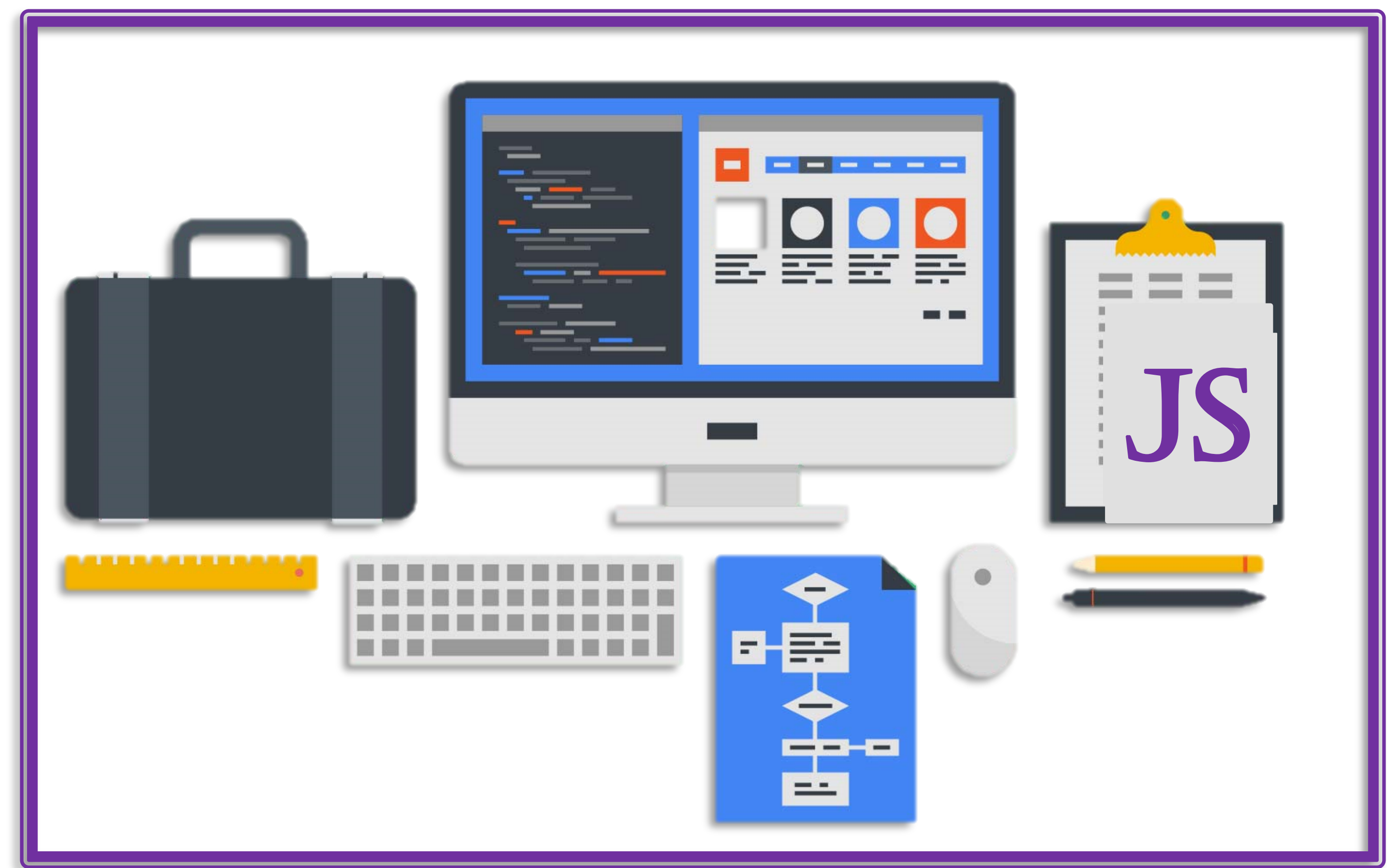
Մանուկյան Դիանա



JavaScript

Ծրագրավորման լեզվի տարրերը

ՄԱՍ 1

Երևան 2019

Բովանդակություն

Ներածություն .... էջ 4

Գլուխ 1. JavaScript-ով աշխատանքի սկզբունքը

1.1. Ինչ է JavaScript-ը, JavaScript-ի արդիականությունը .... էջ 5
1.2. Առաջին ծրագիրը `JavaScript- ում .... էջ 6
1.3. Javascript կոդի կատարելագործում .... էջ 8
1.4. Արտաքին javascript ֆայլի տեղադրումը .... էջ 13
1.5. Վահանակով բրաուզեր, console.log և document.write .... էջ 16

Գլուխ 2. Javascript-ի հիմունքներ

2.1 . Փոփոխականներ և հաստատուններ ……………….......................... էջ 20
2.2 . Տվյալների տիպերը .... էջ 22
2.3 . Գործողություններ փոփոխականներով .... էջ 25
2.4 . Տվյալների փոխակերպումները ……….. էջ 32
2.5 . Զանգվածներ .... էջ 36
2.6 . Պայմանական կառույցներ .... էջ 39
2.7 . Ցիկլեր .... էջ 44

Գլուխ 3. Ֆունկցիոնալ ծրագրավորում

3.1 . Ֆունկցիաներ ……………... էջ 48
3.2 . Ֆունկցիայի պարամետրերը .... էջ 50
3.3 . Տարբեր տեսանելիությամբ փոփոխականներ .... էջ 55
3.4 . Կապակցում և IIFE ֆունկցիան………... էջ 60
3.5 . Մոդուլի կառուցվածքը .... էջ 63
3.6 . Ռեկուրսիվ ֆունկցիաներ .... էջ 65
3.7 . Վերափոխվող ֆունկցիաներ .... էջ 66

3.8 Hoisting .... էջ 67
3.9 Փոխանցվող պարամետրերը ըստ արժեքի և հղման .... էջ 69
3.10 Ուղորդված ֆունկցիաներ .... էջ 71

Գլուխ 4. Օբյեկտ-կողմնորոշված ծրագրավորում

3.1 . Օբեկտներ.. էջ 73
3.2 . Օբյեկտներում ներկառուցված օբյեկտներ և զանգվածներ .... էջ 78
3.3 . Հասանելիության ստուգում և մեթոդների ու հատկությունների որոնումը .... էջ 81
3.4 . Օբյեկտները ֆունկցիաներում .... էջ 83
3.5 . Մոդուլի կառուցվածքը .... էջ 62
3.6 . Ռեկուրսիվ ֆունկցիաներ .... էջ 64
3.7 . Վերափոխվող ֆունկցիաներ .... էջ 65

Նախաբան

Ծրագրավորումը, համակարգչային ծրագրի ստեղծման գործընթաց է: Ըստ Նիկլաուս Վիրտի հայտնի արտահայտության՝

