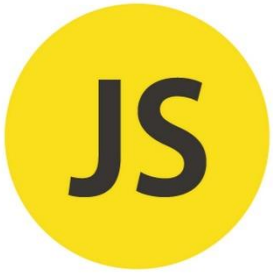
Azerbaijan Javascript Developers Groups



JAVASCRİPT

Nail Mammadov

2020

Mündəricat

1. JavaScript-ə Giriş

Bu kitab Azərbaycan dilində ən sadə yazılmış Javascript dərsliyidir. Javascript haqqında hər şey yəni, onların xüsusiyyəti, nələr edə biləcəyimiz və hansı texnologiyaların istifadə edildiyinə baxacıq.

JavaScript nədir?

Javascript başlanğıcda veb səhifələrin interaktiv (canlı) etmək üçün istifadə edilir.

Bu dildə fayl tipinə script (əmr) faylları deyilir. Veb səhifədə HTML içərisinə Javascript yazıla bilər və səhifə yükləndikdə avtomatik olaraq işə düşəcəkdir. Script faylları işə salmaq üçün xüsusi format vəya compailer-ə ehtiyac yoxdur.

Javascript Java adlanan dildən çox fərqlidir.

Niyə Javascript?

Javascript ilk əvvəllər başqa adla bilinirdi: “LiveScript”. Ancaq Java o vaxtlar çox məhşur idi. Bu səbəbdən yeni dili Javanın “kiçik qardaşı” kimi adlandırılmasına qərar verildi. Ancaq inkişaf etdikcə Javascript Ecmascript adlı müstəqil dil oldu və artıq Java ilə heç bir əlaqəsi yoxdur. Bu gün Javascript yalnız brauzerdə deyil həmdə serverdə vəya əslində Javascript engine adlanan framework (karkas) olan cihazda işləyə bilər. Brauzerdə bəzən “Javascript Virtual Machine” adlı xüsusiyyəti vardır.

Enginlərin kod adları brauzlərə görə aşağıdakı kimidir.Məsələn:

V8 - Chrome və Operada.

SpiderMonkey - Firefoxda.

“Trident” və “Chakra”- fərqli İE versiyalarda olur.

“ChakraCore” - Microsoft Edge-da.

“Nitro” və “SquirrelFish” – Safarida.

Yuxarıdakı terminləri yadda saxlamaqda fayda vardır. Çünki developer və özümün məqalələr yazdığım medium hesabımda çox istifadə edilir. Məsələn “X xüsusiyyəti V8 tərəfindən dəstəklənir” isə ehtimal ki Chrome və Opera-dada işləyir.

Enginlər necə işləyir?

Enginlər qarışıqdır. Ancaq ilkin anlayışlar asandır. Enginlər (Brauzerdə olarsa) scripti oxuyur (“ayırır”). Sonra scripti maşın dilinə çevirir (“hazırlayır”). Və sonda maşın dilində olan kod sürətli işləməyə başlayır. Engine prosesini hər addımında optimizasya işləri aparılır. Ayrılmış script işləyərkən belə onu izləyir,içərsində olan məlumatları analiz edir və məlumatlara əsaslanaraq maşın dilindəki kodu optimizasiya edər. Bu bitdiyində javascript sənədləri çox sürətli işləyirlər.

Brauzer üçün Javascript nə edə bilər?

Modern Javascript “güvənli” proqramlama dilidir.Yaddaş vəya CPU-ya görə zəif bağlantı qurmaz çünki başlanğıcda bu parametrlərin lazım olmadığı brauzerlər üçün nəzərdə tutulmuşdu.

Javascriptin bacarıqları ən çox olduqları mühitə görə dəyişir. Məsələn Node.js – Javascriptin istənilən sənədləri oxumasına/yazmasına, şəbəkə istəklərinin reallaşdırılmasına və s. proseslərə icazəni dəstəkləyir. Brauzer içərisindəki Javascript veb səhifə manipulyasiyası,istifadəçi əlaqəsi və veb server ilə əlaqəli hər şey edilə bilir.

Məsələn Javascript brauzer içərisində bunları edə bilər:

Səhifəyə yeni HTML əlavə etmək, mövcud məlumatların dəyişdirilməsi, formatını dəyişdirmək.

İstifadəçilərin apardığı əməliyyatalara reaksiyası. Mausun kliklənməsi ilə, ox işarələri, qutuların kliklənməsi və s.

Şəbəkə ilə başqa serverlərə istək göndərilməsi, sənədlərin göndərilməsi və yüklənməsi (AJAX və COMET texnologiyaları).

Cookie-lərin hazırlanması ilə istifadəçilərə sualların verilməsi, mesajların göstərilməsi və s.

Müştərinin məlumatlarının yaddaşda saxlanması (“local storage”).

Javascriptlə brauzerdə nələr etmək olmaz?

Javascriptin brauzerdəki bacarıqları, istifadəçinin güvənlilik tərəfindən məhduddur. Məqsəd istifadəçinin məlumatlarının alınmasının vəya oğurlanmasının qarşısının alınmasıdır.

Bu hallara aid nümunələr:

Veb səhifəsindəki JavaScript sabit diskdə özbaşına sənədləri oxumağa / yazmağa, onları nüsxəsini çıxatmağa və ya proqramları işə sala bilməz. OS sisteminin funksiyalarına birbaşa çıxışı yoxdur.

Müasir brauzerlərdə sənədləri oxumağa icazə verilir. Ancaq məhdud və yalnız istifadəçi sənədi brauzer pəncərəsinə “buraxmaq” vəya <input> teqi ilə seçərək istifadə edə bilər.

Kamera/mikrofon və digər cihazlarla əlaqəni yaratmağın yolları vardır. Ancaq bunlar üçündə istifadəçinin icazəsi lazımdır. Bu səbəblə Javascript veb səhifədə veb kameranı gizlicə işə salıb və məlumatları göndərə bilməz.

Fərqli tablarda/ pəncərələrdə ümumiyyətlə bir-birini görmür. Bəzən bir pəncərənin digərini açmaq üçün Javascript istifadə edildiyi zaman edilir. Ancaq bu halda belə səhifədəki Javascript fərqli səhifələrdən gəlirsə, digərindən istifadə edə bilinmir.

Buna “Same Origin Policy” deyilir. Bunun həll yolu hər iki səhifə arasında razılıq olmalıdır və onun üçün isə xüsusi Javascript kodu yazılmalıdır. Bunu sonrakı mövzularda ətraflı izah ediləcəkdir.

Bu məhdudiyyətlə istifadəçinin güvənliyi üçündür. İstifadəçinin açdığı http://hərhansı.com ünvanından olan səhifə http://gmail.com URL-si olan başqa pəncərədən məlumat oğurlamağa imkan vermir.

Javascript şəbəkədən öz səhifəsinin olduğu server-ə rahatlıqla əlaqə qura bilər. Ancaq digər səhifələrdən məlumat alma xüsusiyyəti yoxdur. Javascript brauzerdən kənarda məsələn serverdə istifadə edilirsə, bu məhdudiyyətlər olmayacaqdır. Müasir brauzerlər geniş icazə tələb edilməyəcək əlavələrə icazə verir.

Javascripti oxşarsız edən nədir?

JavaScripti oxşarsız edən 3 səbəb vardır:

HTML / CSS ilə tam bağlıdır.

Sadə şeylər çox asanlıqla həll olunur.

Bütün brauzerlər tərəfindən dəstləklənir və hal-hazırda istifadə olunur.

Javascript 3-nüdə birləşdirən tək brauzer texnologiyasıdır.

Javascriptidə oxşarsız edən budur. Brauzerlər üçün ən çox istifadə edilən yol budur. Yeni texnologiyanı öyrənməyi planlaşdırdıqda baxış bucağınıza nəzarət etməklə daha faydalı ola bilər. Əgər bu sizdə belədirsə, müasir trendlərə keçid etmək olar.

JavaScript üzərindən dillər

Javascript sintaksisi bütün ehtiyacları ödəmir. Hər bir insan üçün fərqli xüsusiyyətin olmasını istəyər.Bu hal proyektlərin və istənilənin hər kəs üçün fərqli olması idi.

Bu səbəblə son zamanlarda Javascriptdə sintaksisində modifikasiya edərək yeni dil ortaya çıxarır.

Müasir alətlərdə çox sürətli və avtomatik olaraq çevrilmələrin aparılmasına imkan yaradır.

Ən çox istifadə edilənlər:

CoffeeScript – Javascript üçün “sugar sintaksis” dir.Daha qısa və dəqiq kod yazmağımıza icazə verən sintaksis imkanını verir.Ümumilikdə developerlər bunu Ruby-də də bəyənirlər.

TypeScript – qarışıq sistemlərin yazılması və hazırlanmasını rahatlaşdırmaq üçün yəni, “strict” məlumatların əlavə etməyə imkan verir.Mikrosfot üçün hazırlanmışdır.

Flow - başqa data tipləri yazmağa imkan verir. Facebook tərəfindən hazırlanmışdır.

Dart - brauzerdən kənarda (mobil proqramlarda) işləyə bilən öz engininə sahib olan ancaq eyni zamanda Javascriptdə müstəqil dildir. Google tərəfindən hazırlanmışdır.

Daha çox var. Təbii ki bunlardan əvvəl Javascripti bilməliyik.

Nəticə

Javascript başlanğıcda yalnız brauzer dili olaraq yaradıldı. Ancaq indi bir çox başqa məqsədlər üçün istifadə edilir.İndiki vaxtda Javascript HTML/CSS ilə tam bağlı olan ən məhşur brauzer dilidir.Javascriptdə “transpiled” və müəyyən xüsusiyyəti verən bir çox dil var. Javascripti anladıqdan sonra ən azında qısa şəkildə onlara baxmağınız məsləhətdir.

2. Əlavələr və xüsusiyyətlər

Bu kitabı 0-dan başlıyan tələbələrin Javascripti mənimsəməsidir. Kitab sadə, başa düşülən və anlaşılan dillə yazılmışdır. Ancaq ilkin bilikləri öyrədikdən sonra başqa mənbələrə ehtiyacınız olacaqdır.

Spesifikasiya

ECMA-262 spesifikasiyası – Javascript ilə bağlı ən dərin, izahlı və rəsmi məlumatları özündə saxlayır. Dili 0-dan izah edir. Ancaq burada yazılanlar rəsmi şəkildə yazıldığı üçün anlamaqda çətinlik çəkə bilərsiniz. Bir sözlə, Javascipt haqqında hər məlumatları xırda detallarına qədər bu kitabdan tapa bilərsiniz.

Ən son versiya bu linkdədir: https://tc39.es/ecma262/

Faydalı

MDN (Mozilla) Javascript-də haqqında məlumat, nümunələr və digər məlumatlar vardır. Dərinləməsinə öyrənmək üçün əla bir yerdir.

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference

Yenədə bunun yerinə internet axtarış etmək daha yaxşısıdır. Sorğuda sadəcə “MDN [termin]" istifadə edin. Məsələn parseİnt metodunu axtarmaq üçün https://google.com/search?q=MDN+parseInt.

MSDN- Javascript (çox zaman Jscript olara bilinir) daxil olan məlumat bazasıdır. İnternet Explorer üçün axtarırsınızsa “RegExp MSDN” vəya “RegExp MSDN jscript” kimi ifadələr yazaraq tapa bilərsiniz.

Yenilik dəstəyi

Javascript inkişaf etməkdə olan dildir. Yeni xüsusiyyətlər davamlı olaraq əlavə edilir.

Brauzerlər arasında dəstəklənmələrini görmək üçün baxın :

xüsusiyyətin dəstəklənmə cədvəli - http://caniuse.com

Hansı brauzerlərin müasir şifrələmə metodlarını dəstəklədiyini görmək üçün- http://caniuse.com/#feat=cryptography.

Dil xüsusiyyətinin və bunları dəstəkləyən vəya dəstəkləməyən brauzerlərin cədvəli - https://kangax.github.io/compat-table

Bütün bunlar sizin ehtiyacınız olacaqdır.

3. Kod redaktorları

Kod redaktoru developerlərin vaxtlarının çoxunu sərf etdiyi yerdir. Kod redaktorlarının iki əsas növü var: IDE və sadə redaktorlar. Developerlərin hər bir növdən fərqli məqsədlər üçün istifadə edirlər.

IDE

IDE (Integrated Development Environment) termini adətən bir "bütün layihədə" fəaliyyət göstərən bir çox xüsusiyyətləri olan güclü bir redaktordur. Adından göründüyü kimi, bu, yalnız bir redaktor deyil, tam miqyaslı "Development Environment".

IDE layihəni yükləyir (bir çox fayl ola bilər), fayllar arasında naviqasiya təmin edir, bütün layihəyə əsasən avtomatik tamamlamağa təmin edir (yalnız açıq fayl) və bir versiya idarəetmə sistemi (git kimi), test mühiti və digər "layihə səviyyəsində" istifadə edilir.

Hələ İDE seçməyibsinizsə, aşağıdakıları nəzərdən keçirə bilərsiniz:

Visual Studio Code (cross-platform, pulsuz).

WebStorm (cross-platform, pullu).

Windows üçün "Visual Studio Kod" ilə “Visual Studio” qarışdırmamaq lazımdır. “Visual Studio” ödənişli və güclü redaktorudur. Visual Studio Community-in pulsuz versiyası da var. Javascript üçün yaxşıdır. Bir çox IDE ödənişlidir, lakin sınaq müddəti var.

Sadə redaktorlar

"Sadə redaktorlar" IDE-lər kimi güclü deyil, lakin onlar sürətli və sadədirlər. Onlar əsasən bir fayl açmaq və redaktə etmək üçün istifadə olunur. "Sadə redaktoru" və "IDE" arasındakı əsas fərq, IDE layihənin həcminə görə seçilməlidir. Yüngül işlər üçün isə “sadə redaktor” – lardan istifadə edilir. Sadə redaktorlarda analizatorlar və avtomatik tamamlama əlavələri olduğundan çox istifadəyə yararlıdır.

Aşağıdakılardan birini seçə bilərsiniz:

Atom (cross-platform, pulsuz).

Sublime Text (cross-platform, shareware).

Notepad ++ (Windows, pulsuz).

Vim və Emacs-dan da istifadə edə bilərsiniz.

4. Console.log

Yazdığınız koddakı səhvlərin izahı üçün isitfadə edilir. Çox böyük ehtimal ki, səhvlər edəcik. Bu normal haldır. Ancaq brauzerdə istifadəçilər xətaları görmürlər. Bu səbəbdən scriptdə səhv olduqda, nəyin işləməyə mane olduğunu və bunu necə düzəltməyin yolunu bilmək üçün birinci növbədə səhvi görməliyik. Xətaları görmək üçün və script haqqında məlumat almaq üçün brauzerlərə “Developer Tools” əlavə olunmuşdur. Developerlər Chrome və ya FireFox-dan istifadə edilir.Çünki ən yaxşı “Developer Tools”-a sahibdirlər. Amma bəzi brauzerlərdə xüsusi “Developer Tools”-a sahib ola bilər.Ancaq ümumilikdə bunların hamısı “Chrome” və ya “FireFox”-da olacaqdır.

Başlamaq üçün xətalara baxıb Javascript scriptlərinin necə işlətməyi öyrənəcik.

Google Chrome

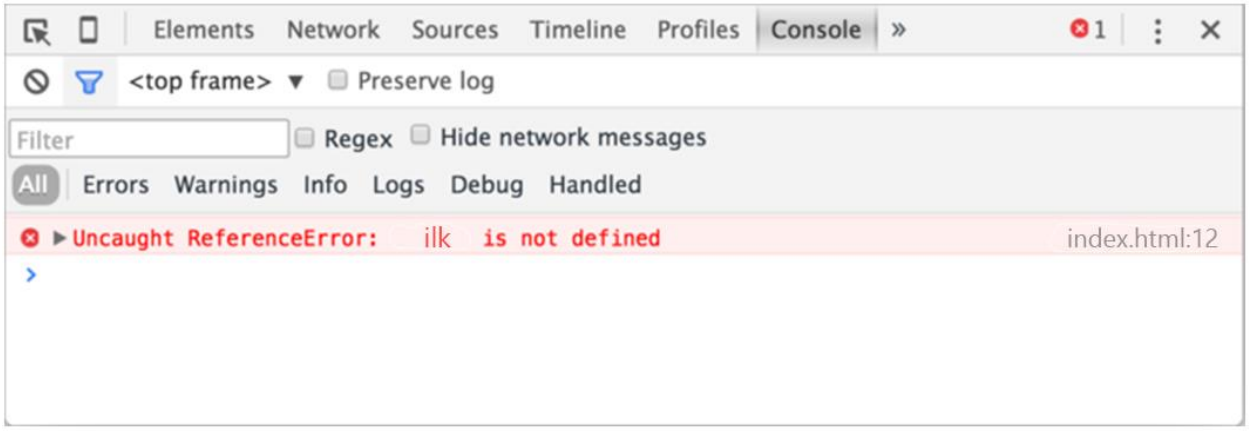
index.html sənədi yaradın.

Sənədin içərisində script daxilində let lalala; yazın və brauzerdə işə salın. Javascript kodunda xəta var. İstifadəçilər tərəfində gizlədilmişdir. Bu səbəbləri onu görmək üçün “Developer Tools” açmalıyıq.

Mac istifadə edirsinizsə Cmd+Opt+J – a eyni anda basın.

Developers Tools consol bölməsində açılacaqdır.

Buna bənzəyəcəkdir:



Developer Tools tam görüntüsü Chrome-un versiyasına bağlıdır. Zaman keçdikcə yenilənir. Burada qırmızı rəngli xəta mesajıdır. Bu halda script faylınızda bilinməyən "ilk" əmri vardır.

Sağ tərəfdə xətanın olduğu sətir nömrəsi göstərilmişdir.

index.html:12 klikləyib baxa bilərsiniz.

Xəta mesajının altında mavi rəngi > simvol vardır.

Javascript kodlarını yaza biləyimiz "kod sətri"-ni işarələyir. Onları işə salmaq üçün enter basın (Çox sətirlik kod yazmaq üçün shift+enter-i basın).

İndi xətaları görə bilirik və bu başlanğıç üçün yetərlidir.

Daha sonra Developer Tools-a geri qayıdacıq və Chrome-da xətaların həllinə baxacıq.

Firefox, Edge və digərləri

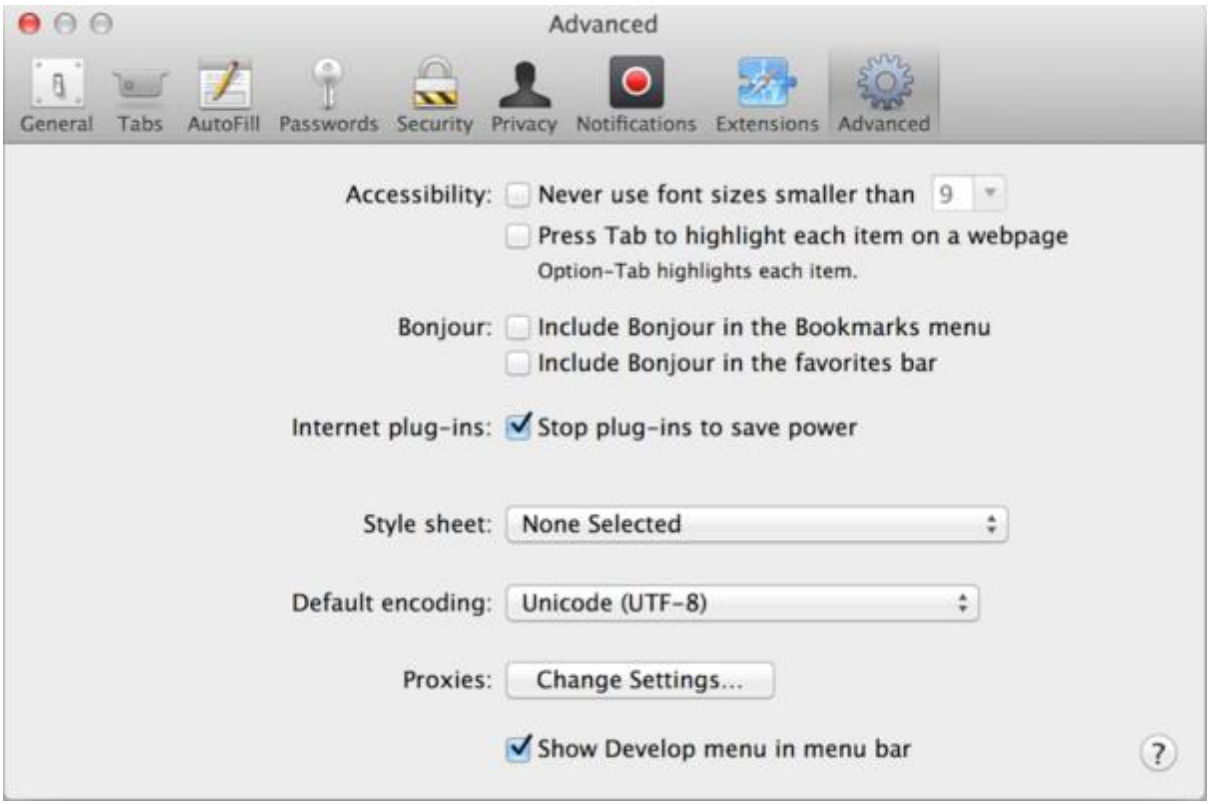
Digər brauzerlərin çoxunda "Developer Tools"-u açmaq üçün F12-i istifadə edilir.

