli massiv, orta element
müəyyən
                            // edilməmişdir.
    var undefs = [,];       // 2 elementli massiv, hər iki
element müəyyən
                            // edilməmişdir.
```

Massivin yaradılması Array() konstruktorunun çağırışı ilə də mümkündür. Konstruktoru üç müxtəlif üsulla çağırmaq olar:

- *Konstruktorun arqumentlərsiz çağırışı:*

```
var a = new Array();
```

Bu zaman boş massiv yaradılacaq və bu [] hal literalına ekvivalentdir.

- *Massiv elementlərinin qiymətlərini konstruktorda açıq-aydın göstərmək olar:*

```
var a = new Array(5, 4, 3, 2, 1, "testing, testing");
```

Bu halda konstruktorda arqumentlər siyahısını əmələ gəlir. Hər bir arqument isə, elementin qiymətini müəyyən edir və bu arqumentlər istənilən tipdə ola bilər. Massiv elementlərinin nömrələnməsi 0-dan başlanır. Massivin length xüsusiyyəti sayəsində konstruktora verilmiş elementlərin miqdarı müəyyən edilir.

• *Massivin uzunluğunu müəyyən edən bir ədəd arqumentli çağırış:*

```
var a = new Array(10);
```

Bu forma göstərilən miqdarda massiv elementi yaradır (hansı ki, bu elementlərin hər biri *undefined* qiyməti ehtiva edir) və arqumentdə göstərilmiş ədəd massivin length xüsusiyyətinin qiyməti olaraq təyin edilir. Massiv uzunluğu qabaqcadan məlumdursa Array()-konstruktoruna bu cür müraciətin forması, ilkin yerləşdirmə üçün istifadə oluna bilər. Analoji vəziyyətdə massivlərin literallarının narahatdır.

7.6. Massivin elementlərinin oxunması və yazılması

massivin elementlərinə Giriş [] operatorun köməyilə həyata keçirilir. Mötərizələrdən solda massivə istinad olmalıdır. Mötərizələrin daxilində mənfi olmayan tam qiymətə malik sərbəst ifadəni olmalıdır. Bu sintaksis yararlıdır oxuma üçün yararlı olduğu kimi, massivin elementinə qiymətin yazılılması üçün də yararlıdır. Beləliklə, aşağıdakı göstərilən JavaScript-təlimatlarından istifadə etmək mümkündür:

```
value = a[0];
a[1] = 3.14;
i = 2;
a[i] = 3;
a[i + 1] = "hello";
a[a [i]] = a[0];
```

Bəzi dillərdə massiv elementlərinin indeksləşməsi 1-dən başlayır. Ancaq JavaScript-də (C, C++ və Java-da olduğu kimi) massivin birinci elementi indeksi 0-a bərabərdir.
Artıq qeyd edildiyi kimi, [] operatoru həmçinin obyektin adlandırılmış xüsusiyyətlərinə giriş üçün istifadə oluna bilər:

```
my['salary'] *= 2;
```

Bir halda ki, massivlər obyektlərin ixtisaslaşmış sinifidir, onda massivlərə istənilən qədər xüsusiyyətlərini müəyyən etmək və onlara nöqtə və [] operatorları vasitəsilə müraciət etmək olar.

Yadınızda saxlayın ki, massiv 2^{32} – 1 indeksli mənfi olmayan ədəd olmalıdır.

Əgər indeks ədədi çox böyükdürsə, mənfidirsə və ya həqiqi ədəddirsə (və ya məntiqi, obyekt və başqa qiymətdirsə), JavaScript onu sətirə dəyişdirəcək və dəyişdirilmiş sətirə massiv indeksi kimi deyil, obyekt xüsusiyyəti kimi baxacaq. Beləliklə, aşağıdakı sətirdə yeni massiv elementi deyil, – 1.23" adlı yeni xüsusiyyət yaradılacaq

```
a[1.23] = true;
```

7.6.1. Massivə yeni elementin əlavə edilməsi

C və Java kimi dillərdə, massiv elementləri təsbit edilmiş saya malikdir, hansı ki, massivin yaradılması zamanı müəyyən edilməlidir. Bu JavaScript-ə aid deyil – JavaScript-də massiv elementləri istənilən miqdarda ola bilər və bu miqdarı istənilən vaxt dəyişdirmək olar.
Massivə yeni element əlavə etmək üçün, ona qiymət mənimsəmək kifayətdir:

```
a[10] = 10;
```

JavaScript-də Massivlər seyrəldilmiş ola bilər. Bu isə o deməkdir ki, massiv indeksləri mütləq ədədlərin fasiləsiz diapazonuna aid deyil; JavaScript reallaşdırması yalnız faktiki mövcud olan massiv o elementləri

yaddaş ayırır. Buna görə də aşağıdakı fraqmentin icrası nəticəsində JavaScript interpretatoru ehtimal ki, yalnız 0 və 10 000 indeksli massivin elementləri yaddaş ayıracaq, onların arasında olan 9 999 element üçün yaddaş ayrılmayacaq:

```
a[0] = 1;
a[10000] = "bu element 10,000";
```

Nəzərə alın ki, massiv elementləri həmçinin obyektlərə də əlavə edilə bilər:

```
var c = new Circle (1,2,3);
c[0] = "bu obyektdə massivin elementidir!";
```

Bu nümunə sadəcə "0" adlı obyektin yeni xüsusiyyətini müəyyən edilir. Ancaq obyektə massiv elementinin əlavə edilməsi obyekti massiv etmir.

7.6.2. Massiv elementlərinin silinməsi

delete operatoru massivin elementinin qiymətini *undefined* təyin edir və bu halda massiv elementi öz mövcudluğunu davam edir. Belə ki, elementlərin silinməsi, qalan elementləri də düzgün yerləşdirmək üçün, massiv metodlarından istifadə etmək lazımdır. Array.shift() metodu birinci elementini, Array.pop() metodu isə massivin son elementini silir.
Array.splice() metodu – elementlərin fasiləsiz diapazonda silir. Bu funksiyalar fəsilin sonunda təsvir ediləcək.

7.6.3. Massivin uzunluğu

İstər `Array()`-konstruktorunun köməyilə, istərsə də massiv literalının köməyilə müəyyən edilmiş bütün massivlər, elementlərin miqdarını ehtiva edən spesifik length xüsusiyyətinə malikdir. Bir halda ki, massivlər qeyri-müəyyən miqdarda elementlərə malik ola bilər, bu fikri daha dəqiq və qısa ifadə edək: length xüsusiyyəti həmişə massiv elementinin ən böyük nömrəsi qədər vahiddən böyükdür. Adi obyekt xüsusiyyətlərindən fərqli olaraq, massivin length-i xüsusiyyəti massivə yeni elementlərin

əlavə edilməsi zamanı avtomatik yenilənir. Bu şərait aşağıdakı fraqmentdə təsvir edilir:

```
var a = new Array();          // a.length == 0 (heç bir
element müəyyən
                              // edilməmişdir)
a = new Array(10);            // a.length == 10 (0-dan 9-a
qədər müəyyən edilmiş
                              // boş element)
a = new Array(1,2,3);         // a.length == 3 (0-dan 2-yə
qədər müəyyən edilmiş
                              // element)
a = [4, 5];                   // a.length == 2 (0 və 1
elementləri müəyyən
                              // edilmişdir)
a[5] = -1;                    // a.length == 6 (0, 1 və 5
elementləri müəyyən
                              // edilmişdir)
a[49] = 0;                    // a.length == 50 (0, 1, 5 və 49
elementləri müəyyən
                              // edilmişdir)
```

Xatırladaq ki, massiv indeksləri 2^{32} - 1 aralığında olmalıdır, yəni, length xüsusiyyətinin mümkün olan maksimal xüsusiyyət qiyməti 2^{32} - 1 bərabərdir.

7.6.4. Massiv elementlərinin dövrü

length xüsusiyyətindən əksər hallarda dövrlərdə massiv elementlərinin sadalanması üçün istifadə olunur:

```
var fruits = [" manqo", "banan", "albalı", "şaftalı"];
for (var i = 0; i < fruits.length; i++)
     alert (fruits[i]);
```

Əlbəttə, bu nümunədə güman edilir ki, massiv elementləri fasiləsiz yerləşdirilmişdir və element 0-dan başlanırlar. Əgər bu belə deyilsə, massivin hər bir elementinə müraciətdən əvvəl, elementin müəyyənliyini yoxlamaq lazımdır:

```
for (var i = 0; i < fruits.length; i++)
if (fruits [i]! = undefined)
    alert (fruits [i]);
```

Analoji yanaşma Array() konstruktorunun çağırışı ilə yaradılmış massiv elementlərinin inisializasiyası üçün də istifadə oluna bilər:

```
var lookup_table = new Array (1024);
for (var i = 0; i < lookup_table.length; i++)
lookup_table [i] = i * 512;
```

7.6.5. Massivin artımı və ixtisarı

Massivin length xüsusiyyəti oxuma üçün əlçatan olduğu kimi, yazı üçün də əlçatandır. Əgər qiymətə length xüsusiyyətini təyin etsək, massivin uzunluğu cari uzunluğdan, yeni uzunluğa qədər qısaldılır; yeni diapozona istənilən element, indeks ixtisara salınır və qiymətləri itir. Əgər length xüsusiyyətini cari uzunluğdan böyük təyin etsək, onda, massivin cari qiyməti, yeni ölçüyə qədər artır və massivə yeni qeyri-müəyyən elementlər əlavə edilir.
Nəzərə alın ki, hobyektlərə massiv elementləri mənimsədilə bilməsinə baxmayaraq, obyektlər length xüsusiyyətini dəstəkləmir. Bu xüsusiyyət və onun xüsusi davranışı – məxsusi olaraq massivlərə aiddir və bu onları obyektlərdən fərqləndirən əlamətlərdən biridir. Massivləri obyektlərdən fərqləndirən əlamətlər – Array sinifi vasitəsilə müəyyən edilən və ***bölmə 7.7-də*** təsvir edilən müxtəlif massiv metodlardır.

7.6.6. Çoxölçülü massivləri

JavaScript-də bu cür massivləri, massivin massivləri adlandırsaq daha düzgün çıxar, çünki JavaScript "əsl" çoxölçülü massivləri dəstəkləmir. Massivlərin massivindəkı məlumat elementinə giriş üçün [] operatorundan iki dəfə istifadə etmək kifayətdir. Məsələn, fərz edək ki, matrix dəyişəni – ədəd massivlərinin massividir. matrix[x] istənilən elementi – ədədlərin massividir. Massivdə müəyyən ədədə giriş üçün matrix[x][y] yazmaq lazımdır. Aşağıda konkret nümunə göstərilmişdir ki, nümunədə ikiölçülü massivdən vurma cədvəli kimi istifadə olunur:

```
// Çoxölçülü massiv yaradırıq
 var table = new Array(10); // Cədvəldə 10 sütun var
```

```
for(var i = 0; i < table.length; i++)
table[i] = new Array(10); // Hər sütunda 10 sətir var

// Massivin inisializasiyası
for(var row = 0; row < table.length; row++) {
  for(col = 0; col < table[row].length; col++) {
     table[row][col] = row*col;
  }
}

// Çoxölçülü massivin köməyilə 5*7 hasilinin hesablanması
var product = table[5][7];  // 35
```

7.7 Massiv metodları

[] massivlərlə Array sinifiylə verilən müxtəlif metodlara vasitəsi ilə işləmək olar. Bu metodlar aşağıda bölmələrdə təqdim edilmişdir. Metodlardan bir çoxu Perl proqramlaşdırma dilindən götürülmüşdür; ona görə bu metodlar Perl-lə işləmiş proqramçılara onlar tanış görünə bilər.

7.7.1. join() metodu

Array.join() metodu massivin bütün elementlərini sətirlərə dəyişdirir və onları bir sətirdə birləşdirir. Alınmış yekun sətirdə elementlərin bir-birindən ayrılması üçün nəzərdə tutulmuş vacib olmayan sətir arqumentini də göstərmək olar. Əgər ayırıcı göstərilməyibsə, ayırıcı qismində vergüldən istifadə olunur. Məsələn, aşağıdakı fraqment nəticədə "1,2,3" sətirini verir:

```
var a = [1, 2, 3]; // Göstərilən elementlləri ehtiva edən
massiv yaradılır
var s = a.join();  // s == "1,2,3"
```

Növbəti nümunədə vacib olmayan ayırıcı təyin edilir, başqa nəticəyə alınır:

```
s = a.join(", "); // s == "1, 2, 3" vergüldən sonra boşluğa
diqqət edin.
```

Array.join() metodu sətirin bölünməsi yolu ilə massiv yaradan String.split() metodunun əksidir.

7.7.2. reverse() metodu

Array.reverse() metodu massivdəki elementlərin ardıcıllıq istiqamətini əks istiqamətdə dəyişdirir və alınmış yeni massivi qaytarır. Bu metod əməliyyatı massiv üzərində edir, başqa sözlə, elementləri elementlər üzərində işləyərkən yeni massivi yaratmır, əməliyyat mövcud massivin üzərində edir. Məsələn, aşağıdakı fraqmentdə, reverse() və join() metodları istifadə olunur, nəticədə isə proqram "3,2,1" sətirini verir:

```
    var a = new Array (1,2,3);  // a[0] = 1, a[1] = 2, a[2] = 3
    a.reverse();                        // indi isə, a[0] = 3,
a[1] = 2, a[2] = 1
    var s = a.join();           // s =="3,2,1"
```

7.7.3. sort() metodu

Array.sort() metodu massiv üzərində massivin elementləri çeşidləyir və çeşidlənmiş massivi qaytarır. Əgər sort() metodu arqumentlərsiz çağırılırsa, onda o massivin elementlərini əlifba sıra ilə (zəruri olduqda müvəqqəti preobrazuya onlar müqayisənin icra edilməsi üçün sətirlərə) çeşidləyir:

```
    var a = new Array (" banana", "cherry", "apple");
    a.sort();
    var s = a.join (", "); // s == "apple, banana, cherry"
```

Çeşidləmə zamanı qeyri-müəyyən elementlər massivin sonuna keçir.
Əlifba sırasından başqa hər hansı sırada çeşidləmə üçün sort() metoduna arqument təyin edərək müqayisə funksiyası qurmaq olar. Qurulan funksiyada, göstərilmiş iki arqumentdən biri çeşidlənmiş siyahının əvvəlində olmalıdır. Əgər birinci arqument ikincidən əvvəl olmalıdırsa, müqayisə funksiyası mənfi rəqəmi qaytarır. Əgər çeşidlənmiş massivdə birinci arqument ikincidən sonra gəlməlidirsə, onda funksiya sıfırdan ən böyük ədədi qaytarır. Və əgər iki qiymət bir-birinə ekvivalentdirsə (yəni, onların yerləşməsə sırası əhəmiyyət daşımırsa), müqayisə

funksiyası 0 qiymətini qaytarır. Buna görə də, məsələn, elementin əlifba sırasında deyil, ədəd sırasında çeşidləməsi üçün aşağıdakı formadan istifadə etmək olar:

```
    var a = [33, 4, 1111, 222];
    a.sort();           // Əlifba sırası: 1111, 222, 33, 4
    a.sort(function(a,b) {  // Ədəd sırası: 4, 33, 222, 1111
          return a-b;    // < 0, 0, və ya > 0 olan ədədləri
qaytarır
         });
```

a və b çeşidləmə qaydasından asılı olaraq nəzərə alın ki, bu fraqmentdə funksional literaldan istifadə etmək olduqca rahatdır. Müqayisə funksiyası yalnız bir dəfə çağırıldığına görə, funksiyaya ad verməyə ehtiyac yoxdur. Massivin elementlərinin çeşidləməsinin daha bir nümunəsi kimi registrə həssas olmayan əlifba sırasını yerinə yetirə bilərsiniz, bunun üçün sözügedən metoda müqayisə funksiyası verib, müqayisədən əvvəl hər iki arqumenti kiçik hərf registrinə (toLowerCase() metodunun köməyi ilə) dəyişdirmək lazımdır. Bundan əlavə bir çox çeşidləmə funksiyaları fikirləşmək olar, məsələn: mənfi ədədlər, tək ədədlərin cüt ədəddən əvvəl gəlməsi və s. kimi sıralanmaya uyğun çeşidləmə yerinə yetirmək olar. Massivin müqayisə edilən elementləri sadə elemenlər deyil (ədəd və sətir kimi), obyektlərdən təşkil olunarsa, bu metod vasitəsilə maraqlı imkanlar reallaşdırmaq mümkündür

7.7.4. concat() metodu

Array.concat() metodu massivin əvvəlki elementlərini özündə ehtiva etməklə, metodda göstərilmiş arqumentlərə uyğun yeni massiv yaradır. Əgər bu arqumentlərdən hər hansı biri özü massivdirsə, yekunlaşdırıcı massivə onun elementlərini əlavə edilir. Ancaq nəzərə alın ki, massivlərdən massivlərin rekursiv bölünməsi mümkün deyil. Aşağıda bir neçə nümunə nümayiş etdirilir:

```
    var a = [1,2,3];
    a.concat(4, 5)            // [1,2,3,4,5] qaytarır
    a.concat([4,5]);        // [1,2,3,4,5] qaytarır
    a.concat([4,5],[6,7])   // [1,2,3,4,5,6,7] qaytarır
    a.concat(4, [5,[6,7]])  // [1,2,3,4,5,[6,7]] qaytarır
```

7.7.5. slice() metodu

Array.slice() Metodu tətbiq edildiyi massivin göstərilən arqumentlərə uyğun fraqmentini və ya altmassivini qaytarır. Metodun iki arqumenti qaytarılan fraqmentin başlanğıc və son nöqtələrini təyin edir. Qaytarılan massiv, göstərilmiş birinci arqumentdən sonuncu arqumentə qədər (sonuncu arqumentin elementi daxil olmamaqla) element dəstindən ibarət olur. Əgər yalnız bir arqument göstərilmişdirsə, bu arqumentdən başlayaraq massivin sonuna kimi bütün elementləri ehtiva edən massiv yaradılır. Əgər arqumentlərdən hər hansı biri mənfidirsə, onda hesablanma massivin s massivin sonundan başlayır. Belə, arqumentə misal olaraq, – 1, massivin son elementini, – 3 isə axırdan üçüncü elementini verir. Aşağıda metodun tətbiq edildiyi bir neçə nümunə göstərilmişdir:

```
var a = [1,2,3,4,5];
a.slice(0,3);          // [1,2,3] qaytarır
a.slice(3);             // [4,5] qaytarır
a.slice(1,-1);         // [2,3,4] qaytarır
a.slice(-3,-2);         // [3] qaytarır
```

7.7.6. splice() metodu

Array.splice() Metodu – massivin elementlərinin yerləşdirməsi və ya silinməsi üçün universal metoddur. Bu metod əməliyyatı massivi üzərində edir, slice() və concat() metodlarından fərqli olaraq yeni massiv yaratmır. Nəzərə alın ki, splice() və slice() çox oxşar adlara malik olsa da, müxtəlif əməliyyatları yerinə yetirirlər.
splice() metodu massivdən elementləri silə, massivə yeni elementlər daxil edə bilər və ya eyni zamanda hər iki əməliyyatı yerinə yetirə bilər. Massivin elementləri zəruri olduqda yerini dəyişir ki, yerləşdirmədən və ya silinmədən sonra kəsintisiz ardıcıllıq yaransın. splice() metodunun birinci arqumenti yerləşdirmənin və / və ya silinmənin başlandığı massiv mövqesini təyin edir. İkinci arqument isə massivdən silinəcək elementlərin miqdarını təyin edir. Əgər ikinci arqument göstərilmişsə,

göstərilən mövqedən massivin sonuna kimi ilkdən massivin bütün elementləri silinir. splice() metodu silinmiş elementlərin massivini və ya boş massiv (əgər heç bir elementlərdən silinməmişdirsə) qaytarır. Məsələn:

```
    var meyveler = ['alma', 'armud', 'heyva', 'çiyələk'];

    // massivə heç bir element silməmək şərtilə (0) 2-ci indeksə
yeni “gilas”
    // elementinin əlavə olunması. Bu zaman mövcud 2-ci
indeksdəki elementdən
    // (indiki halda bu “heyva” elementindən başlayaraq) bütün
massiv
    // elementlərinin indeksi bir vahid artır
    var removed = meyveler.splice(2, 0, 'gilas');
    // indi meyveler massivi ['alma', 'armud', ‘gilas’, 'heyva',
'çiyələk']
    // elementlərini ehtiva edir və massivdən heç bir element
silinmir

    // meyveler massivinin 3-cü indeksindən başlayaraq 1
elementin silinməsi
    removed = myFish.splice(3, 1);
    // indi meyveler massivi ['alma', 'armud', ‘gilas’,
'çiyələk']
    // elementlərini ehtiva edir. Silinən massiv elementi:
“heyva”

    // meyveler massivinin 2-ci indeksindən başlayaraq 1
elementin silinməsi
    // və buraya yeni “alça” elementinin əlavə olunması
    removed = meyveler.splice(2, 1, 'alça');
    // indi meyveler massivi ['alma', 'armud', ‘alça’, 'çiyələk']
    // elementlərini ehtiva edir. Silinən massiv elementi:
“gilas”

    // meyveler massivinin 0-cı indeksindən başlayaraq 2
elementin silinməsi
    // və buraya yeni “ərik”, “şaftalı”, “üzüm” elementlərinin
əlavə olunması
    removed = meyveler.splice(0, 2, 'ərik', 'şaftalı', 'üzüm');
    // indi meyveler massivi ['ərik', 'şaftalı', 'üzüm', ‘gilas’,
'çiyələk']
    // elementlərini ehtiva edir. Silinən massiv elementlərləri:
“alma”, “armud”
```

```
    // meyveler massivinin 2-ci indeksindən başlayaraq 2
elementin silinməsi
    removed = meyveler.splice(meyveler.length -3, 2);
    // indi meyveler massivi ['ərik', 'şaftalı', 'üzüm', ‘gilas’,
'çiyələk']
    // elementlərini ehtiva edir. Silinən massiv elementlərləri:
“üzüm”, “gilas”
```

Nəzərə alın ki, concat()-dan fərqli olaraq, splice() metodu ötürülən massiv-arqumentinin ayrı-ayrı elementlərini ayrılıqda silmir. Yəni əgər yerləşdirmə üçün metoda massiv ötürülürsə, metod özünü massiv elementləri kimi deyil, massiv kimi göstərir.

7.7.7. push() və pop() metodları

push() və pop() metodları massivlərlərin steklər kimi işləməsinə şərait yaradır. push() metodu massivin sonuna bir və ya bir neçə yeni element əlavə edir və onun yeni uzunluğunu qaytarır. Pop() metodu isə əksinə olaraq, massivin son elementi uzaqlaşdırır, massivin uzunluğunu azaldır və massivi silinmiş qiyməti qaytarır. Nəzərə alın ki, sözügedən hər iki metod əməliyyatı cari massivin üzərində edirlər və əlavə massiv surəti yaratmırlar.

```
    var stack = [];              // stek: []
    stack.push(1,2);              // stek: [1,2]              2
qaytarır
    stack.pop();               // stek: [1]               2 qaytarır
    stack.push(3);               // stek:[1,3]               2
qaytarır
    stack.pop();               // stek: [1]               3 qaytarır
    stack.push([4,5]);         // stek: [1,[4,5]]         2 qaytarır
    stack.pop()                     // stek: [1]                [4,5]
qaytarır
    stack.pop();               // stek: []               1 qaytarır
```

7.7.8. unshift() və shift() metodları

unshift() və shift() metodları bir çox məqamda push() və pop() metodlarına oxşayır, lakin bu metodlar massivin başlanğıcına müvafiq

olaraq element əlavə edir və silirlər.
unshift() metodu massiv başlanğıcına element əlavə edir və mövcud elementlərin yerini artan indeksə uyğun dəyişdirərək, massivin yeni uzunluğunu qaytarır.
Shif() metodu massivin başlanğıcındakı elementi silir və qalan elementləri azad olan (silinmiş) yerdə yerləşdirir.