Ծրագրեր = ալգորիթմներ + տվյալների կառուցվածքներ

Այլ խոսքով, ծրագրավորման անմիջական խնդիրներն են ալգորիթմների և տվյալների կառուցվածքների ստեղծումը և կիրառումը: Ավելի լայն իմաստով ծրագրավորումը հասկացվում է որպես ամբողջ գործունեություն, որը կապված է աշխատանքային միջավայրում ծրագրերի ստեղծման ու պահպանման՝ ծրագրային ապահովման հետ: Այստեղ մտնում են խնդրի վերլուծությունն ու ներկայացումը, ծրագրի նախագծումը, ալգորիթմների կառուցումը, տվյալների կառուցվածքային մշակումը, ծրագրի տեքստի գրառումը, ծրագրի կարգավորումն ու տեստավորումը, փաստաթղթավորումը, ուրվագծումը, վերջնական մշակումը: Ծրագրավորումը հիմնվում է ծրագրավորման լեզուների օգտագործման վրա, որոնցով գրառվում են հրահանգները համակարգչի համար:

Ծրագրավորման միջավայրում տեքստային խմբագրիչը կարող է ունենալ հատուկ ֆունկցիաներ, այնպիսին, ինչպիսիք են անունների ինդեքսավորում, փաստաթղթերի պատկերումը, օգտագործողի ինտերֆեյսի վիզուալ ստեղծումը և այլն: Տեքստային խմբագրիչի օգնությամբ ծրագրավորողը դուրս է բերում ստեղծվող ծրագրի տեքստի հավաքակազմն ու խմբագրումը, որն անվանում են ելակետային կոդ: Ծրագրավորման լեզուն որոշում է ելակետային կոդի շարահյուսությունն ու նախասկզբնական իմաստաբանությունը:

Ծրագրավորման լեզուներից են՝ PHP-ն, C++-ը, JavaScript-ը և այլն:

Գլուխ 1. JavaScript-ով աշխատանքի սկզբունքը

Ինչ է JavaScript-ը, JavaScript-ի արդիականությունը

Այսօրվա աշխարհը դժվար է պատկերացնել առանց JavaScript լեզվի: JavaScript-ն այն է, ինչ կայացնում է վեբ էջերը, որը մենք ամեն օր նայում ենք մեր վեբ բրաուզերում: JavaScript-ը աշխատում է բոլոր հանրահայտ բրաուզերներում, որոնց թվում են **Ինտերնետ Էքսպլորերը**, **Mozilla Firefox**-ը, **Google Chrome**-ը, **Օպերան** և **Սաֆարին**:

JavaScript-ը ստեղծվել է 1995 թվականին և առաջին անգամ կիրառվել է Netscape Communications-ի կողմից Netscape Navigator 2 բրաուզերի վրա: Սկզբում լեզուն կոչվել է **LiveScript**, բայց որոշ ժամանակ անց այն վերանվանվեց **Java LiveScript** այնուհետև՝ **JavaScript**: Սակայն, այս անվանումը երբեմն հանգեցնում է ինչ-որ շփոթության, քանի որ որոշ սկսնակ ծրագրավորողներ կարծում եմ, որ Java և JavaScript լեզուները գրեթե նույնն լեզուն են: Սակայն դրանք բոլորովին երկու տարբեր լեզուներ են, և միակ ընդհանուր բանը որ նրանց միջև կա՝ դա նրանց անվանումն է:

Երբ ստեղծվում էր JavaScript-ը՝ նպատակը միայն մեկն էր, ավելացնել մի փոքր վարքագիծը վեբ էջերին: Սակայն երբ սկսեց վեբ միջավայրի զարգացումը, և առաջ եկան **HTML5** և **Node.js** տեխնոլոգիաները, ավելի մեծ հորիզոններ բացվեցին JavaScript-ի համար: Այժմ JavaScript-ը շարունակում է օգտագործվել կայքերի ստեղծման համար, բայց այժմ այն ավելի շատ հնարավորություններ է ընձեռում:

Այն օգտագործվում է նաև որպես սերվերային լեզու, այսինքն, եթե նախկինում Javascript օգտագործելիս մենք ստիպված էինք օգտագործել այնպիսի տեխնոլոգիաներ, ինչպիսիք են **PHP**-ն, **ASP.NET**-ը, **Ruby**-ն, **Java**-ն ապա այժմ մենք կարող ենք կարգավորել սերվերի բոլոր խնդրանքները շնորհիվ **Node.js**-ի:

Վերջին տարիներին JavaScript-ը ապրում է բում բջջային զարգացում: Այսինքն՝ բջջային հեռախոսների, սմարթֆոնների և պլանշետների ծրագրերի ստեղծման համար մենք կարող ենք օգտագործել նաև JavaScript:

Ավելին, շնորհիվ **Windows** օպերացիոն համակարգի նոր ընտանիքի՝ **Windows 8 / 8.1 / 10**-ի թողարկման, մենք կարող եք օգտագործել այս ծրագրավորման լեզուն՝ այդ օպերացիոն համակարգերի համար ծրագրեր մշակելու համար: Այսինքն, JavaScript- ը արդեն անցել է վեբ բրաուզերի սահմանները, որը նախանշվել է դրա ստեղծման մեջ:

Այսպիսով, մենք կարող ենք հանդիպել JavaScript-ի կիրառման գրեթե ամենուր: Այսօր, սա իսկապես ամենատարածված ծրագրավորման լեզուներից է, և լայն ժողովրդականություն է վայելում, որը իր հերթին նպաստում է լեզվի շարունակական աճին:

Սկզբում եղել են մի քանի վեբ բրաուզերներ (**Netscape**, **Internet Explorer**), որոնք տարբեր լեզուների ծրագրեր են իրականացրել: Եվ **ECMA** կազմակերպության ղեկավարության ներքո լեզուների ստանդարտացման համար տարբեր կրճատումներ

իրականացնելու համար, մշակվել է ECMAScript ստանդարտը: Սկզբունքորեն JavaScript-ը և ECMAScript-ը գրեթե նույն լեզուն են:

Ներկայումս ECMA-ն մշակել է մի շարք լեզուների ստանդարտներ, որոնք արտացոլում են դրանց զարգացումը: Վերջին անգամ ընդունված ստանդարտը դա ECMAScript 2015 է, որը այլ կերպ անվանում են նաև ES 6: Բայց ես պետք է ասեմ, որ այս ստանդարտի կիրառումը հայտնի վեբ բրաուզերներում նախնական փուլում է և դրա լիարժեք իրականացումը մի քանի տարի կտևի: Հետևաբար, այս աշխատանքում հիմնականում կանդրադառնանք ES5 ստանդարտին և ֆունկցիոնալությանը, որն արդեն հասանելի է բոլոր հայտնի բրաուզերում:

Զարգացման գործիքները

JavaScript-ի մշակման համար մեզ հարկավոր է տեքստային խմբագիր, որը պետք է գրի կոդը եւ վեբ բրաուզեր, փորձարկելու համար: Որպես տեքստային խմբագիր, խորհուրդ եմ տալիս օգտագործել այնպիսի ծրագիր, ինչպիսիք են Notepad ++-ը, Brackets-ը և այլն: Թեև դրանք կարող է լինել ցանկացած այլ տեքստային խմբագիրներ:

Կան նաև տարբեր զարգացման միջավայրեր, որոնք աջակցում են JavaScript-ին և հեշտացնում են այս լեզվով աշխատանքը, օրինակ, Visual Studio, WebStorm, Netbeans և այլն: Ցանկության դեպքում մենք կարող ենք նաև օգտագործել այդ զարգացման միջավայրերը:

Առաջին ծրագիրը՝ JavaScript-ում

Եկեք ստեղծենք առաջին ծրագիրը javascript-ում: Նախ, եկեք սահմանենք մեր հավելվածը: Օրինակ, եկեք ստեղծենք app թղթապանակը: Այս թղթապանակում ստեղծել index.html անունով ֆայլ: Այսինքն, այս ֆայլը կներկայացնի վեբ էջ:

Բացենք այս ֆայլը տեքստային խմբագրիչով, օրինակ, Notepad++-ով և ֆայլի մեջ սահմանեք հետևյալ կոդը.