Bunların görüntüsü və funksionallığı olduqca bənzərdir. Chrome ilə başlıya bilərsiniz.

Safari

Safari (Mac brauzeridir, Windows / Linux tərəfində dəstəklənmir) burada biraz xüsusidir. Bunun “Developer menyusu” nü aktiv etmək lazımdır.

Seçimləri açın və “Developer” bölməsinə daxil olun. Altda seçim qutusu var:



Cmd + Opt + C konsolu dəyişdirir. Həmdə “ Developer ” adlı yeni üst menyu bölməsi görünəcəkdir. Əmr və seçimlər vardır.

Çox sətirli giriş

Konsolda bir kod sətiri yazdıqdan sonra enter basıldıqda kodumuz işləyəcəkdir.

Birdən çox sətir əlavə etmək üçün shift+enter -lə edə bilərsiniz.

Nəticə

Developer xətaları görməyimizə, kod yazmaq üçün, dəyişənlərə baxmaq üçün və daha çox funksiyaları vardır.

Windows-dakı brauzerlər üçün F12 – ilə açılır. Mac üçün Chrome-un Cmd+Opt+J

Safari: Cmd + Opt + C (əvvəlcə aktiv edilməlidir)

İndi demək olarki hazırıq. Bundan sonrakı bölmədə Javascriptə keçəcik.

5. Salam, dünya!

Kitabın bu hissəsində dilin özü olan başlanğıc səviyyə Javascript haqqında danışacıq. Daha sonra Node.js və onun digər frameworkləri haqqında öyrənəcəksiniz.

Javascript kodlarımızı işləməsi üçün ilk öncə kodu yazmaq üçün yer təyin edilməlidir. Javascript kodlarımız aşağıdakı yollarla əlavə edə bilərik :

"Script" teqi

Javascript kodları HTML sənədində hər hansı bir yerində <script> teqindən istifadə edərək əlavə edə bilərsiniz.

```
<! DOCTYPE HTML>
<html>
<body>
  <p> Əvvəl ... </p>
  <script>
    alert (Salam,dünya!');
  </ script>
  <p> ... Sonra. </p>
</ body>
</ html>
```

Yazılmış bu nümunəni html sənədi içərsinə yazıb, brauzerdə işə sala bilərsiniz.

<script> teqi daxilindəki kod brauzer teqi işlədildiyində avtomatik olaraq işə salınan Javascript kodudur.

Müasir görünüm

<script> teqi köhnə kodlarda istifadə edilən bəzi xüsusiyyətə sahibdir:

Type atributu: <script type =...>

Əvvəlki HTML standartı olan HTML4-də formatın tipini mütləq yazılmasını istəyirdi. "= text / javascript" formasında idi. Artıq lazım deyil. Həmdə müasir HTML standartı olan HTML 5 bunun xüsusiyyətini dəyişdirdi. Javascript modulları üçün istifadə edilir. Modullar haqqında sonrakı dərslərdə ətraflı danışacıq.

Dil xüsusiyyəti: <script language =...>

Bu xüsusiyyət scriptin dilini göstərmək üçündür. Bu xüsusiyyət artıq lazım deyildir. Çünki Javascript tək istifadə edilir. İstifadə etməyə ehtiyac yoxdur.

Şərh teqləri

<script> teqləri daxilində bu şəkildə şərh yaza bilərsiniz.

```
<script type = "text / javascript"> <! -
    ... // -> </ script>
```

Bu müasir Javascriptdə istifadə edilmir. Şərhlər Javascript kodunu <script> teqini necə işləyəcəyini bilməyən brauzerlər üçün gizlədilmişdir.

Script sənədləri

Javascript kodlarınız çoxdursa, ayrı sənəddə qoya bilərsiniz.

Script faylları HTML-ə src atributu ilə əlavə edilir:

<script src = "/ burdan / ya / script.js"> </script>

Burada /buradan/ya/script.js – script sənədinə gedən yoldur. Html sənədinin olduğu yerdədirsə, sadə yazılışsa yaza bilərsiniz. Məsələn src = "script.js" eyni yerdə olduqları mənasını verir.

URL verə bilərik. Məsələn:

<script src =
"https://cdnjs.cloudflare.com/ajax/libs/lodash.js/3.2.0/lodash.js"> </script>

Birdən çox script sənədi əlavə edə bilərik:

<script src = "/ js / script1.js"> </script>

<script src = "/ js / script2.js"> </script>

...

Yaddan çıxarmıyın:

Qayda olaraq sadəcə ən sadə javascript sənədləri HTML-ə əlavə edilir. Daha qarışıq olanlar ayrı sənəddə olur.

Ayrı olmasının üstünlüyü brauzerin sənədi edirməsi və yaddaşında saxlamasıdır.

Script-ə referans verən digər HTML sənədlərini yükləmək əvəzinə yaddaşda alar. Bu o deməkdir ki , bir dəfə yükləyəcəkdir. Bu sürəti artırar və səhifənin yüklənməsini sürətləndirər. Scr olarsa, script teq daxilini diqqətə almaz. <script> həm teq həmdə src atributunu eyni zamanda istifadə edə bilməz

İşləməyəcəkdir :

```
<script src = "file.js">
    alert (1); // daxili oxumayacaqdır.
</ script>
```

Ya xarici <script src = "..."> yada daxili <script> teqini seçməliyik.

Nəticə

• Səhifəyə Javascript kodu əlavə etmək üçün <script> teqi istifadə edilir.

• Növ və dil xüsusiyyəti lazım deyil.

• Xarici script üçün <script src = "path / to / script.js"> </script> ilə əlavə edilir.

6.Kod strukturu

Kod yazılmadan əvvəl onun strukturunu bilməliyik.

Dəyişənlər

Dəyişənlər sintaksis yaradıcısıdır və funksiyaları yerinə yetirmək üçün istifadə edilir.

Biz “Salam, dünya!” mesajını göstərən bildiriş gördük. Kodumuzda istədiyimiz qədər ifadə verə bilərik. İfadələr nöqtəli vergüllə ayrıraraq yaza bilərik.

Məsələn burda “Salam Dünya”-nı iki yerə ayırdıq :

```
alert ( “Salam”);
alert ( 'Dünya');
```

Ümumilikdə kodun daha yaxşı oxunması üçün ifadələri ayrı-ayrı sətirlərə yazılır:

```
alert ( “Salam”);
alert ( 'Dünya');
```

Nöqtəli vergül

Yalnız nöqtəli vergül olduğu halda nöqtəli vergüllər yazılmaya bilər:

```
 alert ( 'Salam')
alert ( 'Dünya')
```

Burada Javascript sətir sonu “bağlanış”-i nöqtəli vergüllə olaraq bildirir. Buna avtomatik olaraq vergül əlavə etmə deyilir. Çox zaman nöqtəli vergül yeni sətir olduğunu bilidirir.

Nöqtəli vergülün yeni sətir olduğunun mənasını vermədiyi hallar.Məsələn:

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

Kod nəticə olaraq 6-nı verəcəkdir. Çünki Javascript bura nöqtləli vergül əlavə etməz. Sətir “+” ilə bitirsə, bunun yarım ifadə olduğunu anlayır. Bu səbəblə nöqtəli vergül lazım deyildir. Bu halda normal işləyəcəkdir. Ancaq Javascriptin ən vacib məqamı nöqtəli vergülün olmaması hallarında səhvlərin yaranmasıdır.

Bunun kimi hallarda xətaları tapmaq və düzəltmək çox çətindir.

Xəta nümunəsi:

```
[1, 2].forEach(alert)
```

Hələ [] və forEach-in istifadəsi izah edilməmişdir. Bunları sonra daha ətraflı məlumat verilibdir. İndi sadəcə kodun nəticəsinə baxın: 1 və 2- ni göstərəcəkdir.

İndi koddan əvvəl bildiriş əlavə edək və nöqtəli vergüllə bitirməyək:

```
alert ("Xəta olacaqdır")
```

İndi kodu işə salsaq sadəcə bildiriş göstəriləcək və sonra bir xətamız olacaqdır.

Ancaq bildirişdən sonra nöqtəli vergül əlavə etsək hər şey qaydasında olacaqdır.

alert(“indi hər şey qaydasındadır”);

[1, 2]. forEach(alert)

İndi “indi hər şey qaydasındadır” mesajı və sonrasında 1 və 2 mesajı gəldi.

Nöqtəli vergül olmayan xətalar isə Javascriptdə [...] bundan sonra nöqtəli vergülün olmamasıdır.

Bu səbəblə nöqtəli vergül avtomatik olaraq yerləşdirilmədiyində ilk nümunədəki kod tək ifadə olaraq görünəcəkdir. Brauzerdə görünməsi:

alert ("xəta olacaqdır") [1, 2]. forEach (alert)

Ancaq normalda bir yox iki ifadə olmalıdır. Bu halda belə birləşmə səhvdir. Başqa hallarda ola bilər.

Yeni sətirlərlə ayrılmış olsalar belə ifadələr arasında nöqtəli vergül qoymağınızı məsləhətdir. Bu qayda developer arasında qəbul edilmişdir. Bir dəfə də qeyd edək – çox zaman vergüldən kənar yazmaq olar. Yeni başlayanlar üçün onları istifadə etmək məsləhətdir.

Şərhlər

Zaman keçdikcə yazdığınız kodlar qarışıq hala düşməyə başlayır. Kodun nə işə yaradığını və səbəbini izah edən şərhlər əlavə etmək lazım olur.

Şərhlər istənilən yerdə yazıla bilər. Heç bir təsiri yoxdur və brauzerlər tərəfindən oxunmur.

Tək sətirlik şərhlər üçün - //.

Bundan sonra başlayan yazı şərhdir. Öz tutduğu sətri şərh halına salır.

Bunun kimi:

```
// Bu şərh tək sətrlikdir.
alert ( Salam);
alert ( 'Dünya'); // Bu şərhin davamı ola bilər.
```

Çox sətirlik şərhlər üçün / əyri xətt və * ulduz işarəsi ilə başlayar. * / - Ulduz və əyri xətt ilə bitər.

Bunun kimi:

```
 / * Çox sətirli
Şərh teqinə nümunə.
* /
alert ( Salam);
alert ( 'Dünya');
```

Şərhlərdə yazılan nə varsa, işləməz bu səbəblə / *... * / yazılmaz.

müəyyən vaxtda işləməsini dayandırmaq istədikdə

```
/ * Şərhdə kod
alert ( Salam);
* /
alert ( 'Dünya');
```

Qısa yollar

Çox editorda tək sətirlik kod CTRL+/qısa yolu ilə

Çox sətirlik kod üçün CTRL+Üst simvol+/ qısa yolu ilə

Mac üçün isə CTRL yerinə Cmd istifadə edilir.

İç-içə şərh teqləri olmaz

/*...*/, başqa /*...*/ içərisində olmaz.

Bu kod xəta nəticəsini verəcəkdir.

```
/ *
 / * iç-içə şərh? * /
* /
alert ('Dünya');
```

Kodlarınızın izahını şərhdə yazmaqda unutmayın.

Şərhlət ümumi olaraq kod sətrini artıracaqdır.Bu isə problem deyildir. Kodları bitirdikdən sonra həcmini kiçildən bir çox alət vardır. Şərhlər silinər və beləcə işləyən kodlar qalar.

Kitabın sonrakı mərhələsində daha yaxşı şərhlər necə yazılacağını izah edən Kod keyfiyyəti bölməsi olacaqdır.

7. Use strict

Uzun müddət Javascriptdə uyğunsuzluq problemi olmadan inkişaf etmişdir. Dilə yeni metodlar əlavə edildikdən sonra nə əvvəlki metodlar dəyişdirilir, nədə ki istifadədən çıxarılırdı. Javascript kodlarının qırılmaması onun üstünlüyü sayılırdı. Ancaq bu belə davam etmədi. Javascriptin yaradıcıları tərəfindən qəbul edilmiş xətanın vəya boşluğun sonsuza qədər dildə qalması buna gətirib çıxardı.

ECMAScript 5-in (ES5)-ə qədər bu hal davam etdi. Dilə yeni xüsusiyyətlər əlavə olundu və olunanların bəzilərida dəyişdirildi. Əvvəlki kodları saxlamaq üçün **“use strict”-** dən istifadə edilməyə başlanıldı.

“use strict” script sənədində ən yuxarıda yerləşdirilməsi bütün script faylının “müasir” formada işləməsinə gətirib çıxaracaqdır.

Gələcək yönümlü layihələr üçün **“use strict”** bütün scriptin əvvəlinə yazılmalıdır. Developerlərin çoxu bütün funksiyaların əvvəlində istifadə edir.

“use strict”-in işləməsi üçün mütləq ən əvvəldə yazılmalıdır. **“use strict”** – i dayandırmaq mümkün deyildir. Brauzeri köhnə metodlarda işləməsi üçün metod yoxdur.

Brauzer konsolunda metodları test edərkən, **“use strict”** yazmağı unutmayın.Birdən çox sətirlik kod yazmaq üçün shift+ enter kombinasiyasında istifadə edərək yaza bilərsiniz.

```
“use strict”
<üst simvol + yeni sətir üçün>
//sizin kodunuz
<işə salmaq üçün enter>
```

FireFox və Chrome kimi çox brauzerlərdə işləyir.

Bu yol alınmazsa, konsola yazmaq ən yaxşı metoddur.

```
Function(){
“use strict”;
// sizin kodunuz
})
```

Hələlik sizə “use strict” ilə “default”-un arasında fərqi izah etmədik. Bundan sonrakı bölmələrdə dil xüsusiyyətlərini öyrənərkən “use strict” və “default” arasında fərqi izah edəcik.

Əslində “use strict” bizim işimizi asanlaşdırır. Hələlik ümumi olaraq aşağıdakılar bilməyimiz yetərlidir.

“use strict” metodu ilə müasir xüsusiyyətlərin brauzerdə işləməsinə kömək edir. “use strict” bütün brauzerlər tərəfindən dəstəklənir. Bu kitabdakı bütün nümunələr əksi deyilmədiyi halda “use strict” halındadır.

8. Dəyişənlər

Javascriptdə məlumatlarla işləmək üçün dəyişənlərdən istifadə edilir. Buna sadə iki nümunə:

1. Onlayn mağaza – satılan malların məlumatı və səbət aid edilə bilər.
2. Danışıq proqramı – istifadəçi məlumatları və s.

Bu məlumatları saxlamaq üçün dəyişənlərdən istifadə edilir. Dəyişən – adlandırılmış baza deməkdir. Mallar, istifadəçilər və digər məlumatları saxlamaq üçün dəyişənlərdən istifadə edə bilərik.

Dəyişən yaratmaq üçün let açar sözündən istifadə edilir.

Aşağıdakı ifadə "message" adında dəyişən yaradır:

let message;

İndi isə dəyişənə dəyər vermək üçün = istifadə edə bilərik:

let message;

message = "Salam"; // məlumatı saxlayır

Yazılmış yazı dəyişənlə əlaqələndirilib onun bazasına əlavə edilmişdir. Daha sonra isə dəyişən adını istifadə edərək yazını çağıra bilərik:

let message;

message = Salam';

alert(message); // dəyişənin dəyərini göstərəcəkdir.

Qısaca dəyişənin bildirişlə və dəyəri verməyi bir sətirdə yaza bilərik.

let message = Salam!'; // dəyişənə dəyər ver.

alert(message); // Salam!

Eyni zamanda birdən çox dəyişəni bir sətirdə yaza bilərik:

İstifadəci = Nail, yash = 25, mesaj = Salam;

Bu daha qısa görünə bilər. Ancaq bunu məsləhət görmürük. Daha yaxşı oxuna bilməsi üçün hər dəyişən üçün ayrı sətirlərdə qeyd edin.

Çox sətirli yazı metodu biraz uzun ola bilər. Ancaq oxunması daha asandır:

```
let istifadəçi = Nail;
let yash = 25;
let mesaj = Salam;
```

Bəzi developerlər çox sətirli metodda birdən çox dəyişən verirlər:

```
let istifadəci = 'John' olsun
let yash = 25,
let message = Salam;
```

... Və ya "əvvəldə vergül" tərzinda ola bilər:

```
let istifadəci = 'John'
 , let yaş = 25
 , let mesaj = Salam;
```

Texnik olaraq bütün dəyişənlər eyni funksiyanı daşıyır. Bu səbəbdən bu zövq və rahatlıq məsələsidir.

var yerinə let

Əvvəlki script sənədlərində let yerinə başqa açar söz görə bilərsiniz. Bu var-dır:

```
var mesaj = Salam;
```

Var açar sözü let ilə eynidir. Eyni şəkildə dəyişən qəbul edər, ancaq biraz "köhnə" metod şəklindədir.

Let və var arasında incə məqamlar var. Bunlar hələki bizim üçün vacib deyildir. Köhnə "var" bölməsində onlar daha ətraflı izah edəcəyik.

Real həyata uyğun oxşarlıq

"Dəyişən" termini məlumat üçün "qutu" olduğunu xəyal etsək, qutunun üzərinəki adla bunu asanlıqla seçə bilərik.

Məsələn mesaj dəyişənindəki "Salam!" dəyəri üçün bir qutu olduğunu düşünə bilərik.

Qutuya hər hansı dəyər əlavə edə bilərik. Həmdə istədiyimiz qədər dəyişdirə bilərik:

```
let mesaj;
mesaj = Salam!';
mesaj = 'Dünya!'; // dəyər dəyişdir
alert (mesaj);
```

Dəyər dəyişdirildiyində əvvəki dəyər silinir:

Həmdə iki dəyişən verilə bilər və məlumatlar birindən digərinə kopyalaya bilərik.

```
let salam = 'Salam dünya';
mesaj;


// 'Salam dünya'-nı salam dəyişənindən mesaja kopyala
mesaj = salam;


// indi iki dəyişənində dəyəri eynidir.
alert (salam); // Salam dünya
alert (mesaj); // Salam dünya
```

Funksional dillər

Dəyişənin dəyərlərini dəyişdirməyi qadağan edən Scala və ya Erlang kimi funksional programlaşdırma dilləri olduğunu qeyd etmək maraqlı ola bilər. Bu cür dillərdə dəyər verildikdən sonra sonsuza qədər orada qalır. Başqa yeni

dəyər lazım olarsa yeni qutu yaratmağa məcbur edəcəkdir. Əvvəlki təkrar istifadə edilməz.

İlk baxışda qəribə görünsədə bu dillə mükəmməl bacarıqlara sahibdir. Bundan əlavə bu qadağanın müsbət halı odur ki , paralel hesablama işlərində rahatçılığıdır. Zehni inkişaf üçün belə dillərdən istifadə edilməyi məsləhətdir.

Dəyişənin adlandırılması

Javascriptdəki dəyişənin adlarında iki məhdudiyyət vardır:

1. Ad sadəcə hərflərfdən, rəqəmlərdən vəya $ və _ simvollarından ibarət olmalıdır.

2.İlki simvol rəqəm olmamalıdır.

Uyğun adlara nümunələr:

```
let userName123;
```

```
let test123;
```

Ad iki vəya daha çox kəlmədən ibarət olarsa, camelCase formasında yazılmalıdır. İlk yazılmış kəlimə kiçik qalan sözlərin ilk hərfi böyük yazılmalıdır:

```
buAdÇoxBöyükdür.
```

Maraqlı olan – dollar işarəsi '$' və alt xətt '_' də adlandırılmada istfadə edilə bilər. Bunlar xüsusi mənası olmayan eyni hərflər kimi normal simvollardır. Bu adlar uyğundur:

```
$ = 1 ; // "$" adında dəyişən yaratdı
```

_ = 2 ; // və "_" adında dəyişən yaratdı

alert ($ + _); // 3

Səhv yazılmış dəyişən adlarına nümunələr:

let 1a; // rəqəmlə başlıya bilməz

mənim-adim; // tire '-' adına icazə verilmir.