```
    var a = [];                    // a:[]
    a.unshift(1);              // a:[1]            Qaytarır: 1
    a.unshift(22);              // a:[22,1]       Qaytarır: 2
    a.shift();                    // a:[1]             Qaytarır:
22
    a.unshift(3,[4,5]);        // a:[3,[4,5],1]       Qaytarır: 3
    a.shift();                    // a:[[4,5],1]       Qaytarır: 3
    a.shift();                    // a:[1]             Qaytarır:
[4,5]
    a.shift();                    // a:[]              Qaytarır: 1
```

unshift() metoduna bir neçə arqument ötürərərkən bir neçə məqama diqqət yetirməlisiniz: Arqumentlər bir-bir ötürülmür (splice() metodunda olduğu kimi). Bu isə o deməkdir ki, yekun massivdə onlar arqument siyahısına müvafiq olaraq sıralanacaq. Bir-bir əlavə edilərsə onlar əks istiqamətdə sıralanacaq.

8

Funksiyalar

Funksiya – bir dəfə təyin edilən və istənilən qədər çağrıla bilən proqram kodunun blokudur. Funksiyalar parametrlərə və ya arqumentlərə malik ola bilər. Bunlar lokal dəyişən və qiymətləri funksiya çağırışı zamanı təyin edilir. Funksiyalar qaytarılan qiymətin hesablanması üçün tez-tez öz arqumentlərindən istifadə edir. Qaytarılan qiymət funksiya çağırış ifadəsinin qiymətidir. Əgər funksiya obyekt kontekstində çağırılırsa, onda o metod adlanır və obyektin özü ona naməlum arqument şəklində ötürülür. Çox güman ki, siz artıq funksiya konsepsiyası ilə tanışsınız, əgər belə anlayışlarla rast gəlinird, alt proqram və prosedur kimi.

Bu fəsildə biz şəxsi JavaScript-funksiyalarının təyini və çağırışına istiqamətlənəcəyik. Xatırladaq ki, JavaScript-də `eval()`, `parseInt()` kimi bir neçə inteqrasiya edilmiş funksiya var. JavaScript-in kliyent dilində də `document.write()` və `alert`() kimi başqa funksiyalar mövcuddur. İnteqrasiya edilmiş JavaScript-funksiyaları da eynilə istifadəçi tərəfindən müəyyən edilən funksiyalar kimi tətbiq edilir.

JavaScript-də funksiyalar və obyektlər bir-birilə sıx bağlıdır. Bu səbəbdən biz funksiyaların bəzi imkanlarının müzakirəsini 9-cu fəsilə qədər təxirə salacağıq.

8.1. Funksiyaların təyini və çağrılması

6-cı fəsildə qeyd edildiyi kimi, funksiyanı *function* təlimatından istifadə edərək təyin. Təlimat *function* açar sözündən ibarətdir və aşağıdakı tərkibdə olur:

- funksiyanın adı;
- yumru mötərizələrlə bağlanmış və bir-birilə vergüllərlə ayrılmış parametr adlarının vacib olmayan siyahısı;
- fiqurlu mötərizələrə əhatələnmiş funksiyanın gövdəsini təşkil edən JavaScript-təlimat(lar)ı

Nümunə 8.1-də təyin edilmiş bir neçə funksiya nümunəsi göstərilmişdir. Hərçənd ki, bu funksiyalar qısa və sadədir, lakin bu funksiyalar burada bütün sadalanan bütün nüansları ehtiva edir. Nəzərə alım ki, funksiyalara müxtəlif miqdarda arqument müəyyən edilə bilər, funksiyalar həmçinin özündə saxlaya və ya *return* təlimatını özündə saxlamaya bilər. *Return* təlimatı 6-cı fəsildə təsvir edilmişdir; bu təlimat funksiyanın icrasını dayandırır və təlimatda göstərilmiş ifadənin (əgər varsa) qiymətini funksiyanı çağıran tərəfə qaytarır; ifadə yoxluğunda bu təlimat *undefined* qiymətini qaytarır. Əgər funksiya *return* təlimatını ehtiva etmirsə, onda bu funksiya sadəcə öz gövdəsində olan bütün təlimatları yerinə yetirir və qeyri-müəyyən qiymət (*undefined*) qaytarır.

Nümunə 8.1. *JavaScript-funksiyaların təyini*

```
// Funksiya-üzlüyü, bəzən bundan document.write()-ın yerinə
istifadə edilir.
// Bu funksiyada return təlimatı yoxdur, buna görə də,
funksiya qiymət
// qaytarmır.
function print(msg)
{
  document.write(msg, "<br>");
}

// İki nöqtə arasındakı məsafəni hesablayan və hesablanmış
nəticəni qaytaran
// funksiya.
function distance(x1, y1, x2, y2)
{
  var dx = x2 - x1;
  var dy = y2 - y1;
```

```
  return Math.sqrt(dx*dx + dy*dy);
}

// Faktorialın hesablanması üçün istifadə edilən
// rekursiv funksiya (özünə istinad edən).
// Xatırladaq ki, burada x faktorial, yəni x! – x və x-dən
kiçik bütün müsbət
// tam ədədlərin hasilidir.
function factorial(x)
{
  if (x <= 1)
  return 1;
  return x * factorial(x 1);
}
```

Müəyyən edilmiş funksiya, 5-ci fəsildə təsvir edilmiş () operatorunun köməyi ilə çağırıla bilər. xatırlayırsınızsa, mötərizələrdən əvvəl funksiya adı, sonra isə bir-birindən vergüllə ayrılan vacib olmayan arqument siyahısı göstərilir (faktiki olaraq yumru mötərizələrdən əvvəl istənilən JavaScript ifadəsi göstərilə bilər, hansı ki, qiyməti (mənanı) funksiyanı qaytarır). ***Nümunə 8.1***-də müəyyən edilmiş funksiyalar aşağıdakı qaydada çağırıla ola bilər:

```
print(" Salam, " + name);
print("Mənim ana səhifəmə xoş gəlmisiniz!");
total_dist = distance (0,0,2,1) + distance (2,1,3,5);
print("Bu ehtimal bərabərdir: " + factorial (5) / factorial
(13));
```

Funksiya çağırışı zamanı mötərizələrin daxilində göstərilmiş bütün ifadələr hesablanır və alınmış qiymətlər funksiyanın arqumentləri kimi istifadə olunur. Bu qiymətlər, uyğun olaraq adları funksiyanın gövdəsində təyin edilən parametrlərə mənimsədilir və funksiya, göstərilən adlar üzrə parametrlərə istinad edərək onlarla işləyir. Nəzərə alın ki, bu dəyişən-parametrlər yalnız funksiya yerinə yetirilən zamanı müəyyən edilir; funksiyanın işi bitdikdən sonra bu dəyişənlər saxlanılmır (bu əhəmiyyətli istisna, bölmə 8.8-də ətraflı təsvir edilir).

JavaScript – *qismən* tipləşdirilmiş dildir, buna görə də funksiya parametrlərinin tipini göstərmək tələb olunmur və JavaScript məlumat tipinin funksiya tələblərinə uyğunluğunu yoxlamır. Əgər arqumentin tipi sizin üçün əhəmiyyətlidirsə, siz verilən arqumenti `typeof` operatorunun

köməyilə müstəqil yoxlaya bilərsiniz. Bundan başqa, JavaScript, funksiyaya parametrlərin lazım olan miqdarda verildiyini yoxlamır. Əgər arqument miqdarı tələb olunandan çoxdursa, onda, artıq olan qiymətlərə baxılmır. Əgər arqument miqdarı tələb olunandan azdırsa, onda göstərilməyən arqumentlərə *undefined* qiyməti mənimsədilir. Funksiyalar, elə yazıla bilər ki, arqument çatışmazlığına önəm verməsin. Yaxud, elə yazıla bilər ki, arqumentlərin təyin olan miqdarda olmasını tələb etsin. Biz fəsilin davamında arqumentlərin miqdarı yoxlamaq və arqumentləri girişi onların adları ilə, sıra nömrələri ilə təşkil etmək üsulu ilə tanış olacağıq.

Nəzərə alın ki, nümunə 8.1-dəki print() funksiyasında (*return* təlimatı yoxdur, buna görə də bu funksiya həmişə *undefined* qiymətini qaytarır və sözügedən funksiyanı mürəkkəb ifadənin bir hissəsi kimi istifadə etməyin mənası yoxdur. Amma distance() və factorial() funksiyaları əvvəlki nümunələrdən göstərildiyi kimi mürəkkəb ifadələrdə istifadə edilə bilər.

8.1.1. Daxili funksiyalar

JavaScript-də funksiyalar başqa funksiyaların daxilində təyin edilə bilər. Məsələn:

```
function hypotenuse(a, b)
{
  function square (x)
  {
     return x*x;
  }
  return Math.sqrt(square (a) + square (b));
}
```

Daxili funksiyalar yalnız yuxarı səviyyəli funksiyaların kodunda təyin edilə bilər. Bu isə o deməkdir ki, *dövrlərin və ya şərt ifadələrinin daxilində funksiya təyin etmək olmaz.*[20] Nəzərə alın ki, bu məhdudiyyətlər yalnız *function* təlimatının köməyilə yaradılan funksiyalara aiddir. Funksional literallar (növbəti bölmədə təsvir ediləcək) istənilən ifadənin daxilində ola bilər.

8.1.2. Funksional literallar

JavaScript funksiyaları funksional literallar şəklində müəyyən etməyə imkan verir. 3-cü fəsildə deyildiyi kimi, funksional literal – adlandırılmayan funksiyanı müəyyən edən ifadədir. Funksional literalının sintaksisi bir çox məqamda *function* təlimatının sintaksisinə bənzəsə də, bu təlimat deyil, literaldır və funksional literalda funksiya adı tələb edilmir

Aşağıdakı iki kod sətirində *function* təlimatının və funksional literalın köməyilə az-çox bir-birinə bənzəyən funksiyalar müyyən edilmişdir:

```
function f(x){ return x*x;}                    // function
təlimatı
function var f = function(x){ return x*x; };    // Funksional
literal
```

Funksional literallar adlandırılmayan funksiyaları yaradır, amma sintaksis funksiya adının göstərilməsinə də şərait yaradır. Bu özünə istinad edən rekursiv funksiyalarının yazılışı zamanı lazım ola bilər. Məsələn:

```
var f = function fact(x){ if (x <= 1) return 1; else return
x*fact (x1); };
```

kodun bu sətiri adlandırılmayan funksiyanı müəyyən edir və onu *f* dəyişənində saxlayır. Funksional literal əslində fact adlı funksiyanı yaratmır, sadəcə bu adın köməyilə funksiyanın gövdəsində öz-özünə istinad etməsinə şərait yaradır. Qeyd edək ki, JavaScript 1.5 versiyasına qədər olan versiyalarda adlandırılmış funksional literallar tam düzgün işləmir.
Funksional literallar təlimatlarla deyil, JavaScript-ifadələri ilə yaradılır və buna görə də bu literallardan daha ustalıqla istifadə oluna bilərlər. Bu literallar xüsusilə yalnız bir dəfə çağrılan və adlandırmaya ehtiyac olmayan funksiyalarda istifadə edilir. Məsələn, funksional literal ifadəsinin köməyilə müəyyən edilmiş funksiya, dəyişəndə saxlanmış, başqa funksiyaya ötürülmüş və ya hətta bilavasitə çağırılmış ola bilər:

```
f[0] = function (x){ return x*x;};        // Funksiyanın
müəyyən edilməsi və
                                          // dəyişəndə saxlanılması
a.sort(function (a, b){ return a b;});       // Funksiyanı
müəyyən edilməsi və
                                          // digər funksiyaya
ötürülməsi
var tensquared = (function (x){ return x*x;}) (10);         //
Funksiyanın müəyyən
                                                    // edilməsi və
                                                    // çağrılması
```

8.1.3. Funksiyaların adlandırılması

Funksiyanın adlandırılması zamanı istənilən mümkün JavaScript identifikatorundan istifadə oluna bilər. Funksiyalara kifayət qədər başa düşülən və qısa ad seçmək lazımdır. Qısalıq və informativlik arasında balansın saxlanılması qabiliyyəti təcrübə nəticəsində yaranır. Düzgün seçilmiş funksiya adları kodun asan oxunmasını və başa düşülən olmasını əhəmiyyətli dərəcədə yüksəldə bilər (deməli, və müşayiətin sadəliyini).

Əksər hallarda funksiya adları kimi feillər və ya feillərdən düzələn birləşmələrdən istifadə edilir. Hamı tərəfindən qəbul edilmiş razılaşmaya əsasən funksiya adları kiçik hərflə başlayır. Əgər ad bir neçə sözdən ibarətdirsə, razılaşmalardan birinə uyğun olaraq, bu sözlər bir-birindən sətir xətti simvolu ilə ayrılır, məsələn: `like_this()`, digər razılaşmaya əsasən, birinci sözdən başqa, yerdə qalan bütün sözlər böyük hərflərlə başlayır və sözlər bitişik yazılır, məsələn: `likeThis()`. Güman edilən funksiyaların adları, hansılar ki, necə güman edilir, yad gözlərdən daxili, gizli funksionallığı reallaşdırırlar, bəzən altından xətt çəkmənin (qeyd etmənin) simvolundan başlanırlar.

Bəzi proqramlaşdırmanın stillərində və ya aydın müəyyən edilmiş proqram platformalarında sıx istifadə edilən funksiyalara çox qısa adlar verilir. Nümunə olaraq JavaScript-in kliyent dilinin `Prototype` platformasını (***http://prototype.conio.net***) göstərmək olar ki, burada `document.getElementById()` adını sadəcə `$()` adlı funksiya əvəzləyib və bu çox rahatdır. (2-ci fəsildə deyildiyi kimi, JavaScript

identifikatorlarında dollar və sətir xətti simvollarından istifadə etmək olar.)

8.2. Funksiyaların arqumentləri

JavaScript-də funksiyalar təyin edilib-edilməməsindən asılı olmayaraq istənilən qədər arqumentlə çağırıla bilər. Bir halda ki, funksiyalar qismən tipləşdirilmişdir, giriş arqumentlərinin tiplərini göstərmək imkanı yoxdur, bununla əlaqədar olaraq, istənilən tipdə olan qiymətləri istənilən funksiyaya ötürmək mümkün hesab edilir. Bütün bu məsələlər növbəti yarımfəsillərdə müzakirə edilir.

8.2.1. Vacib olmayan arqumentlər

Funksiyada göstərilən arqumentlərin miqdarından daha az arqument təyin edilərsə, çatışmayan arqumentlər *undefined* qiymətini alır. Bəzi hallarda bəzi arqumentləri qeyri-icbari təyin etmək lazım olur ki, funksiya çağırışı zamanı bu arqumentlər ixtisara salınsın. Belə hallarda arqumentə susmaya görə qiymətlər mənimsətmək çox səmərəli üsuldur (və ya *null* qiymətiylə verilmişdir). Məsələn:

```
// a massivinə o obyektinin sadalanan xüsusiyyətlər adları
əlavə edilərək, a
// massivini qaytarmaq. Əgər a massivi göstərilməmişdirsə və
ya null-a
// bərabərdirsə, yeni boş a massivi yaratmaq

function copyPropertyNamesToArray(o, /* vacib olmayan */ a)
{
  if (! a) a = [];   // Əgər massiv müəyyən edilməmişdirsə və
ya null
                     // qiymətilə alınmışdırsa onda, boş a
massivi yaradırıq.
  for (var property in o) a.push(property);
  return a;
}
```

Beləliklə funksiya müəyyən edildikdən sonra, bu funksiyaya müxtəlif müraciət imkanları meydana çıxır:

```
// o və p obyektlərinin xüsusiyyətlərinin adlarını əldə
edilməsi

var a = copyPropertyNamesToArray(o);    // o obyektində massiv
şəklində
                                        // əldə etmək
copyPropertyNamesToArray(p, a)             // massivə p
obyektinin
                                        // xüsusiyyətlərinin əlavə
etmək
```

Bu funksiyanın birinci sətirində *if* təlimatının yerinə [] operatorundan aşağıdakı qaydada istifadə etmək olar:

```
a = a || [];
```

5-ci fəsildə qeyd edildiyi kimi, || operatoru birinci arqument *true*-ya bərabərdisə və *true* məntiqi qiymətinə dəyişdirilə bilirsə, birinci arqumentin qiymətini qaytarır. Əks təqdirdə ikinci arqumentin qiymətini qayıdır. İndiki halda bu operator, əgər a dəyişəni müəyyən edilmişdirsə və *null* qiyməti hətta o hadisəyə özündə saxlamır, əgər *a* – bu boş massivdir. Əks təqdirdə o yeni boş massivi qaytaracaq.
Nəzərə alın ki funksiyaların elanı zamanı vacib olmayan arqumentləri arqument siyahısının sonuna yerləşdirmək lazımdır ki, onları ixtisara salmaq rahat olsun. Çünki sizin funksiyanıza müraciət edən bir başqa proqramçı, birinci arqumenti (hansı ki, vacib olmayan) ixtisara salaraq, ikinci arqumenti verə bilməyəcək. Bu halda o birinci arqumentin qiymətinə açıq-aydın *undefined* və ya *null* qiymətini verməlidir.

8.2.2. Dəyişən uzunluqlu arqument siyahıları: Arguments obyekti

Funksiyanın gövdəsində arguments identifikatoru hər zaman xüsusi mənaya malikdir;

arguments – *Arguments* obyekti kimi tanınan obyektə istinad edən çağırış obyektinin spesifik xüsusiyyətidir.

Arguments obyekti – funksiyalara verilmiş qiymətin adları üzrə, nömrəsi üzrə çıxarışını reallaşdıran massivə oxşar obyektdir (bölmə 7.8 baxmaq). *Arguments* obyekti həmçinin əlavə olaraq callee əlavə xüsusiyyətini müəyyən edir ki, bu xüsusiyyət növbəti bölmədə təsvir edilmişdir.

JavaScript-də adlandırılmış funksiya, təsbit edilmiş arqument sayı ilə təyin edilməsinə baxmayaraq, çağırış zamanı ona istənilən qədər arqument ötürülə bilər. *Arguments* obyekti arqumentlərin (hətta adı olmayan arqumentlərə) qiymətlərinə tam girişi təmin edir. Fərz edək, bir x arqumentini tələb edən *f* funksiyası müəyyən edilmişdi. Əgər bu funksiyanı iki arqumentlə çağırsaq, onda birinci arqument funksiyada x adında və ya *arguments[0]* parametri kimi əlçatan olacaq. İkinci arqument isə funksiyada yalnız *arguments[1]* kimi əlçatan olacaq. Bundan başqa massivlərdə olduğu kimi, *arguments*-də də elementlərin miqdarını göstərən *length* xüsusiyyəti mövcuddur. Yəni iki arqumentlə çağırılan *f* funksiyasının gövdəsində *arguments.length* xüsusiyyəti 2 qiyməti ehtiva edir.

Arguments obyekti müxtəlif istiqamətlərdə istifadə oluna bilər. Aşağıdakı nümunədə, funksiya çağrılan arqumentlərin tələb olunan miqdarının necə yoxlanıldığı göstərilmişdir. Axı, JavaScript bunu sizin üçün etməyəcək:

```
function f (x, y, z)
{
// Əvvəlcə, verilən arqument miqdarının düzgünlüyü yoxlanılır

   if(arguments.length! = 3)
   {
     throw new Error("f funksiyası "+ arguments.length +
"arqumentlə
                    çağrılmış, lakin funksiyanın icrası üçün
                    3 arqument      tələb olunur.");
   }
   // Funksiyanın kodu...
}
```

Arguments obyektində JavaScript-funksiyalarının əhəmiyyətli imkanları yer alır: funksiyalar elə yazıla bilər ki istənilən arqument sayı ilə işləsin. Elə aşağıda göstərilən nümunədə *max()* adlı funksiya istənilən miqdarda qəbul edərək, bu arqument içərisində ən böyük arqumenti qaytarır (analoji əməliyyatı *Math.max()* inteqrasiya edilmiş funksiyası da edə bilər):

```
function max(/*.. .*/)
{
  var m = Number.NEGATIVE_INFINITY; // Bütün arqumentlərin
üzrə dövr,
                                    // axtarış və arqumentlərdən
                                 ən
                                    // böyüyünün m dəyişənində
                                 saxlanılması
  for (var i = 0; i < arguments.length; i++)
      if (arguments[i] > m) m = arguments[i];
  return m;                                  //Ən böyük
arqumenti qaytarırıq
}

var largest = max (1, 10, 100, 2, 3, 1000, 4, 5, 10000, 6);
```

Buna oxşar və bu kimi istənilən miqdarda arqument qəbul edən funksiyalara ***dəyişən miqdarlı arqument funksiyaları*** (*variadic functions*, *variable arity functions* və ya *varargs functions*) deyilir. Bu termin C proqramlaşdırma dilinin meydana çıxması ilə birlikdə yaranmışdır. Nəzərə alın ki, dəyişən miqdarlı arqument funksiyalarını arqumentsiz çağırmaq olmaz. Funksiya yazılışı arguments[] obyektindən istifadə edəcək icbari və vacib olmayan arqumentlərin miqdarını təsbit etmək tamamilə düzgün və səmərəli addımdır.

Unutmayın ki, arguments faktiki olaraq massiv deyil – bu sadəcə *Arguments* obyektidir. *Arguments*-in hər bir obyektində nömrələnmiş massiv elementləri və length xüsusiyyəti olmasına baxmayaraq, texniki nöqteyi-nəzərdən massiv deyil. *Arguments*-ə bir neçə nömrələnmiş xüsusiyyətlərə malik obyekt kimi baxmaq daha yaxşıdır. ECMAScript spesifikasiyası hər hansı reallaşdırması Arguments obyektini massiv kimi dəstəkləmir. Hərçənd, məsələn, *arguments.length* xüsusiyyətinə qiymət mənimsətməyə icazə verir, amma ECMAScript bunu obyektdə müəyyən edilmiş massiv

elementlərinin sayının real dəyişikliyi üçün dəstəkləmir. (əsl *Array* obyektləri üçün *length* xüsusiyyətinin spesifik davranışı ***bölmə 7.6.3-də*** təsvir edilir.)

Arguments obyektində bir çox qeyri-adi xüsusiyyət vardır. Funksiyada adlandırılmış arqumentlər olan zaman, Arguments obyektinin massiv elementləri, funksiyanın arqumentlərini ehtiva edən lokal dəyişənlərin sinonimidir. Arguments massivi *[]* və adlandırılmış arqumentlər – bir dəyişənə müraciətin iki müxtəlif formasıdır. Arqumentin adı vasitəsilə arqumentin qiymətinin dəyişikliyi *arguments[]* massivi vasitəsilə çıxardılan qiyməti dəyişdirir. *arguments[]* massivi vasitəsilə arqumentin qiymətinin dəyişikliyi arqument adında çıxardılan qiyməti dəyişdirir. Məsələn:

```
function f(x){
  print(x);             // Arqumentin ilk qiymətini
ekranlaşdırılır
  arguments[0] = null;       // Massivin elementlərini
dəyişdirərkən, biz
                   // həmçinin x-də dəyişdiririk
  print(x);                 // İndi isə "null" ekranlaşdırılır
}
```

Şübhəsiz ki, bu əsl massivə xas davranış deyil. Bu halda *arguments[0]* və *x* eyni qiymətə istinad edə bilərdi, amma bir istinadın dəyişikliyi o birisinə təsir göstərməməlidir.

Nəhayət, nəzərə almaq lazımdır ki, arguments yalnızca adi JavaScript-identifikatorudur və bu ehtiyatda saxlanılan söz sayılmır. Əgər funksiyada eyni adlı arqumenti və ya lokal dəyiəni müəyyən etsək, onda Arguments obyekti əlçatmaz olacaq. Bu səbəbdən arguments sözünü ehtiyata saxlanmış söz hesab etmək və bu adlı dəyişənlər yaratmaqdan çəkinmək lazımdır.

8.2.2.1. callee xüsusiyyəti

Arguments obyekti öz massiv elementlərindən başqa hal-hazırda icra edilən funksiyaya istinad edən *callee* xüsusiyyətini ehtiva edir. Bu xüsusiyyətdən adlandırılmayan funksiyaların rekursiv çağırışı üçün istifadə etmək olar. Aşağıda fatorialı hesablayan adlandırılmayan funksional literalının nümunəsi göstərilmişdir:

```
function (x){
  if (x <= 1)
  return 1;
  return x * arguments.callee (x 1);
}
```

8.2.3. Obyekt xüsusiyyətlərindən arqumentlər kimi istifadə

Funksiya üçdən çox arqument ifadə edən zaman, bu arqumentlərini düzgün sırasını yadda saxlamaq çətin olur. Bu səbəbdən yaranan xətaların qarşısını almaq üçün arqumentlər funksiyaya *"ad - qiymət"* cütlükləri şəklində ötürülə bilər. Bu cür imkanı reallaşdırmaq üçün, funksiyanın təyini zamanı tək arqument kimi obyektin ötürülməsini tətbiq etmək lazımdır. Bu stil sayəsində, istifadəçilər funksiyanı obyekt literalı kimi tətbiq edə biləcək ki, bu formada lazımlı *"ad-qiymət"* cütlükləri təyin ediləcək. Aşağıdakı fraqmentdə belə bir funksiyanın nümunəsi göstərilmiş. Nümunədə həmçinin arqumentlər üçün susmaya görə qiymətlər təyin etmək qaydası da göstərilir:

```
// Elementlərin length qiymətini from massivindən to massivinə
kopyalanması.
// Kopyalanma from massivindəki from_start elementindən
başlanır
// to massivindəki to_start elementin başlayaraq və
elementlərə tətbiqi
// edilir. Bu cür funksiyanın arqument ardıcıllığını yadda
saxlamaq kifayət
// qədər çətindir.

function arraycopy(/* array */ from, /* index
*/ from_start,
                   /* array */ to, /* index */ to_start,
                   /* integer */ length)
{
   // burada funksiyanın realizasiyası baş verir
}
// Bu funksiya isə, yuxarıdakına nisbətən az effektiv olsa da,
// arqument ardıcıllığını yadda saxlamağı tələb etmir,
from_start
```

```
// və to_start arqumentləri isə susmaya görə 0 qiymətini qəbul
edir.

function easycopy(args)
{
  arraycopy(args.from, args.from_start || 0,        //
Diqqətli olun, məhz burada
                                        // susmaya görə qiymətlər
                                        // təyin olunur
  args.to, args.to_start || 0,
  args.length);
}

// Aşağıda isə easycopy() funksiyanın çağırış nümunəsi
göstərilir:
var a = [1,2,3,4];
var b = new Array(4);
easycopy({from: a, to: b, length: 4});
```

8.2.4. Arqumentlərin tipləri

JavaScript qismən tipləşdirilmiş dil olduğuna görə, arqumentlər tiplərinin göstərişi olmadan elan edilir və arqumentlərin funksiya qiymətlərinə ötürülməsi zamanı onların hansı tipdə olduğu yoxlanılmır. Siz öz proqramınızda nümunə 8.3-dəki `arraycopy()` funksiyasında edildiyi kimi funksiya arqumentləri üçün təsvir adları seçərək, şərhdə müvafiq göstəriş daxil edə bilərsiniz. Vacib olmayan arqumentlər üçün şərhə *"vacib olmayan" ("optional")* sözünü əlavə etmək olar. Amma əgər metod arqumentlərin ixtiyari miqdarını qəbul etmək imkanı nəzərdə tutursa, nöqtələrdən istifadə etmək olar:

```
function max(/* miqdar...*/){ /* funksiyanın gövdəsi */ }
```

3-cü fəsildə qeyd edildiyi, ehtiyac olduğu halda JavaScript ***tiplərin dəyişikliyini*** yerinə yetirir. Beləliklə, əgər siz sətir arqumentlərini qəbul etməyə yönəlmiş funksiyanı yaratsanız, sonra da bu funksiyaya sətir olmayan hər hansı tipdə arqument ötürsəniz, arqumentin qiyməti sətirə dəyişdiriləcək və funksiya bu arqumentə sətir müraciət edəcək. İstənilən elementar tip sətirə dəyişdirilə bilər. Obyektlər üçün isə, `toString()` (real olaraq, həmişə faydalı olmayan) metodları mövcuddar. buna görə da xəta riski aşağıdır.

Ancaq belə yanaşma həmişə yararlı olmaya bilər. Yenidən əvvəlki nümunədə nümayiş etdirilmiş *arraycopy()* metoduna baxaq. Bu metod birinci arqumentin massiv olmasını gözləyir. Əgər funksiyaya müraciət edərkən birinci arqument massiv (və ya massivə oxşar obyekt) olmazsa, funksiyaya müraciət uğursuzluqla nəticələnir. Əgər funksiya bir neçə dəfə çağırılacaqsa, onda bu funksiyaya arqument tiplərinin uyğunluğunun yoxlanılmasını əlavə etmək lazımdır. Xətalı arqument tiplərinin ötürülməsi zamanı istisna yaradılmalıdır, ki, bu xəta barədə məlumat versin. Xətalı arqument tiplərinin ötürülməsi zamanı funksiyanın icrasının müvəffəqiyyətsizliyə uğrayacağını nəzərə alaraq, funksiya çağırışını dərhal kəsmək lazımdır. Məsələn, aşağıdakı fraqmentdə funksiya ədəd arqumentinin köməyilə massiv elementinə giriş əldə etməyə çalışır:

```
// a, massivin (və ya massivə oxşar obyektin) elementlərinin
cəmini
// qaytarır. massivin Bütün elementləri ədəd olmalıdır, bu
halda null və
// undefined qiymətlərinə icazə verilmir.