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <title>JavaScript</title>
</head>
<body>
    <h2>Первая программа на JavaScript</h2>
    <script>
```

```
    alert('Привет мир!');
    </script>
</body>
</html>
```

Այստեղ մենք սահմանում ենք ստանդարտ **html** տարրեր: **head**-ում սահմանում ենք էլեմենտների տիպը **utf-8** կոդավորմամբ և էջի անվանումը: **body**-ում սահմանում է վեբ էջի մարմինը, բաղկացած վերնագրի տարրից **<h2>** և **<script>** բանալինային բառից:

Javascript կոդը կապվում է **html** էջի հետ **<script>** (բանալինային բառի) թեգի օգնությամբ: Այն պետք է տեղադրված լինի կամ վերնագրում(**<head>** **</head>** թեգերի միջև), կամ մարմնում (**<body>** և **</body>** թեգերի միջև): Հաճախ, սցենարների(ծրագրային կոդերի) կապը տեղի է ունենում փակման **</body>** թեգից առաջ՝ վեբ էջի բեռնումը օպտիմալացնելու համար:

Նախկինում անհրաժեշտ էր **<script>** թեգում նշել սցենարի տեսակը, քանի որ այս պիտակը օգտագործվում էր ոչ միայն javascript հրահանգներին միանալու, այլև այլ նպատակների համար: Այսպիսով, նույնիսկ հիմա մենք կարող ենք գտնել որոշ վեբ էջերում սցենարների (էլեմենտների)տարրերի նման սահմանում.