Daha vacib məqam

Alma və AlMA adlı dəyişənlər fərqli dəyişənlərdir. Latınca olmayan hərflərə icazə verilir. Ancaq mələhət edilmir. Kiril hərfləri vəya hiroqlif olan hər hansı dili istifadə etmək mümkündür. Nümunə:

let имя = '...';

let 我 = '...';

Texniki olaraq burada xəta yoxdur. Sadəcə ingiliscənin qlobal olaraq qəbul edilməsinə görə məsləhət görülür. İşlərimizin davamında başqa developerlərin çətinlik çəkməməsi üçün qəbul edilib.

İstifadə edilməyən adlar (Reserve Name)

Dilin özü tərəfindən istifadə edildiyi üçün bəzi sözləri dəyişən kimi istifadə etmək olmaz.

Məsələn: let, class, return və function olmaz.

Aşağıdakı kod strukturu xəta verəcəkdir:

let = 5 ; // "let" dəyişən olaraq adlandırılmaz

return = 5; // "return" dəyişən olaraq adlandırılmaz.

"use strict"

Normalda dəyişəni istifadə etməmişdən əvvəl dəyər verilməlidir. Ancaq əvvəlki versiyalarda let ilə dəyər yaradıb dəyər vermədən istifadə etmək mümkün idi.

Ancaq "use strict" buna imkan vermir. Əvvəlki versiyada yazmaq istəyirsinizsə, "use strict" yazmadan həll edə bilərsiniz.

// qeyd: bu nümunədə "use strict" yoxdur.

num = 5; // "num" dəyişəni yoxsa yaradıldı

alert (num); // 5

Bunu "use strict" yazıb yoxlasanız xətaya səbəb olacaqdır:

"use strict";

num = 5; // xəta: num tapılmadı

Const

const (sabit) dəyişəni bildirmək üçün let yerinə const istifadə edə bilərsiniz:

const adGunum = '18.04.1982 ';

const istifadə edilən dəyişənlərə “sabit” deyilir. Dəyişdirilə bilməz. Dəyişdirmək istədikdə xətaya səbəb olacaqdır:

const adGunum = '18.04.1982 ';

const adGunum = '16.11.1996 '; //xəta, sabitə yeni dəyərlə əvəz edə bilməzsiniz

Developer dəyişənin əsla dəyişməyəcəyindən əmin olduğunda qarantiya almaq və dəqiqlik üçün const ilə yaza bilər.

Böyük hərf sabitləri

Proqralamdan əvvəl yada salmaqda çətinlik yaradan dəyərlərin adlarını sabit olaraq istifadə etmək çox işimizə yarayacaqdır.

Bu sabitlər böyük hərf və alt xətt istifadə edilərək adlandırılır.

Məsələn “veb” üçün rəngləri (on altılıq) halında sabitlər yaradaq:

const COLOR_RED = "#F00";

const COLOR_GREEN = "#0F0";

```
const COLOR_BLUE = "#00F";

const COLOR_ORANGE = "#FF7F00";


// ... Bir rəng seçmək lazım olduqda

rəng = COLOR_ORANGE;

alert (rəng); // # FF7F00
```

Üstünlüyü:

• COLOR_ORANGE,xatırlamaq “"# FF7F00”-dan daha rahatdır.

• "# FF7F00" yerinə COLOR_ORANGE yazmaq daha rahatdır

• Oxunuşu – COLOR_ORANGE, # FF7F00 dən mənalıdır.

Böyük hərflər ilə yazılmış dəyişəni nə zaman sabit olaraq və nə zaman normal olaraq adlandırmalıyıq?

“Sabit” olmaq sadəcə dəyişənin dəyərinin heç vaxt dəyişməyəcəyi mənasına gəlir. Ancaq bu sabitlər də iki yerə ayrılır:

1. İş prosesi zamanı yaranan sabitlər
2. Əvvəlcədən bilinən sabitlər

Məsələn:

*const pageLoadTime = / * səhifənin brauzerdə yüklənmə vaxtıdır * /;*

PageLoadTime -un dəyəri əvvəlcədən bilinmədiyi üçün normal olaraq adlandırılmışdır.

Ancaq yenədə sabitdir. Çünki yükləndikdən sonra dəyişməz.

Başqa sözlə desək böyük hərflərlə yazılmış sabitlər yalnızca “kodlaşdırılmış” dəyərlər üçün istifadə edilir.

Adlar haqqında

Dəyişənlər haqqıda danışsaq əsas məqamlar vardır. Dəyişənin adı təmiz və açıq şəkildə mənaya sahib olmalıdır. Saxladığı dəyərlərə uyğun ad olmalıdır.

Dəyişən adlandırılması qarışıq prosesdir. Adlandırılması sizin profesional yoxsa junior olduğunuzu göstərən əlamətlərdən biridir.

Real proyektdə çox zaman sıfırdan başlamaq yerinə mövcud kodları dəyişdirələrək və genişləndirilərək proses davam etdirilir. Bir müddət keçdikdən sonra əvvəl yazdığımız kodlara baxdığımızda kodların başa düşülməsində yardımcı olur.

Dəyişən adları seçərkən mütləq fikirləşmək üçün vaxt ayırın . Bu sizə gələcəkdə vaxt itkisinin qarşısını almağa imkan verəcəkdir.

Bəzi qaydalar :

• userName vəya shoppingCart kimi qlobal adlardan istifadə edin.

• Nə qoyacağınızı bilmirsinizsə qısaltmalardan və ya a, b, c kimi qısa adlardan istifadə etmiyin.

• Adları maksimum mənalı edin.

Pis ad nümunələrinə misal olaraq dəyər və məlumat adlarıdır. Bu cür adlar heç bir məna kəsb etməz.

• Komandanızda və özünüzdə şərtlər qəbul edin.

Veb səhifədən istifadə edənə "istifadəçi" deyilirsə, o zaman currentVisitor vəya newManInTown yerinə currentUsers vəya newUser dəyişən adlarını yazmaq mənalıdır.

Nəticə

var, let vəya const açar sözlərindən istifadə edib dəyişənlər yaratmaq mümkündür.

• let – müasir açar sözdür. Chrome-da (V8) let istifadə etmək üçün "use strict" olmalıdır.

• var – köhnə açar sözdür. Normalda heç vaxt istifadə etməyəcəyik. Ancaq əvvəlki "var" bölməsi olduğundan, sadəcə ehtiyac hiss etdiyimizdə kiçik dəyişiklər üçün istifadə edəcəyik.

• const – let kimidir. Ancaq dəyişənin dəyəri dəyişdirilə bilməz.

Dəyişənlərin içərisində nə olduğunu anlamaq üçün başa düşülən adlandırılma aparılmalıdır.

9. Data tipləri

Javascriptdəki dəyişən istənilən tipdə olan məlumat saxlıya bilər. Dəyişən bir yerdə yazı, fərqli yerdə isə rəqəm ola bilər:

```
// səhv yoxdur
let mesaj = "salam";
mesaj = 123456;
```

Bu cür dəyişikliklərə icazə verən proqramlaşdırma dillərinə “dinamik dillər” deyilir. Yəni məlumat tipləri vardır. Ancaq heç birindən asılı deyildir. Javascriptdə 7 məlumat tipi var. Bu bölmədə onları ümumi olaraq danışacıq və sonrakı bölmələrdə hər biri haqqında ətraflı danışacıq.

Rəqəm (Number)

```
let n = 123;
n = 12.345;
```

Rəqəmlər həm rəqəm həm də kəsrlər ola bilər.

Rəqəmlərin üzərində operatorlarla əməliyyatlar aparmaq olar:

*Vurma *, Bölmə /, Toplama +, Çıxma -*

Normal rəqəmlərdən əlavə xüsusi dəyərlərdə vardır :

İnfinity, - İnfinity və NaN.

• Riyazi sonsuzluq ∞ . Hər hansı rəqəmdən böyük dəyərdir.

Sıfıra bölünmənin nəticəsidir:

alert (1/0); // Sonsuzluq

Və ya sadə olaraq verilə bilər:

alert (İnfinity); // Sonsuzluq

• NaN hesablama zamanı xəta baş verdikdə olur. Məsələn xətalı və ya bilinməyən riyazi əməliyatın nəticəsidir:

alert ("rəqəm deyil" / 2); // NaN – bu xətalı dəyər NaN döndərəcəkdir:

NaN la aparılan operator əməliyyatları NaN dəyərini döndərəcəkdir:

alert (NaN/ 2 + 5); // NaN

Bu səbəblə riyazi ifadədə NaN varsa, nəticədə NaN olacaqdır.

Javascriptdə bütün riyazi əməliyyatları aparmaq mümkündür. Hər şey mümkündür: sıfıra bölünmə, rəqəm olmayan dəyər və s.

die()

Bu əməliyyat xəta ilə durmuyacaqdır. (“die”). Ən pis halda NaN olacaqdır. Xüsusi dəyərlər rəqəm tipinə aiddir. Rəqəmlə bölməsində daha ətraflı məlumat görəcəyik.

String (yazı)

JavaScriptdəki yazılar dırnağı daxilində olmalıdır.

let str = "Salam";

let str2 = 'Tək dırnaqlar içərisində yazıla bilər';

let phrase = `$ {str}` belə də yazılaraq dəyişən köçürülür;

JavaScriptdə 3 növ yazılış şəkli var.

1. İki dırnaq: ”Salam”

2. Tək dırnaq: 'Salam'.

3. Köçürmələr: `Salam`.

Cüt və tək dırnaq sadə dırnaqlardır. Javascriptdə aralarında fərq yoxdur.

Köçürmə şəkli isə inkişaf etdirilmiş funksional dəyərlərdir. Dəyişənləri və ifadələri $ {...} içərisinə yerləşdirməyə icazə verir. Məsələn:

let name = "Nail"; // dəyişən yaradıldı

alert (`Salam, $ {name}!`); // Salam Nail!

// ifadəyə daxil edildi

alert (`nəticə $ {1 + 2}` dir); // nəticə 3

$ {...} içərisindəki ifadə dəyərini köçürür. Bunun içərisinə istədiyimiz tip yaza bilərik.Bunu yalnız “ `` ” köməkliyi ilə edə bilərsiniz. Digərlərin bu xüsusiyyət yoxdur.

alert ("nəticə $ {1 + 2}"); // nəticə $ {1 + 2} 'dir (çüt dırnağın təsiri olmayacaqdır)

Stringlər bölməsində daha ətraflı məlumat veriləcəkdir.

Simvol tipi yoxdur.

Bəzi dillərdə tək simvol üçün xüsusi "simvol" növü vardır. Məsələn C dilində və Javada char-dır. Javascriptdə belə tip yoxdur. Yalnız string vardır. String bir vəya bir necə simvolda ibarət ola bilər.

Boolean (məntiqi tip)

Boolean tipi iki dəyərə sahibdir: düzgün və yanlış. Bu tip ümumilikdə bəli/xeyr dəyərlərini dəyərləri üçün istifadə edilir. True- düzgün, bəli və False – xeyr, yanlış deməkdir.

Məsələn:

nameFieldChecked = true; // bəli, adlar yoxlanıldı mənasını verir.

ageFieldChecked = false; // xeyr, yaşlar yoxlanılmadı mənasını verir.

Boolean dəyərləri həmdə müqayisənin nəticəsi kimi ola bilər:

let Greater = 4 > 1;

alarm (Greather); // true (müqayisənin nəticəsi "bəli" dir)

Məntiqi operatorlar bölməsində Boolean tipini daha dərindən öyrənəcik.

“Null” dəyəri

Xüsusi null dəyəri yuxarıda izah edilən tiplərdən heç birinə bənzəmir.

Yalnızca boş dəyər üçün ayrı tip vardır :

yash = null;

Javascriptdə null “Mövcüd olamayan” bir şey mənasını verir. Bu sadəcə “heç bir şey”, “boş” xüsusi dəyərdir.

Yuxarıdakı yazılmış kodda bəzi səbəblərə görə yaşın boş olduğunu görürük. Bu bizə “null” dəyərini verəcəkdir.

Undefined

Bilinməyən dəyər. Eyni ilə boş göndərilmiş tip kimidir. Undefined-in mənası “dəyəri olmayan”, “bilinməyən” dir. Bir dəyişəni göstərir. Ancaq dəyəri yoxdur:

let x;

alert (x); // "undefined" göstərəcəkdir.

Texniki olaraq hər hansı dəyişənə undefined etmək mümkündür:

let x = 123;

x = undefined;

alert (x); // "Undefined"

Amma bunu məsləhət görmürük. Normalda dəyişənə “boş” və ya “bilinməyən” dəyər vermək üçün null istifadə edərik və dəyişənin olub olmadığını yoxlamaq üçün undefined-dən istifadə edirik.

Objectlər və Simvollar

Object tipi xüsusi tipdir. Digər bütün bütün tiplərə başlanğıç səviyyə deyilir. Çünki bu tiplərdə tək bir şey aid ola bilər (bir rəqəm, yazı və s.). Buna görə də objectlər vardır. Burada qarışıq məlumatların saxlanması üçün istifadə edilir.

Başlanğıc səviyyəni ətraflı öyrəndikdən sonra daha sonra objectlər haqqında öyrənəcik.

Simvol tipi – objectlər üçün oxşarı olmayan dəyər yaratmaq üçün istifadə edilir. Daha yaxşı anlamaq üçün göstərilməlidir. Ancaq object-dən sonra bu tipə baxmaq məsləhətdir.

Typeof operatoru

Typeof operatoru seçilmiş dəyərin tipini göstərir. Fərqli tiplərdəki dəyərləri fərqli şəkilə çevirmək istədikdə və ya sadəcə yoxlama məqsədli istifadə edilir.

İki cür yazılış forması var:

1. Operator olaraq: typeof x.

2. Funksiya olaraq: typeof (x).

Başqa sözlə mörtərizə oldu olmadı işləyəcəkdir. Nəticə eynidir.

typeof x bizə növün tipinə uyğun string geri qaytaracaqdır:

```
typeof undefined //
"undefined"


typeof 0 // "rəqəm"


typeof true // "boolean"


typeof "foo" // "string"


typeof Symbol ("id") // "simvol"


typeof Math // "object" (1)


typeof null // "object" (2)


typeof alert // "function" (3)
```

Son üç sətirin əlavə izah ehtiyacını ola bilər:

1.Riyazi əməliyyatları aparmaq üçün operatorlardan istifadə edəcik. Bunun haqqında sonrakı bölmədə ətraflı izah veriləcəkdir.

2. Null tipinin yoxlasaq nəticəsi “object” dir. Bu səhvdir. Rəsmi olaraq tanınmış xətadır. Təbii ki , null object deyildir. Nəticə olaraq dildə olan xətadır.

3. typeof alert-in nəticəsi funksiyadır. Çünki dilin funksiyasıdır. Javascriptdə xüsusi “funksiya” tipi olmadığı sonrakı bölmələrdə baxacıq. Funksiyalar object tipinə aiddir. Rəsmi olaraq səhvdir. Ancaq praktika olaraq istifadə edilir.

Nəticə

Javascriptdə 7 Data tipi var.

• Rəqəm (Number) tipi: tam ədəd və ya kəsr.

• Yazı (String) tipi: hərf vəya hərflər çoxluğu.

• Boolean tipi: düzgün/ yanlış.

• Null tipi: boş dəyər. Yalnız boş dəyər null olur.

• Undefined tipi: dəyəri bilinməyən.

• Obejct tipi: qarışıq struktura sahib olur. Eyni zamanda həm number həmdə string-ə sahib ola bilər.

• Simvol- oxşarsız simvollar üçün.

Typeof operatoru, hər hansı dəyişəndə nə tip olduğunu öyrənməyimizə icazə verir.

• İki cür: typeof x və ya typeof (x).

• "string" kimi tipin adıyla geri qayıdır.

• Null üçün “object” -geri qaytarır. Rəsmi olaraq bu xətadır. Null object deyildir.

Sonrakı bölmələrdə başlanğıç dəyərləri ətraflı öyrənəcik. Daha sonra isə objectlərə keçid edəcik.

10.Data tiplər arasında keçidlər

Operatorlar və funksiyalar onlara verilən dəyərləri avtomatik olaraq düzgün tipə çevirirlər. Məsələn alert metodu avtomatik olaraq dəyəri göstərmək üçün hər hansı dəyəri stringə çevirir. Riyazi əməliyyatlarda isə dəyərləri rəqəmlərə çevirir. Düzgün olarsa dəyəri gözlənilən tipə çevirmək müəyyən hallarda lazım olur. Hələ object haqqında danışmayacıq. Bunun yerinə başlanğıc səviyyə tiplərə baxacağıq. Daha sonra objectlər haqqında məlumat verdikdən sonra objecti çevirmə haqqında danışacağıq.

ToString

Yazı tipinə çevirmə. Dəyərin string tipində ehtiyacımız olduğu halda istifadə edilir.

Məsələn, alert(deyer) deyeri göstərmək üçün çevirmə aparacaqdır.

Dəyəri stringə çevirmək üçün String(value) metodunu da istifadə edə bilərik:

```
let deyer = true;
alert (typeof deyer); // boolean


deyer = String (deyer); // indi dəyər “true” stringidir.
Alert (typeof deyer); // string
```

String çevirmələri olduqca sadədir.

ToNumber

Ədədə çevirmək üçün istifadə edilir. Riyazi funksiyalar və ifadələrdə avtomatik olaraq çevirilir.

Məsələn, bölünmə/ ədəd olmayan tipdə istifadə edildikdə:

alert ("6" / "2"); // 3, stringlər ədədlərə çevirildi.

Dəyəri ədədə çevirmək üçün Number() metodunu istifadə edə bilərsiniz:

```
let str = "123";
alert (typeof str); // string
num = Number (str); // 123 rəqəmdir
alert (typeof num); // ədəd
```

Bu çevrilmələr ümumilikdə forumda əgər istifadəçinin daxil etdiyi dəyərin yalnız ədəd istədiyimiz hallarda istifadə edə bilərik.

String rəqəm deyilsə, çevrilmənin nəticəsi NaN olacaqdır. Məsələn:

```
let age = Number ("ədəd yerinə istənilən yazı");

alert (age); // NaN, çevrilmə uğursuz oldu
```

Ədəd çevrilmələrinin qaydaları:

Dəyər	Gözlənilən
undefined	*NaN*
null	*0*
true və false	*1 ve 0*
string	Başlanğıcdan və sonra boşluq silinir.

Qalan string boşdursa, nəticə 0 -dır. Əks təqdirdə string oxunacaqdır. Xəta olduqda NaN olacaqdır.

Nümunələr:

alert (Number ("123")); // 123

alert (Number ("123z")); // NaN ("z" hərfi xətaya səbəb oldu)

alert (Number (true)); // 1

alert (Number (false)); // 0

Yadda saxlayın null və undefined tiplərinində fərqli olur. Null tipi 0 qaytaracaq. Undefined isə NaN geri qaytaracaqdır.

'+' Stringləri birləşdirir.

Demək olarki, bütün riyazi operatorlar dəyərləri ədədə çevirir. İstisna hal olaraq toplama + operatoru birləşdirmə funksiyasıı yerinə yetirir. Ancaq dəyərlərdən biri stringdirsə , digərinidə stringə çevirib birləşdirir:

alert (1 + '2'); // '12' (sağda string)

alert ('1' + 2); // '12' (solda string)

Bu sadəcə dəyişənlərdən ən az biri string olduğunda olur. Əks halda dəyərlər number olacaqdır.

ToBoolean

Boolean çevrilməsi ən sadəsidir.

Məntiqi metodları reallaşdırır (daha sonra halları test edir və oxşar şeyləri yerinə yetirir). Ancaq bunu Boolean() ilə etmək mümkündür.

Çevrilmə qaydası:

• Null olan dəyər həmçinin 0-da ola bilər,(null, undefined və NaN) dəyərləri səhv olur.

• Digər dəyərləri true olaraq geri qaytaracaqdır.

Məsələn:

alert (Boolean (1)); // düzgün

alert (Boolean (0)); // yanlış

alert (Boolean ("salam")); // düzgün

alert (Boolean (" ")); // yanlış

Diqqət: Sıfır “0” olan string səhvdir. Bəzi dillərdə (yəni PHP) “0” olan string düzgündür. Ancaq Javascriptdə boş olmayan string hər zaman düzdür.

alert (Boolean ("0")); // false

alert (Boolean ("")); // boşluq, həm də true (boş olmayan hər hansı string true geri qaytarcaqdır)

Nəticə

Ən çox istifadə olunan 3 növ çevrilmə vardır. String, Number və Booleana keçiddir.