function sum(a){
 if ((a instanceof Array) ||                          // əgər
bu massivdirsə
 (a && typeof a == "object" && "length" in a))         // və
ya massivə oxşar
                                                  // obyektirsə
 {
   var total = 0;
   for (var i = 0; i < a.length; i++)
   {
     var element = a[i];
     if (!element)
     continue;             // null və undefined  qiymətlərinə
icazə verməmək
     if (typeof element == "number")
      total += element;
     else
      throw new Error("sum(): Massivin bütün elementləri ədəd
     olmalıdır");
   }
   return total;
 }
 else
   throw new Error("sum(): Arqument massiv olmalıdır");
}
```

sum() metodu giriş arqumentlərinin tiplərinin yoxlamasına olduqca ciddi nəzarət edir və tələblərə uyğun olmayan arqument daxil edildiyi halda, kifayət qədər informativ bildirişlərə malik istisnalar meydana çıxır. Bununla belə metod əsl massivlərlə yanaşı, massivə oxşar obyektlər ilə işləyə bildiyinə, və *null*, *undefined* qiymətlərinə imkan vermədiyinə görə öz elastikliyini saxlayır.

JavaScript – olduqca elastik və qismən tipləşdirilmiş dil olduğuna görə, bu dildəki funksiyalar istənilən miqdarda və tipdə arqument daxil edilməsinə dözümlüdür. Bu yanaşma aşağıdakı flexsum() metodunda öz əksini tapır. Məsələn, bu metod istənilən miqdarda giriş arqumenti qəbul edir və massivləri özləri ilə rekursiv olaraq emal edir. Bunun nəticəsində metod, massivi dəyişən miqdarlı arqument və ya arqument şəklində qəbul edə bilər. Bundan başqa, metod istisna yaramazdan ədəd olmayan arqumentləri maksimum səylə ədədə çevirməyə çalışır:

```
function flexisum(a) {
 var total = 0;
 for(var i = 0; i < arguments.length; i++)
 {
   var element = arguments[i];
       if (!element)
               continue; // null və undefined  qiymətlərinə
icazə verməmək

   // Arqumentin tipinə uyğun olaraq onu ədəd tipinə
dəyişdirmək
       var n;
       switch(typeof element) {
           case "number":
               n = element;       // Dəyişikliyə lüzum yoxdur
               break;
           case "object":
               if (element instanceof Array) // Massivin
rekursiv dövrü
                 n = flexisum.apply(this, element);
               else
                 n = element.valueOf( );       //Digər
obyektlər üçün
                                             //valueOf metodunu
çağırmaq
               break;
           case "function":
```

```
                n = element( ); // Funksiyanı çağırmağa
çalışmaq
                break;
            case "string":
                n = parseFloat(element); // Sətiri dəyişdirməyə
çalışmaq
                break;
            case "boolean":
                n = NaN; // Məntiqi qiyməti ədədə dəyişdirmək
olmur!
               break;
   }

   // Əgər normal ədəd əldə edə bilsək, həmin ədədi cəmə
əlavə edilir.
   if (typeof n == "number" && !isNaN(n))
        total += n;
   // Əks təqdirdə istisna yaradılır
   else
        throw new Error("sum(): " + element + " qiymətinin
     ədədə
                        dəyişdirilməsi xətası");

 }
 return total;
}
```

8.3. Məlumat qismində funksiyalar

Əvvəlki bölmədə göstərildiyi kimi, funksiyaların təyin edilməsi və çağrılması ən əhəmiyyətli xüsusiyyətləridir. Funksiyaların təyini çağırışı – JavaScript və digər proqramlaşdırma dillərinin əksəriyyətinin sintaktis vasitələridir. Ancaq JavaScript-də funksiya – yalnız sintaktis konstruksiya sayılmır, o həm də məlumat ehtiva edir və bu onu bildirir ki, onlar dəyişənlərə mənimsədilə, obyekt xüsusiyyətlərində və ya massiv elementlərində saxlanıla, arqumentlər kimi digər funksiyalara ötürülə və s. əməliyyatlar edilə bilər.
JavaScript-də funksiyaların sintaktis konstruksiyalar və məlumatlarla eyni zamanda necə işlədiyini aşağıdakı funksiya təyinində göstərilir:

```
function square(x){ return x*x; }
```

Bu təyin funksiyanın yeni obyektini yaradır və ona square dəyişənini mənimsəyir. Funksiyanın adı əslində qeyri-maddi sayılır və bu sadəcə funksiyanın adını ehtiva edən dəyişəndir. Funksiya başqa dəyişənə də mənimsədilə bilər:

```
var a = square(4);            // a 16 qiymətini ehtiva edir
b= square;                  // b dəyişəni square funksiyasına
istinad yaradır
var c = b(5);              // c 25 qiymətini ehtiva edir
```

Funksiya qlobal dəyişənlərdən əlavə, həm də obyekt xüsusiyyətlərinə mənimsədilə bilər. Bu halda funksiyanı *metod* adlandırırlar:

```
var o = new Object;
o.square = Function(x){ return x*x;};      // funsional
literal
y = o.square(16);                          // y 256-a
bərabərdir
```

Funksiyalar, həm də massivin elementlərinə mənimsədilə bilər:

```
var a = new Array(3);
a[0] = function (x){ return x*x;}
a[1] = 20;
a[2] = a[0](a[1]);                      // a[2] 400 qiymətini
ehtiva edir
```

Sonuncu nümunədəki funksiya çağırışının sintaksisi sizə qeyri-adi görünə bilər, lakin JavaScript-də bu növ çağırışlar `()` operatorunun köməyilə mümkündür! ***Nümunə 8.2-də*** funksiyaların məlumatlar qismində necə çıxış etdiyi təfərrüatı ilə göstərilmişdir. Nümunədə funksiyaların digər funksiyalara necə ötürüldüyü göstərilir. Hərçənd ki, nümunə sizə bir qədər mürəkkəb görünə bilər, amma kodu şərhlərlə tanış olaraq nəzərdən keçirsəniz prosesi tamamilə anlayacaqsınız.

Nümunə 8.2. *Funksiyalardan məlumatlar kimi istifadə*

```
function add(x, y){ return x + y;}
function subtract(x, y){ return x y;}
function multiply(x, y){ return x * y;}
function divide(x, y){ return x / y;}
```

```
// Bu funksiya yuxarıda göstərilən funksiyalardan birini qəbul edir
// və onu iki operandlara çağırırır

function operate (operator, operand1, operand2)
{
  return operator(operand1, operand2);
}

// Beləlikdə biz (2+3)+(4*5) ifadəsinin qiymətinin hesablanması üçün bu
// funksiyanı aşağıdakı kimi çağıra bilərik:
var i = operate(add, operate (add, 2, 3), operate (multiply, 4, 5));

// Bu nümunədən istifadə edərək, biz bu funksiyanı obyekt literalının
// daxilində funksional literallar vasitəsilə yenidən realizə edirik.

var operators ={
    add: function (x, y){ return x+y;},
    subtract: function (x, y){ return x y;},
    multiply: function (x, y){ return x*y;},
    divide: function (x, y){ return x/y;},
    pow: Math.pow       // Burada hətta qabaqcadan müəyyən edilmiş funksiyalar
                        // da iştirak edə bilər
};

// Bu funksiya operatorun adını qəbul edir, obyektdə həmin operatoru axtarır
// və sonra da tapılmış operatoru ona verilmiş operandlar üzərində tətbiq
// edir. Funksiyanın operatorunun çağırış sintaksisinə diqqət yetirin.

function operate2(op_name, operand1, operand2){
   if (typeof operators [op_name] == "function")
       return operators[op_name] (operand1, operand2);
   else
       throw "Naməlum operator";
}

// Buda belə, indi biz qiymətin hesablanması üçün bu funksiyanı çağıra
// bilərik.
// (" hello" + " " + "world"):
```

```
var j = operate2("add", "hello", operate2 (" add", " ",
"world"))

//  Qabaqcadan müəyyən edilmiş Math.pow() funksiyasından
istifadə edirik:
var k = operate2("pow", 10, 2);
```

Əgər əvvəlki nümunədə funksiyaların arqumentlər qismində necə rahat ötürüldüyünə və digər funksiyalarda bu üsuldan sıx istifadə edildiyinə şübhəniz varsa, `Array.sort()` funksiyasına nəzər yetirin. Bu funksiya massivin elementlərini çeşidləyir. Bir çox çeşidləmə növləri (ədəd, əlifba, tarix, artan, azalan və s. üzrə) mövcuddur, buna görə də `sort()` funksiyası vacib olmayan arqument kimi digər funksiya qəbul edir ki, bu funksiyada hansı qaydada çeşidləmə yerinə yetiriləcəyi göstərilir. Bu cür funksiyaların iş prinsipi çox sadədir. Belə funksiyalar adətən massivin iki elementi qəbul edərək, onları müqayisə edir və sonra da alınan nəticəyə uyğun olaraq elementlərdən hansının birinci olacağı müəyyən edilərək, qaytarılır. Bu funksiya vasitəsilə qaytarılan arqument sayəsində Array.sort() metodu çox universal və elastik hala gəlir ki, bu imkandan istifadə edərək istənilən məlumat tipini düşünülə bilən istənilən sırada çeşidləmək olar! (`Array.sort()` funksiyasından istifadə nümunəsi ***bölmə 7.7.3-də*** mövcuddur.)

8.4. Metodlar qismində funksiyalar

Metodlar – digər funksiyalar fərqli olaraq, obyektin xüsusiyyətində saxlanılır və bu obyekt vasitəsilə çağırılır. Unutmayın ki, funksiyalar – müəyyən edilmiş adlarda saxlanılan məlumatların qiymətləridir və funksiyaların davranışında qeyri-adi heç nə yoxdur. Buna görə də funksiyalar istənilən dəyişənə mənimsədildiyi kimi, obyekt xüsusiyyətlərinə mənimsədilə bilər. Məsələn, əgər `f` funksiyası və `o` obyekti varsa, `m` adlı metodu bu cür müəyyən etmək tamamilə doğrudur:

```
o.m = f;
```

`o` obyektində `m()` metodunu müəyyən etdikdən sonra, bu metoda aşağıdakı qaydada müraciət etmək olar:

```
o.m();
```

Və ya, əgər *m()* metodu iki arqument qəbul etmək istəyirsə:

```
o.m (x, x+2);
```

Metodlar bir çox əhəmiyyətli xüsusiyyətə malikdir: metodun çağrıldığı obyektin qiymətini, metodun gövdəsində *this* açar sözünün köməyilə əldə etmək mümkündür. Yəni, *o.m()* metodu çağırılan zaman, metodun gövdəsində *this* açar sözünün köməyilə o obyektinə giriş əldə etmək olar. Aşağıdakı nümunədə bunun şahidi ola bilərsiniz:

```
var calculator = {              // Obyekt literalı
    operand1: 1,
    operand2: 1,
    compute: function (){
        this.result = this.operand1 + this.operand2;
    }
};
calculator.compute(); // 1+1 neçə edir?
print(calculator.result); // Nəticə
```

this açar sözü çox əhəmiyyətli rolu oynayır. Metod kimi çağırılan istənilən funksiya çağrıldığı obyekti naməlum arqument kimi öz sərəncamına alır. Bir qayda olaraq, metodlar bəzi əməliyyatları bu obyektin üzərində yerinə yetirir, beləliklə, metodların çağırış sintaksisi onu deməyə əsaslanır ki, funksiya obyektlə əməliyyat edir. Aşağıdakı proqramın ikisətirini müqayisə edin:

```
rect.setSize(width, height);
setRectSize(rect, width, height);
```

Bu iki sətirdə çağırılan hipotetik funksiyalar tamamilə eyni əməliyyatları *rect* obyekti (hipotetik) üzərində edir, amma birinci sətirdəki metodun çağırış sintaksisi *rect* obyektinin diqqət mərkəzində olduğunu daha əyani göstərir. (Əgər birinci sətir sizə daha qeyri-adi gəldisə, deməli hələki sizdə obyekt yönümlü proqramlaşdırma təcrübəsi yoxdur)
Funksiya metod kimi deyil, funksiya kimi çağırılarkən, *this* açar sözü qlobal obyektə istinad edir. Ən qəribəsi odur ki, bu yanaşma hətta mənbəyi metod olan və metodların daxildə qurulan funksiyalara (əgər

onlar funksiya kimi çağırılırlarsa) da şamil edilir. `this` açar sözü funksiyada təkmil bir qiymətə malikdir və qoyulduğu funksiyanın gövdəsində qlobal obyektə istinad edir (görünüş cəhətdən hiss olunmasa da). Yadınızda saxlayın ki, `this` – dəyişən və ya xüsusiyyət adı deyil məhz açar sözü sayılır. JavaScript sintaksisi `this` element qiymət mənimsədilməsinə icazə vermir.

8.5. Konstruktor-funksiyası

Konstruktor – obyekt xüsusiyyətlərinin inisializasiyasını yerinə yetirən və `new` təlimatı ilə birgə istifadə edilməsi nəzərdə tutulan funksiyadır. Konstruktorların ətraflı təsviri 9-cu fəsildə yer alıb. Ancaq qısa olaraq onu demək olar ki, `new` təlimatı yeni `Function` obyekti yaradır, sonra da funksiya-konstuktoruna yenidən yaradılmış obyekti `this` açar sözünün qiyməti kimi verərək çağrılmasına səbəb olur.

8.6. Funksiyaların xüsusiyyətləri və metodları

Biz artıq, JavaScript-də, funksiyaların qiymətlər qismində necə istifadə olunduğunun şahidi olduq. `typeof` təlimatı funksiyalar üçün "`Function`" sətirini qaytarmasına baxmayaraq, JavaScript-dəki funksiyalar əslində xüsusi növ obyektdir. Və belə olduqda, funksiyalar digər obyektlər kimi xüsusiyyətlərə və metodlara malik olur.

8.6.1. length xüsusiyyəti

Əvvəlki bölmələrdə qeyd edildiyi kimi, funksiyanın gövdəsində arguments massivinin length xüsusiyyəti bu funksiyaya verilmiş arqumentlərin miqdarını müəyyən edir. Ancaq funksiyanın özünün length xüsusiyyəti başqa mənaya malikdir. Funksiyanın length xüsusiyyəti funksiyaya verilməli olan və yalnız oxuma üçün əlçatan olan

arqumentlərin (funksiya müəyyən edilərkən siyahıda təyin edilən parametrlər) miqdarını qaytarır. Xatırladaq ki, funksiya istənilən miqdarda arqumentlə çağırıla bilər və bu arqumentləri miqdarından asılı olmayaraq arguments massivi vasitəsilə çıxardıla bilər. *Function* obyektinin length xüsusiyyəti də bu cür işləyir, yəni funksiyada miqdarından asılı olmayaraq mövcud olan parametrlər çıxardıla bilər. Nəzərə alın ki, arguments.length xüsusiyyətindən fərqli olaraq, funksiyanın length xüsusiyyəti həm funksiya daxilində, həm də funksiyadan kənar əlçatandır.
Aşağıdakı fraqmentdə başqa funksiyadan arqumentlər massivini əldə edən check() adlı funksiya təyin edilir. Bu funksiya *Function.length* (*arguments.callee.length* kimi də əlçatan) xüsusiyyətiylə arguments.length xüsusiyyətini müqayisə edir və funksiyaya gözlənilən miqdarda arqumentin verilib-verilmədiyini yoxlayır. Əgər arqument sayı gözlənilən miqdarda deyilsə, istisna yaradılır. check() funksiyasını bu funksiyanın çağırışının sırasını göstərən *f*() test funksiyası müşaiyət edir:

```
function check(args) {
var actual = args.length;      // Faktiki ötürülən arqument miqdarı
var expected = args.callee.length; // Gözlənilən ötürülən arqument
miqdarı
if (actual != expected) {      // Əgər gözlənilən arqument miqdarı
// olmasa, istisna yaradırıq
throw new Error("düzgün olmayan arqument miqdarı: gözlənilir: " +
expected + "; faktiki ötürülüb " + actual);
}
}
function f(x, y, z) {
// Faktiki ötürülən arqument miqdarının gözlənilən arqument
miqdarına
// uyğunluğunu yoxlayırıq
// Əgər uyğun deyilsə, istisna yaradılır
check(arguments);
// İndi isə, funksiyanın qalan hissəsini adi qaydada yerinə
yetiririk

return x + y + z;
```

```
}
```

8.6.2. prototype xüsusiyyəti

İstənilən funksiya qabaqcadan müəyyən edilmiş obyekt-prototipinə istinad edən *prototype* xüsusiyyətinə malikdir. Prosesə daxil olan bu obyekt, funksiya new operatoru vasitəsilə konstruktor kimi istifadə olunan zaman, obyektlərin yeni tiplərinin təyini prosesində mühüm rol oynayır. Biz 9-cu fəsildə *prototype* xüsusiyyətini təfərrüatı ilə öyrənəcəyik.

8.6.3. Funksiyaların şəxsi xüsusiyyətlərinin təyini

Funksiya çağırışları arasında dəyişənin qiymətinin saxlanılması tələb olunduqda, ad fəzasında qlobal dəyişənlərin təyininə icazə verməyən *Function* obyektinin xüsusiyyətlərindən daha çox istifadə edilir. Fərz edək ki, hər bir çağırışda unikal identifikator qaytaran funksiya yazmaq lazımdır və funksiya heç vaxt eyni qiyməti iki dəfə qaytarmamalıdır. Bunun üçün, funksiya son qaytarılmış qiyməti yadda saxlayır və bu informasiya onun çağırışları arasında qüvvədə qalır. Hərçənd ki, göstərilən informasiya qlobal dəyişəndə də saxlanıla bilər, amma buna heç bir lüzum yoxdur və bu informasiyanın *Function* obyektinin xüsusiyyətində saxlanılması daha yaxşıdır, çünki saxlanılan informasiya yalnız funksiyanın özündə istifadə olunur. Aşağıda hər bir çağırışında unikal tam ədəd funksiya nümunəsi göstərilmişdir:

```
// "Statik" dəyişəni yaradırıq və inisializasiya edirik.
// Funksiyaların elanı kodun icrasına qədər emal edilir,
// buna görə də biz funksiyanın elanına qədər bu mənimsəməni
yerinə yetirə
// bilərik

uniqueInteger.counter  = 0;

// Funksiyanın özü. Bu funksiya hər kəs çağırışında müxtəlif
qiymətlər
```

```
// qaytarır və son qaytarılmış qiymətin müəyyən edilməsi üçün
şəxsi "statik"
// xüsusiyyətindən istifadə edir.

function uniqueInteger(){
    return uniqueInteger.counter++; // "statik" dəyişəninin
qiymətini
                                    // artırırıq və alınan qiyməti
qaytarırıq
}
```

8.6.4. apply və call() metodları

ECMAScript-də bütün funksiyalar üçün müəyyən edilmiş iki metod mövcuddur. Bu metodlar *call()* və *apply()* metodlarıdır və funksiyanın hər hansı obyekt metodu kimi çağrılmasına şərait yaradır. *call()* və *apply()* metodlarının birinci arqumenti – funksiyanın yerinə yetirildiyi obyektdir və bu obyekt funksiyanın daxilində *this* açar sözünün qiyməti olur. *call()* metodun qalan arqumentləri – çağırılan funksiyaya ötürülən qiymətlərdir. Belə ki, *f*() funksiyasına iki rəqəmini ötürmək və bu funksiyanı obyekt metodu kimi onu çağırmaq üçün, aşağıdakı formadan istifadə etmək olar:

```
f.call(o, 1, 2);
```

Bu üsul aşağıdakı sətirlərinə analojidir:

```
o.m = f;
o.m (1,2);
delete o.m;
```

apply() metodu *call()* metoduna bənzəyir, lakin metod vasitəsilə arqumentlər funksiyaya massiv şəklində ötürülür:

```
f.apply(o, [1,2]);
```

Məsələn, ədəd massivindəki ən böyük ədəd tapmaq üçün, massivin elementlərini Math.max() funksiyasına *apply()* metodunun köməyilə ötürmək olar:

```
var biggest = Math.max.apply(null, array_of_numbers);
```

8.7. Funksiyaların praktik nümunələri

Bu bölmədə obyektlərlə və massivlərlə işləyə bilən, praktiki dəyərə malik bir neçə funksiya nümunəsi göstərilmişdir. ***Nümunə 8.3-də*** Obyektlərlə işləyə bilən funksiyaları göstərilmişdir.

Nümunə 8.3. *Obyektlərlə işləyə bilən funksiyalar*

```
// "o"  obyektinin sadalana bilən ad xüsusiyyətini ehtiva edən
massivin
// qaytarılması

function getPropertyNames(/* obyekt */ o)
{
  var r = [];
  for(name in o)
      r.push(name);
  return r;
}

// "from" obyektinin sadalana bilən xüsusiyyətlərinin "to"
obyektinə
// köçürülməsi.
// Əgər "to" arqumenti null qiymətinə bərabərdirsə, onda yeni
obyekt
// yaradılır.
// Funksiya "to" obyektini və ya yeni yaradılmış obyekti
qaytarır.

function copyProperties(/* obyekt */ from, /* vacib olmayan
obyekt */ to)
{
  if (!to)
  to ={};
  for (p in from)
      to[p] = from[p];
  return to;
}

// "from" obyektinin sadalana bilən xüsusiyyətlərinin "to"
obyektinə
// köçürülməsi. Ancaq burada yalnız  "to" obyektində müəyyən
edilməmiş
```

```
// xüsusiyyətlər köçürdülür. Bu bizə o vaxt lazım ola bilərki,
məsələn:
// "from" obyekti "to" obyektinə köçürdüləcək xüsusiyyətlər
arasında susmaya
// görə qiymətlər ehtiva edə bilər və bu qiymətlər "to"
obyektində artıq
// müəyyən edilmiş ola bilər. Eyni qiymətləri təkrar
köçürdülməməsi üçün bu
// funksiyadan istifadə etmək  lazımdır.
function copyUndefinedProperties(/* obyekt */ from, /* obyekt
*/ to)
{
  for (p in from)
  {
    if (!p in to)
       to[p] = from[p];
  }
}
```

Növbəti nümunədə massivlərlə işləyə bilən funksiyalar göstərilmişdir.

Nümunə 8.4. *Massivlərlə iş üçün funksiyalar*

```
// a massivinin hər bir elementinin yoxlama funksiyasına
ötürülməsi.
// Yalnız yoxlama nəticəsi true olan elementləri ehtiva edən
massivi
// qaytarılması

function filterArray (/* massiv */ a, /* yoxlama funksiyası */
predicate)
{
  var results = [];
  var length = a.length;        // əgər yoxlama funksiyası
length
                          // xüsusiyyətini dəyişdirərsə!
  for (var i = 0; i < length; i++)
  {
    var element = a[i];
    if (predicate(element))
       results.push(element);
  }
  return results;
}

// Hər bir elementi "f" funksiyasına ötürülən qiymətlər
massivinin
// qaytarılması
```

```
function mapArray (/* massiv */ a, /* funksiya */ f)
{
  var r = [];               // nəticələr
  var length = a.length; // əgər f funksiyası length
xüsusiyyətini
                    // dəyişdirərsə!
  for (var i = 0; i < length; i++)
      r[i] = f(a[i]);
  return r;
}
```

Və son olaraq, ***nümunə 8.5-də*** funksiyalar funksiyalarla iş üçün nəzərdə tutulmuşdur. Faktiki olaraq burada daxili funksiyalar istifadə edilir və qaytarılır. Daxili funksiya vaxtilə "qapanma" adını almış üsulla qayıdır. Qapanmalar, sizə biraz mürəkkəb və buna görə də bu mövzuya növbəti bölmədə baxılır.

Nümunə 8.5. *Funksiyalarla iş üçün funksiyalar*

```
// Sərbəst funksiya qaytarır, hansı ki, bu funksiya da öz
növbəsində "f"
// funksiyasını "o" obyektinın metodu kimi çağırır.
Funksiyaya metod ötürmək
// zərurəti yarandıqda bu funksiyadan istifadə etmək olar.
// Əgər metod obyektlə bağlı olmasa, assosiasiya itiriləcək və
metod
// verilmiş funksiyaya adi funksiya kimi ötüriləcək.

function bindMethod(/* obyekt */ o, /* funksiya */ f)
{
  return function()
  {
    return f.apply(o, arguments)
  }
}

// Sərbəst funksiyanı qaytarır, hansı ki, bu funksiya da öz
növbəsində
// "f" funksiyasına verilmiş arqumentləri və əlavə olaraq
// qaytarılan funksiyaya verilən arqumentlər ötürür.
// (Bu üsul bəzi hallarda "currying" adlanır.)
function bindArguments(/* funksiya */ f, /* ilk arqumentlər...
*/)
{
  var boundArgs = arguments;
  return function()
```

```
  {

    // Arqumentlər massivinin yaradılması. Massiv əvvəl
müəyyən edilmiş
    // arqumentlərlə başlayacaq və indi  verilmiş
arqumentlərlə bitəcək
    var args= [];
    for (var i = 1; i < boundArgs.length; i++)
    args.push(boundArgs [i]);
    for (var i = 0; i < arguments.length; i++)
    args.push(arguments [i]);
    return f.apply(this, args); // İndi isə funksiyanı yeni
arqument
                                // siyahısıyla çağırırıq
  }
}
```

8.8. Funksiyaların və qapanmanın görünmə sahəsi

4-cü fəsildə qeyd edildiyi kimi, JavaScript-də funksiyanın gövdəsi lokal görünmə sahəsində işləyir. Qapanmalar daxil olmaqla bu bölmədə görünmə sahəsi ilə bağlı məsələlərə baxılır.

8.8.1. Leksik görünmə sahəsi

JavaScript-də Funksiyalar dinamik olmayan, lakin leksik görünmə sahəsinə malikdir. Bu onu bildirir ki, funksiyalar icra zamanı deyil, təyin edilmə zamanı yaradılan görünmə sahəsində həyata keçirilir. Funksiyaların təyin edilmə anında görünmə sahələrinin cari zənciri funksiyanın daxili vəziyyətinin hissəsində saxlanılır. Yuxarı səviyyədə görünmə sahəsi sadəcə qlobal obyektdən ibarətdir və bunları leksik görünmə sahəsinə aid etmək olmaz. Ancaq daxili funksiya elan edilən zaman, onun görünmə sahələri zənciri bütöv funksiyanı əhatə edir. Bu onu bildirir ki, daxili funksiya, bütöv funksiyada olan bütün arqumentlərə və lokal dəyişənlərə giriş imkanına malikdir. Nəzərə alın ki, görünmə sahələri zənciri funksiyanın təyin edilmə anında təsbit edilməsinə baxmayaraq, müəyyən edilmiş xüsusiyyətlərin siyahısı bu zəncirdə təsbit

edilmir. Görünmə sahələri zənciri dəyişikliklərə meyillidir və funksiyanın icrası zamanı mövcud olan bütün elementlərə müraciət edə bilər.

8.8.2. Çağırış obyekti

JavaScript interpretatoru funksiyanı çağıran zaman, ilk növbədə görünmə sahələri zəncirinə müvafiq olaraq funksiyanın icrası zamanı qüvvədə olan görünmə sahəsi qurulur. Sonra zəncirin başlanğıcına çağırış obyekti kimi bilinən yeni obyekti əlavə edilir. Bu obyekti, ECMAScript spesifikasiyasında ***aktivləşdirmə obyekti*** *(activation object)* termini ilə adlandırılır. Daha sonra funksiyanın Arguments obyektinə istinad edən arguments xüsusiyyəti çağırış obyektinə əlavə edilir. Bundan sonra isə, çağırış obyektinə funksiyanın adlandırılmış arqumentləri əlavə edilir. var təlimatının köməyi ilə elan edilmiş istənilən lokal dəyişən, həmçinin çağırış obyektinin daxilində də təyin edilir. Madam ki, çağırış obyekti görünmə sahələri zəncirinin başlanğıcında yerləşir, onda bütün lokal dəyişənlər, funksiya arqumentləri və Arguments obyekti funksiyanın gövdəsindən görülən olur. Hər şeydən əlavə bu onu bildirir ki, bütün eyni adlı xüsusiyyətlər görünmə sahəsi xaricində məlum olur. Nəzərə alın ki, arguments-dən fərqli olaraq, `this`– çağırış obyektinin xüsusiyyəti deyil, sadəcə açar sözüdür.

8.8.3. Çağırış obyekti ad fəzası qismində

Bəzən olur yalnız funksiyanı ona görə yaratmaq rahat, çağırışın obyektini almaq üçün, hansı ki, müvəqqəti ad sahəsi kimi hərəkət edir (qüvvədə olur), harada op1 Bu bölmə olar yüksəldilmiş çətinliyin materialını özündə saxlayır, hansı ki, birinci oxuma vaxtı buraxmaq olar. redelyat dəyişənlər və xüsusiyyətlər, qlobal ad sahəsiylə mümkün münaqişələr haqqında narahat olmadan. Fərz edək, müxtəlif JavaScript-proqramlarında (və ya, əgər proses JavaScript kliyent dilinə aiddirsə, müxtəlif veb səhifələrdə) istifadə edilməsinə ehtiyac yaranan, JavaScript dilində yazılmış kod faylı mövcuddur. Tutaq ki, həmin bu kodda, istənilən digər kodlarda olduğu kimi, aralıq hesablamaların nəticələrinin saxlanılması

üçün nəzərdə tutulmuş dəyişən təyin edilib. Problem isə ondan ibarət ola bilər ki, madam bu kod müxtəlif proqramlarda istifadə olunacaq, onda həmin proqramların özlərində müəyyən edilən dəyişən adları ilə bu proqramda müəyyən edilən dəyişən adları arasında münaqişə yarana bilər. Belə münaqişələrdən qaçmaq üçün, idxal edilən kodu funksiya daxilinə yerləşdirmək və sonra da bu funksiyaya müraciət etmək olar. Bunun sayəsində dəyişənlər funksiyanın çağırış obyektinin daxilində təyin ediləcəklər:

```
function init()
{
  // Burada idxal edilən proqram kodu yerləşir.
  // Elan edilmiş istənilən dəyişən çağırış obyektinin
xüsusiyyəti olacaq.
  // Bununla da qlobal ad fəzasında münaqişə ehtimalı sıfıra
enəcək.
}
init(); // Funksiyanı çağırmağı unutmayın!
```

Bu fraqment qlobal ad sahəsinə funksiyaya istinad edən bir ədəd init xüsusiyyəti əlavə edir. Əgər bir ədəd xüsusiyyətin əlavə edilməsi də sizə artıq görünürsə, funksiyanı bir ifadə şəklində anonim olaraq müəyyən etmək və sonra bu funksiyanı çağırmaq olar. Aşağıdakı fraqment, bu prosesi əks etdirir:

```
(function (){
   // Bu adsız funksiyadır.
   // Burada idxal edilən proqram kodu yerləşir.
   // Elan edilmiş istənilən dəyişən çağırış obyektinin
xüsusiyyəti olacaq.
   // Bununla da qlobal ad fəzasında münaqişə ehtimalı sıfıra
enəcək.
}) (); // Funksional literalın sonu və funksiyanın çağırışı.
```

Funksional literalı əhatə edən yumru mötərizələrə nəzər yetirir. JavaScript sintaksisi hər zaman bu mötərizələri tələb edir.

8.8.4. Daxili funksiyalar qapanma qismində

JavaScript funksiyaların daxilində funksiyaların elanına imkan verməklə, funksiyalardan adi məlumatlar kimi istifadə etməyə də şərait yaradır və həmçinin funksiyaların görünmə sahələri zəncirləri arasında qarşılıqlı təsiri təşkil etməklə, proqramdan maraqlı və effektiv nəticələrin alınmasına səbəb olur. İzaha başlamazdan əvvəl, *f* funksiyasının daxilində təyin edilən *g* funksiyasına baxaq. *f* funksiyası çağırılan zaman, bu funksiyanın görünmə sahələri zənciri qlobal obyekti müşahidə edən çağırış obyektini ehtiva olunur. *g* funksiyası *f* funksiyasının daxilində təyin edilir, beləliklə, *f* funksiyanın görünmə sahələri zənciri *g* funksiyasının təyin hissəsi kimi saxlanılır. *g* funksiyası çağırılan zaman, onun görünmə sahələri onun zənciri artıq üç obyekti ehtiva edir: şəxsi çağırış obyekti, *f* funksiyasının çağırış obyekti və qlobal obyekti.
Daxili funksiyalar, müəyyən edildikləri leksik görünmə sahəsindən çağırılan zaman, onların işləmə qaydası tamamilə aydın olur. Məsələn, aşağıda fraqmentdə qeyri-adi heç nə yoxdur:

```
var x = "qlobal";
function f ()
{
  var x = "lokal";
  function g()
  {
    alert(x);
  }
  g();
}
f(); // Bu funksiyaya müraciət edən zaman "lokal" sözü
ekranlaşdırılacaq
var x = "qlobal";
function f ()
{
  var x = "lokal";
  function g()
  {
    alert(x);
  }
  g();
}
f(); // Bu funksiyaya müraciət edən zaman "lokal" sözü
ekranlaşdırılacaq
```

JavaScript-də funksiyalara adi məlumatlar kimi baxıldığına görə, funksiyaları başqa funksiyalardan qaytarmaq, obyekt xüsusiyyətlərinə

mənimsətmək, massivlərdə saxlamaq və s. olar. Daxili funksiyaların yoxluğu halında burada da qeyri-adi heç nə yoxdur. Aşağıdakı fraqmentə baxaq. Bu fraqmentdə müəyyən edilən funksiya, daxili funksiyanı qaytarır. Bu funksiyaya hər bir müraciətdə funksiya qaytarılır. Bu halda kodu JavaScript-in özü dəyişmir, amma çağırışdan çağırışa görünmə sahəsini dəyişə bilər, çünki, əsas funksiyaya hər bir müraciətdə, onun arqumentlərini dəyişə bilərlər. (Yəni, görünmə sahələri zəncirində əsas funksiyanın çağırış obyekti dəyişəcək.) Əgər qaytarılan funksiyaları massivdə saxlasaq və sonra da onların hər birini çağırsaq, bu funksiyalar müxtəlif qiymətlər qaytaracaq. Çünki, funksiyanın proqram kodu belə olan halda dəyişmir və funksiyaların hər biri öz görünmə sahələrində çağırılır və bu müəyyən edilmiş funksiyaların görünmə sahələri arasında fərqə gətirib çıxarır:

```
// Bu funksiya başqa funksiyanı qaytarır
// Daxili funksiyanın müəyyənin edildiy çağırışdan çağırışa öz
görünmə
// sahəsini dəyişir
function makefunc(x)
{
  return function()
  {
    return x;
  }
}
// makefunc() funksiyasını bir neçə dəfə çağırılması və
nəticələrin massivdə
// saxlanılması:
var a = [makefunc (0), makefunc (1), makefunc (2)];

// İndi isə funksiyaları çağırırıq və onlardan alınmış
qiymətləri
// ekranlaşdırırıq. Hərçənd ki, funksiyaların gövdəsi
dəyişmir, amma
// bu funksiyaların görünmə sahələri dəyişir və hər bir
çağırışda onlar
// müxtəlif qiymətləri qaytarır:
alert(a[0]());            // 0 ekranlaşdırılacaq
alert(a[1]());              // 1 ekranlaşdırılacaq
alert(a[2]());            // 2 ekranlaşdırılacaq
```

Bu fraqment gözlənilən nəticəni verir və iş prosesi leksik görünmə sahəsinin qaydasına uyğundur: funksiya müəyyən edildiyi görünmə

sahəsində icra edilir. Ancaq ən maraqlısı ondan ibarətdir ki, görünmə sahələri əsas funksiyadan çıxıldıqdan sonra da öz mövcuduğunu davam etdirir. Bu adi vəziyyətdə olmur. Funksiya çağırılarkən, çağırış obyekti yaradılır və bu obyekt funksiyanın görünmə sahəsində yerləşir. Funksiya işini bitirdiyi zaman, çağırış obyekti çağırış zəncirindən silinir. Daxili funksiyaların yoxluğunda halında, çağırış obyektinə yeganə istinad görünmə zənciridir. Obyektə olan istinad zəncirdən silindiyi zaman, prosesə tullantı toplayıcısı daxil olur. Ancaq daxili funksiyaların mövcudluğu halında vəziyyət dəyişir. Daxili funksiya təyin edilən zaman, çağırış obyektinə əsaslanır, çünki, bu obyekt görünmə sahələri zəncirinin başlanğıcında, yəni funksiyanın təyin edildiyi yerdə yerləşir. Əgər daxili funksiya yalnız əsas funksiyanın daxilində istifadə olunursa, onda daxili funksiyaya yeganə istinad çağırış obyektidir. Proqramın idarəsi xarici funksiyadan qaytarılan zaman, daxili funksiya çağırış obyektinə istinad edir, amma çağırış obyekti daxili funksiya timsalında heç bir funksiyaya istinad etmir, bunun sayəsində tullantıların tənzimlənməsi mexanizmi funksiyalara əlçatan olur.

Əgər daxili funksiyaya istinad qlobal görünmə sahəsində saxlanılırsa vəziyyət tamamilə dəyişir. Bu cür hallar, daxili funksiya əsas funksiyanın qaytarılan qiyməti şəklində ötürüldükdə və ya hər hansı başqa obyektin xüsusiyyəti şəklində saxlanıldığı zaman müşahidə edilir. Bu zaman daxili funksiyaya xarici istinad meydana çıxır və belə olan halda da daxili funksiya əsas funksiyanın çağırış obyektinə istinad etməyə davam edir. Nəticədə əsas funksiyaya hər belə müraciət zamanı yaradılmış bütün çağırış obyektləri daxil olmaqla, funksiya arqumentlərinin və lokal dəyişənlərin adları və qiymətləri də öz mövcudluğunu davam etdirir. JavaScript-də çağırış obyektinə birbaşa təsir etmək imkanları mövcud deyil, amma bu obyektin xüsusiyyətləri, daxili funksiyaya ixtiyari müraciət zamanı yaradılan görünmə sahələri zəncirinin bir hissəsidir. (Nəzərə alın ki, əgər əsas funksiyada iki daxili funksiyaya qlobal istinadlar mövcuddursa, bu daxili funksiyalar eyni çağırış obyektini birgə istifadə edəcək və bu funksiyalardan hər hansı birinə müraciət nəticəsində edilmiş dəyişikliklər o biri funksiyada da ediləcək.)

JavaScript-də funksiyalar icra edilən proqram kodunun və görünmə sahəsinin kombinasiyasından olunub. Kompüter terminologiyası üzrə olan ədəbiyyatlarda proqram kodunun görünmə sahəsinin bu cür kombinasiyası

qapanma (closure) adlanır. JavaScript-də bütün qapanmadır. Ancaq bütün bu qapanmalar yalnız bir marağı mövcuddur, yəni biraz əvvəl bəhs etdiyimiz kimi, daxili funksiya yalnız müəyyən edildiyi görünmə sahəsi hüdudlarında ixrac edilir. Beləliklə istifadə edilən daxili funksiyalar adətən aşkar qapanma adlandırılır. Qapanmalar – çox maraqlı və güclü proqramlaşdırma texnikasıdır. Hərçənd ki, qapanmalar nadir hallarda istifadə olunur, onları öyrənmək lazımdır. Əgər siz qapanma mexanizmini başa düşsəniz, görünmə sahələrini asanlıqla anlayacaqsınız və təcrübəli JavaScript proqramçısı olacaqsınız.

8.8.4.1. Qapanma nümunələri

Bəzən funksiya çağırışları arasında bir neçə qiymətin yadda saxlanılmasına ehtiyac. Qiymət lokal dəyişəndə saxlanıla bilməz, çünki, funksiya müraciətlər arasında öz çağırış obyektini saxlamır. Qlobal dəyişən belə vəziyyətdə çıxış yolu ola bilər, amma bu ad fəzasının dolmasına gətirib çıxarır. Bölmə 8.6.3-də *`uniqueInteger()`* funksiyası göstərilmişdir, hansı ki, belə vəziyyətlər üçün funksiyaya şəxsi xüsusiyyəti cəlb edilmişdir. Odna16 sonra getmək və neisçezayuşey dəyişənin şəxsi (private-ı) yaradılması üçün qısa qapanmadan istifadə etmək olar.
Aşağıda başlanğıc üçün bu funksiyanın qapanmasız nümunəsi göstərilmişdir:

```
//Hər bir çağırışda  müxtəlif qiymətlərini qaytarılır
uniqueID= function(){
if(!arguments.callee.id) arguments.callee.id = 0;
    return arguments.callee.id++;
};
```

Burada problem ondadır ki, *`uniqueID.id`* xüsusiyyəti funksiya xaricində əlçatandır və qiymət 0 olaraq təyin edilsə, "funksiyanın heç vaxt eyni qiyməti iki dəfə qaytarmamaq tapşırığı" uğursuzluğa düçar olur. Bu problemi həll etmək üçün qiyməti qapanmada saxlamaq olar və belə olan halda qiymət giriş yalnız bu funksiyadan mümkün olacaq:

```
uniqueID = (function()
{
  // Qiymət funksiyanın çağırış obyektində saxlanılır
```

```
  var id = 0;

  // Bu xüsüsi dəyişən, öz qiymətini funksiya çağırışları
arasında saxlayır
  // Xarici funksiya bu qiymətə giriş edə bilən
  // daxili funksiyanı qaytarır. Bu daxili funksiya uniqueID
dəyişənindən
  // yuxarıda saxlanılır.

  return function() { return id++; }; // Qiymətin artırılması
və qaytarılması
}) (); // Xarici funksiyanı təyin etdikdən sonra çağırışı.
```

Nümunə 8.6-da daha bir qapanma nümunəsi göstərilmişdir. Nümunədə xüsusi dəyişənlər qismində nümayiş etdirilən dəyişənləri, əvvəl göstərilmişdi, bir neçə (bir qədər) funksiyayla birgə istifadə oluna bilərlər.

Nümunə 8.6. *Qapanmaların köməyilə şəxsi xüsusiyyətlərin yaradılması*

```
// Bu funksiya verilmiş adlara uyğun "o" obyektinin
xüsusiyyətinə
// giriş metodlarını əlavə edir.
// Metodlar get<name>  və set<name> əldə edir. Əgər əlavə
olaraq yoxlama
// funksiyası verilərsə, yazı metodu saxlanmadan öncə qiymətin
yoxlanılması
// üçün bu funksiyadan istifadə edəcək. Əgər yoxlamanın
funksiyası false
//qaytararsa,  yazı metodu istisnanı yaradır.
//
// Belə yanaşmanın qeyri-adiliyi ondadır ki, metodlara əlçatan
olan
// xüsusiyyət qiyməti, "o" obyektinin xüsusiyyəti şəklində
deyil bu
// funksiyanın lokal dəyişəni şəklində saxlanılır.
// Bundan başqa, giriş metodları lokal miqyasda müəyyən
edilmişdir, buna görə
// də metodlar bu funksiyaya və bu lokal dəyişənə giriş edə
bilirlər.
//
// Nəzərə almaq lazımdır ki, qiymət yalnız bu iki metod üçün
əlçatandır və
// yazı metodu kimi təyin edilə və ya hər hansı formada
dəyişdirilə bilməz.
```

```
function makeProperty (o, name, predicate)
{
  var value;  // Bu xüsusiyyətin qiymətidir

  // Oxuma metodu sadəcə qiyməti qaytarır.
  o["get" + name] = function(){ return value; };

  // Yazı metodu əgər yoxlama funksiyasının nəticəsi true
olarsa qiyməti
  // saxlayır,əks təqdirdə istisna yaradır.
  o ["set" + name] = function (v){
  if (predicate &&! predicate (v))
     throw "set" + name + ": səhv qiymət"+ v;
  else
     value = v;
  };
}
// Növbəti fraqment makeProperty() metodunun işini göstərir.
var o ={};          // Boş obyekt

// getName() və setName() adlı xüsusiyyətə giriş metodlarını
əlavə etmək
// yalnız sətir qiymətlərinin mümkünlüyünü təmin etmək
makeProperty(o, "Name", function (x){ return typeof x ==
"string"; });

o.setName("Frank");     // Xüsusiyyətin qiymətini təyin etmək
print(o.getName ());    // Xüsusiyyətin qiymətini əldə etmək
o.setName(0);             // Xüsusiyyətə xətalı tipdə qiymət
təyin etmək cəhdi
```

Qapanmalara aid mənə məlum olan ən praktik nümunə – ***Stiv Yen*** *(Steve Yen)* tərəfindən hazırlanmış və ***TrimPath*** kliyent platformasının bir hissəsi kimi ***http://trimpath.com*** saytında dərc edilən dayanma nöqtələri mexanizmidir.

Dayanma nöqtəsi – funksiyanın daxilində elə bir nöqtədir ki, bu nöqtədə proqramın icrası dayanır və developerə (proqram istehsal edən) dəyişənlərin qiymətlərini gözdən keçirmək, ifadələri hesablamaq və ilaxır funksiyaları çağırmaq və s. imkanlara şərait yaradılır. Stiv Yens tərəfindən düşünülmüş dayanma nöqtələri mexanizmində qapanmalar cari funksiyanın (lokal dəyişənlər və giriş arqumentləri daxil olmaqla) icra kontekstinin saxlanılmasına xidmət edir və qlobal `eval()` funksiyasının köməyilə bu kontekstin tərkibini gözdən keçirməyə imkan

verir. `eval()` funksiyası JavaScript dilində sətirlərdən qiymətlər alınmasına xidmət edir. Aşağıda öz icra kontekstini yoxlayan, qapanma kimi işləyən daxili funksiya nümunəsi göstərilmişdir:

```
// Cari konteksti yadda saxlamaq və onu eval() funksiyasının
köməyi ilə
// yoxlamağa icazə vermək
var inspector = function ($){ return eval ($);}
```

Funksiya arqument adı kimi nadir hallarda istifadə edilən $ identifikatorundan istifadə edir ki, bu da yoxlanılan görünmə sahəsində olan adların münaqişə ehtimalı aşağı salır. Nümunə 8.7-də göstərilən qapanmanı verməklə, dayanma nöqtəsini yaratmaq olar.

Nümunə 8.7. *Qapanmalar əsasında dayanma nöqtələri*

```
// Bu funksiya dayanma nöqtəsinin realizasiyasıdır. Funksiya,
// istifadəçiyə ifadəni daxil etmək təklifi edir, daxil edilən
ifadəni
// qapanmalar vasitəsilə hesablayır və nəticəni ekranlaşdırır.
İstifadə
// edilən qapanma yoxlanan görünmə sahəsinə giriş imkanı
verir, beləliklə
// istənilən funksiya öz şəxsi qapanmasını yaradacaq.
//
//breakpoint() funksiyasının obrazı və bənzərliyi üzrə Stiv
Yens tərəfindən
// realizasiya olunub.
// http://trimpath.com/project/wiki/TrimBreakpoint

function inspect (inspector, title)
{
  var expression, result;

  // Bu funksiyada "ignore" xüsusiyyətinin yaradılması
hesabına
  // dayanma nöqtələrini kəsmək olar.

  if ("ignore" in arguments.callee) return;
  while(true)
  {
    // Sorğunu istifadəçidən əvvəl müəyyən edirik
    message = "";
    // Əgər title arqumenti verilmişdirsə, onda birinci onu
çıxarırıq
    if(title) message = title + "\n";
```

```
    // Əgər ifadə artıq hesablanmışsa, onu qiymətiylə
birlikdə çıxarırıq
    if (expression) message += "\n" +expression+" ==>
"+result+" \n";
    else expression = "";
    // İstifadəçini həmişə daxil edilməyə dəvət etmək
lazımdır:
    message += "Zəhmət olmasa hesablmamaq istədiyiniz ifadəni
daxil edin:";
    // İstifadəçinin daxil etdiyi informasiyanı əldə etmək,
dəvəti silmək və
    // son ifadəni susmaya görə qiymət kimi istifadə etmək
    expression = prompt(message, expression);

    // Əgər istifadəçi heç nə daxil etmədisə ( və ya imtina
düyməsini
    // kliklədisə), işi dayanma nöqtəsində bitmiş hesab etmək
olar
    // və bu zaman idarəetmə qaytarılır.
    if (!expression) return;

    // Əks təqdirdə icra kontekstində yoxlanılan
    // qapanma vasitəsilə ifadəni hesablamaq.

    // Nəticələr növbəti iterasiyada ekranlaşdırılacaq.
    result = inspector(expression);
  }
}
```

Nəzərə alın ki, informasiyanın çıxarılması və istifadəçinin sətir daxil etməsi üçün nümunə 8.7-dəki *`inspect()`* funksiyası *`window.prompt()`* metodunu cəlb edəcək.

Ədədin fatorialını hesablayan və dayanma nöqtələri mexanizmindən istifadə edən funksiya nümunəsinə baxaq:

```
function factorial(n)
{
  // Funksiya üçün qapanmanın yaradılması
  var inspector = function($)
  {
    return eval($);
  }
  inspect(inspector, "factorial() funksiyasına giriş");
  var result = 1;
  while(n > 1)
  {
```

```
    result = result * n;
    n--;
    inspect(inspector, "factorial () loop");
  }
  inspect (inspector, "factorial() funksiyasından çıxış");
  return result;
}
```

8.9. Function() konstruktoru

Əvvəldə də deyildiyi kimi, funksiyalar adətən *Function* açar sözünün köməyilə və ya funksiyanın təyinləri formasında, və ya funksional çap səhvinə vasitəsi ilə təyin edilir. Ancaq bundan başqa Function() konstruktorunun köməyilə də funksiyaların yaradılması imkanı mövcuddur. Function() konstruktorunun köməyilə funksiyaların yaradılması digər üsullara nisbətən mürəkkəb hesab edildiyinə görə, çox az istifadə olunur. Aşağıda oxşar tərzdə yaradılmış funksiya nümunəsi göstərilmişdir:

```
var f = new Function(" x", "y", "return x*y;");
```

Bu sətir aşağıdakı sintaksisə ekvivalent yeni funksiyanı yaradır:

```
function f(x, y){ return x*y;}
```

Function() konstruktoru istənilən miqdarda arqumenti sətir şəklində qəbul edir. Sonuncu arqument – funksiyanın gövdəsində. Bu arqument bir-birindən nöqtəli vergüllərlə ayırılmış istənilən qədər JavaScript-təlimatı ehtiva edə bilər. Konstruktorun qalan bütün arqumentləri müəyyən edilən funksiyanın parametr adlarını verən sətirlərdən təşkil olunur. Əgər siz funksiyanı arqumentlərsiz müəyyən etmək istəyirsinizsə, onda konstruktor yalnız bir sətir – funksiyanın gövdəsini ötürmək lazımdır. Nəzərə alın ki, Function() konstruktoruna yaradılan funksiyanın adını verən arqument ötürülmür. Function()-konstruktorunun köməyilə yaradılmış adlandırılmayan funksiyalar bəzən anonim funksiya da adlanır. Function()-konstruktoru ilə bağlı bir neçə məqamı xüsusi olaraq diqqətinizə çatdıraq:

• `Function()` konstruktoru proqramın icrası zamanı dinamik olaraq funksiyaların yaradılması və onların kompilyasiya olunmasına şərait yaradır.
• `Function()` konstruktoru yeni funksiyanı hər çağırışda kompilyasiya edir və yaradır. Əgər konstruktor çağırışı dövr və ya tez-tez çağırılan funksiyanın gövdəsində olarsa, bu proqramın məhsuldarlığında mənfi təsir edə bilər. Buna zidd olaraq dövrün daxilində olan funksional literallar və ya daxili funksiyalar hər iterasiyada yenidən kompilyasiya olunmur, bundan başqa, literal halında funksiyanın yeni obyekti yaradılmır. (Hərçənd, bu halda artıq qeyd edildiyi kimi, funksiyanın müəyyən edildiyi leksik konteksti ehtiva edən yeni qapanma yaradılması daha yaxşıdır.)
• Və sonda çox vacib məqam: funksiya `Function()`-konstruktorunun köməyilə yaradılan zaman, cari leksik görünmə sahəsi nəzərə alınmır – bu yolla yaradılmış funksiyalar həmişə yuxarı səviyyəli funksiyalar kimi kompilyasiya edilirlər ki, növbəti fraqment bunu əyani olaraq nümayiş etdirir:

```
var y = "qlobal";
function constructFunction()
{
  var y = "lokal";
  return new Function ("return y"); // lokal konteksti
saxlamır!
}
// Aşağıda nəticə olaraq "qlobal" sözü ekranlaşdırılacaq,
çünki Function()
// funksiya, lokal kontekst istifadə etmir.
// konstruktoru ilə əgər funksiya literal şəklində müəyyən
edilsəydi,
// nəticə olaraq "lokal" sözü ekranlaşdırılacaqdı.
alert(constructFunction()()); // "qlobal" sözünü ekranlaşdırır
```

9

Siniflər, konstruktorlar və prototiplər

7-ci fəsildə JavaScript- obyektlərinə müqəddimə verilmiş və qeyd edilmişdi ki, hər bir obyekt istənilən digər obyektdən öz unikal xüsusiyyət dəsti ilə fərqlənir. Obyekt yönümlü proqramlaşdırma dillərinin əksəriyyətində obyektlərin siniflərini müəyyən etmək və sonra bu siniflərin nüsxələri qismində ayrı-ayrı obyektlər yaratmaq imkanı mövcuddur. Məsələn, kompleks ədədləri təqdim etmək və bu ədədlər üzərində hesab əməllərini yerinə yetirmək üçün çağırılmış Complex sinifini elan etmək olar, onda Complex obyekti bir kompleks ədədi təqdim edərdi və bu sinifin nüsxəsi kimi yaradıla bilərdi. JavaScript dili Java, C++, C# və bu kimi dillərə nisbətən tam olaraq sinif dəstəyinə malik deyil. Bununla belə JavaScript-də funksiya-konstuktorları və obyekt prototipləri kimi instrumental vasitələrin köməyilə psevdosiniflər müəyyən etmək imkanı mövcuddur. Bu fəsildə JavaScript-də olan konstruktorlardan və prototiplərdən bəhs edilir və bir sıra psevdosinif və hətta psevdoaltsinif nümunələri göstərilir. Bu fəsildə "sinif" sözündən qeyri-rəsmi istifadə edilir, buna görə diqqətli olun və JavaScript dəstəklənməyən, lakin əksər OYP dillərində olan əsl siniflərlə bu qeyri-formal "siniflər" qarışdırmayın. Qeyd edək ki, JavaScript 2.0-versiyasının sinif mexanizmini tam olaraq dəstəkləməsi planlaşdırılır.

9.1. Konstruklar

7-ci fəsildə {} literalının, həmçinin `new Object()` ifadəsinin köməyilə yeni boş obyektin yaradılmasını şahidi olduq. Bundan başqa,

digər tiplərdə olan obyektlərin yaradılması qaydası təxminən aşağıdakı formada nümayiş etdirilmişdi:

```
var array = new Array(10);
var today = new Date();
```

new operatorunda funksiya-konstruktorunun adı göstərilməlidir. new operatoru yeni obyekti heç bir xüsusiyyəti olmadan yaradır və yaradılmış obyekti *this* açar sözünün qiymәti şəklində funksiyaya ötürür və bu funksiyanı çağırır. new operatoru ilə birgə tətbiq edilən həmin funksiya funksiya-konstuktoru və ya sadəcə konstruktor adlanır. Konstruktorun əsas işi yeni yaradılmış obyektin inisializasiyasından ibarətdir yəni, konstruktora qədər inisializasiya edilәsi bütün xüsusiyyətləri qruplaşdıraraq, obyekti proqramda istifadə olunacaq şəklə salmaqdır. Şəxsi konstruktor müəyyən etmək üçün, obyektə, *this* açar sözünə istinad etməklə yeni xüsusiyyətlər əlavə edən funksiya yazmaq kifayətdir. Aşağıdakı fraqmentdə iki obyektin konstruktorunun təyini göstərilir:

```
// Konstruktoru müəyyən edirik.
// Nəzərə alın ki, obyekt "this" açar sözünün köməyilə
inisializasiya olunur.

function Rectangle(w, h){
     this.width = w;
     this.height = h;
}

// İki Rectangle obyektinin yaradılması üçün konstruktoru
çağırırıq. Biz
// konstruktora en və hündürlük veririk ki, hər iki yeni
obyekti düzgün
// insiallaşdırmaq mümkün olsun.
var rect1 = new Rectangle(2, 4);       // rect1 ={ width:2,
height:4 };
var rect2 = new Rectangle(8.5, 11);   // rect2 ={ width:8.5,
height:11 };
```

Nəzərə alın ki, konstruktor obyektin *this* açar sözünə istinad edən xüsusiyyətlərinin inisializasiyası üçün öz arqumentlərindən istifadə edir. Burada biz sadəcə uyğun olan funksiya-konstuktorunu yaradıb obyektlərin sinifini müəyyən etdik – *Rectangle()*-konstruktorunun köməyilə

yaradılmış bütün obyektlər zəmanətli olaraq inisializasiya edilmiş *width* və *height* xüsusiyyətlərinə malik olacaq. Bu onu bildirir ki, şəraitdən asılı olaraq, eyni işi *Rectangle* sinifinin bütün obyektləri ilə təşkil etmək olar. Bir halda ki, hər bir konstruktor obyektlərin sinifini ayrılıqda müəyyən edir, funksiya-konstuktoruna bu cür ad mənimsətmək çox yaxşıdır, hansı ki, bu halda funksiya-konstuktorunun köməyilə yaradılan obyektlərin sinifi açıq-aydın əks olunacaq. Məsələn, düzbucaqlı dördbucağın obyektini yaradan `new Rectangle(1, 2)` sətiri `new init_rect(1, 2)` sətirinə nisbətən daha aydın görünür.
Adətən funksiya-konstuktorları heç nə qaytarmır, onlar yalnız *this* açar sözünün qiyməti qismində alınmış obyekti inisializasiya edirlər. Ancaq konstruktorlar üçün obyekt qaytarmaq imkanı mövcuddur və qaytarılan obyekt `new` ifadəsinin qiyməti olur. Bu halda konstruktora *this* açar sözünün qiyməti şəklində verilmiş obyekt sadəcə olaraq silinir.

9.2. Prototiplər və varislik

8-ci fəsildə deyilirdi ki, metod – obyektin xüsusiyyəti kimi çağırılan funksiyadır. Funksiya bu yolla çağırılan zaman, çağırış obyekti, *this* açar sözünün qiyməti olur. Fərz edək ki, bizə *Rectangle* obyekti ilə təqdim edilmiş düzbucaqlı dördbucağın sahəsini hesablamaq lazımdır. Bunun üçün mümkün üsullardan biri aşağıda göstərilir:

```
function computeAreaOfRectangle(r){ return r.width * r.height;
}
```

Bu funksiya öz işinin öhdəsindən layiqincə gəlir, lakin bu funksiya obyekt yönümlü deyil. Obyektlə işləyərkən, obyekt metodlarını çağırmaq və obyektləri arqumentlər qismində yad funksiyalara verməmək daha yaxşıdır. Bu yanaşma aşağıdakı fraqmentdə nümayiş etdirilir:

```
// Rectangle obyektinin yaradılması
var r = new Rectangle(8.5, 11);
// Obyektə metod əlavə edirik
r.area = function() { return this.width * this.height; }
// İndi, obyektin metodunu çağırmaqla sahəni hesablaya bilərik
```

```
var a = r.area();
```

Əlbəttə ki, obyektin istifadəsindən əvvəl ona yeni metodu əlavə etmək narahatdır. Ancaq, əgər *area* xüsusiyyətini funksiyası-konstruktorunda inisializasiya etsək bu vəziyyəti yaxşılaşdırmış olarıq. Deməli belə, *Rectangle()* konstruktorunun təkmil realizasiyası aşağıdakı qaydada olur:

```
function Rectangle(w, h) {
  this.width = w;
  this.height = h;
  this.area = function( ) { return this.width * this.height; }
}
```

Eyni alqoritmi konstruktorun yeni versiyasında digər formada realizasiya etmək olar:

```
// U.S kağız formatında olan vərəqin kv. düymlə sahəsinin
tapılması.
var r = new Rectangle(8.5, 11);
var a = r.area();
```

Bu cür həll daha yaxşı görünə bilər, amma bu həll əvvəlki kimi optimal deyil. Yaradılmış hər bir düzbucaqlı dördbucaq üç xüsusiyyətə malik olacaq. *width* və *height* xüsusiyyətləri hər bir düzbucaqlı dördbucaq üçün unikal qiymətlərə malik ola bilər, amma *Rectangle* obyektinin ayrılıqda götürülmüş hər bir *area* xüsusiyyəti həmişə eyni funksiyaya istinad edəcək (əlbəttə, bu xüsusiyyəti iş prosesində dəyişdirmək olar, amma, bir qayda olaraq, obyekt metodları dəyişdirilməməlidir). Obyekt metodlarının eyni sinifin bütün nüsxələrində istifadə olunması üçün ayrı-ayrı xüsusiyyətlərdə saxlanılması çox effektsiz həll üsuludur.

Ancaq, bu problemi həll etmək olar. Məlumdur ki, JavaScript-də bütün obyektlər prototip adlanan daxili istinadı özündə saxlayır. Prototipin istənilən xüsusiyyəti prototipi olan digər obyektin xüsusiyyəti olur. Yəni, başqa sözlə desək, ixtiyari JavaScript-obyekti öz prototipinin xüsusiyyətlərini müşahidə edir.

Əvvəlki bölmədə, new operatorunun boş obyekti yaratdığı və sonra funksiya-konstuktorunu çağırdığı barədə bəhs edilmişdi. Amma bununla iş bitmir. Boş obyektin yaradılmasından sonra new operatoru bu obyektdə prototip istinadı qurur. Konstruktor funksiyasının *prototype* xüsusiyyət

qiyməti obyektin prototipidir. Bütün funksiyalar təyin edilərkən inisializasiya olunan *prototype* xüsusiyyətinə malikdir. Bir xüsusiyyətli obyekt bu xüsusiyyətin ilk qiymətidir. Həmin bu xüsusiyyət *constructor* adlanır və onun qiyməti prototipi assosiasiya edən funksiya-konstruktoruna istinad edir. (*constructor* xüsusiyyəti 7-ci fəsildə ***təsvir olunmuşdur***, burada isə, sadəcə hər bir obyektin nəyə görə *constructor* xüsusiyyətinə malik olduğu izah olunur.) Prototipə əlavə edilmiş istənilən xüsusiyyət avtomatik olaraq konstruktor tərəfindən inisializasiya edilən obyektin xüsusiyyəti olur. Aşağıdakı nümunədə bunu daha aydın izah etmək olar. Nümunədə *Rectangle()* konstruktorunun yeni versiyası göstərilir:

```
// Konstruktor funksiyası hər biri ayrı-ayrı nüsxə üçün unikal
qiymətə malik
// olan xüsusiyyətləri inisializasiya edir
function Rectangle(w, h){
   this.width = w;
   this.height = h;
}

// Obyekt-prototipi metodları və başqa xüsusiyyətləri özündə
// saxlayır, hansı ki, bunlar sinifin bütün nüsxələrində birgə
istifadə oluna
// bilər.
Rectangle.prototype.area = function(){
   return this.width * this.height;
}
```

Konstruktor "sinifi" obyektləri müəyyən edir və sinifin hər bir nüsxəsi üçün fərqli ola bilən *width* və *height* kimi xüsusiyyətləri inisializasiya edir. Obyektin prototipi konstruktora bağlıdır və konstruktor tərəfindən inisializasiya edilmiş prototipi olan hər bir obyektin xüsusiyyətlər dəstinə varis olur. Bu isə o deməkdir ki, obyekt prototipi – metodlar və başqa sabit-xüsusiyyətlər üçün mükəmməl yerdir.
Nəzərə alın ki, varislik - xüsusiyyət qiymətinin axtarış prosesinin bir hissəsi kimi avtomatik həyata keçirilir. Xüsusiyyətlər yeni obyektə prototip obyektindən yamsılanmır; onlar sadəcə olaraq, obyekt xüsusiyyətləri qismində olurlar. Buradan iki əhəmiyyətli nəticə çıxır. Birincisi, obyekt-prototipindən istifadə, hər bir obyekt üçün tələb edilən yaddaşın həcmini önəmli ölçüdə azalda bilər çünki, bu halda obyektlərin

öz xüsusiyyətlərindən bir çoxu varis ola bilər. İkincisi, hətta obyektin yaradılmasından sonra prototipə əlavə edilmiş obyekt xüsusiyyətləri də varis olur. Bu mövcud siniflərə yeni metodları əlavə etmək imkanının mövcudluğunu bildirir (hərçənd bu tamamilə düzgün deyil).
Varis olunmuş xüsusiyyətlər adi obyekt xüsusiyyətlərindən heç nəylə fərqlənmir. Onlar *for*/*in* dövründə sadalana və *in* operatorunun köməyilə yoxlana bilər. Onları adi obyekt xüsusiyyətlərindən yalnızca *Object.hasOwnProperty()* metodunun köməyilə ayırmaq mümkündür:

```
var r = new Rectangle(2, 3);
r.hasOwnProperty("width"); // true: width – "r"
r.hasOwnProperty("area");  // false: area – "r" bilavasitə
xüsusiyyətidir
"area" in r;               // true: area – "r" varis olunmuş
xüsusiyyətidir
```

9.2.1 Varis olunmuş xüsusiyyətin oxunması və yazılması

Hər bir sinifdə bir xüsusiyyət dəsti olan bir obyekt-prototipi mövcuddur, amma potensial olaraq sinifin bir çox nüsxəsi mövcud ola bilər, hansı ki, bu nüsxələrin hər biri prototip xüsusiyyətlərinə varis olur. Prototip xüsusiyyəti bir çox obyektlərlə varis oluna bilər, buna görə də JavaScript interpretatoru xüsusiyyət qiymətlərinin oxunması və yazılışı arasında fundamental assimmetriyanı təmin etməlidir. Siz *o* obyektinin *p* xüsusiyyətini oxuduğunuz zaman, JavaScript əvvəlcə *o* obyektində *p* adlı xüsusiyyətin mövcudluğunu yoxlayır. Əgər burada belə xüsusiyyət yoxdursa, onda, obyekt-prototipində *p* adlı xüsusiyyətin mövcudluğu yoxlanılır. Prototiplər əsasında varislik bu cür işləyir. Ancaq xüsusiyyətə qiymət mənimsədilən zaman, JavaScript obyekt-prototipindən istifadə etmir. Gəlin bunun niyə belə olduğunu düşünək: fərz edək, siz *o.p* xüsusiyyətinə qiymət təyin etmək istəyirsiniz, amma *o* obyektində *p* adlı xüsusiyyətlər mövcud deyil. İndi isə, fərz edək ki, JavaScript *p* xüsusiyyətinin axtarışının obyekt prototipdə davam etdirir və protipində *p* xüsusiyyətinin mövcudluğu halında sizə prototipin xüsusiyyət qiymətini

dəyişdirməyə imkan verir. Nəticədə siz *p* qiymətini obyektin bütün sinifləri üçün dəyişdirirsiniz, lakin sizdən bu tələb olunmurdu.
Buna görə də, xüsusiyyətlərin varisliyi yalnız xüsusiyyət qiymətlərinin oxunması zamanı mövcud olur. Əgər siz o obyektində *p* xüsusiyyətini qurursunuzsa (hansı ki, bu xüsusiyyətə öz prototipindən varis olur), yadınızda saxlayın ki, yeni *p* xüsusiyyətin yaradılması bilavasitə obyektdə olacaq. İndiki halda, o obyekti *p* adlı şəxsi xüsusiyyətə malik olduğu zaman, o prototipdən p qiyməti daha varis olmur. Siz p qiymətini oxuduğunuz zaman, JavaScript onu əvvəlcə o obyektinin xüsusiyyətlərində axtarır. Bu zaman, o obyektində müəyyən edilmiş *p* xüsusiyyəti tapılır və xüsusiyyətin obyekt-prototipdə axtarılması ehtiyac qalmır. Biz bəzən bu hadisəni, *p* xüsusiyyətinin obyekt-prototipin *p* xüsusiyyətini "kölgədə qoyması" kimi adlandırırıq. Prototiplərin varisliyi sözlərlə ifadəsi dolaşıq görünə bilər, buna görə də yuxarıda göstərilən bütün nüansların şəkil formasında təsviri daha yaxşıdır.

Burada *area* xüsusiyyətinin müəyyən edilməsi yerləşir. Əgər sözügedən xüsusiyyət *c* obyektinin özündə mövcuddursa, onda bu qiymət qaytarılır
c.area()
c obyektindən *area*
xüsusiyyətini əldə edir
area xüsusiyyəti c obyektinin özündə müəyyən edilmir, buna görə də c obyektinin obyekt-prototipi ilə asossasiya edildiyini yoxlamaq lazımdır.

`Circle` obyekti, `c`
`r=1.0`
`x=2.0`
`y=3.0`

c.pi=4;
c obyektindəki *pi*
xüsusiyyətinə qiymət
mənimsədilməsi

pi xüsusiyyəti müəyyən edilməyib, buna görə də *c* obyektinin özündə yeni xüusiyyət yaradılır.

Obyekt-prototipi,
Object.prototype

area=Circle_area
pi=3.14159

Circle obyekti, c
r=1.0
x=2.0
y=3.0
pi=4;

pi və *r* xüsusiyyətləri *c* obyektinin özündə müəyyən edilib, buna görə də qaytarılan qiymət burada yerləşir və artıq obyekt-prototipinə müraciət edilmir.
`a=c.pi*c.r*c.r;`
c obyektindəki *pi*
və *r* xüsusiyyətinin oxunması

Circle obyekti, d
r=2.0
x=0.0
y=0.0

Burada *area* xüsusiyyətinin müəyyən edilməsi yerləşir. Əgər sözügedən xüsusiyyət *c* obyektinin özündə mövcuddursa, onda bu qiymət qaytarılır
`a=d.pi*d.r*d.r;`
d obyektindəki *pi*
və *r* xüsusiyyətinin oxunması
pi xüsusiyyəti müəyyən edilməyib, buna görə *d* obyekti ilə assosiasiya edilmiş obyekt-prototipinə müraciət edilir.

Şəkil. 9.1. *Obyektlər və prototiplər*

Prototipin xüsusiyyətləri sinifin bütün obyektləri ilə birgə istifadə olunur, buna görə də, bir qayda olaraq, onları yalnız sinifin bütün obyektləri üçün uyğun olan xüsusiyyətlərin təyini üçün tətbiq etmək lazımdır. Bu prototiplər metodların təyini üçün mükəmməldir. Digər sabit xüsusiyyət qiymətləri (məs., riyazi sabitlər) başqa xüsusiyyətlər həmçinin prototipin xüsusiyyətləri kimi təyin üçün yarayır. Əgər sinifdə tez-tez istifadə edilən susmaya görə qiymət ehtiva edən xüsusiyyəti müəyyən edilirsə, onda bu xüsusiyyəti və onun susmaya görə qiymətini obyekt-prototipində müəyyən etmək olar. Belə olan halda, susmaya görə qiymətləri dəyişdirmək imkanı olan obyektlər, özündə bu xüsusiyyətin surətini yarada və surətin qiymətini susmaya görə qiymətlərə toxunmadan dəyişə bilər.

9.3. Obyekt yönümlü proqramlaşdırma

JavaScript obyekt adlandırdığımız məlumat tipi dəstəkləməsinə baxmayaraq, bu dildə formal sinif anlayışı yoxdur. JavaScript-in bu xüsusiyyəti onu C++ və Java kimi klassik OYP dillərindən fərqləndirir. Obyekt yönümlü dillərin ümumi xüsusiyyəti ciddi tipləşdirmə və siniflər əsasında varisliyin mexanizminin mövcudluğudur. JavaScript bu meyarlara cavab verməsə də obyekt yönümlü proqramlaşdırma dillərinə aid edilir. Digər tərəfdən, biz gördük ki, JavaScript obyektlərdən fəal istifadə edir və prototiplər əsasında varisliyin xüsusi tipinə malikdir. Bu dilin bəzi realizasiyaları (nisbətən az tanınanları) yönəlmiş dillər obyekt, hansılarda ki, siniflər əsasında varisliyin yerinə prototiplər əsasında varislik reallaşdırılmışdır.

JavaScript – siniflərə əsaslanmayan obyekt yönümlü dil olmasına baxmayaraq, Java və C++ dilləri kimi siniflər əsasında dillərin imkanlarını təqlid edir yaxşıdır, belələr Java və C++ kimi. Bu fəsildə "sinif" terminindən qeyri-formal olaraq istifadə edilir.[21]

Gəlin, öncə bəzi baza terminlərindən başlayaq. Bildiyimiz kimi *obyekt – adlandırılmış məlumatların müxtəlif fraqmentlərini, həmçinin məlumatların bu fraqmentləriylə işləmək üçün metodlar ehtiva edən*

məlumatlar strukturudur. Obyekt vahid nizamlı qiymətləri və metodları qruplaşdırır və bu da öz növbəsində kodun çoxqat istifadəsi üçün modulluğun və imkanın dərəcəsini artıraraq proqramlaşdırma prosesini yüngülləşdirir. JavaScript-də obyektlər istənilən miqdarda xüsusiyyətə malik ola bilər və xüsusiyyətlər obyektə dinamik əlavə edilə bilər. Java və C++ kimi ciddi tipləşdirilmiş dillərdə bu belə deyil. Onlarda istənilən obyekt qabaqcadan müəyyən edilmiş xüsusiyyət[22] dəstinə malikdir və hər bir xüsusiyyətin tipi qabaqcadan müəyyən edilməlidir.

JavaScript-obyektləri vasitəsilə obyekt yönümlü proqramlaşdırma üsullarını təqlid edərək, bir qayda olaraq, qabaqcadan hər bir obyekt və məlumat tipində olan xüsusiyyət üçün xüsusiyyətlər dəsti müəyyən edirik

Java və C++-da sinif, obyekt strukturunu müəyyən edir. Sinif, obyektin ehtiva etdiyi etdiyi sahəni və məlumat tipini dəqiq göstərir. Həmçinin sinif, obyektlə işləmək üçün metodlar müəyyən edir. JavaScript-də formal sinif anlayışı yoxdur, amma, biz gördük ki, bu dildə sinif imkanlarına konstruktor və obyekt-prototiplərin köməyilə reallaşır.

JavaScript, siniflərə əsaslanan bir çox OYP dilləri kimi bir sinifin daxilində bir neçə obyektinin mövcudluğunu icazə verir. Biz adətən deyirik ki, obyekt sinifin nüsxəsidir. Beləliklə, eyni zamanda istənilən sinifin bir çox nüsxəsi mövcud ola bilər. Bəzən obyektin yaradılması prosesinin təsviri üçün (yəni. sinifin nüsxəsi) nüsxənin yaradılması terminindən istifadə olunur. Java proqramlaşdırma dili ilə ilk tanışlıq zamanı sinif adlarının böyük, obyekt adlarının isə kiçik hərflərlə başlanması qaydası tələqin edilir. Bu qayda mənbə mətnlərindəki siniflər və obyektləri ayırmağa kömək edir. Bu qaydanı JavaScript-də də tətbiq etməyin arzuolunandır. Məsələn, əvvəlki bölmələrdə biz `Rectangle` sinifini müəyyən etdik və bu sinifin nüsxələrini rect kimi adlarla yaratdıq.

Java-da sinif üzvləri dörd əsas tipdən birinə aid ola bilər: nüsxə xüsusiyyətləri, nüsxə, sinif xüsusiyyəti və sinif metodları metodları. Növbəti bölmələrdə biz bu tiplərin JavaScript-də necəliyinə və onlar arasında fərqlərə baxacağıq.

9.3.1. Nüsxə xüsusiyyətləri

Hər bir obyekt nüsxə xüsusiyyətlərinin şəxsi surətlərinə malikdir. Başqa sözlə, əgər bir sinifin 10 obyekti varsa, nüsxənin hər bir xüsusiyyətinin 10 surəti var. Məsələn, bizim `Rectangle` sinifimizdə istənilən `Rectangle` obyekti düzbucaqlı dördbucağın enini müəyyən edən `width` xüsusiyyətinə malikdir. İndiki halda `width` nüsxə xüsusiyyətidir. Və bir halda ki, hər bir obyekt nüsxənin xüsusiyyətinin şəxsi surətinə malikdir, onda bu xüsusiyyətlərə ayrı-ayrı obyektlər vasitəsilə giriş almaq olar. Məsələn, əgər r – özündə `Rectangle` sinifinin nüsxəsini təqdim edən obyektdirsə, biz aşağıdakı qaydada onun enini ala bilərik:

```
r.width
```

JavaScript-də susmaya görə obyekt xüsusiyyəti nüsxə xüsusiyyətidir. Ancaq OYP-yə əsaslansaq, JavaScript-də nüsxə xüsusiyyətləri əslində funksiya-konstruktoru tərəfindən yaradılan və/və ya inisializasiya olunan xüsusiyyətlərdir.

9.3.2. Nüsxə metodları

Nüsxə metodu bir çox əlamətinə görə nüsxə xüsusiyyətinə bənzəsə də, bu qiymət deyil, metoddur. (JavaScript-dən fərqli olaraq Java-da funksiya və metodlar məlumat sayılmır, buna görə də bu fərqi Java-da daha aydın görmək olar.) Nüsxə metodları obyektə və ya nüsxəyə bağlı çağırılır. Bizim `Rectangle` sinifimizdə `area()` metodu nüsxə metodudur. O `Rectangle` obyekti ücün aşağıdakı qaydada çağırılır:

```
a = r.area()
```

Nüsxə metodları `this` açar sözünün köməyilə obyektə və ya nüsxəyə istinad edir. Nüsxə metodu sinifin istənilən nüsxəsi üçün çağırıla bilər, amma bu o demək deyil ki, nüsxə hər bir obyekt metodunun şəxsi surətini ehtiva edir. Bunun yerinə nüsxənin hər metodu sinifin bütün nüsxələriylə birgə istifadə olunur. Biz JavaScript-də funksiyanın mənimsənməsi yolu ilə sinifin nüsxə metodunu obyekt-prototipinin konstruktorunda xüsusiyyət kimi müəyyən edirik.
Belə ki, cari konstruktor tərəfindən yaradılmış bütün obyektlər, funksiyaya varis olunmuş istinaddan birgə istifadə edir və göstərilən metod *çağırış sintaksisinin* köməyilə bu funksiyanı çağıra bilər.

9.3.2.1. Nüsxə metodları və this açar sözü

Əgər siz Java və ya C++ kimi dillərlə tanışsınızsa, onda bu dillərdəki nüsxə metodları ilə JavaScript-dəki nüsxə metodları arasındakı bir əhəmiyyətli fərqi yəqin etmisiz. Java və C++-da nüsxə metodlarının görünmə sahəsi `this` obyektini daxil edir. Belə ki, məsələn, Java-da `area` metodu daha sadə realizasiya oluna bilər:

```
return width * height;
```

Ancaq JavaScript-də xüsusiyyət adlarından əvvəl `this` açar sözünü qoymaq lazımdır:

```
return this.width * this.height;
```

Əgər nüsxənin hər bir xüsusiyyət adından əvvəl `this` qoymaq narahatdırsa, with (***bölmə 6.18-də*** təsvir edilən) təlimatından istifadə etmək olar, məsələn:

```
Rectangle.prototype.area = function (){
  with (this){
    return width*height;
  }
}
```

9.3.3. Sinif xüsusiyyətləri

Java-da sinif xüsusiyyəti – sinifin nüsxələri ilə deyil, yalnızca özü ilə bağlı olan xüsusiyyətdir. Yaradılan sinif nüsxələrinin miqdarından asılı olmayaraq, hər bir sinif xüsusiyyətinin yalnız bir surəti var.

Nüsxə xüsusiyyətləri sinif nüsxəsi üçün əlçatan olduğu kimi, sinif xüsusiyyətləri də sinif üçün əlçatandır. Number.MAX_VALUE – JavaScript-də sinif xüsusiyyətinə müraciətin bir nümunəsidir. Burada, MAX_VALUE xüsusiyyəti Number sinifi vasitəsilə əlçatandır. Çünki hər bir sinif xüsusiyyətinin yalnız bir surəti var və sinif xüsusiyyətləri əsasən qlobaldır. Ancaq onların üstünlüyü ondan ibarətdir ki, onlar siniflə bağlıdır və onların JavaScript ad sahəsindəki mövqeyi, çətin ki, başqa

xüsusiyyətlərlə toqquşacaq. Göründüyü ki, JavaScript-də sinifin xüsusiyyətləri özünü, konstruktorun-funksiyasının xüsusiyyətinin sadə təyini kimi göstərir. Məsələn, 1x1-ölçülü bir düzbucaqlı dördbucağın saxlanılması üçün `Rectangle`.UNIT sinifinin xüsusiyyəti bu cür yaratmaq olar:

```
Rectangle.UNIT = new Rectangle(1,1);
```

Burada `Rectangle` – funksiya-konstruktorudur, amma bir halda ki, JavaScript-də funksiyalar obyektlərdən təşkil olunub, onda biz istənilən digər obyekt xüsusiyyəti kimi funksiyanın xüsusiyyətini yarada bilərik.

9.3.4. Sinif metodları

Sinif metodu – birbaşa siniflə bağlı olan metoddur; bu metodlar sinifin konkret nüsxəsi ilə deyil, bilavasitə sinif vasitəsilə çağırılır. Məsələn, Date.parse() metodu – sinif metodudur. O həmişə Date sinifinin konkret nüsxəsi vasitəsilə deyil, Date konstruktorunun obyekti vasitəsilə çağırılır. Bir halda ki, sinif metodları funksiya-konstuktoru vasitəsilə çağırılır, onlar sinifin hər hansı konkret nüsxəsinə istinad üçün `this` açar sözündən istifadə edə bilmir, çünki, indiki halda `this`-in özü funksiya-konstuktoruna istinad edir. (Adətən sinif metodlarında `this` açar sözü heç istifadə olunmur.)
Sinif xüsusiyyətləri kimi, sinif metodları da qlobaldır. Sinif metodları konkret nüsxəylə işləmir, buna görə də sinif vasitəsilə çağırılan metodlara sadəcə funksiyalar kimi baxmaq lazımdır. JavaScript-də sinif metodunu müəyyən etmək üçün, müvafiq funksiyanı konstruktor xüsusiyyətində yaratmaq lazımdır.

9.3.5. Nümunə: Circle sinifi

Nümunə 9.1-də dairə obyektlərinin yaradılması üçün istifadə edilən funksiya-konstuktorunun və obyekt-prototipinin proqram kodu nümayiş

etdirilir. Burada nüsxə xüsusiyyətləri, nüsxə metodları, sinif xüsusiyyətləri və sinif metodları nümunələri ilə tanış ola bilərsiniz.

Nümunə 9.1. *Circle sinifi*

```
// Konstruktordan başlayaq.
function Circle (radius){
  // r – nüsxə xüsusiyyətidir, o konstruktor tərəfindən
  // təyin edilir və  inisializasiya olunur.
this.r = radius;
}
// Circle.PI – sinif, yəni funksiya-konstruktorunun
xüsusiyyətidir.
Circle.PI = 3.14159;
// Dairənin sahəsini hesablayan nüsxə metodu.
Circle.prototype.area = function (){ return Circle.PI * this.r
* this.r;}
// Sinif metodu – Circle sinifinin iki obyektini qəbul edir və
böyük radiuslu obyekti qaytarır.