```
<script type="text/javascript">
```

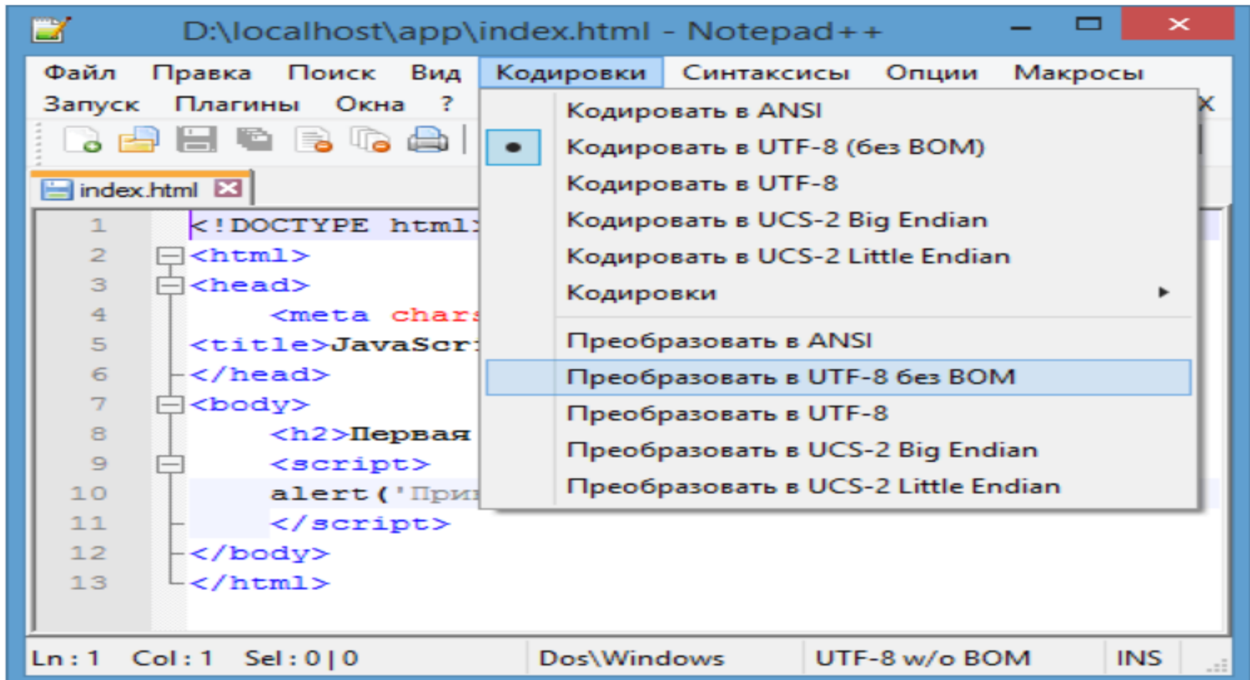
Սակայն հիմա գերադասելի է չնշել սցենարի տեսակը, քանի որ բրաուզերները հասկանում են, որ սցենարը պարունակում է javascript հրահանգներ:

Ինչպես տեսնում եք javascript կոդում մենք օգտագործել ենք մեկ արտահայտություն.

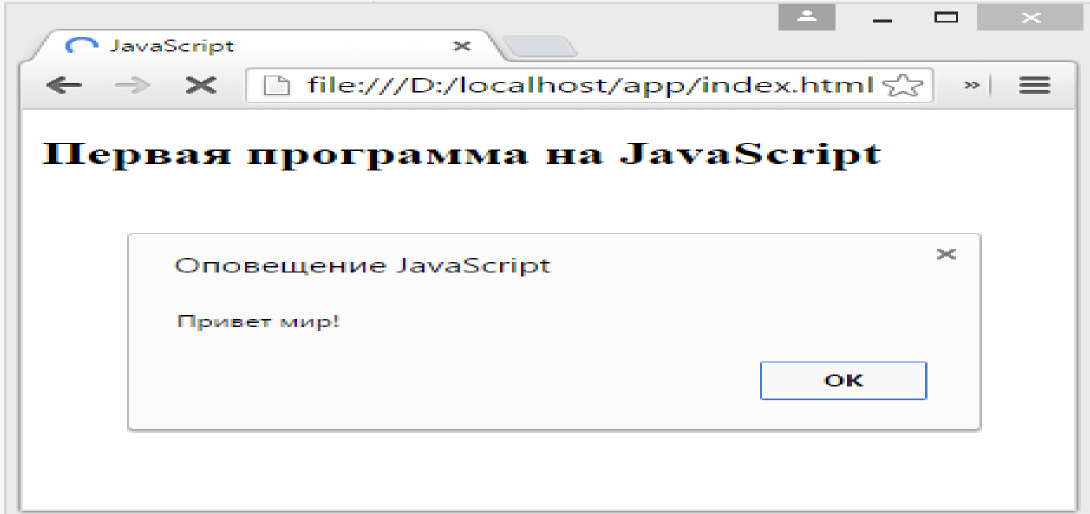
```
alert('Привет мир!');
```

Javascript կոդը կարող է պարունակել բազմաթիվ հրահանգներ և յուրաքանչյուր հրահանգ ավարտվում է կետ-ստորակետով(;): Այս օրինակում մենք կանչել ենք alert() մեթոդը, որը 'Привет мир!' տողը ցուցադրում է որպես հաղորդագրություն:

Երբ աշխատում ենք տեքստային խմբագրիչում և օգտագործում ենք տարբեր տիպի սիմվոլներ, մենք պետք է հաշվի առնենք կոդավորումը: Այս դեպքում մենք մեր վեբ էջում օգտագործում ենք **utf-8** կոդավորում: Եվ դուք նույնպես պետք է այս կոդավորումը տեղադրեք վեբ էջի և տեքստային խմբագրիչի համար:



Կոդավորման մենյուում մենք պետք է ընտրենք "Преобразовать в UTF-8 без BOM" կետը: Այժմ, երբ մեր վեբ էջը պատրաստ է, բացենք այն մեր վեբ բրաուզերում:



Եվ վեբ բրաուզերը կցուցադրի այն ուղերձը, որը մենք ուղարկել էինք alert() մեթոդին javascript կոդի մեջ:

Javascript կոդի կատարելագործում

Երբ բրաուզերը ստանում է javascript պարունակող HTML կոդը, սկսում է մեկնաբանել այն: Տարբեր տարրերի տեսքով մեկնաբանության արդյունքը՝

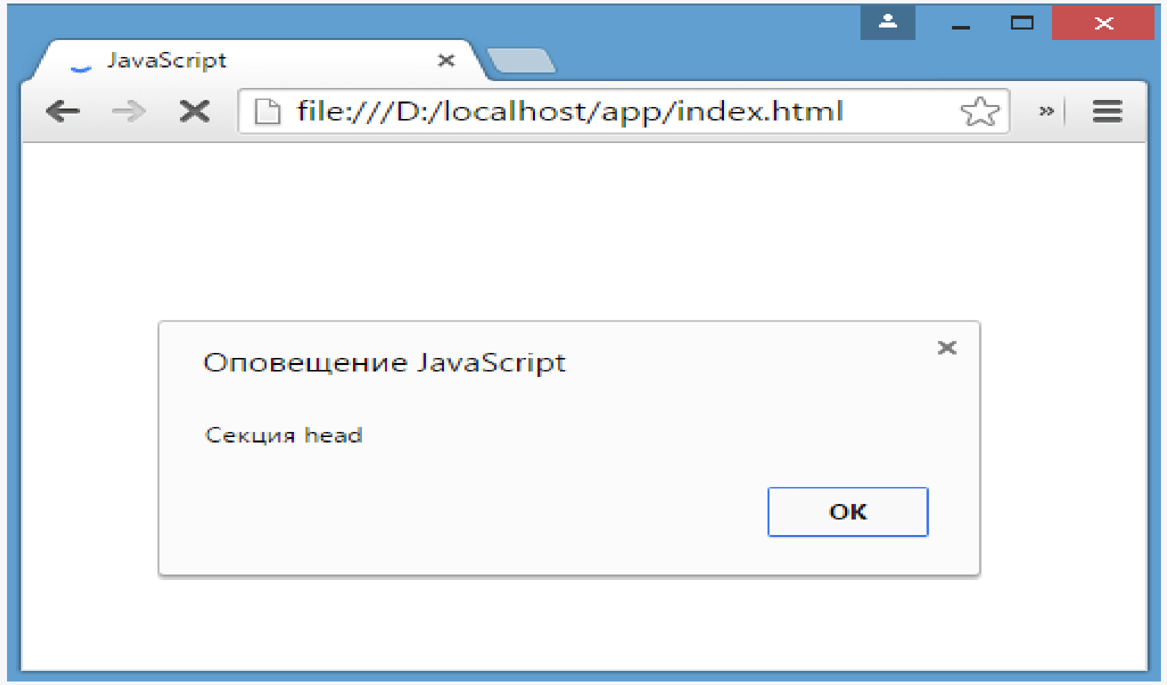
կոճակները, մուտքային դաշտերը, տեքստային բլոկները և այլն, մենք տեսնում ենք մեր բրաուզերի վրա: Վեբ էջի մեկնաբանությունը հետևողականորեն վերևից ներքև է:

Երբ բրաուզերը կոդում հանդիպում է javascript-ի <script> տարրին, ներկառուցված javascript-ի թարգմանիչը ուժի մեջ է մտնում: Եվ մինչև չի ավարտում աշխատանքը, վեբ էջի հետագա մեկնաբանումը չի կատարվում:

Քննարկենք մի փոքր օրինակ և դա անենք, փոխելով index.html ֆայլը հետևյալ կերպ.

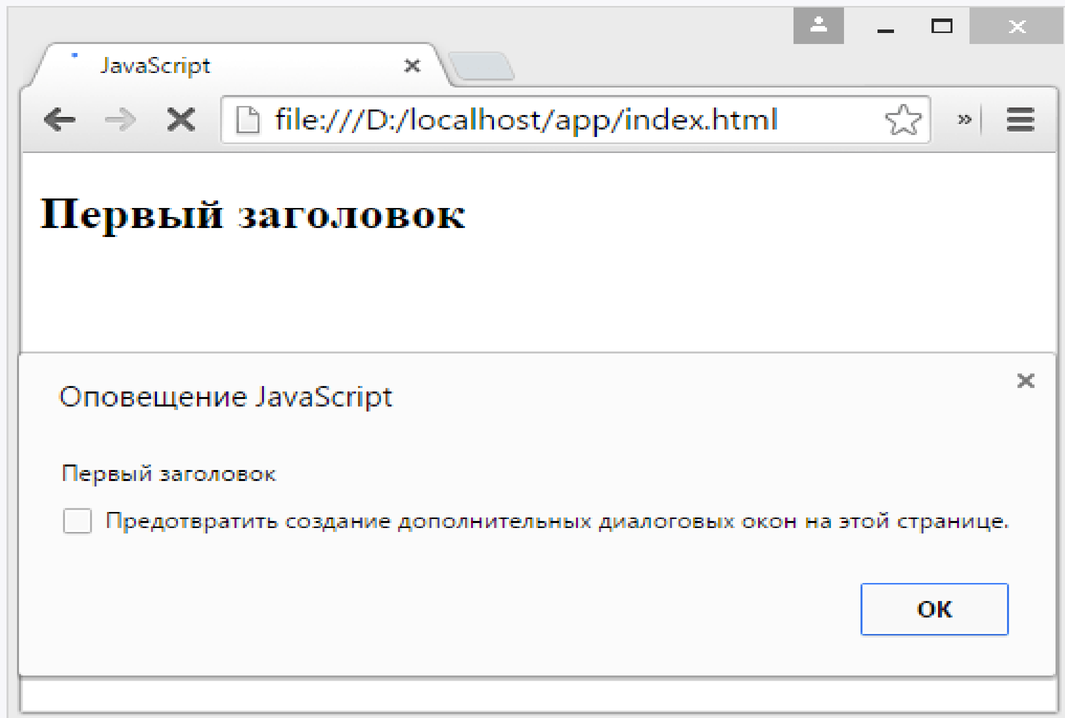
```
<!DOCTYPE html>
<html>
<head>
   <meta charset="utf-8" />
   <title>JavaScript</title>
   <script>
   alert("Секция head");
   </script>
</head>
<body>
   <h2>Первый заголовок</h2>
   <script>
   alert("Первый заголовок");
   </script>
   <h2>Второй заголовок</h2>
   <script>
   alert("Второй заголовок");
   </script>
</body>
</html>
```

Այս կոդում կա javascript-ի 3 ներդիր՝ մեկը <head> բաժնում, իսկ մյուս երկուսը <body>-ում: Բացենք այն մեր բրաուզերում:

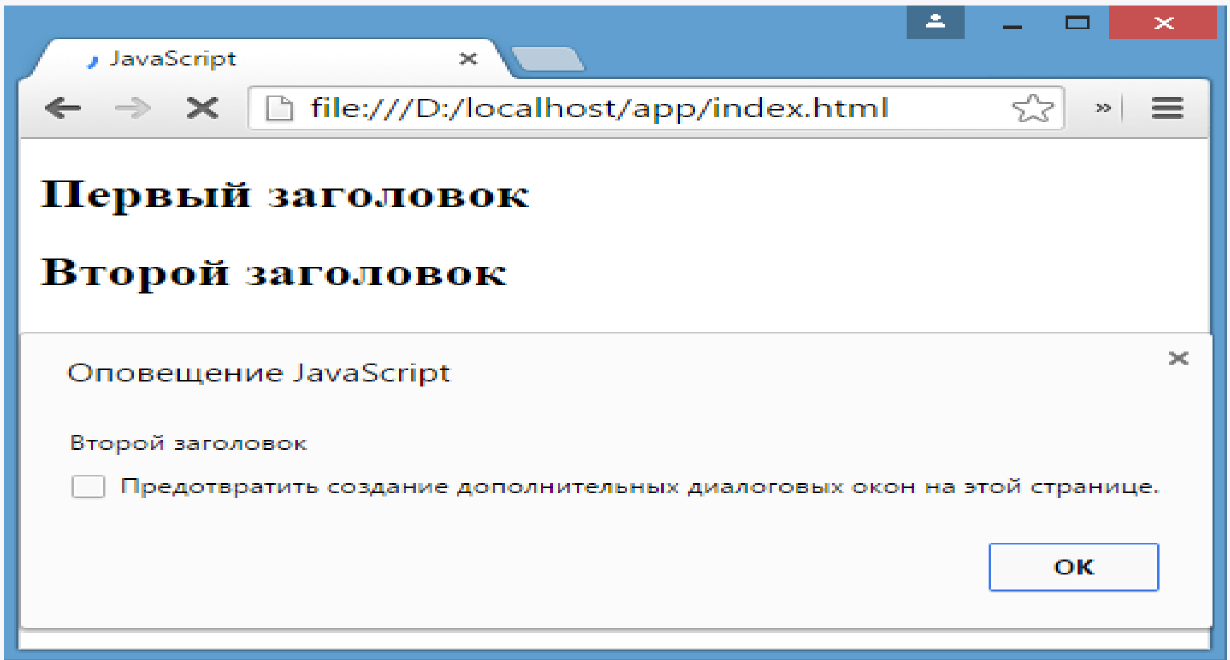


Չնայած այն հանգամանքին, որ մեր վեբ էջում(կոդում) հայտարարվում է երկու վերնագիր, սակայն դրանք դեռևս տեսանելի չեն, քանի որ <head> բաժնում իրականացվում է javascript կոդը: Եվ մինչև մենք չփակենք հաղորդագրության վանդակը, վեբ-էջի մեկնաբանումը չի շարունակվի:

Հաղորդագրությունով պատուհանը փակելուց հետո բրաուզերը կցուցադրի էջին առաջին վերնագիրն ու կրկին կդադարեցվի հաջորդ բլոկի javascript կոդի մեկնաբանումը.



Հաղորդագրության պատուհանը փակելուց հետո բրաուզերը կտեղափոխվի վեր էջ և այնտեղ կավելացնի էջի երկրորդ վերնագիրն և կանգ կառնի javascript կոդի երրորդ բլոկում.



Հաղորդագրության երրորդ պատուհանը փակելուց հետո, բրաուզերը կավարտի վեր էջի մեկնաբանումը, իսկ վեր-էջը լիովին բեռնված կլինի: Այս կետը շատ կարևոր է, քանի որ դա կարող է ազդել կատարողականի վրա: Հետևաբար, հաճախ javascript կոդերը փակվում են </ body> թեգից առաջ, երբ էջի հիմնական մասը արդեն բեռնված է բրաուզերում: Այսինքն, մեր դեպքում մենք կարող էինք բոլոր javascript կոդերը գրել մեկ <script> թեգի մեջ:

```
<!DOCTYPE html>
<html>
<head>
   <meta charset="utf-8" />
   <title>JavaScript</title>
</head>
<body>
   <h2>Первый заголовок</h2>
   <h2>Второй заголовок</h2>
   <script>
   alert("Секция head");
   alert("Первый заголовок");
   alert("Второй заголовок");
   </script>
</body>
</html>
```

Javascript լեզվի հիմունքները

Նախքան javascript ծրագրավորման լեզվի հիմունքների մանրամասն ուսումնասիրմանը անցնելը, հաշվի առնենք դրա մի քանի հիմնական կետերը:

Javascript կոդը բաղկացած է հրահանգներից, որոնք միմիանցից առանձնանում են ստորակետով (;):

```
alert("Вычисление выражения"); var a = 5 + 8; alert(a);
```

Այնուամենայնիվ, ժամանակակից բրաուզերները հեշտությամբ կարող են առանձնացնել անհատական հրահանգները, եթե դրանք պարզապես տեղադրվում են առանձին տողերում՝ առանց ստորակետի:

```
alert("Вычисление выражения")
var a = 5 + 8
```

```
alert(a)
```

Սակայն ծրագրային կոդի ընթեռնելիության բարելավման և հնարավոր սխալների նվազեցման համար խորհուրդ է տրվում յուրաքանչյուր javascript հրահանգը սահմանել առանձին տողում և ավարտել այն ստորակետով (;), ինչպես առաջին տարբերակում:

Javascript կոդում կարող ենք օգտագործել նաև մեկնաբանություններ(Comments): Մեկնաբանություններ կարող են լինել մեկտողանի, որոնց համար օգտագործվում է կրկնակի թեք գիծ(slash //):

```
// հաղորդագրության թողարկում
alert("Вычисление выражения");
// թվաբանական գործողություն
var a = 5 + 8;
alert(a);
```

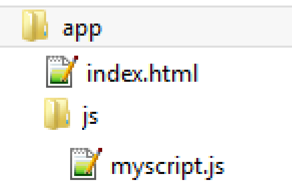
Մեկտողան մեկնաբանություններից բացի, կարող են օգտագործվել նաև բազմատողանի մեկնաբանություններ: Նման մեկնաբանությունները գրվում են հետևյալ սիմվոլների միջև՝ /* */: Օրինակ՝

```
/*  հաղորդագրության թողարկում և
թվաբանական գործողություն */
alert("Вычисление выражения");
var a = 5 + 8;
alert(a);
```

Արտաքին javascript ֆայլի տեղադրումը

Javascript կոդը վեբ էջին միացնելու եղանակներից է՝ կոդի ներկայացումը արտաքին ֆայլում, և դրանք կապումը <script> թեգի միջոցով:

Այսպիսով, նախորդ թեմայում մենք ստեղծեցինք html էջ index.html անունով, որը գտնվում է ծրագրի գրացուցակում: Հիմա այս ցուցակում ստեղծենք նոր ենթացուցակ: Եկեք այն անվանենք js: Այն նախատեսված է javascript կոդով ֆայլերը պահելու: Այս ենթացուցակում մենք կստեղծենք նոր տեքստային ֆայլ, որը մենք կկոչենք myscript.js : Javascript կոդով ֆայլերը ունեն .js ընդլայնումը: Այսինքն, ծրագրի թղթապանակում մենք ունենք հետևյալ կառուցվածքը.



Տեքստային խմբագրիչում բացենք myscript.js ֆայլը և այնտեղ սահմանեք հետևյալ կոդը.