ToString – Bir şey yazdığımızda ortaya çıxır. String(deyer) ilə edilə bilər. String keçid ümumilikdə mümkündür.

ToNumber – Riyazi əməliyyatlarda ortaya çıxır. Number() ilə çevirmək mümkündür.

Dəyər	Nəticə
Undefined	*NaN*
Null	*0*
true / false	*1/0*
string	String, “olduğu kimi” oxunur, hər iki tərəfdən boşluğu silər. Boş string 0 olur. Xəta olduqda NaN olur.

ToBoolean – Məntiqi tipə çevirir. Boolean(value) ilə edilə bilər.

Qaydalar:

Dəyər	Nəticə
0, null, undefined, NaN, " "	*false*
Başqa hər hansı dəyər	*true*

Bu qaydaların çoxunu başa düşmək və yadda saxlamaq asandır. Developerlərin ümumilikdə xətaları bunlardır:

• undefined və NaN numberdir, 0 deyildir.

• "0" və "" kimi striglər boolean olaraq true olaraq geri qaytarır.

Objectlər barəsində məlumat verilməmişdir. Onlara daha sonra Javascript haqqında başlanğıc səviyyəni öyrəndikdən sonra, yalnızca objectlərə ayrılmış bölmədə izah verilmişdir.

11. Operatorlar

Məktəbdə operatorlar haqqında bilirik. Bunlar toplama +, vurma *, çıxma – və s. Bu bölmədə məktəbdə görmədiyimiz operatorların xüsusiyyətinə baxacıq.

Terminlər: “unary”, “binary”, “operand”

Davam etmədən əvvəl, bəzi terminlərin izahına baxaq:

Operator- əməliyyat aparmaq üçün istifadə edilir. Məsələn 5*2 -burada 2 həddi vardır. Bəzən “həddi” yerinə “arqument” deyilir.

Unary- tək elementə aid olan operatordur. Ədədi tərsinə çevirir.

```
let x = 1;

x = -x;

alert (x); // -1, unary operatoru istifadə edildi.
```

Normalda operatorlar 2 ədəd arasında olur.

```
x = 1, y = 3;

alert (y - x); // 2, çıxma operatoru işlədildi.
```

İki fərqli operator haqqında danışacıq: tək tərəfli: tərs çevirmə operatoru (unary) və çıxma operatoru

String (yazı birləşdirmə) operatoru +

İndi, Javascript operatorlarının məktəb operatorlarında fərqli xüsusiyyətlərinə baxaq. Ümumilikdə toplama operatoru ədədləri toplayır.

Ancaq , stringlərdə istifadə olunarsa, bunları birləşdirəcəkdir:

```
s = "my" + "string";

alert (s); // mystring
```

ng digəri isə ədəddirsə, digərinidə stringə çevriləcəkdir.

Məsələn:

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

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

String arqumentinin ilk vəya ikinci olması fərq etmir. Qayda sadədir- Hər hansı arqumentdən biri stringdirsə, ikincisidə stringə çevriləcəkdir.

Ancaq əməliyyatın soldan sağa tərəf işlədiyini yaddan çıxarmıyın. Stringdən əvvəl ədədlər varsa, əvvəlcə onları toplayıb sonra birləşdirəcəkdir:

```
alert (2 + 2 + '1'); // "41"-dir. Nəticə "221" deyildir.
```

String birləşdirmə və çevirmə + -un xüsusiyyətidir. Digər operatorlarda sadəcə ədədlər üçün istifadə edilir.

Məsələn çıxma və bölmə:

```
alert (2 - '1'); // 1

alert ('6' / '2'); // 3
```

Unary-də + operatorundan istifadə

+ iki əməliyyat üçün istifadə edilir.Yuxarıda istifadə etdiyimiz və Unary-də

Unary -də istifadə olunan + operatoru heç bir təsiri yoxdur. Ancaq arqument rəqəmdə deyilsə, unary + ilə ədədə çevriləcəkdir.

Məsələn:

```
//Rəqəmlərə təsiri yoxdur.

let x = 1;

alert (+ x); // 1



let y = -2;

alert (+ y); // -2



// Rəqəmləri çevirməz

alert (+ true); // 1

alert (+ ""); // 0
```

Əslində **number()** ilə eyni funksiyanı yerinə yetirir. Ancaq daha qısadır.

Stringləri ədədlərə çevirmək çox qarşımıza çıxacaqdır. Məsələn HTML formda gələn dəyərlər ümumi halda stringlərdir.

Onları toplamaq istəsək bəs?

+ bunları string olar əlavə edər:

```
 let alma= "2";

let portağal = "3";

alert (alma + portağal); // "23", stringləri birləşdirdi.
```

Onları ədəd olaraq istəyiriksə, çevirmələyik və sonra toplamalıyıq:

```
let alma= "2";

let portağal = "3";

// unary + ilə ədədlər ədədlərə çevriləcəkdir.

alert (+ alma + + portağal); // 5

// uzun versiya

// alert (Number (alma) + Number (portağal)); // 5
```

Riyaziyyatçılara görə + çoxluğu qəribə görünə bilər. Ancaq developerə görə heçdə qəribə deyildir. İlk əvvəl unary + çevirir, stringlər ədədlərə çevrili. Sonra isə toplanacaqdır.

Niyə unary+-lar dəyərdən əvvələ yazılır? Bunun əsas səbəbi vardır.

Operatorların sırası

İfadədə birdən çox arqument vardırsa, operator sırasına görə əməliyyatlar sıralanacaqdır.

Məktəbdən bilirik ki , 1+2*2 ifadəsindəki vurma toplamadan əvvəl hesablanmalıdı. Bu operator sıralamasıdır. Vurmanın daha əvvəl aparılması bilinir.

Mörtərizələr hər hansı sıranı pozar. Bu səbəblə sıranı xoşlamadıqda dəyişdirmək mümkündür. Məsələn (1+2)*2 yazın.

Javascriptdə çox operator vardır. Hər operatorun sırası vardır. İlk əvvəl sırası böyük olan işləyəcəkdir. Əgər eyni operatordursa, soldan sağa doğru işləyəcəkdir.

Operator sıralama cədvəli (bunu yadda saxlamağa ehtiyac yoxdur,ancaq unary operatorlarunun nədən böyük olduğuna fikir verin):

Operator sıralaması

... ...

16 unary plus +

16 unary negation -

14 vurma *

14 bölmə /

13 toplama +

13 çıxma -

... ...

3 bərabərlik =

... ...

Gördüyünüz kimi , “unary +”, 13

Gördüğümüz kimi, "unary +"(16) cəm(13)-dən böyükdür. Bu səbəblə “+alma++portağal” ifadəsində unary+ toplamadan əvvəl işləyir.

Bu səbəblə, "+ almalar ++ portağallar" ifadesinde, unary cəmdən əvvəl işləyir.

Bərabərlik

Bərabərlik = eyni zamanda operator olduğunu bilməliyik. Cədvəldə 3-dür. Bu səbəblə, x=2*2+1 kimi dəyişənə bərabər etdikdə, əvvəl hesablama aparılır və sonradan nəticə x-ə əlavə edilir.

```
let x = 2 * 2 + 1;
alert (x); // 5
```

Bərabərlikləri çoxaltmaq mümkündür:

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


alert (a); // 4
alert (b); // 4
alert (c); // 4
```

Çoxaltdıqda əməliyyat sağdan sola doğru aparılacaqdır. İlk əvvəl ən sağdakı 2+2 ifadəsi hesablanacaq və sonra soldakı dəyişənlərə veriləcəkdir: c, b və a. Sonda isə bütün dəyişənlər eyni dəyərə sahib olacaqdır.

Bərabərlik operatoru “=” dəyər geri qaytarır.

Bu əməliyyat zamanı həmişə bir dəyər geri qayıtmalıdır. Bu toplama+ vəya vurma * kimi çox əməliyyata aiddir. Ancaq bərabərlik operatoru də qayda ilə işləyir.

x=dəyər çağırdıqda, dəyəri x-in içinə yazar və sonra geri qaytarar.

Qarışıq ifadədənin bir hissəsi olan nümunədə bərabərliyə baxaq :

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

Yuxarıdakı nümunədə ifadənin nəticəsi (a=b+1), a(3) olaraq dəyərini almışdır. Daha sonra başqa ifadələrdə istifadə ediləcəkdir.

Kodun necə işlədiyini başa düşməliyik. Çünki Javascript kitablarında nümunələr görürük və başa düşmürük. Bunun üçün kodları birdə özünü yazmalısınız. Yoxlayın.

Qalıq (Modules) %

Qalıq operatoru % faizlə əlaqəsi yoxdur.

Qalıq – a və b arasındakı bölmə əməliyyatı nəticəsində qalan ədəddir.

Məsələn:

```
alert ( 5 % 2); // 1, 5 böl 2
alert ( 8 % 3); // 2, 8 böl 3
alert ( 6 % 3); // 0, 6 böl 3'ün qalığıdır.
```

Qüvvət **

Qüvvət – a-nın b dəfə bir-birinə vurulmasıdır.

Nümunə:

```
alert (2 ** 2); // 4 (2 * 2)
alert (2 ** 3); // 8 (2 * 2 * 2)
alert (2 ** 4); // 16 (2 * 2 * 2 * 2)
```

Operator tam ədəd olmayan üçündə işləyir.

Nümunə:

*alert (4 ** (1/2)); // 2 (1 / 2'nin kvadratı kökaltı ilə eynidir, riyaziyyat)*

*alarm (8 ** (1/3)); // 2 (1/3 kvadratı kub kökaltıya bərabərdir)*

Artırma/azaltma

Ədədi bir-bir artımaq vəya azaltmaq ən çox istifadə edilən əməliyyatlardan biridir.

Yəni, bunun üçün xüsusi operator var:

Artırma ++ , dəyişənin artımı 1 olacaqdır:

• let counter = 2;

• counter ++; // counter = counter + 1 ilə eyni şəkildə işləyəcəkdir. Ancaq daha qısa yazılış şəklidir.

alert (counter); // 3

• Azaltma – dəyişənin azalması 1 olacaqdır:

• let counter = 2;

• counter--; // counter = counter - 1 ilə eyni şəkildə işləyəcəkdir. Ancaq daha qısa yazılışdır.

alert (counter); // 1

Qeyd: artma/azalma sadəcə dəyişənlərə istifadə edilə bilər. 5++ kimi istifadə də xətaya səbəb olacaqdır. ++ və -- operatorları dəyişəndən əvvəl vəya sonra yazıla bilər. Operator dəyişəndən sonra yazıldıqda “postfix form” adlanır.

Operator dəyişəndən əvvəl yazıldıqda “prefix form” adlanır. Bu ifadələrin hər ikisidə eyni əməliyyatı aparır: counteri 1 artırın. Fərq varmı? Bəli, ancaq ++/- ni istifadə etsək görə biləcik.

İzah edim : Bildiyimiz kimi bütün operatorlar bir dəyər geri qaytarır. Artma/azalma isə istisna deyildir. Postfix yeni dəyəri verərkən, prefix form bizə əvvəlki dəyəri geri qaytarır.

Fərqi görmək üçün nümunə :

let counter = 1;

a = ++ counter olsun; // (*)

alert (a) ' // 2

Sətirdə (*), prefix form ++ counter- counterin dəyərini artıracaqdır və dəyəri 2 döndərəcəkdir.

İndi postfix istifadə edək:

let counter = 1;

a = counter ++ olsun; // (*) ++ counter- counter ++ ilə dəyişdirildi

alert (a) ' // 1

Sətirdə (*) postfix form counter ++ da counter-i artırır ancaq əvvəlki dəyəri geri qaytarır (artımdan əvvəl). Yəni, alert 1 göstərir.

Nəticə

• Artırma / azaltma nəticə istifadə edilməyəcəksə, formun seçimində fərq yoxdur:

let counter = 0;

counter ++;

++ counter;

alert (counter); // 2, yuxarıdakı nümunə ilə eyni oldu.

• Dəyəri artımaq və nəticəni istifadə etmək istəyiriksə, post form -a ehtiyacımız vardır:

let counter = 0;

alert (++ counter); // 1

• Dəyəri artırıb və nəticəni istifadə etmək istəmiriksə, prefix form- a ehtiyacımız var:

let counter = 0;

alert (counter ++); // 0

Digər operatorlar arasında artış/azalış

++ / -- operatorları ifadələrin içərisində də istifadə etmək olar. Digər operatorlardan əvvəl yerinə yerirlir.

Nümunə:

let counter = 1;

*alert (2 * ++ counter); // 4*

Müqayisə edək:

let counter = 1;

*alert (2 * counter ++); // 2, çünkü counter ++ əvvəlki dəyəri qaytaracaqdır*

Yəni alert 1 göstərəcəkdir.

"Tək sətir – tək əməliyyat" yazı tərzi:

let counter = 1 ;

*alert (2 * counter);*

counter ++;

Aralıq operatorları

Aralıq operatorları arqumentləri 32 bitlik say sistemi olaraq qəbul edir və ikili say sistemində işləyir.

Bu operatorlar yalnız Javascriptə aid deyildir. Çox proqramlaşdırma dilləri dəstləkləyir.

Operatarların siyahısı:

•VƏ (&)

• VƏYA (|)

• XOR (^)

• DEYİL (~)

• SOL SİMVOL (<<)

• SAĞ SİMVOL (>>)

• SIFIRDAN SAĞ SİMVOL (>>>)

Bu operatorlar çox nadir hallarda istifadə edilir. Onları başa düşmək üçün aşağı say sisteminə keçid etmək lazımdır və indi halda bunu etmək lazım deyil. Çünki bunlara ehtiyacımız olmaycaqdır.

Yerindəcə dəyişdirmə

Ümumilikdə dəyişən üzərində operatordan istifadə etdikdə yeni dəyəri eyni dəyişəndə saxlamaq lazımdır.

Məsələn:

```
let n = 2;

n = n + 5;

n = n * 2;
```

Bunun qısa yolu += və *= operatorlarından istifadə edərək yazmaqdır:

```
let n = 2;

n + = 5; // indi n = 7 (n = n + 5 ilə eynidir)

n * = 2; // şimdi n = 14 (n = n * 2 ilə eynidir)

alert (n); // 14
```

Bütün operatorların və aralıq operatorların qısa yazılış forması var: / =, - =, və s.

Bu operatorlar normal bərabərlik kimi üstünlüyə malikdir. Bu səbəblə digər operatorlardan sonra işləyirlər:

```
let n = 2;

n * = 3 + 5;

alert (n); // 16 (əvvəl sağ hissə hesablandı, n * = 8 ilə eynidir)
```

Vergül

Vergül operatoru maraqlı operatorlardan biridir. Qısa kod yazmaq üçün istifadə edilir. Hər ifadədən sonra vergüldən istifadə edilir. Ancaq sonuncu ifadə üçün istifadə edilmir.

Məsələn:

```
let  a = (1 + 2, 3 + 4);

alert (a); // 7 (3 + 4-ün nəticəsidir)
```

Burda ilk ifadə 1+2 hesablanır və nəticə atılır. Daha sonra 3+4 hesablanır və nəticə olaraq göstərir.

Vergül ən aşağı üstünlüyə malikdir

Vergül operatoru = operatorunda belə aşağı üstünlüyə malikdir. Bu səbəblə yuxarıdakı nümunədə mörtərizənin olması vacibdir.

Bunlar olmadan : a= 1+2, 3+4 əvvəl + operatoru hesablandı. a=3, 7 oldu. Sonra isə a=3 nəticəsini götürəcəkdir və sonrasına diqqət etmiyəcəkdir. (a=1+2),3+4 kimi

Əgər sonuncunu nəzərə almıyacaqdısa niyə ehtiyacımız var buna?

Bəzən developerlər bir necə əməliyyatı bir sətirdə etmək üçün qarışıq struktur hazırlayar.

Nümunə:

```
// bir sətirdə 3 əməliyyat
 (a = 1, b = 3, c = a * b; a <10; a ++) {
 ...
}
```

Bu cür texnikanı bir çox Javascript frameworkun-də istifadə edilir. Bu səbəblə bunun izahını verdim.

12. Müqayisə operatorları

Riyaziyyatda bir çox müqayisə operatoru bilirik:

• Böyük / kiçik : a> b, a <b.

• Böyük / kiçik və ya bərabərdir: a>= b,

a<= b.

• Bərabərlik: a == b (diqqət çüt bərabərlikdir. Tək bir a = b simvolu bir dəyişəni başqa dəyişənə bərabər etmə).

• Bərabər deyil. Riyaziyyatda ≠- bu cür yazılır. Ancaq Javascriptdə fərqli olaraq nida vardır: a! = B.

Boolean nəticəsi

Digər bütün operatorlar kimi müqayisədə də dəyər verir. Bu halda da dəyər booleandır.

• true - "bəli", "düzgün" və ya "real" mənasını verir.

• false - "xeyr", "yanlış" veya "real olmayan" mənasını verir.

Məsələn:

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

Müqayisənin nəticəsi hər hansı dəyişənə verilə bilər:

```
let netice = 5> 4; // müqayisənin nəticəsi

alert (netice); // düzgün
```

String müqayisəsi

Bir stringin digər stringdən böyük olub olmadığını yoxlamaq üçün Javascriptdə “yazı” vəya “leksik” müqayisə edilir.

Başqa sözlə desək stringlər hərf hərf ilə müqayisə edilir.

Müqayisə:

```
alert ('Z'> 'A'); // düzgün

alert ('Glow'> 'Glee'); // düzgün

alert ('Arı'> 'Ol'); // düzgün
```

İki stringin müqayisə alqoritması sadədir:

1. Hər iki stringin ilk simvolunu müqayisə edir.

2. İlk stringdən gələn simvol digər stringdən daha böyüksə (və ya daha az), ilk string ikincidən böyükdür (və ya daha az). Yoxlanılmışdır.

3. Əks təqdirdə hər iki stringin də ilk simvolu eynidirsə, ikinci simvol eyni şəkildə müqayisə edilir.

4. Hər hansı stringin sonuna qədər davam edir.

5. Hər hansı string də eyni uzunluğa sahibdirsə , o zaman bərabərdir. Əks təqdirdə , daha böyük string böyükdür.

Yuxarıdakı nümunlərdə ,'Z'> 'A'

Qarşılaşdırılması ilk addımda nəticə verərkən, "Glow" ve "Glee" dəki simvolları qarşılaşdırılacaq:

1. G, G ilə eynidir.

2. l, l ilə eynidir.

3. o, e'dən böyükdür. Burada isə ilk string daha böyükdür.

Bu qarşılaşdırma hərfə görə deyil Unicode-a görə edilir.

Məsələn: Böyük hərflə “A”, kiçik hərf “a” ilə eyni deyildir. Hansı daha böyükdür bəs? Kiçik hər “a” böyükdür. Səbəb? Çünki kiçik hərfli simvol Javascriptin istifadə etidiyi daxili şifrələmə cədvəlində daha böyük index-sə sahibdir (Unidecode). Bunun String bölməsindəki spesifik əlamətlərinə və qiymətlərinə geri qayıdacıq.

Fərqli tiplərin müqayisəsi

Fərqli tiplərin dəyərlərinin müqayisəsi zamanı Javascript dəyərləri ədədlərə çevrilir.

Məsələn:

```
alert ('2'> 1); // true, '2' stringi 2 ədədinə çevriləcəkdir.
alert ('01' == 1); // true, '01' stringi 1 ədədinə çevriləcəkdir.
```

Boolean dəyərlər üçün true 1 və false 0 olur.

Məsələn:

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

Qəribə nəticələr

Eyni zamanda mümkündür:

• İki dəyər bərabərdir.

• Biri bir boolean olarak, diğeri ise bir boolean dəyər deyildir.

Məsələn:

```
a = 0;
alert (Boolean (a)); // səhv

b = "0";
alert (Boolean (b)); // düzgün

alert (a == b); // düzgün
```

JavaScriptə görə bu nəticə olduqca normaldır. Bərabərlik operatoru ədəd olaraq çevirib yoxlayır. (bu səbəblə "0" 0 olur)

Qatı bərabərlik

Normal bərabərlik yoxlamasında səhvlik vardır. 0-i səhvdən ayıra bilmir.

alert (0== false); //düzgün

Eyni nəticə boş stringlədə olur :

alert (" " == false); // düzgün

Bunun səbəbi fərqli tipdəki dəyişənləri == operatoru ədədə (number)-ə çevirməsidir. Boş string və 0 yanlış dəyəri kimi olacaqdır.

0 – ı yanlışdan ayırmaq üçün nə etməliyik?

Qatı bərabərlik operatoru ===, tipi çevirmədən bərabərliyi yoxlayacaqdır.

Başqa sözlə əgər a və b fərqli tiplərdədisə, o zaman a===b bərabərliyindən çevirmədən false dəyərini döndərəcəkdir.

Yoxlayaq:

alert (0 === false); // false, çünki tiplər fərqlidir

Eyni zamanda "qatı bərabərlik" operatoruda vardır. Qatı operator ilə daha dəqiq və xətalara daha az vermək üçün istifadə edilir.

Null və undefined müqayisəsi

İndi isə bu tiplərin müqayisəsinə baxaq

Boş və bilinməyən dəyərlərin digər dəyərlərlə qarşılaşdırılması zamanı məntiqə uymayan nəticələr ortaya çıxır.

Qatı bərabərlik zamanı === bu dəyərlər fərqlidir, çünki hər biri fərqli tipdir.

Qatı olmayan bərabərlik zamanı üçün == isə xüsusi qayda vardır. Bu iki tip bir-birinə bərabərdirlər.

alert (null == undefined); // düzgün

Riyazi və digər müqayisələr üçün <> <=> =

null / undefined ədədlərə çevrilir: null 0 , undefined NaN olacaqdır.

İndi bu qaydaları gördüyümüzdə qəribə hallara baxaq. Və daha vacibi onlara necə aldanmamaq lazımdır.

Maraqlı nəticə

Null və 0-ı müqayisə edək:

alert (null> 0); // (1) yanlış

alert (null == 0); // (2) yanlış

alert (null> = 0); // (3) düzgün

Riyazi olaraq bu çox gözlənilməzdir. Son nəticə sıfırın sıfırdan böyük vəya bərabər olduğunu düzgün qəbul edir. Bu səbəblə yuxarıdakı müqayisəni birinin düzgün olması lazımdır. Ancaq ikisidə yanlışdır.

Bunun səbəbi bərabərlik operatorunun == və müqayisə operatorlarının > <> = <= fərqli işləmə üsulunun olmasıdır. Müqayisədə null dəyəri rəqəmə çevrilərək 0 olacaqdır.

(3) null> = 0 düzgündür və (1) null> 0 yanlışdır.

Həmdə bilinməyən və null üçün bərabərlik yoxlaması == hər hansı çevrilmə olmadan bərabər olacaqdır və başqa heç bir şeyə ehtiyac qalmayacaqdır. Bu səbəblə (2) null==0 səhvdir.

Bərabəri olmayan undefined

Bilinməyən dəyər digər dəyərlərlə müqayisə edilməməlidir:

alert (undefined> 0); // yanlış (1)

alert (undefined <0); // yanlış (2)

alert (undefined == 0); // yanlış (3)

Niyə sıfırdan xoşu gəlmir? Hər zaman yanlış geri qaytarır.

Müqayisələr (1) və (2) yanlış dəyəri geri qaytarır. Çünki undefined NaN-a çevrilir və NaN a bütün müqayisələr üçün false geri qaytaran xüsusi number tipidir.

Bərabərlik yoxlamasında (3) false geri qaytarır, çünki bilinməyən dəyər sadəcə boş , bilinməyən və başqa dəyərlərə bərabər deyildir.

Problemlərdən yan keçmə

Niyə bu nümunələrə baxdıq? Bu xüsusiyyətləri hər zaman yadda saxlamalıyıqmı ?

Əslində indi çətin görünən bunlar zamanla yaddaşınıza yazılacaqdır. Ancaq onlarla yaranan problemlərdən yan keçməyin metodu vardır:

Hər hansı müqayisəyə undefined/ null dəyərlərlə, qatı bərabərlik ===. İstisna hallardan kənarda isə == operatoru ilə yoxlayın.

Müqayisələrlə > => <<= nə etdiyini tam olaraq bilmirsinizsə, null/undefined olacaq dəyər istifadə etməyin. Bu dəyişən bu dəyərlərə sahibdirsə, bunları ayrı ayrı yoxlayın.

Nəticə

Müqayisə operatorları Boolean dəyərləri geri qaytarır.

Stringlər – “söz” sırasında hərf hərf müqayisə edilir.

Fərqli tiplərin dəyərləri müqayisə edildiyində, çevrilmələr olur. (Qatı bərabərlikdən başqa)

Dəyərlər əgər null və undefined-ə bərabərdirsə, digər heç bir dəyərə bərabər olmayacaqdır.

> Vəya < kimi müqayisə istifadə edilərkən boş və ya undefined dəyərlər qarşı diqqətli olun. Null/undefined ayrı ayrı yoxlamaq məsləhətdir.

13. Dialoq pəncərələri : alert, confirm, prompt

Kitabın bu hissəsində real nümunələr olmadan Javascript dilində danışacıq. Brauzerinizdən istifadə edərək dialoqların necə qurulacağına baxmalıyıq. Bu bölmədə əsas brauzer funksiya bildirişləri, istək və təsdiqi haqqında öyrənəcik.

Alert

Sintaksisi

alert (yazı);

Bu kod yazı göstərəcəkdir və istifadəçi "OK" -a klikləyənə kimi bütün funksiyalar dayanacaqdır.

Məsələn:

```
alert ( "Salam");
```

Yazının olduğu kiçik pəncərəyə **modal window** deyilir.

Prompt

Bu metod iki arqument qəbul edir:

netice = prompt (başlıq, [default]);

Modal pəncərədə istifadəçidən bir dəyər istəyər və “OK/CANCEL” - düymələrindən birinə kliklənməsini gözləyər.

Başlıq - İsitfadəçiyə göstəriləcək yazı olacaqdır.

Default - İstəyə görə ikinci parametrdir. Başlanğıç dəyər kimi qəbul edilir.

İsitfadəçi inputun daxilinə bir şey yazıb, “OK”-təsdiq və “Cancel”-ə vəya ESC – düyməsini sıxıb imtina edə bilərsiniz.

Yazılmış nəticəni göndərir vəya yazılmayıbsa null dəyərini göndərəcəkdir.

Məsələn:

let yash = prompt ('Neçə yaşınız var?', 75);

alert (`$ {yash} yaşınız var!`); // 75 yaşın var!

İE-də : hər zaman default dəyər olmalıdır.

Bu əslində istəyə bağlı parametrdir. Ancaq İnternet Explorer “undefined” olaraq görəcəkdir.

Nəticəni görmək üçün İnternet Explorerdə işə salın :

let test = prompt ("Test");

Bu səbəblə, İE-də həmişə default dəyəri yazmaq məsləhət görülür:

let test = prompt ("Test", ''); // <- IE üçün

Confirm

Sintaksisi:

netice = confirm (sual);

Funksiyanın təsdiqləməsi üçün istifadə edilən confirm iki düymədən ibarətdir: OK və Cancel

OK düyməsi sıxıldıqda ya düzgün olarsa, True dəyəri, səhv olarsa, False dəyərini verəcəkdir.

Nümunə:

let buLiderdir = confirm ("Lider sizsiniz?")

alert (Mənəm); // OK basılarsa, true

Nəticə

İstifadəçilərlə brauzerdə əlaqə qurmaq üçün 3 metoda baxdıq:

Alert - Bildiriş mesajı göstərəcəkdir.

Prompt - İsitifadəçidən mətn tələb edən mesaj qutusudur. Ok basılarsa, mətni göndərir vəya Cancel vəya ESC basılarsa, null dəyərini döndərəcəkdir.

Confirm - Mesajı göstərmək üçün istifadə edilir. İstifadəçidən "OK" və ya "Cancel"-dən birinin kliklənməsini gözləyir. Ok- true dəyərini, Cancel/ESC false dəyərini verəcəkdir.

Bütün bu metodlar modaldır: açılmış pəncərə bağlanana kimi digər bütün funksiyalardan istifadəyə qadağan qoyacaqdır.

Yuxarıda bütün metod üçün standart vardır:

1.Modal pəncərəsi brauzerdə tərəfindən qəbul edilir. Adi halda mərkəzdə olur.

2. Pəncərənin dizaynını brauzerə uyğun olaraq təyin olunur.

14.Şərt operatorları : if, “?”

Bəzə fərqli şərtlərə görə fərqli nəticələr almaq lazım olur. Bunu etmək üçün “sual işarəsi” operatoru olaraq adlandırılan if ifadəsini və şərt operatorunu istifadə edə bilərik.

“İf” ifadəsi

İf ifadəsi şərtlər qoymaq üçün istifadə edilir və nəticə düzdürsə, bir kod hissəciyinin işləməsi üçün istifadə edilir.

Məsələn:

```
let il = prompt ('ECMAScript-2015 nə zaman buraxılmışdır?' ');


if (il == 2015) alert (Düzdür!');
```

Yuxarıdakı nümunədə şərt sadə yoxlama nümunəsi oldu (year==2015), ancaq çox qarışıqda ola bilər.

Birdən çox ifadə istifadə etmək istəyiriksə, kod hissəsini skopların içərisinə daxil etmək lazımdır:

```
if (il == 2015) {
  alert ("düzgündür!");
  alert ("ağıllısan!");
}
```

İstfadə edəcəyimiz bir ifadə olsa belə kodlar skopların {} daxilində yazılmalıdır. Həmdə oxunması rahat olacaqdır.

Boolean çevrilməsi

İf (…) ifadəsi skop içərisindəki kodları dəyərləndirib, nəticəni boolean tipini çevirib göndərəcəkdir.

Tiplər arasında əlaqə bölməsini yada salaq:

•Dəyər 0, boş string “” , null ,undefined və NaN – false dəyəri alır.

• Qalan dəyərlər isə true olur.

Buna görədə düz olarsa, aşağıdakı kod heç vaxt işə düşməz:

```
if (0) {
// 0 false-dir.
  ...
}
```

… Və bu halda həmişə true dəyəri alır:

```
if (1) {// 1 true
  ...
}
```

Əvvəlcədən düzgün dəyər qeyd edə bilərik. Nümunə:

```
let cond = (il == 2015); // bərabər olarsa düz, əks halda səhv olsun.
if (cond) {
  ...
}
```

“Else” ifadəsi

İf ifadəsin istəyə bağlə “else” bölməsi istifadə edilə bilər. Bu hissədə şərt əgər səhv olan zaman işlədiləcək kod hissəciyi yerləşdirilir.

Nümunə:

```
let il = prompt ('ECMAScript-2015 neçənci ildə buraxılmışdır?' ');


if (il == 2015) {
  alert ('Doğrudur!');
} else {
  alert (Səhvdir'); // 2015 -dən başqa dəyər yazılarsa
}
```

Birdən çox şərt : “else if”

Bəzən şərt qoyarkən bir necə hal üçün yazmaq istəyə bilərik.

Else if buna imkan verir.

Nümunə:

```
let il = prompt ('ECMAScript-2015 neçənci ildə buraxılmışdır?' ');

if (2015 < il) {
  alert ('Bu ildən sonra olub ...');
} else if (il> 2015) {
  alert ('Bu ildən əvvəl olub');
} else {
  alert ('Əlaqəsizdir!');
}
```

Yuxarıdakı kodda Javascript ilk əvvəl <2015 ilini yoxlayacaqdır. Bu olmazsa, sonrakı şərt olan il> 2015 yoxlayacaqdır. Buda səhv olarsa, sonuncu ifadəni işə salacaqdır.

Kod blokları çox ola bilər. Sonuncu istəyə bağlıdır.

Şərt operatoru "?"

Bəzən dəyişən vermək lazım olur.

Məsələn:

```
let girishQadagandir ;
let yash = prompt ('Neçə yaşınız var?', '');

if (yash> 18) {
  girishQadagandir = true;
} else {
  girishQadagandir = false;
}

alert (girishQadagandir);
```

“?” operatoru “İf” operatorunun daha qısa və sadə yazılış şəklidir.

Operator sual işarə ilə təmsil olunur. “Tanery” olaraq adlandırılır, çünki operatorun 3 dəyəri vardır.

Sintaksisi:

Let netice = şert? deyer1: deyer2;

Şərt yoxlanılır: deyer1 düzdürsə deyer 1,əks halda isə deyer2 seçilir.

Məsələn:

let accessAllowed = (yaş> 18) ? true:false

Texniki olaraq > 18 ifadəsi mörtəziləri nəzərə almıya bilər. Sual işarəsi üstünlük cəhətdən aşağıdır, bu səbəblə müqayisə sonra aparılır.

Bu nümunə əvvəlki nümunə ilə eyni nəticəni verəcəkdir:

```
//müqayisə operatoru üstünlüyə sahibdir. “yash>18”
// Mörtərizə içində yazmağa ehtiyac yoxdur
girishQadagandir = yash> 18? true false;
```

Ancaq mörtərizələr kodu daha yaxşı oxumağa imkan verir. İstifadəsi məsləhətdir.

Yadda saxlayın:

Yuxarıdakı nümunədə müqayisə işarəsi true/false dəyəri verdiyindən sual işarəsində istifadə etmiyə bilərsiniz.

// eyni

let girishQadagandir = yash> 18;

Birdən çox şərt olduqda : '?'

Birdən çox şərt qoymaq üçün "?" operatorunda istifadə edə bilərsiniz.

Nümunə:

```
let yash = prompt ('yash?', 18);


let cavab = (yash <3)? 'Salam Dünya!' :
  (yash <18)? Salam!' :
  (yash <100)? Sağol!' :
  'Son!';
alert( cavab );
```

Anlamaqda çətinlik çəkə bilərsiniz. Ancaq daha aydın izah üçün bu hissəni diqqətli oxuyun:

1. İlk sual işarəsi yash<3 olub, olmadığını yoxlayar.

2. Əgər doğrudursa "Salam Dünya" nəticəsini verəcəkdir. Əks təqdirdə 18 yaşını yoxlayacaqdır ":"-əks halda davam etməsini bildirir.

3. Bu düzdürsə, “Salam” dəyərini verəcəkdir. Əks halda <100 yaşını yoxlayacaqdır “:” dan sonra ifadəsi davam edəcəkdir.

4. Əgər doğru olarsa “Sağol” ifadəsini verəcəkdir. Əks halda sonuncu nəticəni döndürəcəkdir.

? yerinə if... else istifadə etsəydik:

```
if (yash <3) {
  cavab = 'Salam Dünya!';
} else if (yash <18) {
  cavab = 'Salam';
} else if (yash <100
cavab = 'Sağol';
} else {
  cavab = 'Son!';
}
```

‘?’-in istifadə edilmədiyi yerlər

Bəzən sual “?” işarəsi if-in əvəzinə istifadə edilir :

```
let shirket = prompt ('Javascripti hansı şirkət yaratdı?', '');


(shirket == 'Netscape')?
   alert ('Düzgün!'): alert ('Səhv.');
```

Şirkətlə bağlı olaraq == “Netscape”, ? ifadəsində birinci ifadə yoxlandıqdan sonra ikinci ifadə yoxlanılacaqdır. Sonda isə alert işə düşəcəkdir.

Burda dəyişənə dəyər olaraq nəticəni vermirik. Sadəcə hala uyğun olaraq fərqli kodların işləməsini istəyirik.

Sual işarəsini bu şəkildə ifadəsi məsləhət deyildir. Bu if ifadəsindən qısa olsada oxunarlılığı heçdə yaxşı deyildir.

Müqayisə üçün sual yerinə if istifadə edək:

```
let shirket = prompt ('Javascripti hansı şirkət yaratdı?', '');
if (shirket == 'Netscape') {
  alert ( 'Düzgün!');
} else {
  ( 'Səhv.') ;
}
```

Gözlərimizlə kodu daha rahat oxumağa imkan verir. Anlaşılan şəkildə yazılışmış bu kod tək sətirlik koddan daha rahatdır.

“?” operatorunu nə məqsədlə istifadə edirik? Nəticəyə uyğun olaraq dəyərin verməsi üçün istifadə edilir.

15. Məntiqi operatorlar

JavaScriptdə 3 məntiqi operator var: || (VƏYA), && (VƏ),! (DEYİL). Məntiqi operatorlar olmağına baxmıyaraq sadəcə boolean tipidə deyil, müxtəlif tiplərdə də istifadə etmək mümkündür. Onlar nəticələridə fərqlidir.

İzahlara keçək

|| (VƏYA)

"VƏYA" operatoru- iki düz xətt ilə işarə olunur:

```
netice = a || b;
```

Klassik proqramlaşdırmada məntiqi OR sadəcə boolean dəyərini istifadə etmək üçün isitfadə edilirdi. Arqumentlərdən düzgün olanını true olaraq. Əks halda isə false dəyərini döndərəcəkdir. Javascriptdə bu operatordan çox istifadə edilir. Əvvəlcə boolean tipində necə nəticə verdiyinə baxaq.

Dört məntiqi kombinasiya vardır:

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

alert (false || true); // true

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

alert (false || false); // false

Gördüyünüz kimi hər ikisidə false olandan başqa hər zaman true dəyərini verir.

Əgər istifadə edilən boolean deyildirsə, nəticə əldə etmək üçün boolean dəyərinə çevirəcəkdir.

Məsələn, 1 – true , 0- false dəyərini verir:

```
if (1 || 0) {// eyni ilə if kimi işləyir (true || false)
  alert ('Düzgündür!');
}
```

Çox zaman VƏYA || ilə verilənlərindən hansının düzgün olub olmamasını if ifadəsində istifadə edilir.

Məsələn:

```
let saat = 9;
if (saat <10 || saat> 18) {
  alert ('Ofis bağlıdır.');
}
```

Daha çox şərt yaza bilərik;

```
let saat = 12;

let hefteSonu = true;

if (saat <10 || saat> 18 || hefteSonu) {

  alert ('Ofis bağlıdır.'); // Həftə Sonu

}
```

VƏYA ilk düzgün dəyərini tapır

Yuxarıda izah edilən məntiq biraz klassikdir. İndi isə Javascriptin "əlavə" xüsusiyyətlərini izah edək. Genişləndirilmiş alqoritma aşağıdakı kimi işləyir.

Birdən çox OR dəyəri verildikdə:

netice= deyer1 || deyer2 || deyer3;

VƏYA || operatoru aşağıdakıları yerinə yetirir:

• Dəyərlər solda sağa doğru yoxlanılır.

• Alınmış nəticə üçün boolean dəyərinə çeviri. Nəticə düzdürsə, çevrilmə dayancaqdır və əsl dəyəri verəcəkdir.

• Bütün dəyərlər yanlışdırsa, son dəyəri verəcəkdir.

Dəyər çevrilməsi olmadıqda original halda geri döndürəcəkdir.

OR "||" ifadəsində true dəyəri olmadıqda ilk və ya son dəyəri verəcəkdir.

Nümunə:

```
alert (1 || 0); // 1 (1 Düzgündür)

alert (true || 'nə olursa olsun'); // (düzgündür)

alert (null || 1); // 1 (1, düzgündür)

alert (null || 0 || 1); // 1 (düzgündür)
```

Sadə, klassik, boolean VƏYA ilə müqayisə etdikdə maraqlı nəticələr əldə etmiş olacıq.

İfadələrdəki düzgün dəyəri almaq

Məlumatın tərkibində null/undefined olan dəyişənlər ola bilər. Düzgün dəyəri necə tapa bilərik?

OR || istifadə edə bilərik.