Circle.max = function(a, b){ if (a.r > b.r) return a; else
return b;}

// Bu sahələrdən hər birindən istifadə nümunələri:
var c = new Circle(1.0);

// Circle sinifinin nüsxəsinin yaradılması
c.r= 2.2; // r nüsxə xüsusiyyətinin  təyin edilməsi
var a = c.area(); // area() nüsxə metodunun çağırışı
var = Math.exp(Circle.PI); // Hesablamalarının icra edilməsi
üçün sinifin
                           // PI xüsusiyyətinə müraciət
var d = new Circle(1.2); // Circle sinifinin başqa nüsxəsinin
yaradılması
var bigger = Circle.max(c, d); // Sinifin max() metodunun
çağırılması
```

9.3.6. Nümunə: Kompleks ədədlər

Nümunə 9.2-də JavaScript-də sinif obyektlərinin təyininin daha bir üsulu göstərilmişdir. Əvvəlki nümunələrə nisbətə daha formal olan bu nümunədə şərhləri gözdən keçirtməklə prosesi anlaya bilərsiniz.

Nümunə 9.2. *Kompleks ədədlər sinifi*

```
/*
```

```
* Complex.js:
* Bu faylda kompleks ədədlərin təqdim etməsi üçün Complex
sinifi təyin edilir. * Xatırladaq ki, kompleks ədəd – həqiqi
və xəyali ədədin cəmidir
* və i xəyali ədədinin kvadrat kökü -1-ə bərabərdir.
*/
/*
* Sinifin təyinində ilk addım – sinifin funksiya-
konstuktorunun təyinidir.
* Bu konstruktor obyekt nüsxəsinin bütün xüsusiyyətlərini
inisializasiya
* etməlidir. Bu "dəyişən vəziyyətləri",bütün müxtəlif sinif
nüsxələrini
* ayrılmaz edir.
*/
function Complex(real, imaginary) {
this.x = real;              // Həqiqi ədəd
this.y = imaginary;       // Xəyali ədəd
}

/*
* Sinifin təyinində ikinci addım – obyekt-protopinin
konstruktorunda
* nüsxə metodlarının (və başqa xüsusiyyətlərin) təyinidir.
* Bu obyektdə müəyyən edilmiş istənilən xüsusiyyət bütün sinif
nüsxələrinə
* varis olunacaq. Nəzərə alın ki, nüsxə metodları gizli olarar
this açar sözü
* ilə işləyir. Bir çox metodlar üçün başqa heç bir arqument
tələb olunmur.
*/
// Kompleks ədədin modulu qaytarılması. Bunun üçün kompleks
müstəvidə
// koordinat  başlanğıcından həmin ədədə qədər olan məsafə
təyin edilir.
Complex.prototype.magnitude = function() {
return Math.sqrt(this.x*this.x + this.y*this.y);
};
// Mənfi işarəli kompleks ədədin qaytarılması.
Complex.prototype.negative = function() {
return new Complex(-this.x, -this.y);
};
// Verilmiş bu kompleks ədədin toplanması və cəmin yeni obyekt
şəklində
// qaytarılması.
Complex.prototype.add = function(that) {
return new Complex(this.x + that.x, this.y + that.y);
```

```
}
// Verilmiş bu kompleks ədədin toplanması və hasilin yeni
obyekt şəklində
// qaytarılması.
Complex.prototype.multiply = function(that) {
return new Complex(this.x * that.x - this.y * that.y,
this.x * that.y + this.y * that.x);
}
// Complex obyektini anlaşılan formada sətirə dəyişdirmək.
// Complex obyektinin sətir kimi istifadə edildikdə
çağırılması.
Complex.prototype.toString = function() {
return "{" + this.x + "," + this.y + "}";
};
// Verilmiş kompleks ədədin bərabərliyinin yoxlanılması.
Complex.prototype.equals = function(that) {
return this.x == that.x && this.y == that.y;
}
// Kompleks ədədin həqiqi hissəsinin qaytarılması.
// Bu funksiya yalnızca Complex obyektinə ədəd qiyməti kimi
baxıldıqda
// çağırılır.
Complex.prototype.valueOf = function() { return this.x; }

/*
* Sinifin təyinində üçüncü addım – konstantları və digər
lazımlı
* xüsusiyyətləri konstuktor-funksiyasının özünün
xüsusiyyətləri
* (obyekt-prototipinin xüsusiyyəti kimi deyil) kimi təyin edən
metodların
* realizasiyadır.
* Nəzərə alın ki, sinif metodları this açar sözündən istifadə
etmir, onlar
* yalnızca öz arqumentləri ilə işləyirlər.
*/
// İki kompleks ədədi toplayır və alınan cəmi qaytarır.
Complex.add = function (a, b) {
return new Complex(a.x + b.x, a.y + b.y);
};
// İki kompleks ədədi vurur və alınan hasili qaytarır.
Complex.multiply = function(a, b) {
return new Complex(a.x * b.x - a.y * b.y,
a.x * b.y + a.y * b.x);
};
// Qabaqcadan müəyyən edilmiş bir neçə kompleks ədəd.
// Bu sinif xüsusiyyətləri “sabitlər” kimi istifadə olunur.
```

```
// (Halbuki, JavaScript-də yalnız oxunmaya əlçatan xüsusiyyət
müəyyən etmək
// mümkün deyil.)
Complex.ZERO = new Complex(0,0);
Complex.ONE = new Complex(1,0);
Complex.I = new Complex(0,1);
```

9.3.7. Şəxsi üzvlər

C++ kimi ənənəvi obyekt yönümlü proqramlaşdırma dilləri kimi, JavaScript-də də şəxsi elan edilmə (private) mövcuddur, hansı ki, bu cür elan nəticəsində metodlar yalnızca sinifin daxilində əlçatan olur. ***Məlumatların inkapsulyasiyası*** adlandırılan məşhur proqramlaşdırma texnikası şəxsi xüsusiyyətlərin yaradılması və bu xüsusiyyətlərə girişi xüsusi metodlar vasitəsi yalnız oxuma/yazma kimi məhdudlaşdırır. JavaScript bu cür davranış qapanmalar (bu mövzu ***bölmə 8.8-də*** müzakirə edilir) vasitəsilə mümkün edir, amma bunu elə etmək lazımdır ki, giriş metodları sinifin hər bir nüsxəsində saxlanılsın və bu səbəbdən də obyekt prototipinə varis oluna bilməsin. Aşağıdakı fraqmentdə bunun qaydası göstərilmişdir. Nümunə, `Rectangle` düzbucaqlı dördbucaq obyektinin realizasiyasını ehtiva edir. Burada obyektə yalnız en və hündürlük əlçatandır və onları yalnız xüsusi metodlara müraciət edərək dəyişdirmək olar:

```
function ImmutableRectangle(w, h) {
 // Bu konstuktor en və hündürlüyünün saxlanıldığı obyekt xüsiyyəti
yaratmır,
 // O sadəcə obyektə əlçatan olan metodları təyin edir
 // Bu metodlar qapanmadan təşkil olunub və en, hündürlük kimi
qiymətlər öz görünmə
 // sahəsi zəncirində saxlanılır.
 this.getWidth = function() { return w; }
 this.getHeight = function() { return h; }
}
// Nəzərə alın ki, sinif, obyekt-prototipində adi metodlar təşkil
oluna bilər.
ImmutableRectangle.prototype.area = function() {
return this.getWidth( ) * this.getHeight();
```

```
};
```

Bu metodikanın ilk açılışı (və ya ilk nəşri), Duqlas Krokforda (Douglas Crockford) aiddir. Bu mövzunu onun müzakirə səhifəsində - ***http://www.crockford.com/javascript/private.html*** tapmaq olar.

9.4. Object sinifinin ümumi metodları

JavaScript-də yeni sinif təyin edilən zaman, sinifin bəzi metodları qabaqcadan müəyyən edilir. Bu metodlar növbəti bölmələrdə təfərrüatı ilə təsvir edilir.

9.4.1. toString() metodu

`toString()` metodunun məğzi ondan ibarətdir ki, sinifdəki hər bir obyekt özünün xüsusi sətir təqdimatına malik olmalıdır və buna görə də obyektlərin sətirə dəyişikliyi üçün uyğun *`toString()`* metodunu müəyyən etmək lazımdır. Yəni sinifi müəyyən edərkən, sinif üçün xüsusi *`toString()`* metodunu təyin etmək lazımdır ki, sinif nüsxələri aydın sətirlərə dəyişdirilə bilsin. Obyekt sətir dəyişikliyi haqqında informasiyanı ehtiva etməlidir, çünki bu kodun gedişatı zamanı lazım ola bilər. Əgər sətir dəyişikliyi üsulu düzgün seçilmişdirsə, o həmçinin özlərində faydalı proqramlar ola bilər. Bundan başqa, *`toString()`* metoduyla qaytarılan sətir dəyişikliyindən sonra parse() statik metodunun şəxsi realizasiyası vasitəsilə əks əməliyyatı yaratmaq olar.
Nümunə 9.2-dəki Complex sinifi artıq *`toString()`* metodunun realizasiyasını ehtiva etdiyinə görə, aşağıdakı fraqmentdə *`toString()`* metodunun Circle sinifi üçün mümkün realizasiyası nümayiş etdirilir:

```
Circle.prototype.toString = function () {
return "[Mərkəzi ("
        + this.x + ", " + this.y + ") nöqtəsində olan çevrənin
radiusu "
         + this.r + " bərabərdir
        .]";
}
```

toString() metodunun bu cür təyinindən sonra Circle obyekti aşağıdakı formada sətirə dəyişdirilə bilər:

```
Mərkəzi (0, 0) nöqtəsində olan çevrənin radiusu 1 bərabərdir
```

9.4.2. valueOf() metodu

valueOf() metodu demək olar ki *toString()* metoduna bənzəyir, amma bu meod obyekti sətirdən başqa hər hansı elementar tipdə (adətən ədəd tipində) olan qiymətə dəyişdirmək tələb olunan zaman çağrılır. Əgər obyekt elementar tipdə olan qiymət kontekstində istifadə olunursa JavaScript interpretatoru bu metodu avtomatik çağırır.

Təyininə görə obyektlər elementar qiymət deyil, buna görə də obyektlərin əksəriyyəti ekvivalent elementar tipə malik deyil. Bunun nəticəsində *Object* sinifiylə susmaya görə müəyyən edilən *valueOf()* metodu hər hansı dəyişiklik yerinə yetirmir və sadəcə çağrıldığı obyekti qaytarır. Number və Boolean və bu kimi siniflərin, aşkar şəkildə elementar ekvivalentliyi mövcuddur, buna görə onlar *valueOf()* metodunu yenidən təyin edirlər ki, metod uyğun olan qiymətləri qaytarsın. Bu səbəbdən Number və Boolean obyektləri bir çox məqamda özünü elementar tipdə olan qiymətlərə ekvivalent göstərə bilər.

Bəzən ağıllı elementar ekvivalentliliyinə malik olan sinif müəyyən edilir. Belə olan halda bu sinif üçün *valueOf()*-un xüsusi metodunu müəyyən etmək zərurəti yaranır. Əgər biz nümunə 9.2-yə qayıtsaq, görərik ki, Complex sinifi üçün *valueOf()* metodu müəyyən edilmişdir. Bu metod sadəcə kompleks ədədin həqiqi hissəsini qaytarır. Buna görə də Complex obyekti özünü, sanki xəyali ədəddən təşkil olunmamış, ədəd kontekstində göstərə bilər. Məsələn, aşağıda fraqmentə baxaq:

```
var a = new Complex(5,4);
var b = new Complex(2,1);
var c = Complex.sum(a,b); // burada c kompleks ədəddir {7,5}
var d = a + b; // d 7-yə bərabərdir
```

Bu metoddan istifadə edərkən bir məqamda ehtiyatlı davranmaq lazımdır: obyektin sətirə dəyişikliyi halında *valueOf()* metodu bəzən *toString()* metodundan yüksək prioritetə malikdir. Buna görə, sinif

üçün *valueOf()* metodu müəyyən edərkən, sinif obyektinin sətirə dəyişdirilməsi halında, *toString()* metodunu sinifin daxilində açıq-aydın çağırışını göstərmək lazımdır. Nümunəni Complex sinifi ilə davam etdirək:

```
alert("c = " + c);          // valueOf() istifadə edir; "c =
7" ekranlaşdırılır
alert("c = " + c.toString()); // "c = {7,5}" ekranlaşdırılır
```

9.4.3. Müqayisə metodları

JavaScript-də müqayisə operatorları obyektləri qiymət üzrə deyil, istinad üzrə müqayisə edir. Belə ki, əgər obyektlərə iki istinad varsa, onların eyni obyektə istinad etdiklərini aydınlaşdırmaq olur, amma müxtəlif obyektlərin eyni xüsusiyyətlərlə eyni qiymətlərə malik olmasını aydınlaşdırmaq mümkün olmur. Obyektlərin ekvivalentliliyini rahatlıqla müəyyən etmək və ya hətta onların ardıcıllıq nizamını təyin etmək mümkündür. Əgər siz sinif müəyyən edərkən bu sinifin nüsxələrini müqayisə etmək istəyirsinizsə, sinif daxilində müvafiq müqayisəni yerinə yetirən metodları müəyyən etmək lazımdır.

Java proqramlaşdırma dilində obyektlərin müqayisəsi metodların köməyi ilə aparılır və bu yanaşmanı müvəffəqiyyətlə JavaScript-də də istifadə etmək olar. Sinif nüsxələrini müqayisə etmək üçün *equals()* adlı nüsxə metodunu müəyyən etmək lazımdır. Bu metod tək arqumenti qəbul edir və əgər verilən arqument metodun çağrıldığı obyektə ekvivalentdirsə, true qiyməti qaytarır. Əlbəttə ki, əvvəlcə, sinif kontekstində "ekvivalent" anlayışını başa düşmək lazımdır.

Adətən obyektlərin bərabərliyi yoxlamaq üçün iki obyekt nüsxəsinin xüsusiyyət qiymətləri müqayisə edilir. Nümunə 9.2-dəki Complex sinifi bu cür formada *equals()* metoduna malikdir. Bəzən müqayisə əməliyyatları obyektlərin ardıcıllıq qaydasını aydınlaşdırmaq üçün tətbiq edilir. Belə ki, bəzi sinif nüsxələrinə kiçik və ya böyük demək olar. Məsələn, Complex sinifindəki obyektlərin ardıcıllıq qaydası magnitude() metodu vasitəsilə qaytarılan qiymət əsasında təyin edilir. Eyni zamanda Circle sinifinin obyektləri üçün "kiçikdir" və "böyükdür" sözlərinin mənasını müəyyən etmək çətindir – bu zaman görəsən hansı parametri -

radius ölçüsünü yaxud X və Y koordinatlarının müqayisəsini nəzərə almaq lazımdır? Bəlkə, hər üç parametri nəzərə almaq lazımdır?
JavaScript-obyektlərini münasibət operatorlarının köməyilə müqayisə edərkən, interpretator əvvəlcə obyektlərin *valueOf()* metodlarını çağıracaq və əgər bu metodlar elementar tipdə olan qiymətləri qaytarsa, bu qiymətlər müqayisə ediləcək. Bir halda ki, Complex sinifi ədədin həqiqi hissəsini qaytaran *valueOf()* metoduna malikdir, Complex sinifinin nüsxələrini xəyali hissəsi olmayan adi həqiqi ədədlər kimi müqayisə etmək olar. Bu sizin seçiminizə uyğun gələ və ya uyğun gəlməyə bilər. Seçiminizə uyğun obyekt nizamını müəyyənləşdirmək üçün (yenə də, Java proqramlaşdırma dilindəki qəbulları örnək götürərək) compareTo() adlı metodu reallaşdırmaq lazımdır. compareTo() metodu tək arqumenti qəbul edir və verilən arqumenti metodun çağrıldığı obyektlə müqayisə edir. Əgər *this* obyekti, compareTo() metoduna arqument qismində ötürülən obyektdən kiçikdirsə, metod sıfırdan kiçik qiymət qaytarmalıdır. Əgər *this* obyekti, compareTo() metoduna arqument qismində ötürülən obyektdən böyükdürsə, metod sıfırdan böyük qiymət qaytarmalıdır. Və əgər hər iki obyekt bərabərdirsə, metod qiyməti sıfıra bərabər qiymət qaytarmalıdır. Qaytarılan qiymət haqqında bu razılaşmalar olduqca əhəmiyyətlidir, çünki münasibət operatorlarının aşağıdakı ifadələrlə əvəzedilməsi mümkündür:

Münasibət ifadələri	**Əvəz edildiyi ifadələr**
`a < b`	`a.compareTo(b) < 0`
`a <= b`	`a.compareTo(b) <= 0`
`a > b`	`a.compareTo(b) > 0`
`a >= b`	`a.compareTo(b) >= 0`
`a == b`	`a.compareTo(b) == 0`
`a != b`	`a.compareTo(b) != 0`

Aşağıda, nümunə 9.2-dəki Complex sinifi üçün compareTo() metodunun mümkün realizasiyalarından biri göstərilmişdir. Burada kompleks ədədlər

modullar üzrə müqayisə edilir:

```
Complex.prototype.compareTo = function(that) {
// Əgər arqumnt ötürülməsə və ya ötürülən arqument magnitude()
metoduna
// əlçatan olmasa istisna yaratmaq lazımdır.magnitude(),
необходимо
// Variant kimi, kompleks ədədin ixtiyari digər ədəddən böyük
və ya kiçik
// olduğunu göstərmək üçün -1, 1 qiymətlərinin qaytarılması
mümkündür.
if (!that || !that.magnitude || typeof that.magnitude !=
"function")
throw new Error("Complex.compareTo() üçün düzgün olmayan
arqument");
// Burada qiymətin kiçikliyini, böyüklüyünü və ya sıfıra
bərabərliyini
// qaytaran çıxma əməliyyatı xüsusiyyətindən istifadə olunur.
// compareTo() metodunun bir sıra realizasiyalarında bu
üsuldan istifadə
// etmək olar.
return this.magnitude() - that.magnitude();
}
```

Sinif nüsxələrinin müqayisə etmə səbəblərindən biri, nüsxə massivinin bir neçə nizamda çeşidləməsi imkanıdır. Array.sort() metodu vacib olmayan arqument şəklində müqayisə funksiyasını qəbul edə bilər, hansı ki, qaytarılan qiymət (məna) haqqında həminki razılaşmaları izləməlidir (getməlidir) ki, və compareTo() metodu. compareTo() metodu mövcudluğu halında təxminən aşağıdakı qaydada kompleks ədəd massivinin çeşidləməsini təşkil etmək olar:

```
complexNumbers.sort(new function(a,b) { return a.compareTo(b);
});
```

Çeşidləmə böyük əhəmiyyətə malikdir və buna görə də istənilən sinifdə compareTo() nüsxə metodunun müəyyən edildiyi yerlərdə statik compare() metodunun reallaşdırmasının imkanına nəzər yetirmək lazımdır. Birinci yazılışı aşağıdakı yazılış vasitəsilə daha sadə realizasiya etmək mümkündür:

```
Complex.compare = function(a,b) { return a.compareTo(b); };
```

Bu metod olduğu halda massiv çeşidləməsi daha sadə realizasiya oluna bilər:

```
complexNumbers.sort(Complex.compare);
```

Nəzərə alın ki, nümunə 9.2-dəki Complex sinifinin təyini zamanı compare() və compareTo() metodlarının realizasiyaları daxil edilməyib. Məsələ ondadır ki, sözügedən metodlar nümunədə təyin *`equals()`* metoduna uyğun gəlmir. *`equals()`* metodu iddia edir ki, Complex sinifinin hər iki obyekti yalnız o halda ekvivalentdir ki, onların həqiqi və xəyali hissələri bərabər olsun. Ancaq compareTo() metodu bərabər modullara malik istənilən iki kompleks ədəd üçün sıfır qiyməti qaytarır. 1+0i və 0+1i ədədlərə eyni modullara malikdir və bu iki ədəd compareTo() metodunun çağırışı zamanı bərabər, *`equals()`* metodunun çağırışı zamanı isə bərabər olmayacaq. Beləliklə, əgər siz eyni sinifdə *`equals()`* və compareTo() metodlarını reallaşdırmağa hazırlaşırsınızsa, bu metodları lazımlı yerlərdə uyğunlaşdırmaq lazımdır. "Bərabərlik" termininin düzgün başa düşülməməsi alqoritmik uyğunsuzluğa və xətalara gətirib çıxara bilər. Gəlin, *`equals()`* metoduna analoji olan compareTo() metodunun realizasiyasına baxaq.

```
// Kompleks ədədlərin müqayisəsi zamanı ilk növbədə həqiqi
hissələr müqayisə
// edilir. Əgər bu hissələr bərabərdisə, xəyali hissələr
müqayisə edilir.
Complex.prototype.compareTo = function(that) {
var result = this.x - that.x;       // Həqiqi ədədlərin çıxma
əməliyatının
                                    // köməyilə müqayisə edilməsi
if (result == 0)                    // Əgər onlar bərabərdisə...
    result = this.y - that.y;       // bu zaman xəyali
hissələr müqayisə edilir
     // İndi nəticə yalnız o halda sıfıra bərabər olacaq ki,
həm həqiqi, həm də
     // xəyali hissələr bərabər olsun.
return result;
};
```

9.5. Üst və altsiniflər

Java, C++ və digər siniflər əsaslı OYP dillərində sərbəst sinif iyerarxiyası konsepsiyası mövcuddur. Hər bir sinif xüsusiyyətlərinin və metodlarının varis olunan üstsinifə malik ola bilər. İstənilən sinif genişləndirilə bilər, yəni onun davranışını varis olduğu yarımsinifə ötürülür. Gördüyünüz kimi, JavaScript siniflər əsasında varisliyin yerinə prototiplər əsasında varisliyi dəstəkləyir. Bununla belə JavaScript-də analoji sinif iyerarxiyasını həyata keçirmək olar. JavaScript-də `Object` sinifi – ən ümumi sinifdir və digər bütün siniflər və ya yarımsiniflər onun ixtisaslaşmış versiyasıdır. Həmçinin `Object` sinifi demək olar ki bütün inteqrasiya edilmiş siniflərin üstsinifidir. Bütün siniflər `Object` sinifinin bir neçə baza metoduna varis olur.

Bilirik ki, obyektlər onların obyekt-prototipinin konstruktorundakı xüsusiyyətlərə varis olur. Bəs, bu xüsusiyyətlər, həmçinin `Object` sinifindən də varis ola bilərmi? Xatırladaq ki, obyektin özü obyekt-prototipindən təşkil olunur; o Object() konstruktorunun köməyilə yaradılır. Bu isə o deməkdir ki, obyekt-prototipindəki xüsusiyyətlərə `Object.prototype`-ə varis olur! Buna görə də Complex sinifinin obyekti Complex.`prototype` obyektinin xüsusiyyətlərinə varis olur, hansı ki, o da öz növbəsində `Object.prototype`-dəki xüsusiyyətlərə varis olur. Complex obyektində hər hansı xüsusiyyətin axtarışı yerinə yetirilən zaman, axtarış əvvəlcə obyektin yerinə yetirilir. Əgər xüsusiyyət tapılmasa, axtarış Complex.`prototype` obyektində davam edir. Və nəhayət, əgər bu obyektdə də xüsusiyyət tapılmasa, axtarış `Object.prototype` obyektində yerinə yetirilir.

Nəzərə alın ki, bir halda ki, axtarış əvvəlcə Complex obyekt-prototipində aparılır, Complex.`prototype` obyektinin xüsusiyyətləri `Object.prototype`-dəki eyni adlı istənilən xüsusiyyətləri gizlədir. Belə ki, nümunə 9.2-də göstərilmiş sinifdə biz Complex.`prototype` obyektində `toString()` metodunu müəyyən etdik. Bu adlı metod həmçinin `Object.prototype`-ində müəyyən edilmişdir, amma Complex obyektləri bunu heç vaxt görməyəcək, çünki, `toString()` ilk olaraq Complex.`prototype`-ində təyin olacaq.

Bu fəsildə göstərilən bütün siniflər, bilavasitə `Object` sinifinin altsiniflər təşkil edir. Bu adətən JavaScript-də proqramlaşdırma üçün; adətən siniflərin daha mürəkkəb iyerarxiyasının yaradılmasında heç bir

ehtiyac yoxdur. Ancaq tələb olunan, istənilən digər sinifin yarımsinifini yaratmaq olar. Fərz edək ki, biz düzbucaqlı dördbucağın koordinatları ilə bağlı xüsusiyyətləri və metodları *`Rectangle`* sinifinə əlavə etmək üçün altsinifini yaratmaq istəyirik. Bunun üçün sadəcə əmin olmalıyıq ki, yeni sinifin obyekt-prototipi *`Rectangle`* nüsxəsini ehtiva edir və buna görə də *`Rectangle.prototype`*-in bütün xüsusiyyətləri varis olunur. Nümunə 9.3-də Rectanle sinifinin sadə təyini göstərilir və sonra da bu təyin yeni *`PositionedRectangle`*-sinifinin yaradılması hesabına genişləndirir.

Nümunə 9.3. *JavaScript-də üstsinifin yaradılması*