```
var date = new Date(); // ընթացիկ ամսաթիվի ստացումը
var time = date.getHours(); // ընթացիկ ժամանակի ստացումը ժամերով
if(time < 13) // ժամանակի համեմատումը 13 թվի հետ
   alert('Доброе утро!'); // եթե ժամանակը 13-ից փոքր է
else
   alert('Добрый вечер!'); // եթե ժամանակը 13-ից մեծ է
```

Որպեսզի համահունչ լինի index.html էջի կոդավորման հետ, javascript կոդով ֆայլի համար նաև ցանկալի է սահմանել utf-8 կոդավորում:

Այս դեպքում արդեն ավելի շատ javascript արտահայտություններ կան: Մեր օրինակում առաջին արտահայտությունն ստանում է ընթացիկ ամսաթիվը և այն վերագրում է date փոփոխականին: Օգտագործելով երկրորդ հրահանգը, մենք ստանում ենք ժամանակ՝ ժամերով: Այնուհետև մենք ստացված ժամանակը համեմատում ենք 13-ի հետ և, կախված ստուգումների արդյունքներից, ցուցադրվում է առաջին կամ երկրորդ հաղորդագրությունը:
Այժմ այս ֆայլը միացնենք վեր էջի index.html փաթեթին:

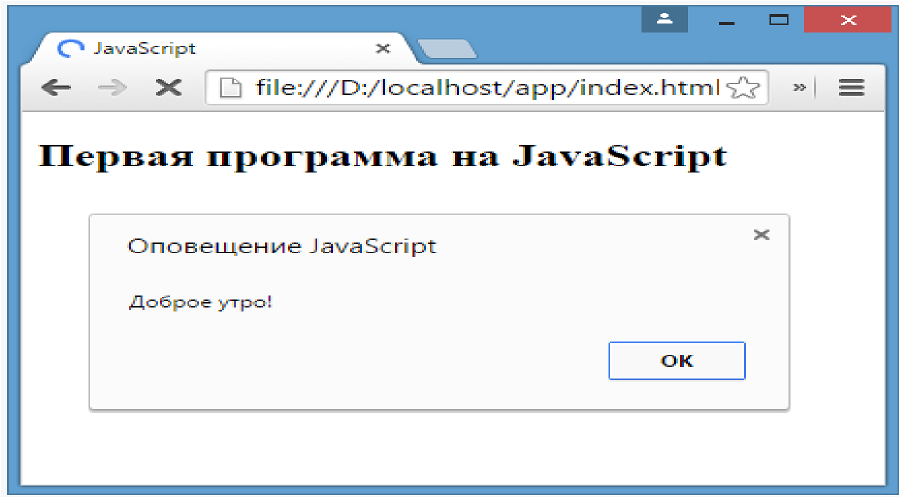
```
<!DOCTYPE html>
<html>
<head>
   <meta charset="utf-8" />
<title>JavaScript</title>
</head>
<body>
   <h2>Первая программа на JavaScript</h2>
   <script src="js/myscript.js"></script>
</body>
```

</html>

Վեբ էջում javascript կոդով ֆայլ կցելու համար օգտագործվում է <script> թեգը, որն ունի src ատրիբուտը: Այս ատրիբուտը նշում է կցվող ֆայլի ուղին: Մեր դեպքում, օգտագործվում է հարաբերական ուղին: Քանի որ վեբ էջը նույն թղթապանակում է, մենք կարող ենք գրել js / myscript.js-ը որպես ուղի:

Պետք է նաև հաշվի առնել, որ բացման <script> թեգին պետք է հետևի փակմանը </script> թեգը:

Իսկ բրաուզերում index.html ֆայլը բացելուց հետո կցուցադրվի հաղորդագրությունը.



javascript կոդը արտաքին ֆայլում պահելը ունի մի քանի առավելություն.

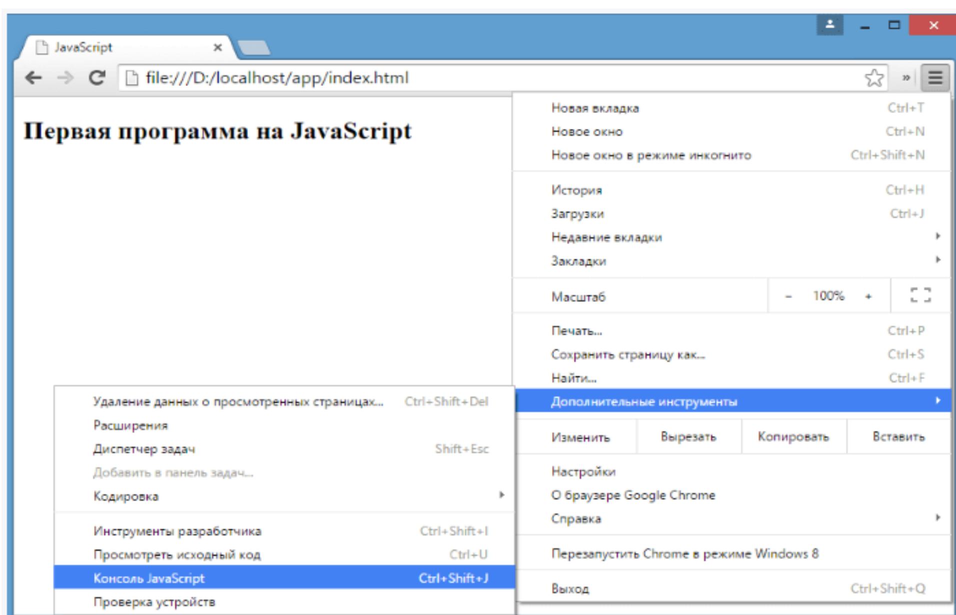
- Մենք կարող ենք նույն կոդն օգտագործել բազմակի վեբ էջերում:
- Բրաուզերի արտաքին javascript ֆայլերը կարող են պահվել, այդ պատճառով հաջորդ անգամ մուտք գործելով էջը, զննարկիչը նվազեցնում է սերվերի վրա բեռը և զննարկիչը պետք է ավելի քիչ տեղեկություններ ներբեռնի:
- Վեբկայքի կոդը մաքուր է, այսինքն այն պարունակում է միայն HTML հրամաններ, և գործողությունների կոդը պահվում է արտաքին ֆայլերում: Արդյունքում կարող եք առանձնացնել html էջի կոդը ստեղծելու գործը javascript կոդի գրելուց:

Հետևաբար, որպես կանոն, նախընտրելի է օգտագործել javascript կոդը արտաքին ֆայլերից, այլ ոչ թե ուղղակի ներդնել վեբ էջի փաթեթում:

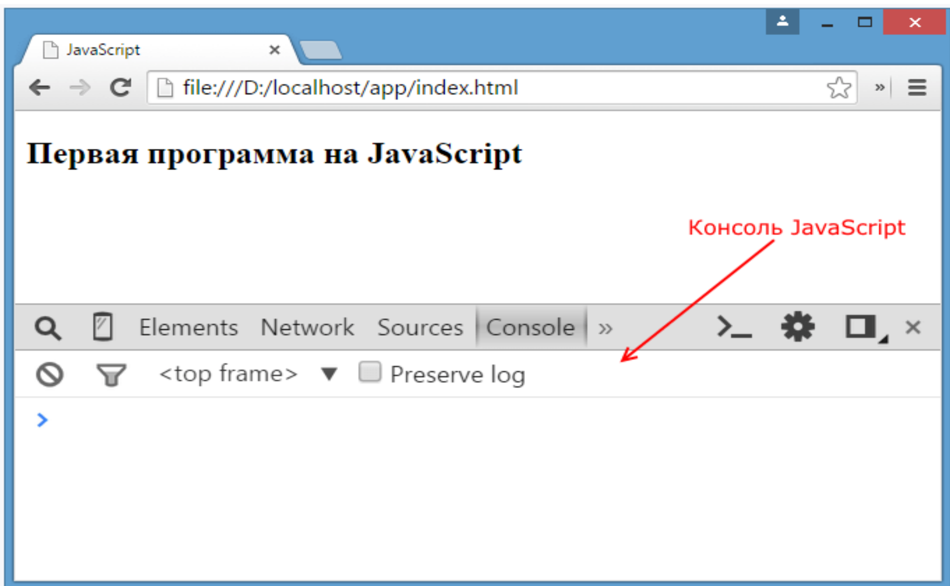
Վահանակով բրաուզեր, console.log և document.write

Երբ բրաուզերը աշխատում է javascript-ի հետ, անհրաժեշտ է այնպիսի գործիք, որը թույլ է տալիս կարգաորել ծրագիրը: Շատ ժամանակակից բրաուզերների ունեն նման վահանակ: Օրինակ, Google Chrome-ի այդ վահանակը բացելու համար մենք պետք է կատարենք քայլերի հետևյալ հաջորդականությունը՝

Дополнительные инструменты -> Консоль JavaScript

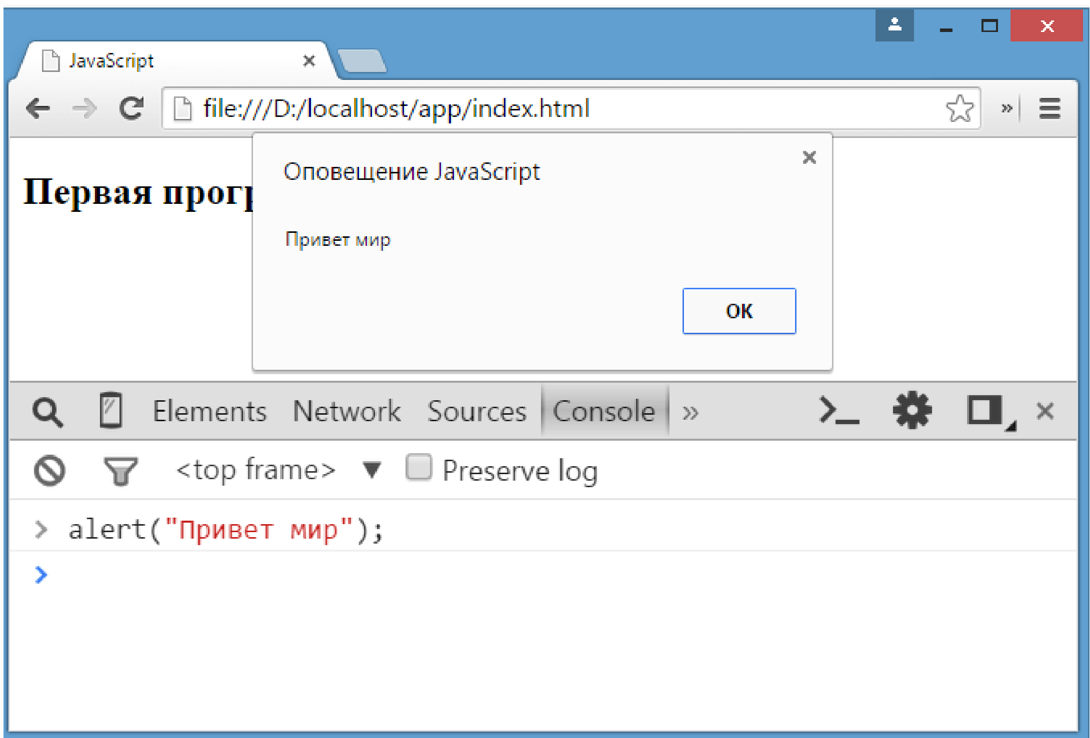


Դրանից հետո բրաուզերի ներքևում կբացվի հետևյալ վահանակը:



Մենք կարող ենք JavaScript-ի արտահայտությունները անմիջապես մուտքագրել բրաուզերի վահանակից(քոնսոլից), և դրանք կկատարվեն: Օրինակ, մուտքագրենք հետևյալ տեքստը քոնսոլում:

```
alert("Привет мир");
```

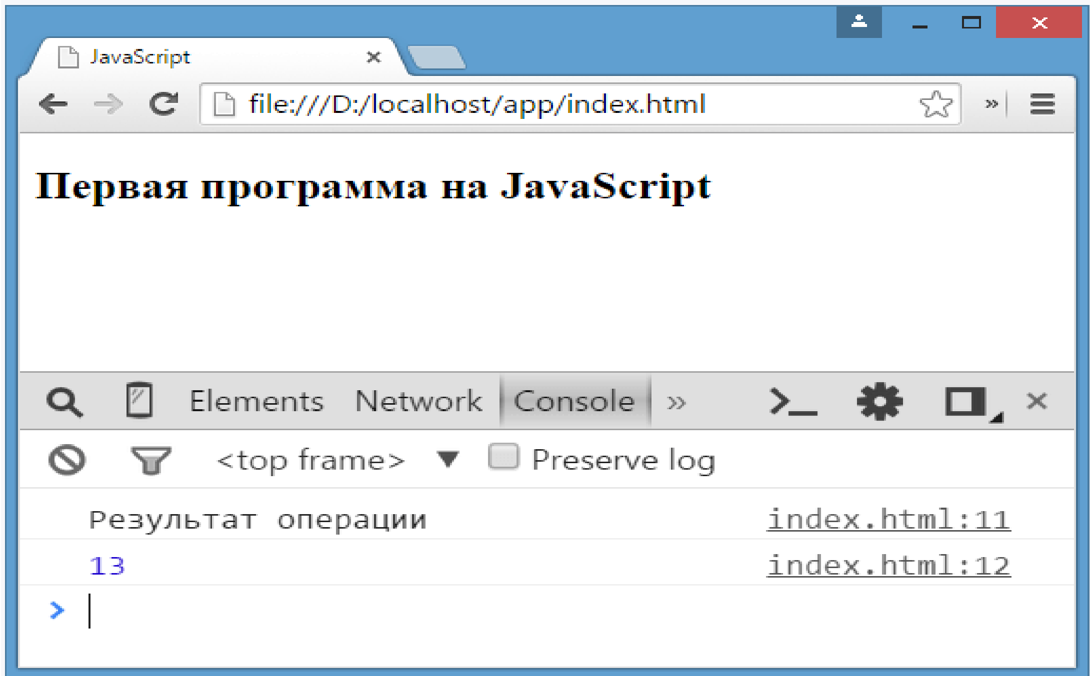


Բրաուզերը կկատարի այս հրամանը և հաղորդագրությունը կցուցադրվի պատուհանին:
Բրաուզերի կոնսոլում տարբեր տեսակի տեղեկատվություն ցուցադրելու համար օգտագործվում է console.log() հատուկ ֆունկցիան: Օրինակ, սահմանեք հետևյալ վեբ էջը.