```
let indikiİstifadeci = null;

let ilkinİstifadəci = "Nail";

let name = indikiİstifadeci || ilkinİstifadəci || "adsiz";

alert (ad); // "Nail" -nu seçəcəkdir. İlkin düzgün dəyər olduğu üçün
```

Həm currentUser həmdə defaultUser yanlışdırsa, “unnamed” nəticəsi olacaqdır.

2. Qısa yolla dəyər alma.

İsitfadə ediləcəklər dəyərlərlə bağlı deyil həmdə ifadələrə də bağlı ola bilər. VƏYa bunları solda sağa dəyərini alıb və test edəcəkdir. Dəyər əgər true olarsa, dayanacaqdır və nəticəni verəcəkdir. Buna "qısa dəyər alma" deyilir.

İkinci arqument olaraq verilən ifadənin dəyişən alması bəzi problemlərin yaranmasına gətirib çıxarır.

Aşağıdakı nümunədə x-ə dəyər verilmir:

```
let x;

true || (x = 1);

alert (x); // undefined, çünki (x = 1) dəyər verilmədi
```

Bunun yerinə ilk arqument səhvdirsə , || ikincisini yoxlayacadır. Beləliklə istədiyimiz nəticəni alırıq:

```
let x;

false || (x = 1);

alert (x); // 1
```

Bir şərt halı çox sadədir. Dəyərlərə görə əgər sonda bir düzgün nəticə almazsa heç bir kod hissəciyi işə düşməyəcəkdir. Gördüyünüz kimi bu halda if istifadə etmək daha qısa yoldur. Dəyərlər boolen nəticələr qaytaracaqdır. İlki true , ikincisi isə false dəyərində işləyəcəkdir. Kodun başa düşülən olması üçün bir "qayda" istifadə etmək daha yaxşıdır. Ancaq bəzi hallar üçün keçərlidir.

&& (VƏ)

AND vəya iki və işarəsi ilə göstərilir &&:

netice = a & b;

Klassik proqramlaşdırmada hər iki dəyişən düzgün vəya səhvdirsə , AND istifadə edilir :

alert (true and true); // true

alert (false and true); // false

alert (true and false); // false

alert (false and false); // false

İf ilə nümunə:

```
let saat = 12;

let deqiqe = 30;

if (saat == 12 && deqiqe == 30) {

  alert ('Saat 12:30-dur');

}
```

OR'da olduğu kimi hər hansı istənilən dəyərə AND metodunda istifadə etmək mümkündür:

```
if (1 && 0) { // true and false olaraq nəticə verəcəkdir

  alert ("İşləməyəcəkdir. Çünki nəticə yanlışdır");

}
```

AND ilk olaraq yanlış dəyəri tapır

Birdən çox AND dəyəri verilə bilər:

```
netice = deyer1 && deyer2 && deyer3;
```

AND && operatoru aşağıdakı kimi işləyir:

• Dəyərlər solda sağa doğru yoxlanılır.

• Hər dəyər üçün boolean dəyəri verir. Nə yanlışdırsa, yoxlamağı dayandırıb, əsl dəyəri verəcəkdir.

• Bütün dəyərlər yoxlanıldıqdan sonra sonuncu dəyər nəticə olaraq veriləcəkdir.

Başqa sözlə desək AND – ilk false dəyəri vəya son dəyər verilir.

Yuxarıdakı qaydalar OR-a üçündə oxşardır. Fərq isə AND-ın ilk false dəyərində nəticənin false olmasıdır.

```
// əgər ilk arqument düzdürsə,

// AND- ikinci arqumentidə yoxlayır:

alert (1 && 0); // 0

alert (1 and 5); // 5

// əgər ilk arqument səhvdirsə,

// AND- nəticəni verəcəkdir. İkinci yox kimi sayılacaqdır.

alert (null & & 5); // null

alert (0 && "nə olursa olsun"); // 0
```

Həmdə ard arda bir necə dəyər də verə bilərsiniz. İlk false dəyərinin necə nəticə verdiyinə baxa bilərsiniz:

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

Bütün dəyərlər düzgün olduğunda son dəyər nəticə olaraq verəcəkdir:

alert (1 && 2 && 3); // 3, sonuncusu

VƏ-nin && operatorundan üstünlüyü

Yəni kod a && b || c &&d əsas olaraq && ifadələrinin mörtərizə daxilində eyni üstünlüyə sahibdir:

(a && b) || (c &&d).

Eyni ilə OR kimi AND-in yerinə && operatoru istifadə edilə bilər.

Məsələn:

```
let x = 1;

(x> 0) && alert ('Sıfırdan böyük!');
```

&& nın sağındakı olan şərt ona keçdiyimiz zaman işə salınacaqdır. Yəni, sadəcə (x> 0) doğru olarsa.

Yəni sadə hala baxsaq bir şərtimiz olur :

```
let x = 1;

if (x> 0) {

  alert ('Sıfırdan böyük!');

}
```

&& operatoru qısa yazılışdır. Ama oxunması rahat olduğu halda istifadə edilir.

Bu səbəblə istənilən halda kod strukturunuza uyğun olanı istifadə edin: istər & &, istərsə də AND.

! (NOT)

Boolean-de xeyr işarəsi nida işarəsi ilə göstərilir.

Sintaksisi çox sadədir:

netice =! deyeri;

Operator tək arqument qəbul edir və aşağıdakıları etməyə imkan verir:

1. Şərti boolean tipinə çevirir: true/false.

2. Tərs geri qaytarır.

Məsələn:

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

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

Yalnız bunlar üçün deyil başqa tipdə olan dəyəri boolean tipinə çevirmək üçün istifadə edilir :

```
alert (!! "boş olmayan string"); // true

alert (!! null); // false
```

Yəni, ilk (!)NOT, dəyəri boolean çevirər yada əksinə, ikincisi də (!)NOT, onu tərsinə çevirəcəkdir. Sonda, boolean dəyəyirinə sadə çevrilməmiz vardır.

Eyni nəticəni almağın daha bir yolu vardır. Boolean-dan istifadə edərək.

```
alert (Boolean ("boş olmayan string")); // true

alert (Boolean (null)); // false
```

! (NOT) bütün məntiqi operatorların ən yüksəyidir. Bu səbəblə && veya || operatorlarından əvvəl olur.

16. Dövr etmə: while və for

Çox vaxt əməliyyatların təkrarlanması lazım olur. Məsələn 1ilə 10 arasında rəqəmləri yazdırmaq lazımdır. Dövr etmə bir neçə dəfə ola bilər.

"While" dövr etməsi

While dövretməsi aşağıdakı sintaksislə yazılır:

```
while (şərt) {
  // kod
  // dövr edilənin yazılacağı yer
```

Şərt düzgün olarsa, dövr etdiriləcək yerdəki kod işə düşür.

Məsələn aşağıdakı dövretmə nəticəsi alarkən <3:

```
let i = 0;
while (i <3) {// 0, sonra 1, sonra 2'yi göstərəcək.
  alert (i);
  i ++;
}
```

Dövr etmə kodun yenilənməsinə deyilir. Yuxarıdakı nümunədə dövretmə 3 dəfə yenilənəckdir.

Əgər yuxarıdakı i++ mənfi olarsa, dövretmə sonsuzluğadək davam edəcəkdir (teoremdə). Layihədə brauzer bu dövretmənin dayandırılmasının yollarını izah edir və sever tərəfdə Javascript metodunu dayandıra bilərsiniz.

Hər hansı ifadə və dəyişən (müqayisələr istisna olmaqla) dövretmə metodu ola bilər. Bunun nəticəsi boolean dəyəri verəcəkdir.

Məsələn while (i!=0) yerinə while(i) yazmaq daha qısa yoldur:

```
i = 3;
(i) {// 0 oldanda şərtə görə dövretmə dayancaqdır.
  alert (i);
  i--;
}
```

Tək sətirlik kod üçün nöqtəli vergüldən istifadə etmək lazım deyildir.

Dövr etmədə skop daxilində bir ifadə varsa, skopları {...} yazmamaq olar:

```
i = 3;

while (i) alert (i--);
```

"do ... while" dövretməsi

Şərt do..while-dan istifadə edərək şərt yazıla bilər:

```
do {
  // kod hissəsi
} while (şərt);
```

Dövretmə əvvəl kodları işə salır sonra isə şərtin düzgünlüyünü yoxlayacaqdır. Bu şərt sonlanana kimi davam edəcəkdir.

Məsələn:

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

Bu üsuldan istədiyiniz vaxt istifadə edə bilərsiniz. Ümumilikdə digər : while (...) {...} şərti istifadə edilir.

"For" dövr etməsi

For dövr etmə metodu ən çox istifadə edilən üsuldur.

Sintaksisi aşağıdakı kimidir:

```
for (başlangıc; şərt; addım) {
  // ... kod hissəsi...
}
```

Aşağıdakı nümunədən bu metodun mənasına baxaq. Aşağıdakı dövretmədə i 0 və 3 arasında alert işə düşür:

```
(i = 0; i <3; i ++) {{0, sonra 1, sonra 2-ni göstərir.
  alert (i);
}
```

For-un hissələrinə bir-bir baxaq:

Hissələr

Başlanğıc - i = 0 Dövretmə başladıqdan sonra bir dəfə işə salınır.

Şərt - i <3 Hər dövretmə yenilənməsində yoxlanılır. Səhv olarsa, dövretmə dayanacaqdır.

Addım – i++ şərtdən sonra işə düşür.

Kod hissəsi alert(i) şərt düzgün olduğu halda təkrar olaraq işə düşəcəkdir.

Ümumi dövretmə alqoritması aşağıdakı kimi işləyir:

İşə sal və başla

Başlat

→ (əgər düzdürsə → kod hissəsini işə sal və addım at)

→ (əgər düzdürsə → kod hissəsini işə sal və addım at)

→ (əgər düzdürsə → kod hissəsini işə sal və addım at)

→ ...

Dövretmələri anlamaqda yenisinizsə, aşağıdakı nümunədə daha açıq şəkildə baxa bilərsiniz:

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


// dövretmə işə başladı
let i = 0
// əgər düzdürsə → kod hissəsini işə sal və addım at
if (i <3) {alert (i); i ++}
// əgər düzdürsə → kod hissəsini işə sal və addım at
if (i <3) { alert (i); i ++}
// əgər düzdürsə → kod hissəsini işə sal və addım at
if (i <3) { alert (i); i ++}
// ... son, if i = = 3 dir.
```

Sətir içində dəyişən verilməsi

Burada “counter” dəyişəni tam dövretmədə bildirilir. Buna “sətir içi” dəyişən deyilir. Bu cür dəyişənlər sadəcə dövretmə içində verilə bilər.

```
  ((i = 0; i <3; i ++) {
  alert (i); // 0, 1, 2
}
alert (i); // xəta belə dəyişən yoxdur
```

Dəyişən yaratmaq yerinə mövcud olanı istifadə edə bilərsiniz:

```
let i = 0;
(i = 0; i <3; i ++) {// olan bir dəyişəndən istifadə edin
  alert (i); // 0, 1, 2
}
alert (i); // 3 dəyər alacaqdır.
```

Kod sturukturunda qısaltmalar

Yazılmış kodda hər hansı bir hissəni yazmaya bilərsiniz. Nümunə olaraq dövretmədə başlanğıcı ata bilərik.

Nümunə :

```
let i=0, // dəyişəni bildirdik
((i <3; i ++) {// "başlama" hissəsi lazım deyil
  alert (i); // 0, 1, 2
}
```

Addım hissəsini də yazmıya bilərik:

```
let i = 0;

(; i <3;) {
  alert (i ++);
}
```

Bu (i<3) ilə eyni dövretməni edəcəkdir.

Əslində sonsuz dövretmə yaradaraq şərti poza bilərik:

```
for (;;) {
  // sonsuz təkrarlama
}
```

Nöqtəli vergülü yazmağı unutmayın. Bu xətaya səbəb ola bilər.

Dövretməni dayandırma

Normalda səhv olarsa, dövretmə davam edir.

Ancaq xüsusi break metodunu istifadə edərək istənilən yerdə dövretməni dayandıra bilərsiniz.

Məsələn aşağı halda prompta heç bir dəyər verilmədikdə “break” şəklində sorğu işə düşəcəkdir:

```
let cem = 0;

while (true) {

  deyer = +  prompt ("ədəd yazın", ' ');

  if (! deyer) break; // (*)

  cem + = deyer;

}
```

İstifadəçi boş dəyər vəya imtina edərsə, break bu (*) sətirdən sonra işə düşəcəkdir. Ondan sonra isə dövretməni dayandıracaqdır. Yəni bildiriş dayanacaqdır.

“sonsuz dövretmə” – də break-dən istifadə çox yaxşıdır. Çünki dövretməyə nəzarət etmək lazım olduqda istifadə edilə bilinir. Break istənilən yerdə və hətta bir necə yerdə istifadə edilə bilər.

Continue ilə davam etmə

Continue break-in daha yüngül bir versiyasıdır. Bütün dövretməni dayandırmaz. Bunun yerinə hal-hazırkı yenilənməni dayandıracaqdır və dövretmənin yenisinin başlanmasını tələb edəcəkdir.

Aşağıdakı dövretmədə sadəcə tək dəyərlərdən istifadə olunur:

```
((i = 0; i <10; i ++) {

// düzgündürsə, qalan hissəni keç

if (i% 2 == 0) continue;


alert (i); // 1, sonra 3, 5, 7, 9

}
```

i-nin dəyərləri üçün , yoxlama müddəti birincidən sonra üçün təsir göstərir. Bu səbəblə alert həmişə tək dəyərlər üçün çağıralacaqdır.

Continue iç-içə yazılışı azaltmağa kömək edir.

Tək dəyərləri göstərən dövretmə aşağıdakı kimi görünür:

```
((i = 0; i <10; i ++) {
if (i% 2) {
alert (i);
}
}
```

Texniki cəhətdən baxdıqda yuxarıdakı nümunə ilə eynidir. Əlbətdə həmişə istifadə etmək yerinə kodu if blokunda yaza bilərik.

Ancaq mənfi cəhəti odurki, iç-içə yazılış şəklidir. Kod bir neçə sətirdən uzundursa, umumilikdə oxunması çətinləşəcəkdir.

break/continue yerinə '?'-dən istifadə

Yadda saxlayın sintaksis cəhətdən başadüşülən şəkildə yazmaq üçün ? -dən istifadə etməyin. Əsasəndə continue/break -də istifadə edilmir.

Məsələn :

```
if (i> 5) {
alert (i);
} else {
continue;
}
```

... və ?-ilə yazılmış versiyası:

(i> 5)? alert (i): continue; // davam etməsinə icazə verilmir.

Kod işləməyi dayandıracaqdır. Bunun kimi kod sintaksisi xəta verəcəkdir.

Continue/break üçün etiketlər

Bəzən eyni anda birdən çox iç içə dövretmədən çıxmaq lazım olur.

Məsələn aşağıdakı kodda i və j dən dövretmə qurmaq olur. (i,j) (0,0)-(3,3)

Koordinatlarını görmək istədikdə :

```
((i = 0; i <3; i ++) {


(j = 0; j <3; j ++) {


let input = prompt (`Kodlardakı dəyər ($ {i}, $ {j})`, '');
// Bu hissədən tamamlandı hissəsinə çıxmaq istədikdə?
}
}
alert ('Tamamlandı');
```

Dövretməni dayandırmaq üçün istifadəçidən dəyər almağa ehtiyacımız var.

Dəyərdən (input) sonra yazılan break dövretməni dayandırar. Ancaq lazımı etiket deyildir.

Etiket dövretməyə verilmiş addır:

```
labelName: for (...){
...
}
```

Aşağıdaki dövretmədə <labelName> ifadesi həmən dövretməni dayandıracaqdır:

```
outer: for (i = 0; i <3; i ++) {
(j = 0; j <3; j ++) {
let input = prompt (`Kodlardakı dəyər ($ {i}, $ {j})`, '');

// boş bir array vəya imtina edilərsə, dövretməni dayandır
if (! input) break outer; // (*)

// dəyər olarsa, işlə ...
}
}
aler (Yaxşı!');
```

Yuxarıdakı kodda outer yuxarı qaldırdıqda.

Beləliklə idarə tamamilə (*)-dan alert-ə doğru gedəcəkdir:

```
outer
 (for i = 0; i <3; i ++) {...}
```

Continue ayrılıqda başqa etiket ilə istifadə edilə bilər. Bu zaman etiket başqa təkrarlanarsa, ona ötürələcəkdir.

Etiketler "goto" deyildir.

Etiketlər kodu istədiyimiz yerdə idarə etməyə imkan verir.

Məsələn adi halda bunu etmək mümkün deyildir:

```
break etiket; // etiket ötürüləcək? Yox.
```

```
etiket: for (...)
```

Etiket continue/break metodunda əvvəldə olmalıdır.

Nəticə

3 növ dövretməyə baxdıq :

• while – Hər yenilənmədə vəziyyəti yoxlayır.

• do..while – Hər yenilənmədən sonra yoxlanılır.

• for (;;) – Şərt hər yenilənmədə yoxlanılır. Əlavə parametrlər əlavə edilə bilinir.

“Sonsuz” dövretmə almaq üçün çox vaxt while (true) dəyəri istifadə edilir. Bunu dayandırmaq üçün break-dan istifadə edilir.

Mövcud yenilənmədə başqa bir şey etmək istəmiriksə, continue-dan istifadə edə bilərik.

Etiket break/continue-dan əvvəl yazılır. Etiketi ilə kənar dövretəməyə break/continue ilə təsir etməyin tək yoludur.

17."Switch"

“Switch” ilə çoxlu şərt (if) qoymaq mümkündür. Bizə çox seçim imkanı verir.

Sintaksisi

“Switch”-də istənilən sayda bloklar və default dəyər olur.

Nümunə:

```
switch (x) {
  case 'value1': // if (x === 'value1')
    ...
    [break]
  case 'value2': // if (x === 'value2')
    ...
    [break]
  default:
    ...
    [break]
}
```

• x'in dəyəri, birincidən, sonradan ikinciyə qədər olan bərabərlik yoxlanılır.

• Bərabərlik olarsa, break-in olduğu yerə qədər ən yaxın kod işləyəcəkdir.

• Heç bir hal uyğunlaşmazsa, default dəyər işləyəcəkdir. (varsa)

Nümunə:

```
let a = 2 + 2;
switch (a) {
  case 3:
    alert ('Çox kiçik');
    break;
  case 4:
    alert ('Orta');
    break;
  case 5:
    alert ('Çox böyük');
    break;
  default:
    alert ("Belə dəyərləri bilinmir");
}
```

Burada 3 şərt üçün yoxlamağa başlayır. Əgər heç biri uyğun olmazsa, default-da olan şərt işə düşəcəkdir.

Break olmazsa, bütün şərtlər yoxlanılaraq sonuncu şərtə kimi davam edəcəkdir. Nümunə:

```
let a = 2 + 2;
switch (a) {
  case 3:
    alert ('Çok kiçik');
  case 4:
    alert ('Orta');
  case 5:
    alert ('Çok böyük');
  default:
    alert ("Belə dəyər yoxdur");
}
```

Yuxarıdakı nümunədə 3 alert-in ardıcıllıqla işə düşməsini görəcəksiniz :

alert ('Orta');

alert ('Çok böyük');

alert ("Belə dəyər yoxdur ");

İstənilən ifadə switch/ case-in arqumenti ola bilər. Həm break həmdə case-siz yazıla bilər.

Nümunə:

```
let a = "1";
let  b = 0;

switch (+ a) {
  case b + 1:
alert ("bu işləyəcəkdir. Çünkü + a 1, təxmini olaraq b + 1 -ə bərabərdir");
    break;

  default:
    alert ("İşləmədikdə işə düşəcək hissə");
}
```

Burada + a 1- verəcəkdir. Bu halda b + 1 ilə müqayisə etdiriləcəkdir və uyğun olan hissədəki kod işə düşəcəkdir.

“Case” qruplaşdırılması

Bir şərt bir necə case aid olduqda qruplaşdırmadan istifadə edə bilərik.

Nümunə olaraq 3 və 5 ci case-ləri qruplaşdırmağa çalışaq:

```
let a = 2 + 2;
switch (a) {
  case 4:
    alert ( 'Doğru!');
    break;
  case 3: // (*) iki case qruplaşdırıldı
    durum 5:
    alert ( 'Səhvdir!') ;
    alert ("Niyə riyaziyyat öyrənəməyə başlamırsan?");
    break;
  default:
    alert ('Nəticə bilinmir.');
}
```

İndi həm 3 həm də 5 dəki kod hissəciyini göstərir.

Qruplaşdırılması case-lərdə breakla işlətmək olmur. Bu onun mənfi cəhətidir. Buna görə də dayanmadan şərtləri yoxlamadan keçirəcəkdir.

Tip uyğunlaşdırılması

Bərabərlik yoxlanılması hər zaman vacibdir. Dəyərlərin uyğunlaşdırılması üçün eyni tipdə olmalıdır.

Məsələn aşağıdakı koda baxaq:

```
let arg = prompt ("Bir dəyər yazın?");
switch (arg) {
  case '0':
  case 1':
    alert ('Bir vəya sıfır');
    break;
  case '2':
    alert ('İki');
    break;
  case 3:
    alert (İşə düşməyəcəkdir!');
    break;
  default:
    alert ('Bilinməyən dəyər');
}
```

1. 0, 1 üçün ilk alert işə düşür.

2. 2 üçün ikinci alert işə düşür.