```
// Sadə düzbucaqlı dörbucaq sinifinin müəyyən edilməsi.
// Bu sinif, sahənin hesablanmasında istifadə olunan en və
hündürlük
// xüsusiyyətlərini ehtiva edir.
function Rectangle(w, h) {
this.width = w;
this.height = h;
}
Rectangle.prototype.area = function( ) { return this.width *
this.height; }
// Növbədə yarımsinifin təyinidir
function PositionedRectangle(x, y, w, h) {
// İlk növbədə yeni obyektin width və height xüsusiyyətlərinin
// inisializasiyası üçün üstsinifin konstruktorunu çağırmaq
lazımdır.
// Burada call metodu istifadə olunur ki, konstruktor
inisializasiya edilən
// obyekt metodu kimi çağırılsın.
// Bu zəncir üzrə konstruktorun çağırışı adlanır.
Rectangle.call(this, w, h);
// Sonra isə, düzbucaqlı dördbucağın sol  yuxarı bucağının
koordinatları
// saxlanılır.
this.x = x;
this.y = y;
}
// Əgər biz PositionedRectangle() konstruktorunun təyini
zamanı yaradılan
// obyekt-prototipindən susmaya görə istifadə etsək, onda
Object sinifinin
// üstsinifi yaradılacaq.
// Rectangle sinifinin yarımsinifini yaratmaq üçün şübhəsiz
ki, obyekt-
```

```
// prototipini yaratmaq lazımdır.
PositionedRectangle.prototype = new Rectangle();
// Biz obyekt-prototipini varislik məqsədilə yaratdıq, amma
bizə Rectangle
// sinifinin bütün obyektləri üçün əlçatan width və height
xüsusiyyətlərinə
// varis olmaq lazım deyil, buna görə onları prototipdən
siləcəyik.
delete PositionedRectangle.prototype.width;
delete PositionedRectangle.prototype.height;
// Bir halda ki, obyekt-prototipi Rectangle() konstruktorunun
köməyilə
// yaradılıb, onda belə çıxır ki, constructor xüsusiyyəti bu
konstruktora
// istinad edir. Amma bizə lazımdır ki, PositionedRectangle
obyektləri digər
// konstruktora istinad etsin, buna görə də aşağıda
constructor xüsusiyyətinə
// yeni qiymətin mənimsədilməsi baş verir.
PositionedRectangle.prototype.constructor =
PositionedRectangle;
// İndi bizim üstsinifimiz düzgün qurulmuş prototipə malikdir,
odur ki, nüsxə // metodlarının əlavə edilməsinə başlamaq olar.
PositionedRectangle.prototype.contains = function(x,y) {
return (x > this.x && x < this.x + this.width &&
y > this.y && y < this.y + this.height);
}
```

Nümunə 9.3-dən göründüyü kimi, JavaScript-də yarımsiniflərin yaradılması `Object` sinifindən varis olunmadan daha mürəkkəbdir. Birinci problem üstsinif konstruktorunun yarımsinif konstruktorundan çağırılması zərurətidir. Bu zaman həm də üstsinifin konstruktorunu yenidən yaradılmış obyekt metodu kimi çağırmaq lazım olur. Biraz hiylə işlədərək üstsinifin obyekt-prototipinin konstruktorunda dəyişiklik etmək olar. Bizə üstsinifin nüsxəsi kimi obyekt-protipinin yaradılması, sonra bu prototipin constructor xüsusiyyətini dəyişdirilməsi lazımdır.[23] Həmçinin obyekt-prototipdə üstsinifin konstruktoru tərəfindən yaradılan istənilən xüsusiyyətləri silmək istəyi meydana çıxa bilər. Bu zaman obyekt-prototipin xüsusiyyətlərinin onun prototipindən varis oluna bilməsi çox əhəmiyyətlidir.

Bu cür təyinə malik `PositionedRectangle` sinifini, proqramlarda təxminən bu qaydada istifadə etmək olar:

```
var r = new PositionedRectangle(2,2,2,2);
```

```
print(r.contains(3,3));       // Nüsxə metodunun çağırılması
print(r.area( ));       // Varis olunmuş nüsxə metodunun
çağırılması
// Sinifin nüsxəsinin sahələri ilə işləmək:
print(r.x + ", " + r.y + ", " + r.width + ", " + r.height);
// Bizim obyektə 3 sinifinin hər birinin nüsxəsi kimi baxmaq
olar.
print(r instanceof PositionedRectangle &&
    r instanceof Rectangle &&
    r instanceof Object);
```

9.5.1. Konstruktor dəyişikliyi

İndi nümayiş etdirilmiş nümunədə *PositionedRectangle()* funksiya-konstuktoru üstsinifin funksiya-konstuktorunu açıq-aydın çağırmalıdır. Bu zəncir üzrəkonstruktor çağırışı adlanır və altsiniflərin yaradılmasının adi praktikasıdır. Siz altsinifin obyekt-prototipinə superclass xüsusiyyətini əlavə edib, konstruktorun sintaksisini sadələşdirə bilərsiniz:

```
// İstinad üstsinif konstuktorunda saxlanılır.
PositionedRectangle.prototype.superclass = Rectangle;
```

Ancaq qeyd etmək lazımdır ki, bu üsulu dayaz varislik iyerarxiyası şərti ilə istifadə etmək olar. Belə ki, əgər B sinifi A sinifinin, C sinifi isə B sinifinin varisidirsə və hər iki sinifdə (B və C) superclass xüsusiyyətindən müraciət üsulundan istifadə olunursa, C sinifinin nüsxəsinin yaradılması zamanı *this*.superclass istinadı B() konstruktorunda görünəcək və nəticədə B() konstruktorunun sonsuz rekursiv sirkulyasiyası baş verəcək.

Buna görə də çalışın ki hər zaman, sadə altsinifsinif üsulundan deyil, zəncir üzrə konstruktor çağırışı üsulundan istifadə edin (***Nümunə 9.3***-ə nəzər yetirin). Xüsusiyyət müəyyən edildikdən sonra, zəncir üzrə konstruktorun çağırışı sintaksisi əhəmiyyətli dərəcədə sadələşir:

```
function PositionedRectangle(x, y, w, h) {
  this.superclass(w,h);
  this.x = x;
  this.y = y;
```

```
}
```

Nəzərə alın ki, funksiya-konstuktoru *this* obyektinin kontekstində çağırılır. Bu onu bildirir ki, üstsinif konstruktorunun çağırışı üçün obyekt metodu kimi call() və ya *apply()* metodundan istifadədən imtina edilir.

9.5.2. Yenidən müəyyən edilmiş metodların çağırışı

Altsinifdə təyin edilən metodun adı üstsinifdəki mövcud metodun adı ilə eynidirsə, onda altsinif (overrides-ə) bu metodu yenidən təyin edir. Mövcud sinifdən altsinifin yaradılması zamanı bu situasiya ilə sıx rastlaşılır. Məsələn, istənilən vaxt sinifin *toString()* metodunu müəyyən etmək və bununla *Object* sinifinin *toString()* metodunu yenidən təyin etmək olar.

Adətən metodların yenidən təyin edilməsi bu metodların tam dəyişdirilməsi məqsədilə edilmir. Burada yalnızca onların funksionallığını genişləndirmək imkanı nəzərdə tutulur. Metod özünün yenidən təyin edilmiş metodunu çağırmaq imkanına malik olmalıdır. Bu qəbulu konstruktor analogiya üzrə müəyyən mənada metodların zəncir üzrə çağırışı kimi adlandırmaq olar. Ancaq yenidən müəyyən edilmiş metodu çağırmaq üstsinif konstruktorunu çağırmaqdan daha rahatdır.

Aşağıdakı nümunəyə baxaq. Fərz edək ki, *Rectangle* sinifi *toString()* metodunu (hansı ki, bu metod qabaqcadan müəyyən edilmiş metoddur) aşağıdakı qaydada müəyyən edir:

```
Rectangle.prototype.toString = function( ) {
  return "[" + this.width + "," + this.height + "]";
}
```

Əgər siz *Rectangle* sinifində *toString()* metodunu realizasiya etsəniz, onda xüsusi olaraq, *PositionedRectangle* sinifində də bu yenidən təyin etmək lazımdır ki, altsinif nüsxələri yalnızca en və hündürlük qiymətlərinin deyil, bütün sinif xüsusiyyətlərinin sətir ilə təqdim edilməsinə malik olsun. *PositionedRectangle* – çox sadə sinifdir və onun bütün xüsusiyyətlərini qaytarmaq üçün *toString()* metodu kifayətdir. Ancaq nümunəyə görə koordinat xüsusiyyətlərinin qiymətlərini sözügedən sinifin özündə, *width* və *height*

xüsusiyyətləri isə üstsinifə göndərilmiş nümayəndə vasitəsilə emal ediləcək. Bunu təxminən aşağıdakı qaydada etmək olar:

```
PositionedRectangle.prototype.toString = function() {
 return "(" + this.x + "," + this.y + ") " +       // bu
sinifin sahəsi
 Rectangle.prototype.toString.apply(this);         //
üstsinifin zəncir üzrə
                                                   // çağırışı
}
```

`toString()`-i metodunun üstsinifdə realizasiyası, üstsinifin obyekt-prototipinin xüsusiyyəti kimi də əlçatandır.
Nəzərə alın ki, biz metodu birbaşa çağıra bilmərik – bunun üçün bizə *`apply()`* metodundan istifadə etmək lazım oldu. Ancaq əgər *`PositionedRectangle.prototype`*-inə superclass xüsusiyyətini əlavə etsək, onda elə etmək olar ki, kod üstsinif tipindən asılı olmasın:

```
PositionedRectangle.prototype.toString = function( ) {
  return "(" + this.x + "," + this.y + ") " +       // bu
sinifin sahəsi
  this.superclass.prototype.toString.apply(this);
}
```

Bir daha nəzərə alın ki, superclass xüsusiyyəti varislik iyerarxiyasında yalnız bir dəfə istifadə oluna bilər. Əgər bu xüsusiyyət sinifə və onun altsinifə cəlb ediləcəksə, bu sonsuz rekursiyaya gətirib çıxaracaq.

9.6. Varislik olmadan genişlənmə

Altsiniflərin yaradılması probleminin əvvəlki müzakirəsində, digər sinif metodlarına varis olan yeni siniflərin yaradılması sırası təsvir edilir. JavaScript dili o qədər elastik ki, altsiniflərin yaradılması və varislik mexanizmi olmadan belə siniflərin funksional imkanlarının genişlənilməsi mümkündür. Bir halda ki, JavaScript-də funksiyalar – sadəcə məlumat qiymətidir, onda onları bir sinifdən başqa bir sinifə asanlıqla ötürmək olar. ***Nümunə 9.4***-dəki funksiya bir sinifin bütün metodlarını alır və onların surətlərini başqa sinifin obyekt-prototipində yaradır.

Nümunə 9.4. *Bir sinifdəki metodun digər siniflər də istifadəsi üçün metodun alınması*

```
// Bir sinifdəki metodun digər siniflər də istifadəsi üçün
metodun alınması
// Arqumentlər siniflərin funksiya-konstuktoru olmalıdır.
// Object, Array, Date və RegExp kimi inteqrasiya edilmiş
siniflər sadalanan
// olmadığı üçün bu funksiyada işləmir
function borrowMethods(borrowFrom, addTo) {
  var from = borrowFrom.prototype; // prototip-mənbə
  var to = addTo.prototype; // prototip-qəbuledici
  for(m in from) { // prototip-mənbədəki bütün xüsusiyyətlərin
dövrü
    if (typeof from[m] != "function") continue; // Funksiya
olmayan
                                                // xüsusiyyətlərə
məhəl
                                                // qoymamaq
    to[m] = from[m]; // Metodu almaq
  }
}
```

Bir çox metodlar siniflə o qədər sıx əlaqəlidir ki, onların digər siniflərdə istifadəsi mənasızdır. Ancaq elə metodlar var ki, onlar kifayət qədər universaldır və istənilən sinifdə işləyə bilər. Nümunə 9.5-də təyin edilən iki sinifin ayrılıqda heç bir faydası yoxdur, amma bu siniflərdə realizasiya olunan metodları, başqa siniflər üçün tətbiq bilər. Xüsusi olaraq alınma məqsədi ilə hazırlanan belə siniflər sinif-qarışığı və ya sadəcə qarışıq siniflər adlanır.

Nümunə 9.5. Alınma üçün nəzərdə tutulmuş universal metodlar ehtiva edən sinif qarışığı

```
// Öz-özlüyündə bu sinif o qədər də yaxşı deyil. Ancaq bu
sinif özündə
// universal toString() metodunu ehtiva edir ki, bu metod
digər siniflərdə də istifadə oluna bilər.
function GenericToString() {}
GenericToString.prototype.toString = function( ) {
  var props = [];
  for(var name in this) {
```

```
    if (!this.hasOwnProperty(name)) continue;
    var value = this[name];
    var s = name + ":"
    switch(typeof value) {
      case 'function':
           s += "function";
           break;
      case 'object':
           if (value instanceof Array) s += "array"
           else s += value.toString( );
           break;
      default:
           s += String(value);
           break;
    }
    props.push(s);
  }
  return "{" + props.join(", ") + "}";
}
// Növbəti sinifdə sadəcə obyektləri müqayisə edən equals()
metodu müəyyən
// edilir.
function GenericEquals() {}
GenericEquals.prototype.equals = function(that) {
   if (this == that) return true;
    // obyektlər o halda bərabərdir ki, this və that
obyektləri eyni
    // xüsusiyyətləri ehtiva etsin və bu xüsusiyyətlərdən
başqa xüsusiyyətə
    // malik olmasın.
    // Nəzərə alın ki, burada dərin müqayisəyə lüzum yoxdur.
    // Qiymətlər sadəcə bir-birinə === (eynilik) olmalıdır.
Buradan belə
    // nəticə çıxır ki, əgər başqa obyektlərə istinad edən
xüsusiyyətlər
    // varsa, bu xüsusiyyətlər equals() metodunun true
qaytardığı obyektlərə
    // deyil, mövcud ən yuxarı obyektə istinad etməlidir.
   var propsInThat = 0;
   for(var name in that) {
      propsInThat++;
      if (this[name] !== that[name]) return false;
   }
    // İndi əmin olmaq lazımdır ki, this obyekti əlavə
xüsusiyyətlərə malik
    // deyil.
   var propsInThis = 0;
   for(name in this) propsInThis++;
```

```
    // Əgər this obyekti əlavə xüsusiyyətlərə malikdirsə,
  onda, obyektlər
    // bərabər deyil.
    if (propsInThis != propsInThat) return false;
    // İki obyekt bərabər götürülərkən...
    return true;
}
```

Deməli belə, sinif-qarışığından *`toString()`* və *`equals()`* metodunu alan sadə *`Rectangle`* sinifi aşağıdakı formada olur:

```
// Sadə Rectangle sinifi
function Rectangle(x, y, w, h) {
  this.x = x;
  this.y = y;
  this.width = w;
  this.height = h;
}
Rectangle.prototype.area = function( ) { return this.width *
this.height; }
// Bir neçə metodun alınması
borrowMethods(GenericEquals, Rectangle);
borrowMethods(GenericToString, Rectangle);
```

Burada təqdim edilmiş sinif-qarışığından heç biri şəxsi konstruktora malik deyil, ancaq bu o demək deyil ki, konstruktorları almaq olmaz. Aşağıdakı fraqmentdə *`ColoredRectangle`* adlı yeni sinif təyini edilir. O *`Rectangle`* sinifinin funksionallığına varis olur və *`Colored`* sinif-qarışığından konstruktor və metod alır:

```
// Bu sinif-qarışığı konstruktordan asılı olan metodu ehtiva
edir. Onların hər ikisi, konstruktor və metod alınmış
olmalıdır.
function Colored(c) { this.color = c; }
Colored.prototype.getColor = function() { return this.color; }
// Yeni sinifin konstruktorunun müəyyən edilməsi
function ColoredRectangle(x, y, w, h, c) {
  this.superclass(x, y, w, h); // Üstsinif konstruktorunun
çağırışı və
  Colored.call(this, c);       // Colored konstruktorunun
alınması
}
```

```
// Rectangle sinifindəki metodlara obyekt-prototipinin
varisliyinin
// tənzimlənməsi
ColoredRectangle.prototype = new Rectangle();
ColoredRectangle.prototype.constructor = ColoredRectangle;
ColoredRectangle.prototype.superclass = Rectangle;
// Colored sinifindəki metodların yeni sinifə nəqli
borrowMethods(Colored, ColoredRectangle);
```

`ColoredRectangle` sinifi `Rectangle` sinifini genişləndirir (və onun metodlarına varis olur), həmçinin `Colored` sinifinin metodlarını alır. `Rectangle` sinifinin özü isə `Object` sinifinə varis olur və GenericEquals və GenericToString siniflərinin metodlarını alır. Hərçənd belə məqamda bu cür analogiyalar yersizdir, amma bunu bir növ "*çoxsaylı varislik*" adlandırmaq olar. Çünki `ColoredRectangle` sinifi `Colored` sinifi metodlarını alır, `ColoredRectangle` sinifinin nüsxələrinə eyni zamanda `Colored` sinifinin nüsxələri kimi baxmaq olar. instanceof operatoru bunu müəyyən edə bilmir, amma bölmə 9.7.3-də yaratdığımız universal metod vasitəsilə göstərilmiş sinifdəki obyekt metodunun varis olunduğunu və kənardan alındığını müəyyən etmək olur.

9.7. Obyektin tipinin təyini

JavaScript dili – qismən tipləşdirilmiş dildir və bundan dolayı JavaScript obyektləri qismən tipləşdirilmişdir. Bununla belə JavaScript-də bir neçə qəbul mövcuddur ki, ixtiyari qiymətin tipinin təyininə xidmət edə bilər. Əlbəttə, ən yayılmış qəbul typeof operatorundan (***bölmə 5.10.2***) istifadədir. İlk növbədə typeof obyektləri və elementar tipləri ayırmağa imkan verir, ancaq bu operator bəzi qəribəliklərə malikdir. Birincisi, typeof `null` ifadəsində nəticə olaraq "object" sətirini verir, halbuki `undefined` ifadəsində "`undefined`" sətirini qaytarır. Bundan başqa istənilən massiv tipini "object" sətiri kimi qeyd edir, bu anlaşılandır ki, bütün massivlər obyektlər təşkil olunub, ancaq ixtiyari funksiya üçün bu operator "function" sətiri qaytarır, hərçənd ki, funksiyalar da faktiki olaraq obyektdir.

9.7.1. instanceof operatoru və konstruktor

Hər hansı qiymətin elementar qiymət və funksiyadan deyil, obyektdən ehtiva olunduğu aydınlaşdırdıqdan sonra, bu qiyməti instanceof operatoruna verərək onun mahiyyətli ətraflı onun öyrənmək olar. Məsələn, x massivdirsə, aşağıdakı true qaytaracaq:

```
x instanceof Array
```

instanceof operatorundan solda yoxlanan qiymət, sağda isə obyektlərin sinifini müəyyən edən funksiya-konstruktorunun adı yerləşir. Nəzərə alın ki, şəxsi-obyekt sinifin və onun bütün üstsiniflərinin nüsxəsi kimi qiymətləndirilir. Beləliklə, istənilən o obyekti üçün o instanceof *Object* ifadəsi həmişə true qiyməti qaytaracaq. Maraqlıdır ki, instanceof operatoru funksiyalarla da işləyə bilir. Belə ki, aşağıda göstərilən bütün ifadələr true qiymətini qaytarır:

```
typeof f == "function"
f instanceof Function
f instanceof Object
```

Ehtiyac olduğu halda hər hansı sinifin bir altsinifin deyil, müəyyən sinifin nüsxəsi oluduğuna əmin olmaq olar – bunun üçün constructor xassə mənasını yoxlamaq kifayətdir:

```
var d = new Date();       // Date obyekti; Date – Object
sinifinin altsinifi
var isobject = d instanceof Object;       // true qaytarır
var realobject = d.constructor==Object; // false qaytarır
```

9.7.2. Object.toString() metodunun köməyilə obyektin tipinin təyini

instanceof operatorunun çatışmazlığı və constructor xüsusiyyətləri yalnız sizə məlum olan sinifləri yoxlamağa imkan verir, amma onlar məsələn, hazırlanma prosesində naməlum obyektlərin tədqiqatı zamanı heç bir faydalı məlumata verə bilmir. Belə vəziyyətdə *Object.toString()* metodu köməyinizə gələ bilər.

7-ci fəsildə deyildiyi kimi, *Object* sinifi *toString()* metodunu susmaya görə ehtiva edir. Şəxsi metod müəyyən etməyən istənilən sinif, susmaya görə bu metoda varis olur. Susmaya görə *toString()* metodun maraqlı xüsusiyyəti mövcuddur ki, sözügedən metod inteqrasiya edilmiş obyektlərin tipi haqqında bir neçə daxili informasiyanı ekranlaşdırır. ECMAScript spesifikasiyası tələb edir ki, *toString()* metodu susmaya görə həmişə məlumatı sətir formatında qaytarsın:

```
[object class]
```

Burada class – obyektin daxili tipidir ki, adətən bu obyektin funksiya-konstuktorunun adına uyğun olur. Məsələn, class, massivlər üçün – "Array", funksiyalar üçün – "Function", tarix/vaxt obyektləri üçün – "Date", inteqrasiya edilmiş Math sinifi üçün "Math" və Error ailəsinin bütün sinifləri üçün – "Error" sətiri olacaq.

JavaScript reallaşdırmasıyla müəyyən edilən JavaScript-in kliyent dilinin obyektlərində və digər ixtiyari obyektlərdə class sətirində realizasiyasiyaya müvafiq sətir (məsələn, "Window", "Document" və ya "Form"). Ancaq daha əvvəl nümayiş etdirilən Circle və Complex kimi istifadəçi tərəfindən müəyyən edilən obyektlərin tipləri üçün class sətirində həmişə "Object" sətiri qaytarılır. Yəni *toString()* metodu obyektlərin yalnız əvvəlcədən qurulmuş tipləri müəyyən edə bilir.
Bir halda ki, siniflərin əksəriyyətində susmaya görə *toString()* metodu yenidən təyin edilir, onda bu metodu bilavasitə obyektdən çağırmaqla obyektin tipini müəyyən etməyi gözləməyin. Bunun yerinə susmaya görə olan *Object*.*prototype* funksiya müraciət etmək və tələb olunan obyekti *apply()* metodu ötürərək onun hansı tipdə olduğunu öyrənmək olar:

```
Object.prototype.toString.apply(o);      // Hər zaman susmaya görə toString()
                                         // metodunun çağırılması
```

Bu qəbuldan ***nümunə 9.6***-da tipin müəyyənləşdirilməsi üzrə genişləndirilmiş imkanlara malik olan funksiyanın təyinində istifadə olunur. Əvvəl də qeyd edildiyi kimi, *toString()* metodu istifadəçi

siniflәriylә işlәmir, belә olan halda aşağıda göstәrilmiş funksiya classname xüsusiyyәtinin qiymәtini yoxlayır vә xüsusiyyәt müәyyәn edildiyi halda onun qiymәtini qaytarır.

```
function getType(x) {
  // Əgər x qiyməti null-a bərabərdisə, "null" sətiri
qaytarılır.
  if (x == null) return "null";

  // typeof operatorunun köməyilə tipin müəyyən edilməsini
yoxlamaq.
  var t = typeof x;

  // Əgər anlaşılmayan nəticə əldə olunsa, t-nin özünü
qaytarmaq
  if (t != "object") return t;

  // Əks təqdirdə, x – obyektdir. Susmaya görə toString()
metodunu susmaya
  // çağırılması və sinfin adının sətiraltı ekranlaşdırılması.
  var c = Object.prototype.toString.apply(x); // "[object
class]" formatında
  c = c.substring(8, c.length-1); // "[object" və "]"
silinməsi

  // Əgər sinifin adı Object deyilsə, c-nin özünü qaytarmaq.
  if (c != "Object") return c;

  // Əgər "Object" tipi alınarsa, x-in həqiqətən bu sinifə aid
olduğunun
  // yoxlanılması
  if (x.constructor == Object) return c; // Həqiqi "Object"
tipi

  // İstifadəçi tərəfindən yaradılmış sinifləri üçün obyekt-
prototipindən
   // varis olan classname xüsusiyyətinin  sətir qiymətinin
çıxardılması.
  if ("classname" in x.constructor.prototype && // varis
olunmuş sinifin adı
       typeof x.constructor.prototype.classname == "string") //
bu sətirdir
       return x.constructor.prototype.classname;

  // Əgər tip müəyyən etmək mümkün deyilsə, aşağıdakı sətir
qaytarılır.
  return "<unknown type>";
```

```
}
```

9.7.3. Kobud tip təyini

Belə bir köhnə fikir mövcuddur: "Əgər hər hansı canlı ördək kimi yeriyirsə və ördək kimi səs çıxarırsa, deməli bu canlı ördəkdir!". Bu aforizmi JavaScript dilinə bu aforizmi uyğunlaşdırmaq kifayət qədər çətindir, ancaq gəlin yoxlayaq: "Əgər obyektdə ixtiyari sinifin bütün metodları reallaşdırılmışsa, deməli, bu obyekt sinifin nüsxəsidir". JavaScript kimi zəif tipləşdirilmiş elastik proqramlaşdırma dillərində, bu "kobud tiq təyini" adlanır: əgər obyekt X sinifinin bütün xüsusiyyətlərinə malikdirsə, onda bu obyektə X sinifinin nüsxəsi kimi baxmaq olar (hətta əgər bu obyekt X() funksiya-konstuktoru ilə yaradılmasa belə).[24]
Kobud tip təyininin, metodlarını başqa siniflərdən "alan" siniflərdə istifadəsi xüsusilə rahatdır. Fəsilin əvvəlində təyin edilən GenericEquals adlı sinifdə `equals()` metodunu alan `Rectangle` sinifi nümayiş etdirilmişdir. `Rectangle` sinifinin istənilən nüsxəsinə GenericEquals sinifinin nüsxəsi kimi baxmaq olar. instanceof operatoru bu faktı müəyyən edə bilmir, amma bizim yaratdığımız bu xüsusi metod (***Nümunə 9.7***) bu faktı müəyyən edə bilir.

Nümunə 9.7. *Verilmiş sinif metodlarının obyektlə alınması faktının yoxlaması*

```
// Əgər c.prototype metodlarının hər biri o obyektiylə
alınarsa true qiymətini
// qaytarır.Burada o – obyekt deyil, funksiyadır. o obyektinin
özünün yerinə
// onun prototipi yoxlanılır.
// Nəzərə alın ki, bu funksiya üçün metodların təkrar
realizasiyası deyil,
// kopiyalanması lazımdır. Əgər sinif, metodu aldısa və bundan
sonra metodu
// yenidən təyin etdisə, bu funksiya false qiymətini
qaytaracaq.
function borrows(o, c) {

   // Əgər o obyekti artıq c sinifinin nüsxəsidirsə, true
  qiymətini qaytarmaq
   // olar
```

```
  if (o instanceof c) return true;