3. Ancaq 3 yazıldıqda 3===”3” -ə barəbarə deyildir. Yəni case 3-də kod hissəciyi işə düşməyəcəkdir. Defaut dəyəri işə düşəcəkdir.

18.Funksiyalar

Çox zaman kodun hər hansı bir hissəsində oxşar olan kod hissəcikləri yazmaq lazım olur. Məsələn istifadəçi hesabına daxil olduqda, çıxdıqda və başqa yerdə oxşar olan mesajı göstərmək lazım ola bilər.

Funksiyalar proqramlaşdırmanın əsas hissəsidir. Kodun təkrar edilmədən bir necə dəfə çağırılmasına imkan verir.

Alert(mesaj), prompt(mesaj, default) və confirm (sual) nümunələrinə baxdıq. Ancaq öz istədiyimizə görə də funksiyalar yaza bilərik.

Funksiya yaradılması

Funksiya yaratmaq üçün function ifadəsindən istifadə edirik.

Aşağıdakı şəkildə yazılır:

```
function gosterMesaji () {
  alert ('Hamıya Salam!');
}
```

Funksiyalarda açar sözlər əvvəldə yazılır. Sonra funksiyanın adı, ondan sonra mörtərizə içərisinə (yuxarıdakı kimi boş olan) parametrlər yazılır və skoplar arasında işləyəcək kod hissəciyi qeyd olunur.

Yeni yaradılmış funksiyamız adı ilə çağırıla bilər : gosterMesaj() .

Nümunə:

```
function  gosterMesaj () {
  alert ('Hamıya Salam!');
}
gosterMesaj ();
gosterMesaj ();
```

gosterMesaj () çağrılması ilə kod işə düşəcəkdir. Burada mesajı iki dəfə görəcəyik.

Bu nümunə funksiyaların əsas məqsədlərindən birinin izahıdır: kod təkrarlanmasının qarşısının alınması.

Mesajı vəya göstərilmə şəklini dəyişdirmək lazım olarsa, kodu tək bir yerdə dəyişdirmək yetərlidir.

Daxili dəyişənlər

Funksiyanın içərisində bildirilmiş dəyişən sadəcə o funksiyanın içərisində görülür.

Nümunə:

```
function gosterMesaj () {
  let mesaj = "Salam, JavaScript-əm!"; // local dəyişən

  alert mesajı );
}
gosterMesaj (); // Salam, JavaScript-əm!
alert (mesaj ); // <- Xəta! Dəyişən localda yerləşdirilmişdir
```

Qlobal dəyişənlər

Funksiya global dəyişənlərdən istifadə edə bilir:

```
let userName = 'Nail';
function gosterMesaj () {
  let mesaj = 'Salam,' + userName;
  alert (mesaj);
}
gosterMesaj (); // Salam Nail
```

Funksiyalar global dəyişənlərə tam olaraq təsir göstərə bilir. Onları dəyişdirə bilirlər. Məsələn :

```
 userName = 'Nail';
function gosterMesaj () {
  userName = "Elmar"; // (1) global dəyişən dəyişdirildi
  let mesaj = 'Salam,' + userName;
  alert (mesaj);
}

alert (userName); // Funksiya çağırılmamışdan əvvəl Nail idi
gosterMesaj ();

alert (userName); // Elmar, dəyər funksiya tərəfindən
dəyişdirildi.
```

Global dəyişən lokalda yalnız bir dəfər istifadə edilir.

Funksiyaların içərisində eyni adlı dəyişən olarsa, global dəyişən nəzərə alınmayacaqdır. Məsələn aşağıdakı kodda funksiya lokal userName adlı dəyişən istifadə edir. Global dəyişəni nəzərə almıyacaqdır:

```
userName = 'Nail';
function gosterMesaj () {
  let userName = "Elmar"; // Lokal dəyər bildirildi.
  let mesaj = 'Salam,' + userName; // Elmar
  alert (mesajı);
}
// funksiya özünün userName-ni istifadə edəcəkdir
gosterMesaj ();
alert (userName); // Nail, dəyişmədi. Funksiya global dəyişəni nəzərə almadı.
```

Global dəyişənlər

Yuxarıdakı kodda global istifadəci adı kimi hər hansı funksiyadan kənarda bildirilmiş dəyişənlərə qlobal dəyişənlər deyilir.

Ümumi dəyişənlər hər hansı funksiyada görə bilərsiniz (lokal dəyişənlər tərəfindən təsir göstərilməzsə)

Global dəyişənlərin istifadəsini ən aza endirmək yaxşı kod yazılışıdır.

Müasir kod strukturlarında çox az qlobal dəyişənlər var və yaxud bəzilərində heç yoxdur. Çox dəyişənlər funksiayaların tərkibində olur. Bəzən də layihədəki məlumatları saxlamaq üçün yaxşıdır.

Parametrlər

Parametrlərdən istifadə edərək funksiayalara istənilən dəyər verə bilərik (bunlara funksiya arqumentləri deyilir).

Aşağıdakı nümunədə funksiyanın iki parametri var : from və text.

```
function gosterMesaj (from, text) {// argumentlər: from, metin
 alert (from +':' + text);
}
```

```
gosterMesaj ('Nail', 'Salam'); // Nail: Salam (*)
gosterMesaj ('Elmar', "Necəsən?"); // Nail: Necəsən? (**)
```

Funksiyalarda (*) və (**) yazılan parametrlərə verilmiş dəyərlər lokal dəyişənlərə ötürülür. Funksiya işlədikdə isə onlar çağırılır.

Daha bir nümunə: dəyişənimiz var və onu funksiyaya əlavə edirik. Diqqət edin funksiya dəyişəcəkdir. Ancaq dəyişiklik görülməyəcəkdir. Çünki hər zaman funksiya dəyərin nüsxəsini alır:

```
function gosterMesaj (from, text) {

 from = '*' + from + '*'; // "from" dan daha görünməyə kömək edir.

  alert (from+ ':' + text);
}

let from = "Nail";

gosterMesaj (from," Salam "); // * Nail *: Salam

// "from" dəyəri eynidir. Funksiyal lokal dəyişəni dəyişmişdir.
alert (from); // Nail
```

Default (İlkin) dəyərlər

Heç bir parametrin uyğun gəlmədiyi halda default dəyər işə düşür. Məsələn yuxarıda göstərilmiş funksiya gosterMesaj(from, text) tək arqumentlə çağırılır:

gosterMesaj ("Nail");

Bu səhv deyildir. Belə çağrılmalarda "Nail: undefined" ilə nəticələnəcəkdir. Text olmadığı üçün text===undefined olduğunu zənn edir. Text heç bir dəyər verməyərək istifadə etmək istəyiriksə, textdən sonra = yazmalıyıq:

```
function gosterMesaj (from, text = " yazı yoxdur")  {
  alert (from + ': "+ text);
}
gosterMesaj ( "Nail"); // Nail: yazı yoxdur.
```

İndi əgər text parametrinə heç bir dəyər verilməzsə “yazı yoxdur” dəyərini alacaqdır. Burada verilmiş nümunədə string olaraq verilmişdir. Bunun daha mürəkkən ifadəsi ilə qarşılaşa bilərsiniz. Yəni aşağıdakı kimi də mümkündür:

```
function gosterMesaj (from, text = anotherFunction ()) {
  // anotherFunction () yalnız text olmaz işləyəcəkdir
  // dəyər sonuncu yazılmış dəyərə bərabər olacaqdır
}
```

Default parametrlərdən istifadə

Javascriptdə funksiya hər dəfə çağırıldıqda əlaqəli olan parametr olmadıqda default parametr işə düşür. Yuxarıdakı nümunədə hər parametrə gosterMesaj () -də text parametrinin dəyəri olmadıqda başqa bir funksiya işə düşür. Default dəyər 1 ədəd olur. Python kimi dillərin əksinə olaraq.

Default parametrlər köhnə formada istifadəsi

Əvvəlki Javascript versiyalarında default dəyərlər dəstəklənmirdi. Bu səbəblə onları əvvəlki formada istifadə edərək dəstəklənməsinə nail ola bilərsiniz.

Məsələn undefined olmağının yoxlanması:

```
function gosterMesaj (from, text) {
  if (text === undefined) {
    text = 'yazı yoxdur';
  }
  alert (from+ ": "+ text);
}
... Vəya || operatoru:
function gosterMesaj (form,text) {
  // if əgər səhv dəyər alarsa, "undefined" dəyərini alacaqdır.
  text = text || 'yazı yoxdur';
  ...
}
```

Geriyə dəyər ötürmə prosesi

Funksiya nəticə olaraq geriyə bir dəyər ötürməlidir.

Ən sadə nümunə iki dəyərin toplanmasına adi funksiya olacaqdır:

```
function cem (a, b) {
  return a + b;
}
let result = cem (1, 2);
alert (netice); // 3
```

Geriyə dəyər ötürmə (Return) funksiyanın hər yerində ola bilər. Funksiya bitdikdə proses dayanır və dəyər geri ötürülür. (Yuxarıdakı nümunə kimi olacaqdır.)

Tək funksiyada bir neçə geri dəyər ötürülməsi ola bilər. Nümunə:

```
funksiya checkAge (yaş) {
  if (yaş> 18) {
    return true;
  } else {
    return confirm ('Ailenizden izniniz var mı?');
  }
}

let age = prompt ('Neçə yaşınız var?', 18);

if (checkAge (yaş)) {
  alert ('İcazə verildi');
} else {
  alert ('İcazə verilmədi');
}
```

Return olmadanda dəyər almaq mümkündür. Bu funksiyanın dərhal çıxmasına səbəb olacaqdır.

Nümunə:

```
function showMovie (yaş) {
  if (! checkAge (yaş)) {
    return;
  }

  alert ("Film başladı"); // (*)
  // ...
}
```

Yuxarıdakı kodda checkAge (yaş) səhv olarsa, ShowMovie-da alert-in dayandırıldığına baxın.

Boş dəyər geri ötürlərsə, undefined dəyərini alacaqdır:

```
function doNothing () {/ * empty * /}

alert (doNothing () === undefined); // düzgün

Boş göndərilmiş dəyər undefined -a bərabərdir:

function doNothing () {

  return;

}

alert (doNothing () === undefined); // düzgün
```

Return ilə dəyər arasında heç vaxt boş sətir qoymayın.

Uzun bir ifadə üçün aşağıdakı kimi yeni bir sətirə keçid etmək xoş görünə bilər:

return

*(bunun+kimi + uzun + ifadə + vəya + hər nə+olursa+olsun * f (a) + f (b))*

Bu işə yaramıyacaqdır. Çünki Javascriptdə return-dən sonra nöqtəli vergül qoyulmalıdır. Bu da təbbi ki, işləməyəcəkdir.

return;

*(bunun+kimi + uzun + ifadə + vəya + hər nə+olursa+olsun * f (a) + f (b))*

Beləcə gördüyünüz kimi boş dəyər alaraq geri döndü. Bunun düzəltmək üçün return ilə dəyəri eyni sətirə qoymaq lazımdır.

Funksiya adlandırılması

Funksiyalar hərəkətdirlər. Yəni onların ümumilikdə fel adlanır. Mümkün olduğu qədər adlandırılma düzgün olmalıdır və nə üçün yazıldığını izah etməlidir. Beləliklə kodu oxuyan developer funksiyanın nə iş gördüyünü anlayacaqdır.

Funksiya adlandırılmasında komandaların istifadə etdiklər prefikslər vardır. Bunlardan istifadə etmək komanda daxilində işin asanlaşdırılmasına səbəb olur.

Məsələn “show” ilə başlıyan funksiya ümumilikdə aşağıdakı fellərin prefiksi ola bilər:

• "get..." - bir dəyər ver,

• "calc..." - bir şey hesabla,

• "creat ..." - bir şeyler yarat,

• "creat " - - bir şeyi yoxla və boolean dəyəri geri qaytar, vb.

Adlandırmaya əlavə nümunələr:

showMessage (..) // bir mesaj göstər

getAge (..) // yaşı ver

calcSum (..) // cəm hesablayır və nəticəni geri qaytarır

createForm (..) // form yaradır

checkPermission (..) // icazəni yoxlayır

Nümunələrin düzgün olması funksiyanın adına baxdıqda nə işə yaradığını anlamaya imkan verir.

Bir funksiya – bir hərəkət

Hərəkətlər tam olaraq adındakı funksiyanı yerinə yetirməlidir. Daha çoxunu istifadə etməməlidir.

İki hərəkəti yaradan funksiya olsa belə, ümumilikdə ortak funksiyalara görə adlandırılmalıdır.

Bu qaydaların pozulmasına dair nümunələr :

• getAge – yaşla bağlı bildiriş göstərsə(sadəcə almalıdır), düzgün olmayacaqdır.

• createForm – form dəyişərsə, özünə əlavə form əlavə edərsə düzgün deyildir.

• checkPermission – qəbul mesajı görünərsə, sadəcə rədd mesajı olduqda görünməlidir.

Bu nümunələr ortaq mənaları olanlar idi. Siz və komandanız digər mənalarda üzərində çalışıb kombinasiya qurmaqda sərbətsiniz. Ama ümumilikdə çox da fərqlənməyəcəkdir. Hər halda prepikslərin təyini və nə məna verdiyi yalnız sizin komandaya aid olacaqdır. Bu adlandırılma bütün komanda üzvlərinə izah edilməlidir.

Ultrashot funksiya adları

Çox istifadə edilən funksiyalar ultrashot adlarına sahibdirlər.

Məsələn jQuery frameworkündə $ ilə funksiyalar adlandırılır. Lodash kitabxanasında _ ilə adlandırılır. Bunlar istisnalardır. Ümumi olaraq adlar qısa və mənalı olmalıdır.

Funskiyalar == Şərhlər

Funksiyalar qısa olmalı və işə yarıyan olmalıdır. Əgər funskiyanız böyükdürsə , onu kiçik funksiyalara bölmək sizə kömək ola bilər. Bəzən bu qaydaları izləmək o qədər də asan olmaya bilər. Amma inanın ki, sizin üçün sadələşməyə kömək edəcəkdir.

Funksiyanı test etmək və xəta tapmaq asan deyildir. Nümunə olaraq aşağıdakı Primes(n) funksiyasını iki hissəyə ayrılmış funksiya ilə müqayisə edək.

İlk dəyişən etiketdən istifadə edir:

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

İkinci dəyişəni test etmək üçün əlavə funksiya olan isPrime (n) istifadə edilir:

```
function showPrimes (n)
  for(i = 2; i <n; i ++) {
    if (! isPrime (i)) continue;


    alert (i) // prime
  }
}
function isPrime (n) {
  for (i = 2; i <n; i ++) {
    if (n% i == 0) return false;
  }
  return true;
}
```

İkinci dəyişənin başa düşülməsi sadədir. Kod hissəsi yerinə funksiyanın nə işə yaradığını görürük (isPrime). Bəzən insanlar öz özlərini izah edən kodlara istinat edirlər.

Beləliklə onları yenidən istifadə etdikdə heç bir çətinliyimiz olmayacaqdır.

Nəticə

Funksiyanın strukturu belə görünəcədir:

```
function adı (parametr, saysız,vergüllə ayrılmış,)
{
  / * kod * /
}
```

• Parametrler lokal dəyişənlərinə ötürülərkən işə düşən dəyərlərdir.

• Funksiya global dəyişənlərdən istifadə edə bilər. Ancaq sadəcə içəridən çölə doğru işləyəcəkdir. Funksiyanın kənarında lokal dəyişənlər görünməyəcəkdir.

• Funksiyalar dəyər ötürə bilirlər. Olmazsa, undefined alır.

Kodu təmiz və başa düşülən hala salmaq üçün funksiyada qlobal dəyişənləri yerinə lokal dəyişənlərdən istifadə etməyiniz məsləhətdir.

Parametrləri olan, bunlarla işləyən və nəticəni geri qaytaran funksiya hər zaman sadə deyildir. Ancaq qlobal dəyişənlər əlavə təsir göstərir.

Funksiyaların adlandırılması:

• Ad funksiyanın nə işə yaradığını izah etməlidir. Kodda funksiya çağırıldığını gördükdə nəyin baş verdiyini anlamağımıza kömək edir.

Funksiya hərəkətdir. Bu səbəblə funksiya adları ümumilikdə fellərdən olur.

Create…, show…, get…, check… və bunlar kimi bir çox tanınmış funksiya adları vardır. Bu funksiyanın nə işə yaradığını göstərmək üçün istifadə edilir.

Funksiyalar görülən işin əsas hissəsidir. İndi sadə formada başa düşdük. Əslində onları artıq yaradıb və istifadə edə bilərik. Amma bu sadəcə yolun başlanğıcıdır. Funksiyalara daha sonra yenidən qayıdacıq. Dərin xüsusiyyətlərinə baxacıq.

19.Expression funksiyaları və oxlar

Javascriptdə funksiya "sehirli dil strukturu" deyildir. Sadəcə xüsusi dəyərdir. Daha əvvəl istifadə etdiyimiz sintaksis ilə funksiyamızı çağıraq :

```
function salamDe () {
  alert ("Salam");
}
```

Funksiya sintaksisinin fərqli yazı tipi vardır. Nümunə:

```
let salamDe = function () {
  alert ("Salam ");
};
```

Burada funksiya digər hər hansı dəyər kimi yaradılır və dəyişənə verilir. Funksiyanın tipi vacib deyildir. Sadəcə salamDe dəyişəninin olduğu dəyərdir.

Bu kod nümunələrinin ikisidə eynidir.

Bu dəyəri bildiriş kimi çağıraq:

```
function salamDe () {
  alert ("Salam ");
}
alert (salamDe); // funksiyada olan kod
hissəciyi işə düşəcəkdir.
```

Son sətrin işləmədiyini yadda saxlayın, çünki salamDe-dən sonra skop yoxdur.

Javascriptdə funksiyanın dəyərdir. Bu səbəblə onunla tip kimi davrana bilərik.

Təbii ki, salamDe() kimi adlandırılan xüsusi dəyərdir.

Ama bu hələ də bir dəyərdir. Beləliklə digər dəyərlərdə olduğu kimi onunlada işləyə bilərik. Funksiyanı dəyişənlə istifadə edək:

```
function salamDe () {    // (1) yarat
  alert ("Salam");
}
let func = salamDe; // (2) dəyişəndə verilmiş halı (nüsxə)
func (); // Salam // (3) dəyişən işlədir
salamDe(); // Salam // normal işləyir
```

Yuxarıdakı funksiyanı daha ətraflı izah edək:

1. Funksiya adlandırılması ilə yaradılır və salamDe adlı dəyişənə ötürülür.

(2) Sətirdə həmin funksiyanı func dəyişəninə ötürülür.

2. Funksiyada salamDe() həm də func () olaraqda adlandırıla bilər.

Həmdə ilk sətirdə salamDe-ni bildirmək üçün funksiya ifadəsi yaza bilərik :

```
let salamDe = function () {...};



func = salamDe;
```

Bütün hamsı eyni zaman işləyəcəkdir. Prosesin necə getdiyini anlamaq üçün kod yazmağa başlayın.

Hər sətirdən sonra niyə nöqtəli vergül var?

Funksiya ifadəsində sonra nöqtəli vergül olmalıdır.

```
  function salamDe () {
  // ...
}
let salamDe = function () {
  // ...
};
```

Cavab çox sadədir:

• Nöqtəli vergül if {...}, for { }, function { } kimi bloklardan sonra qoyulur.

• Funksiya içərisində yazılan ifadədən sonra nöqtəli vergül qoyulur.

let salamDe = ...;. Burada nöqtəli vergül ifadə ilə əlaqəsi yoxdur.Sadəcə ifadəni sonlandırır.

Callback (Geri qayıtma) funksiyaları

Funksiyalara dəyişənə aid etmə və sintaksisinə anlamaq üçün daha çox nümunəyə baxaq.

Funksiyamızda 3 parametr(sual,bəli,xeyr) yazacıq:

```
Sual
Sual yazısı
Bəli
Əgər cavab düzdürsə, işləyəcək kod hissəsi
Xeyr
Əgər cavab yanlışdırsa, işləyəcək kod hissəsi
```

Funksiyanın soruşduğu və istifadəçinin cavabına əsasən beli() vəya xeyr() ifadələri işləyəcəkdir:

```
function  deyir (sual, bəli, xeyr) {
  if (confirm (sual)) bəli ()
  else xeyr();
}

function gösterBeli ()
  alert ("Qəbul etdiniz.");
}

function gosterXeyr () {
  alert ("Proses dayandı.");
}
// use: function gosterBəli, gosterXeyr soruşulan arqumentlər olaraq bildirildi.
deyir ("Qəbul edirsinizmi? ", gosterBəli, gosterXeyr);
```

Bu ifadələrin necə qısa şəkildə yaza bilməzdən əvvəl brauzerdə (bəzi hallarda server tərəfində) bu funksiyaların necə istifadə edildiyini bilməlisiniz. Real həyatda funksiyaların istifadəçi ilə əlaqə qurması sadə təsdiq etmədən daha qarışıq metodların istifadə edilməsi üçündür. Brauzerdə belə bir funksiya ümumilikdə gözəl görünüşlü pəncərə yaradır. Bunun haqqında digər bölmələrdə ətraflı danışacıq.

Deyir funksiyasının arqumentlərinə callback funksiyaları vəya sadəcə callback deyilir.

Callbackların belə anlıya bilərik ki, isitfadəçi suala əgər "bəli" vəya "xeyr" cavabını verərsə, geri qayıdaraq nəticəni ekranda göstərməsidir.

Eyni funksiyanı daha qısa yazmaq üçün Function Expression-dan istifadə edə bilərsiniz:

```
function deyir (sual, beli, xeyr) {
  if (confirm (sual)) beli ()
  else xeyr ();
}
deyir(
  "Qəbul edirsinizmi?",
  function () {alert ("Qəbul etdiniz."); },
  function () {alert ("Prosesi dayandırdınız."); }
);
```

Funksiyada (deyir(...)) sualın sağında yazılmış parametrlər callbacklardır. Onların adı yoxdur və bilinməyən kimi adlandırılır. Bu dəyərlər deyirdən başqa heç nə təsir göstərə bilməz. Tamda istədiyimiz buradadır.

Belə formada kodlaşdırma çox normaldır.Çünki Javascriptin əsas prinsipi elə budur.

Bir funksiya bir prosesi aparan vasitədir.

Burada yazılar vəya rəqəmlər kimi normal dəyərlər məlumatı təmsil edir.

Bir funksiya bir prosesin baş verəcəyini bildirir.

Dəyişənləri istədiyimiz zaman dəyişə bilərik və yaxud boşda göndərə bilərik.

Function Expression və Function Declaration

Function Expression və Function Declaration arasındakı fərqlərə baxaq.

İlk əvvəl sintaksis: Kod yazılışında fərqlər vardır.

Function Declaration: function ilə digər kodlardan ayrılmış kod blokudur.

```
// Function Declaration

funtion cem (a, b) {

return a + b;

}
```

Function Expression: bir ifadənin içərisində vəya başqa ifadələr içərisində yaradılmış funksiyadır. Burada funksiya dəyişənə bərabər olur:

```
// Function Expression
let cem = funksiya (a, b) {
return a + b;
};
```

Ən əsas məqam isə Javascript engine tərəfindən bir funksiyanın yaradılmasıdır.

Function Expression ifadəsinə dəyişənə bərabər edilib və bundan sonra istənilən yerdə istifadə edilə bilər.

Function Declaration fərqlidir.

Bu funksiya scriptdə (vəya kod blokundadırsa, onun tərkibində) istifadə edilə bilər

Başqa sözlə, Javascript faylı vəya kod blokunda yazmağa başladıqda əvvəlcə Funksiyalar (proseslər) yazılır. Bu “başlanğıc” olaraq başa düşə bilərik.

Bütün bunları başa düşdüksən sonra anlayırıq ki , Function Declaration digər funksiyalara görə sürətli çağırılır.

Məsələn nümunə:

```
salamDe ( "Nail"); // Salam Nail


function salamDe (ad) {
  alert (`Salam, $ {ad}`);
}
```

Javascript scripti işə başladığı andan Salam dəyəri yaranmışdır.

Function Expression olsaydı bu proses baş verməzdi:

```
salamDe ( "Nail"); // xəta!


salamDe = function (ad) {// (*) artıq proses xəta aldı!
  alert (`Salam, $ {ad}`);
};
```

Function Expression funksiyalarımda proses onlara çatdıqda yaradılır. Yəni proses bir düz xətt üzərində baş verir. Bu isə çox gecikməyə səbəb olur.

Function decleration kod blokunun içərisində (skope daxilində) hər yerdə görülür.Ancaq blokdan kənarda görünməyəcəkdir.

Bəzən yalnız blokda lokal olaraq funksiya yararlı ola bilər. Ancaq bu xüsusiyyət problemlərə səbəb ola bilər.

Nümunə olaraq proses zamanı aldığımız yaş dəyişəninə bağlı olaraq, xosGelmisiniz() funksiyasını bildirmək lazımdır. Daha sonra isə istifadəyə başlıyaq.

Aşağıdakı kod işləmiyəcəkdir :

```
Let yaş = = prompt ("Yaşınız neçədir?", 18);


// Şərt elavə edin
if (yaş <18) {


 function xosGeldiniz() {
    alert ( "Salam!");
  }


} else {


   function xosGeldiniz() {
   alert ( "Salamlar!");
  }


}
// ... daha sonra istifadə et
xosGeldiniz (); // Xəta: xosGeldiniz undefined
```

Bunun səbəbi Function Declaration yalnız içərisində olduğu kod bloku görməsidir.

Başqa nümunə:

```
let yas= 16; // nümunə olaraq 16


if (yas <18) {
  xosGeldiniz (); // \   (işləyəcək)
 function xosGeldiniz () {
    alert ( "Salam!"); // | Function Declaration kod blokunda görür
  } // | bütün parametrlər blokun daxilində idi
                          // |
  xosGeldiniz(); // /   (işləyəcəkdir)
} else {


  function xosGeldiniz () {// yaş = 16 üçün, " xosGeldiniz" funksiyası işləməz
    alert( "Salamlar");
  }
}
// Skoplar bitti
// Bu səbəblə artıq funksiyamız işləmiyəcəkdir
xosGeldiniz (); // Xəta: xosGeldiniz undefined
```

xosGeldiniz funksiyasının kənardan if-i görmək üçün nə edə bilərik ? Sonun əsas məqama çatdıq. Funksiyadan skop-dan kənarda təsir göstərməyə baxacıq. Salam dəyişəninə dəyər verə biləcik.

Aşağıdakı nümunədə kimi işlədə bilərik:

```
let yash= = prompt ("Yaşınız?", 18);


let salam;


if  (yash <18) {


  salam = function () {
    alert ( "Salam!");
  };


} else {


  salam = function () {
   alert ( "Sağolun!");
  };


}


xosGeldiniz(); // bitdi
```

```
let yash= = prompt ("Yaşınız?", 18);


let salam = (yash <18)?
  function () {alert ("Salam!"); }:
  function () {alert ("Sağolun!"); };


xosGeldiniz(); // tamam şimdi
```

Nə zaman Declaration funksiyası əvəzinə Expression funksiyası istifadə edilməlidir?

İlkin qayda olaraq funksiyamızı yazdıqda diqqətə alınması ən böyük qaydalardan biri funksiyanın işləmə proses ardıcıllığıdır.

Kodumuzun ardıcıllığında heç bir qayda yoxdur. Çünki bu cür funksiyalarımızı bildirmədən belə çağıra bilərik.

Kodda f (...) {...} funksiyasına baxmaq, let f= function (...) {...} funksiyasına nisbətən sadə yazılmışdır.

Decralation funksiyasının işlənmə yerinə bəzən Expression funksiyasıda işlənə bilər.

Arrow (Ox) Funksiyaları

Funksiyaları yaratmaq üçün sadə ox funksiyasıdan istifadə edə bilərik. Bu funksiyaların ox funksiyalarının adlandırılması aşağıdakı nümunədə anlayacaqsınız.

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

... Bu, arg1..argN arqumentlərinə sahib funksiya yaradıldı və sağ tərəfdəki funksiyaya dəyərlər verikdikdən sonra nəticəni verəcəkdir.

Başqa sözlə yuxarıdakı ilə eynidir:

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

... Tək bir fərqi daha qısa olmasıdır.

Nümunəyə baxaq:

```
cem = (a, b) => a + b;


/ * Ox funksiyası daha yazılış şəklidir:
cem = function (a, b) {
 return a + b;
};
* /


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

Tək arqumentimiz varsa, skoplar atıla bilər:

```
// eyni ilə yazırıq
// let double = function (n) {return n * 2}
let = = n => n * 2;
```

Əgər arqument yoxdursa, skoplar boş olmalıdır.(Ancaq onlar olmalıdır):

```
let salam = () => alert ("Salam!");
let salam();
```

Ox funksiyaları Expression funksiyaları ilə eyni şəkildə istifadə edilir.

Nümunə olaraq salam() ilə yazılmış funksiyaya baxaq:

```
let = = prompt ("Yaşınız?", 18);
let salam = (yaş <18)?
  () => uyarı ('Salam'):
  () => alert ("Salam!");
xosGeldiniz(); // bitti
```

Ox funksiyalarına fərqli gələ bilər. İlk baxışda qarışıq gələ bilər. Ancaq gözlər strukturda işlədikcə öyrəcəkdir.

Hər söz üçün fərqli sətirdə yazmaq yorduğundan sadə tək sətirlik funksiya daha rahat olacaqdır.

Çox sətirli ox funksiyaları

Yuxarıdakı nümunələrdə => - in solunda şərt yazıldı və sağ tərəfində ifadə dəyərləri yazıldı.

Bəzən çox ifadəyə eyni zamanda yazmaq lazım olur. Bu cür qarışıq ifadələri yazmaq mümkündür. Ancaq oları dırnaq içərisində yazmalıyıq. Sonra içlərində normal olaraq return-dən istifadə edə bilərik.

Bunun kimi:

```
cem = (a, b) => {// skopların olması çox sətirli funksiyalar üçün açılır.
let result = a + b;
  return result; // skopdan istifadə edirsinizsə, nəticəni almaq üçün returndən istifadə edin.
};
```

Burada ox funksiyaları haqqında qısa məlumat verdik. Sonrakı bölmədə ox funksiyaları haqqında ətraflı məlumat verəcik.

Nəticə

Funksiyalar dəyərdir. Kodun istənilən yerində istifadə edilə bilər, kopyalanabilər və bildirə bilərik.

Funksiyalar kodun tərkibində fərqli ifadə olaraq bildirilsə, buna “Declaration funksiyası” deyilir.

Funksiya ifadənin tərkibindən ayrılmayıb bir bildirilsə, buna “Expression funksiyası” deyilir.

Declaration funksiyaları kod bloku işə salınmadan işə düşür. Blokun hər yerində görünə bilir.

Expression funksiyaları yalnız prosesin onlara çatdığı anda işə düşürlər.

Çox zaman Declaration funksiyasından əvvəlcədən bildirilmə lazım olduqda istifadə etmək daha çox rahatlığı və ümumilikdə oxunarlılığı artırır.

Bu səbəblə Expression funksiyası yalnız Declaration funksiyası uyğun olmadığı hallarda istifadə etməliyik. Bu bölmədə bir neçə nümunə gördük və gələcəkdə daha çox nümunlərə baxacıq.

Ox funksiyaları tək sətirlik kodlar üçün istifadəsi verimlidir. İki növdə yazılır:

1. Skoplar olmadan (...args) => espression – sağ tərəfdə ifadədir: funksiya onun dəyərləndirir və nəticəni göstərir.
2. Skoplarla: (...args) => (...args)=> {body}- skoplar funksiyanın tərkibində çox ifadənin yazılmasına imkan verir. Ancaq nəticəni almaq üçün return-dən istifadə edilməlidir.

20.Xülasə

Bu bölmədə indiyənə qədər öyrədilmiş dərslərin əsas hissələrinə qısa olaraq baxacıq.

Kod strukturu

İfadələr nöqtəli vergüllə sonlandırılır:

```
alert ('Salam');

alert ('Dünya');
```

Ümumlikdə sətrin sonu bitməni bildirir. Bu səbəblə də yenə işləyəcəkdir:

```
alert ('Salam')

alert  ('Dünya')
```

“Avtomatik vergül əlavə etmə” belə problemlərin həllidir. Məsələn:

```
alert ("Bu mesajdan sonra xəta olacaqdır")

[1, 2]. forEach (alert)
```

Developerlərin çoxu hər ifadədən sonra nöqtəli vergül qoymaqla oxunarlılığı artırıldığını söyləyir.

Nöqtəli vergül {...} bunun kimi kod bloklarında sonra lazım deyildir:

```
function f ()

  // funksiyadan sonra nöqtəli vergülə ehtiyac yoxdur.

}

for (;;) {

  // alqoritmadan sonra nöqtəli vergül lazım deyildir.

}
```

... Ancaq istənilən bir yerə bilməyərəkdən "əlavə" nöqtəli vergül qoyarıqsa, diqqətə alınmayacaqdır.

Daha ətraflı: Kod strukturundadır.

Use Strict Mode

Müasir Javascriptin bütün xüsusiyyətlərini tam olaraq işə salmaq üçün script fallarının əvvəlində “use strict” -dən istifadə etməliyik.

```
'use strict';

...
```

Təsir edəcəyi kodun vəya funksiyanın başında yazılmalıdır.

“use strict” olmadan hər şey işləyəcəkdir. Ancaq bəzi xüsusiyyətlər əvvəlki xüsusiyyətlərini əvvəlki şəkildə istifadə edəcəkdir. Ümumi olaraq kodlarda use strict moddan istifadə edirik.

Dilin bəzi müasir xüsusiyyətlərinin (sonrakı bölmələrdəki kimi classlardakı kimi) use strictlə istifadə etmək mümkündür.

Daha ətraflı: use strict mode bölməsinə baxa bilərsiniz.

Dəyişənlər

Aşağıdakılarla bildirilərək istifadə edilə bilər:

• let

• const (sabit, dəyişilməz)

• var (köhnə metod, yazıldıqdan sonra görünəcəkdir)

Dəyişənlərin adlandırılaması :

• Hərflər və rəqəmlər. Ancaq ilk simvol rəqəm olmamalıdır.

• $ və _ simvollarından istifadə edə bilərsiniz.

• Latın əlifbasında başqa hərflərə icazə verilir ancaq çoxda istifadə edilmir.

Dəyişənlər dinamikdir. İstənilən dəyəri saxlaya bilərlər :

```
let x = 5;

x = "Nail";
```

7 məlumat (Data) tipi vardır:

• Rəqəm – kəsr və tam ədədlər,

• Yazılar – String,

• Məntiqi dəyərlər- Boolean (true/false),

• null - "boş" vəya "olmayan" mənasına gələn və yalnız null dəyərinə sahib tipdir.

• undefined – Bilinməyən dəyər olan “ bilinməyən” mənasını verən tipdir.

• Object və symbol – Qarışıq dataların saxlanması və oxşarı olmayan dəyərləri saxlayan data tipidir.

Typeof operatoru iki istisnadan başqa, dəyərin hansı tipdə olduğunu bildirir:

```
typeof null == "object" // dildə xətadır

typeof function () {} == "function" // funksiyalar xüsusi olaraq
```

Daha ətraflı: Dəyişənlər və Data tipləri bölməsindən baxa bilərsiniz.

Alert, Prompt, Confirm

Brauzerdə işləyən sadə Uİ elementləri ilə hazırlanmış sadə funksiyalardır:

prompt(sual, [default])

Sual soruşduqda – istifadəçi tərəfindən yazılmış dəyər vəya cancel -dəyəri geri döndürülür.

confirm (sual)

Sualın nəticəsinin Bəli vəya Xeyr olaraq alınması üçün istifadə edilir.

alert (mesaj)

Bildiriş mesajı çıxacaqdır.

Bütün bu funksiyalar modaldır. Kodun işləməsini dayandırır və istifadəçinin cavab verməsini gözləyir.

Nümunə:

```
let istifadeciAdi = prompt ("Adınız?", "Nail");

let cayİsteyen = confirm ("Çay istəyirsiniz?");



alert ("İstifadəçi:" + istifadeciAdi); // Nail

alert ("Çay istənildi:" + cayİsteyen); // true
```

Daha ətraflı: Alert, Prompt, Confirm bölməsindən baxa bilərsiniz.

Operatorlar

JavaScript aşağıdakı operatorları dəstəkləyir:

Aritmetik

Normal: * + - /, Qalıq % və ** qüvvət operatorları.

+ operatoru iki string birləşdirir. Dəyərlərdən biri stringdirsə, digərinidə stringə çevirəcəkdir:

```
alert ('1' + 2); // '12', string

alert (1 + '2'); // '12', string
```

Bərabərlik

Dəyər vermə : a=b və a*=2 kimi qısaltma vardır.

Bitwise

Bitwise operatorları ən kiçik 32 bit səviyyəsində işləyir. Ehtiyac olarsa dokumentariyasına baxa bilərsiniz.

Şərt

Üç parametrli tək operator : cond? neticeA:neticeB.Nəticə düzgündürsə neticeA əks halda isə neticeB nəticəsini verəcəkdir.

Məntiq operatorları

Məntiqi VƏ && və VƏYA || qısa məntiqi əlaqə qura bilərsiniz və nəticəni istəyinizə uyğun dəyərləndirə bilərsiniz. Məntiq NOT! Operatoru boolean dəyəri alır və tərs dəyəri verir.

Müqayisə operatorları

Bərabərlik yoxlaması etdikdə (==) dəyişən tiplərinin nəticəsi fərqli tipə çevriləcəkdir. (Bir-biri ilə bərabər olan və null vəya undefined istisna olmaqla),

Bu səbəblə bunlar bərabərdir:

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

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

Digər müqayisələrdə ədədə çevirir.

Qatı bərabərlik operatoru (==) : Fərqli tiplərdə olan dəyərlər bu operator üçün fərqli dəyər almasını bildirir.

Null və undefined birbirinə bərabərdir və başqa heçnəyə bərabər olmurlar.

Digər Operatorlar

Vergül operatorları kimi bir neçə operator vardır.

Daha ətraflı: Operatorlar , Müqayisə , Məntiqi Operatorlara baxın.

Alqoritma

Aşağıda 3 fərqli alqoritma yazdıq:

```
// 1
while (şərt) {
...
}
```

```
// 2
do
...
} while (şərt);
```

```
// 3
for (let i = 0; i <10; i ++) {
...
}
```

for (let ...) alqoritmasında verilmiş dəyişən yalnız alqoritma daxilində görünür. Ancaq olan dəyişən buraxa bilərik və yenidən istifadə edə bilərik.

Alqoritmada break/continue ilə istifadə edərək qırmaq və davam etdirmək mümkündür.

Daha ətraflı : Alqoritma: while və for bölməsinə baxın.

Daha sonra isə objectlərlə əlaqələndirmək üçün daha çox alqoritma növü ilə işləyəcəyik.

“Switch” strukturu

“Switch” strukturu birdən çox if olarsa, istifadə edilir.

Müqayisə üçün === istifadə edilir.

Nümunə:

```
let age = prompt ('Yaşınız?', 18);


switch (age) {

  case 18:

    alert ("İşləmiyəcəkdir"); // istənilən dəyər tipi rəqəm deyil string istəyir.


  case: "18"

    alert ("Bu işləyəcəkdir!");

    break;


  default:

    alert ("Yuxarıdakılardan heç biri uyğun olmadığı halda bu işə düşəcək dəyər");

}
```

Daha ətraflı : “Switch” strukturundan baxa bilərsiniz.

Funksiyalar

Javascriptdə funksiya yaratmağın 3 yolu var:

1. Declaration funksiyası: Əsas kodun tərkibində yazılan funksiya

```
function cem (a, b) {
 let cem = a + b;
return cem;
}
```

2. Expression funksiyası:

```
let cem = function (a, b) {
let cem= a + b;
 return cem;
};
```

3. Ox funksiyaları:

```
// İfadə oxdan sağ tərəfdə yazılır

cem = (a, b) => a + b;

 // vəya {...} ilə çox sətirlik kod yaza bilərsiniz. Ancaq
bu zaman returndən istifadə edilməlidir:

let cem = (a, b) => {

 // ...

return a + b;

}

// Asılı olmayan dəyişənlər olmadan

let sayHi = () => alert ("Salam");

// tək arqumentlə

let = = n => n * 2;
```

Funksiyalar local dəyişənlərə sahib ola bilərlər. Öz tərkiblərində yaradılır. Bu dəyişənlər sadəcə funksiyanın daxilində görülür.

Funksiyalar default (ilkin) dəyərə sahib ola bilərlər. Function cem (a=1, b=2) {...}.

Funksiyalar hər zaman dəyər qaytarır. Return ifadəsi yoxdursa, nəticə undefineddir.

Daha ətraflı : Funksiyalar, Funksiya ifadələri və ox funksiyaları bölməsindən oxuya bilərsiniz.