   // İnteqrasiya edilmiş sinif metodlarının alınması faktının
yoxlamasını
   // tam olaraq yerinə yetirmək mümkün deyil, çünki,
inteqrasiya edilmiş
   // tiplərin metodları sadalana bilmir. Belə olan halda
istisana yaradaraq
   // undefined qiymətini qaytarmaq olar. undefined qiyməti
bir çox məqamda
   // özünü false qiyməti kimi aparır, amma əgər undefuned
çağıran proqrama
   // lazımdısa zaman false-dən fərqlənə bilər.
  if (      c == Array || c == Boolean || c == Date || c ==
Error ||
           c == Function || c == Number || c == RegExp || c ==
String)
  return undefined;
  if (typeof o == "function") o = o.prototype;
  var proto = c.prototype;
  for(var p in proto) {

    // Funksiya olmayan xüsusiyyətlərə nəzərə almamaq
    if (typeof proto[p] != "function") continue;
    if (o[p] != proto[p]) return false;
  }
  return true;
}
```

Nümunə 9.7-dəki borrows() metodu kifayət qədər məhduddur: sözügedən metod, yalnız o obyekti c sinifi ilə müəyyən edilən metodların dəqiq surətlərinə malik olan zaman, true qiymətinə qaytarır. Reallıqda isə, kobud tip təyini daha ustalıqla işlənməlidir:

объект o должен рассматриваться как экземпляр класса c, если содержит методы, напоминающие методы класса c. В JavaScript «напоминающие» означает «имеющие те же самые имена» и (возможно) «объявленные с тем же количеством аргументов». В примере 9.8 демонстрируется метод, реализующий такую проверку.

Əgər o obyekti c sinifinin metodlarına xatırladan metodları ehtiva edirsə, onda o obyektinə c sinifinin nüsxəsi kimi baxılmalıdır. JavaScript-də "xatırladan" dedikdə, "eyni adlara malik" və (ola bilər ki,) "eyni arqument

miqdarına malik" başa düşülür. Nümunə 9.8-də bu cür yoxlamanı reallaşdıran metod nümayiş etdirilir.

Nümunə 9.8. *Eyni adlı metodların mövcudluğunun yoxlanılması*

```
// Əgər o obyekti c.prototype sinifi ilə eyni adlara və eyni
arqument
// miqdarına malik metodlar ehtiva edirsə true qiməti
qaytarılır. Əks təqdirdə
// false qiyməti qaytarılır. Əgər c sinifi sadalana bilməyən
inteqrasiya
// edilmiş tipə aid olan metod ehtiva edirsə, istisna
yaradılır.
function provides(o, c) {
  // Əgər o obyekti artıq c sinifinin nüsxəsidirsə, onda o
obyekti c sinifini
  // "xatırladır".
  if (o instanceof c) return true;
  // Əgər obyektin yerinə obyekt konstruktoru ötürülərsə,
obyekt-prototipdən
  // istifadə etmək
  if (typeof o == "function") o = o.prototype;

  // İnteqrasiya edilmiş siniflər metodları sadalana bilmir,
buna görə də
  // undefined qiymətini qaytarılır. Əgər undefined qiyməti
qaytarılmasa,
  // istənilən obyekt hər hansı inteqrasiya edilmiş tipi
xatırladacaq.
  if (      c == Array || c == Boolean || c == Date || c ==
Error ||
           c == Function || c == Number || c == RegExp || c ==
String)
     return undefined;
  var proto = c.prototype;
  for(var p in proto) { // c.prototype xüsusiyyətlərinin dövrü
    // Funksiya olmayan xüsusiyyətlərə nəzərə almamaq
    if (typeof proto[p] != "function") continue;
    // Əgər o obyekti eyni adlı xüsusiyyət ehtiva etmirsə,
false qiyməti
    // qaytarmaq
    if (!(p in o)) return false;
    // Əgər bu funksiya deyil, xüsusiyyətdirsə, false qiyməti
qaytarmaq
    if (typeof o[p] != "function") return false;
    // Əgər hər iki funksiya müxtəlif miqdarda arqumentlərlə
elan
```

```
    // edilmişdirsə, false qiymətini qaytarmaq.
    if (o[p].length != proto[p].length) return false;
  }
  // Bütün hallar yoxlanıldıqdan sonra true qiymətini
qaytarmaq olar.
  return true;
}
```

Kobud tip təyininin və provide() metodundan istifadə nümunəsi kimi ***bölmə 9.4.3***-də təsvir edilmiş compareTo() metoduna baxaq. Bir qayda olaraq, compareTo() metodu alınma üçün nəzərdə tutulmamışdır, amma bəzən compareTo() metodunun köməyilə obyektlərin müqayisə imkanına malik olmasını yoxlamaq tələb olunur. Bu minvalla Comparable sinifini müəyyən edəcəyik:

```
function Comparable( ) {}
Comparable.prototype.compareTo = function(that) {
  throw "Comparable.compareTo() – abstrakt metoddur. Çağırıla
bilmir!";
}
```

Comparable sinifi abstrakt sinifdir: onun metodları çağırış üçün nəzərdə tutulmamışdır, o sadəcə tətbiq interfeysini müəyyən edir. Ancaq bu sinifin təyini olduğu halda iki obyektin müqayisəsi mümkünlüyünü yoxlamaq olar:

```
// o və p obyektlərinin müqayisəsinin mümkünlüyünü yoxlamaq
// Onlar bir tipə aid olmalıdır və compareTo() metoduna malik
olmalıdır
if (o.constructor == p.constructor && provides(o, Comparable))
{
   var order = o.compareTo(p);
}
```

Nəzərə alın ki, bu bölmədə təqdim edilmiş, hər iki funksiyaya - borrows() və provides() funksiyasına JavaScript-in inteqrasiya edilmiş tiplərindən birinə aid olan obyekt (məs., Array) ötürülürsə funksiya `undefined` qiymətini qaytarır. Bunun səbəbi odur ki, inteqrasiya edilmiş tiplərin obyekt-prototiplərinin xüsusiyyətləri for/in dövründə sadalana bilmir.

9.8. Nümunə: defineClass() köməkçi metodu

Bu fəsil, konstruktorlar, prototiplər, yarımsiniflər, metodların alınması və verilməsi haqqında müzakirə edilmiş mövzuları özündə təcəssüm etdirən defineClass()-ın köməkçi metodunun təyin edilməsi ilə bitir. Metodun realizasıyası nümunə 9.10-da göstərilmişdir.

Nümunə 9.10. *Siniflərin təyini üçün köməkçi funksiya*

```
/**
* defineClass()–JavaScript-siniflərinin təyin edilməsi üçün
köməkçi funksiya.
*
* Bu funksiya tək arqument şəklində obyekti almağa gözləyir.
 * Funksiya  bu obyektdəki məlumatlara əsasən yeni JavaScript-
sinifi müəyyən
* edir və yeni sinifin funksiya-konstruktorunu qaytarır. Bu
funksiya
* siniflərin təyiniylə bağlı məsələləri həll edir: obyekt-
prototipdə varislik
* düzgün qurur, başqa siniflərdən metodları köçürür və s.
*
* Arqument kimi verilən aşağıda tələblərin bəzilərinə cavab
verməlidir:
*
* name:             Müəyyən edilən sinifin adı.
*             Əgər ad müəyyən edilərsə, bu ad obyekt-
 prototipin classname
*            xüsusiyyətində saxlanacaq.
*
*
* extend:       Varis olunan sinifin konstruktoru. Yoxluq
halında Object()-
*             konstruktorundan istifadə ediləcək. Bu qiymət
obyekt-
*            prototipin superclass xüsusiyyətində saxlanacaq.
*
*
* construct:       Sinifin funksiya-konstruktoru. Yoxluq
               halında yeni boş
*            funksiyadan istifadə ediləcək. Bu qiymət
               funksiyanın
*            qaytarılan qiyməti olacaqvə bu obyekt-
               prototipinin
```

```
*            constructor-u xüsusiyyətində saxlanılacaq.
*
* methods:       Sinif nüsxəsinin metodlarını (və müxtəlif
                 nüsxələrlə birgə
*            istifadə edilən başqa xüsusiyyətlərləri) müəyyən
                 edən obyekt.
*             Bu obyektin xüsusiyyətləri sinifin obyekt-
prototipinə
*            kopiyalanacaq. Yoxluq halında boş obyektdən
istifadə
*            ediləcək.
*
*
*                  "classname" , "superclass" və "constructor"
xüsusiyyət
*                  adları ehtiyat saxlanılmışdır və bu
obyektdə istifadə
*                  olunmamalıdır.
*
* statics:            Statik metodları ( və digər statik
*                   xüsusiyyətləri) müəyyən edən  sinif. Bu
obyektin
*                  xüsusiyyətləri funksiya-konstruktorun
xüsusiyyəti
*                  olacaq. Yoxluq halında boş obyektdən
istifadə
*                  ediləcək.
*
* borrows:            Funksiya-konstruktor və ya funksiya-
konstruktorların
*                  massivi.
*                  Verilmiş siniflərdən hər birinin nüsxə
*                  metodları bu yeni sinifin obyekt-
prototipinə
*                  kopiyalanacaq, beləliklə yeni sinif
verilmiş
*                  siniflərin hər biri metodlarını alacaq.
Konstruktorlar
*                  sıra ilə emal edilir, bunun nəticəsində
massivin
*                  sonunda duran sinif metodları, yuxarıda
duran sinif
*                  metodlarını yenidən təyin edə bilər. Nəzərə
alın ki,
*                   xüsusiyyətlər və obyektdən yuxarıda
göstərilən
```

```
     *                  metodlar kopiyalana qədər alınan metodlar
     obyekt-
     *                  prototipində saxlanılır.
     *                  Buna görə də, bu obyektlərlə müəyyən edilən
     metodlar
     *                  alınan metodları yenidən təyin edə bilər.
     Bu
     *                  xüsusiyyətin yoxluğunda metodların alınması
     baş
     *                  vermir.
     *
     * provides:                  Funksiya-konstruktor və ya
                        funksiya-konstruktorlarının
     *                        massivi.
     *                  Obyekt-prototip inisializasiya edildikdən
     sonra, bu
     *                  funksiya prototiplə göstərilən sinif
     nüsxələri
     *                  arasında eyni adlı və eyni arqument
     miqdarlı
     *                  metodların mövcudluğunu yoxlayır. Burada
     heç bir metod
       *                  kopiyalanmayacaq,sadəcə əmin olmaq
lazımdır ki, bu
       *                  sinif göstərilən siniflə təmin edilən
funksionallığa
       *                  "imkan verir". Əgər yoxlama uğursuz
olssa, bu metod
       *                  istisna yaradacaq. Əks təqdirdə yeni
sinifin istənilən
       *                  nüsxəsinə göstərilən tiplərin nüsxəsi
kimi  baxmaq
       *                  olar (kobud tip təyini metodikasından
istifadə
       *                  etməklə). Əgər bu xüsusiyyət müəyyən
edilməməsə,
       *                  yoxlama yerinə yetirilməyəcək.
     **/
     function defineClass(data) {
       // Obyekt-arqumentindən qiymət sahələri çıxartmaq.
       // Susmaya görə qiymətlər təyin etmək.
       var classname = data.name;
       var superclass = data.extend || Object;
       var constructor = data.construct || function( ) {};
       var methods = data.methods || {};
       var statics = data.statics || {};
       var borrows;
       var provides;
```

```
  // Alınma tək konstruktordan deyil, həm də konstruktorlar
massivindən
  // həyata keçir.
  if (!data.borrows) borrows = [];
  else if (data.borrows instanceof Array) borrows =
data.borrows;
  else borrows = [ data.borrows ];

  // Eyni verilən xüsusiyyətlər üçün.
  if (!data.provides) provides = [];
  else if (data.provides instanceof Array) provides =
data.provides;
  else provides = [ data.provides ];

  // Sinif prototipi olan obyektin yaradılması.
  var proto = new superclass();

  // Varis olunmamış bütün xüsusiyyətləri yeni obyekt-
prototipindən silmək.
  for(var p in proto)
        if (proto.hasOwnProperty(p)) delete proto[p];
  // Sinif-qarışıqlarından metodları alaraq, prototipə
kopiyalamaq.
  for(var i = 0; i < borrows.length; i++) {
     var c = data.borrows[i];
     borrows[i] = c;
     // c obyektinin prototipindəki metodları bizim
prototipimizə
     // kopiyalamaq
     for(var p in c.prototype) {
        if (typeof c.prototype[p] != "function") continue;
        proto[p] = c.prototype[p];
     }
  }
// Obyekt-prototipinəə nüsxə metodlarını kopiyalamaq
// Bu əməliyyat sinif-qarışıqlarından kopiyalanmış metodları
yenidən təyin
// edə bilər
for(var p in methods) proto[p] = methods[p];

// "constructor", "superclass" və "classname" kim ehtiyata
saxlanılmış
// xüsusiyyət qiymətlərini prototipdə yerləşdirmək
proto.constructor = constructor;
proto.superclass = superclass;
```

```
// classname xüsusiyyətini yalnız verildiyi zaman
yerləşdirmək.
if (classname) proto.classname = classname;

// Əmin olmaq ki, prototip güman edilən bütün metodları verir.
for(var i = 0; i < provides.length; i++) { // hər bir sinifdə
   var c = provides[i];
   for(var p in c.prototype) { // hər bir xüsusiyyətdə
      if (typeof c.prototype[p] != "function") continue; //
ancaq metodlarda
      if (p == "constructor" || p == "superclass") continue;
      // Eyni adla və eyni miqdarlı metodların mövcudluğunu
yoxlamaq.
      // Əgər belə metod varsa, dövrü davam etmək.
      if (p in proto &&
         typeof proto[p] == "function" &&
         proto[p].length == c.prototype[p].length) continue;
      // Əks təqdirdə istisna yaratmaq
      throw new Error("Класс " + classname + " не
предоставляет метод "+
                      c.classname + "." + p);
   }
 }

 // Funksiya-konstruktorla obyekt-prototipini əlaqələndirmək.
 constructor.prototype = proto;
 // Statik xüsusiyyətləri konstruktora kopyalamaq
 for(var p in statics) constructor[p] = data.statics[p];
 // Və axırda funksiya-konstruktorunu qaytarmaq
 return constructor;
}
```

Nümunə 9.11-də defineClass() metodundan istifadə nümunəsi göstərilmişdir.

Nümunə 9.11. *defineClass() metodundan istifadə*

```
// Comparable interfeysinə "malik" sinifləri müəyyən edə
bilən, abstrakt
// metodlar ehtiva edən Comparable sinifi.
var Comparable = defineClass({
  name: "Comparable",
  methods: { compareTo: function(that) { throw "abstract"; } }
});
// Alınma üçün nəzərdə tutulmuş universal equals() metoduna
malik sinif-
// qarışığı
```

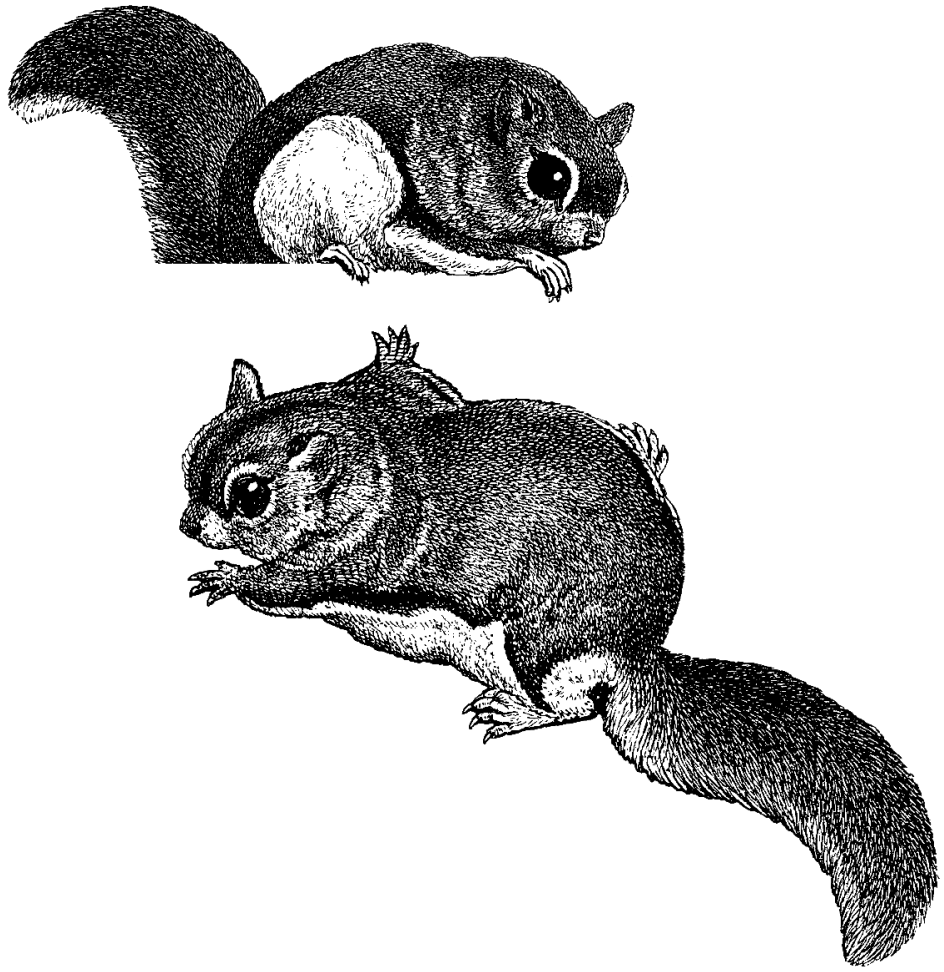
```
var GenericEquals = defineClass({
  name: "GenericEquals",
  methods: {
      equals: function(that) {
         if (this == that) return true;
         var propsInThat = 0;
         for(var name in that) {
            propsInThat++;
            if (this[name] !== that[name]) return false;
         }
         // Əmin olmaq lazımdır ki, this obyekti əlavə
xüsusiyyətlərəə
         // malik deyil
         var propsInThis = 0;
         for(name in this) propsInThis++;
         // Əgər əlavə xüsusiyyətlər varsa, obyektlər bərabər
olmayacaq
         if (propsInThis != propsInThat) return false;
         // Görünür ki, iki obyekt ekvivalentdir.
         return true;
      }
   }
});
// Comparable interfeysinə malik  çox sadə Rectangle  sinifi
var Rectangle = defineClass({
name: "Rectangle",
construct: function(w,h) { this.width = w; this.height = h; },
methods: {
area: function() { return this.width * this.height; },
compareTo: function(that) { return this.area( ) that.area( );
}
},
provides: Comparable
});

// Öz üstsinifinin konstruktorunu zəncir üzrə çağıran,
Rectangle sinifinin
// altsinifi, üstsinifin metodlarına varis olur, öz nüxsə
metodlarını və
// statik metodları müəyyən edir və equals() metodunu alır.
var PositionedRectangle = defineClass({
   name: "PositionedRectangle",
   extend: Rectangle,
   construct: function(x,y,w,h) {
        this.superclass(w,h); // вызов по цепочке
        this.x = x;
        this.y = y;
   },
   methods: {
```

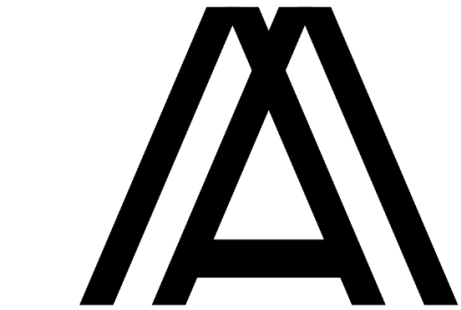
```
        isInside: function(x,y) {
            return x > this.x && x < this.x+this.width &&
            y > this.y && y < this.y+this.height;
        }
    },
    statics: {
        comparator: function(a,b) { return a.compareTo(b); }
    },
    borrows: [GenericEquals]
});
```

Digər nəşrlərimiz:

ɔin Niksonun ***“Learning PHP, JavaScript and MySQL”*** kitabı əsasında hazırlanmışdır.



PHP



AbbasMajidov

Notes

[←1]

ECMAScript v3 həmçinin $ işarəsini də dəstəkləyir, lakin JavaScript 1.1 versiyasına qədər olan versiyalarda, bu

[←2]

Bu format double tipində olan ədədlər formasında Java- proqramçılara tanış olmalıdır. double tipi həmçinin C və C++-ın bütün müasir reallaşdırmalarında istifadə edilir.

[←3]

ECMAScript standartı iddia edir ki, əgər sətir "0" simvolundan başlanırsa ("0x" və ya "0x" istisnadır), parseInt() funksiyası sətiri səkkizlik və onluq say sistemlərində olan ədəd kimi göstərə bilər. Çünki, funksiyanın davranışı aydın müəyyən edilməmişdir. "0 ilə" başlanan sətirlərin interpretasiyası üçün parseInt() funksiyasından istifadədən çəkinmək və ya hesablamalarda sistemin əsasını açıq-aydın göstərmək lazımdır.

[←4]

C-də işləmiş proqramçıların nəzərinə çatdırırıq ki, C dilindən fərqli olaraq JavaScript-də məntiqi qiymətlər üçün ayrı məntiqi məlumat tipi mövcuddur ki, bu məlumat tipi həqiqi ədədlərlə ifadə edilə bilir. Java-proqramçılarının nəzərinə çatdırmaq istəyirik ki, baxmayaraq JavaScript məntiqi məlumat tipi mövcuddur, lakin bu tip Java-dakı boolean tipi qədər “təmiz” işləmir - JavaScript-də məntiqi qiymətlər başqa məlumat tiplərində asan dəyişdiriləcək və buna görə təcrübədə, məntiqi qiymətlərlə işləyərkən, JavaScript proqramlaşdırma Java-ya nisbətdə C proqramlaşdırma dilinə daha oxşardır.

[←5]

C və C++ proqramçılarının nəzərinə çatdırmaq istəyirik ki, JavaScript-də null qiyməti C, C++ və digər proqramlaşdırma fərqli olaraq null qiyməti 0 ədədi deyil. Null qiyməti müəyyən şəraitdə özünü 0 ədədi kimi göstərsə də, heç bir zaman bu qiymətlər bir-birinə ekvivalent deyil!

[←6]

Bu bölmə kifayət qədər mürəkkəb materiala malikdir və bölmə ilə ilk tanışlıq məqsədilə bəzi məlumatları ixtisar edə bilərsiniz.

[←7]

Ancaq bu halda sətir qiymətlərində `eval()` metodundan istifadə etmək lazımdır.

[←8]

Bu bölmə kifayət qədər mürəkkəb material özündə saxlayır və ilk dəfə tanış olma zamanı bu bölməni ixtisara olar.

[←9]

C-də işləməyi bacaran və ümumilikdə bu proqramlaşdırma dilinin göstəricilər konsepsiyası ilə tanış olan proqramçılar, bu kontekstdə istinadların əsas məğzini anlamalıdır. Bununla belə qeyd etmək lazımdır, JavaScript göstəricilər ilə işi dəstəkləmir.

[←10]

Bu mürəkkəb predmetdir və kitabdakı digər fəsillərlə əlaqəli olduğu üçün tam mükəmməl qavranılması tələb olunur. Təzə başlayanlar fəsilin ilk iki bölməsi istisna olmaqla, digər bölmələri ixtisara salıb, 5, 6, 7-ci fəsillərə tanış olduqdan sonra fəsilin qalan bölmələrinə davam edə bilərlər.

[←11]

Əgər bu etməməksə, onda dəyişən gizli özlərinə JavaScript İnterpretatoru tərəfindən elan ediləcək.

[←12]

Bu sadələşməyə, JavaScript-in faktiki reallaşdırmasının təsviri kimi baxmağa lüzum yoxdur.

[←13]

Bu mövzu sizə maraqsız gəldisə, onu ixtisara sala bilərsiniz.

[←14]

Burada “zəncir” sözü “silsilə” söz ilə də əvəz edilə bilər. *(t.r)*

[←15]

JavaScript 1.0 və 1.1-də, əgər sol operandın hesablanması nəticəsində false qiyməti alınarsa, && operatoru sol operandın dəyişdirililməmiş qiymətini qaytarır.

[←16]

JavaScript 1.0 və 1.1-də, əgər sol operand true dəyişdirilə bilirsə, operator true qiymətini qaytarır, əks təqdirdə operandın dəyişdirilməmiş qiymətini qaytarır.

[←17]

C++ proqramlaşdırma dili ilə tanış olan şəxslərin, nəzərinə çatdıraq ki, JavaScript-dəki delete operatoru C++-dakı delete operatorundan tamamilə fərqlənir. JavaScript-də yaddaşın boşalması tullantı toplayıcısı tərəfindən avtomatik yerinə yetirilir və yaddaşın açıq boşalması barədə narahat olmağa lüzum yoxdur. Buna görə də, C++ stilində qalıqsız obyektləri silən delete operatoruna ehtiyac yoxdur.

[←18]

C, C++ və Java-dakı `switch` təlimatı JavaScript-dəki `switch` təlimatından əhəmiyyətli dərəcədə fərqlənir. Bu dillərində `case` ifadələri sabitlər olmalıdır. Bu ifadələrin hesablanması kompilyasiya mərhələsində yerinə yetirilir və `case` ifadələri hamısı eyni tipdə - integer və ya başqa ədəd tipində olmalıdır. Bu isə o deməkdir ki, JavaScript-də switch təlimatı C, C++ və Java-ya daha az effektivdir. Bu dillərdə case ifadələri JavaScript-dəki icra zamanı deyil, kompilyasiya mərhələsində hesablanan sabitlərdən təşkil olunur. Bundan başqa, bir halda ki, C, C++ və Java-da case ifadələri ədəd tipində olur, onda switch təlimatı yüksək təsirli cədvəllər arasında keçidlər zamanı çox effektiv işləyəcək.

[←19]

continue təlimatının müzakirəsi zamanı görəcəyik ki, *while* dövrü *for* dövrünə heç bir halda ekvivalent deyil.

[←20]

JavaScript-in müxtəlif reallaşdırmaları funksiyaların standarta uyğun təyin edilməsi tələblərinə laqeyddir. Məsələn,

Netscape JavaScript 1.5 reallaşdırmaları if təlimatlarının daxilində "şərti olaraq funksiyaların təyin edilməsinə"

imkan verir.

[←21]

Bu bölmənı hətta OYP konsepsiya ilə tanış olmayanların belə oxuması tövsiyə edilir.

[←22]

Java və C++-da bu termin "sahə" adlanmasına baxmayaraq, burada biz onları xüsusiyyəti adlandıracağıq, çünki, JavaScript-obyektlərində belə terminologiya qəbul edilmişdir

[←23]

Rhino-nun (Java dilində yazılmış JavaScript interpretatoru) 1.6r1 və bundan əvvəlki versiyalarında constructor xüsusiyyətinin nizamlanmasını yerinə yetirən proqram kodunda səhv barədə məlumatlar bildirilir. Nəticədə `PositionedRectangle` sinifinin nüsxələri `Rectangle`()-konstruktoruna istinad edən constructor xüsusiyyətinin qiymətinə varis olur. Praktikada bu səhv demək olar ki görünmür, çünki xüsusiyyətlər düzgün varis olunur və instanceof operatoru `PositionedRectangle` və `Rectangle` siniflərinin nüsxələrini düzgün ayırd edir.

[←24]

"Kobud tip təyini" termini Ruby proqramlaşdırma dilindən götürülmüşdür. Terminin əsl adı allomorfizmdir.