```
<!DOCTYPE html>
<html>
<head>
   <meta charset="utf-8" />
<title>JavaScript</title>
</head>
<body>
```

```
        <h2>Первая программа на JavaScript</h2>
        <script>
        var a = 5 + 8;
        console.log("Результат операции");
        console.log(a);
        </script>
    </body>
    </html>
```

Javascript կոդում, var բառի օգտագործմամբ, հայտարարվում է a փոփոխական, որին վերագրվում է 5 և 8 թվերի գումարը՝ var a = 5 + 8: console.log () մեթոդը ցույց է տալիս a փոփոխականի արժեքը: Իսկ բրաուզերում վեր էջը գործարկելուց հետո, կոնսոլում կտեսնենք կոդի կատարման արդյունքը:



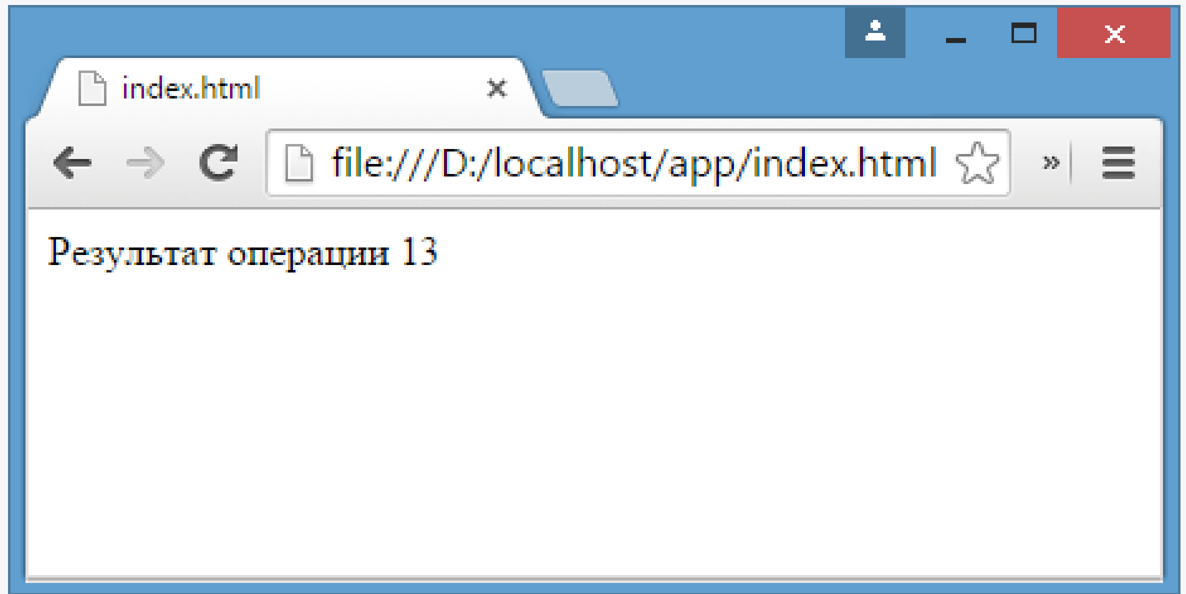
Document.write մեթոդը

Նախնական փուլում մեզ համար օգտակար է Document.write () մեթոդի օգտագործումը, որը գրում է տեղեկատվություն վեր էջում:

Օրինակ, վերցնենք նախորդ օրինակը և փոխենք console.log մեթոդը document.write մեթոդով:

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
<title>JavaScript</title>
</head>
<body>
    <h2>Первая программа на JavaScript</h2>
    <script>
    var a = 5 + 8;
    document.write("Результат операции ");
    document.write(a);
    </script>
</body>
</html>
```

Ծրագիրը ցուցադրում է գործողությունների արդյունքը վեր էջում:



Գլուխ 2. Javascript-ի հիմունքներ

Փոփոխականներ և հաստատուններ

Փոփոխականները օգտագործվում են ծրագրի տվյալները պահպանելու համար: Փոփոխականը հիշողության մեջ տիրույթ է, որի մեջ մենք կարող ենք պահել ինչ-որ արժեքներ, որոնք հետագայում՝ աշխատանքի ընթացքում կարող ենք փոփոխել: Սովորաբար փոփոխականներ ստեղծելու համար օգտագործվում են **var** և **let** բանալինային բառերը: Օրինակ, եկեք հայտարարենք myIncome փոփոխական:

```
var myIncome;
// այլ տարբերակ
let myIncome2;
```

Յուրաքանչյուր փոփոխական ունի անուն: Անունը կամայական թվանշանների հաջորդականություն է, այն կարող է գրվել ներքևի գծիկով(_) կամ դոլարի նշանով ($), բայց կարևոր զգուշացում՝ անունները չպետք է սկսվեն թվանշաններով: Այսինքն, մենք կարող ենք օգտագործել տառեր, թվեր, ներքևի գծիկ, և դոլարի նշան, իսկ մյուս բոլոր նշանները արգելված են: Ստորև ներկայացված են փոփոխականների ճիշտ անվանումներ՝

```
$commision
someVariable
product_Store
income2
myIncome_from_deposit
```

Հետևյալ անվանումները սխալ են և չեն կարող օգտագործվել.

```
222lol
@someVariable
my%percent
```

Բացի այդ, փոփոխականներին չեն կարող տրվել այնպիսի անուններ, որոնք համընկնում են վերապահված բանալինային բառերի հետ: JavaScript-ում բանալինային բառերը այնքան էլ շատ չեն ,այսինքն այս կանոնին հետևել դժվար չէ:

Օրինակ, հետևյալ անունը սխալ կլինի, քանի որ **for**-ը JavaScript-ի բանալինային բառ է.

```
var for;
```

Պահված բառերի ցանկը javascript-ում հռտևյալն է.

abstract, boolean, break, byte, case, catch, char, class, const, continue, debugger, default, delete, do, double, else, enum, export, extends, false, final, finally, float, for, function, goto, if, implements, import, in, instanceof, int, inteface, long, native, new, null, package, private, protected, public, return, short, static, super, switch, synchronized, this, throw, throws, transient, true, try, typeof, var, volatile, void, while, with:

Փոփոխականներին դիմելիս պետք է հաշվի առնել, որ JavaScript-ը « զգայուն » լեզու է, այսինքն, հետևյալ կոդով երկու տարբեր փոփոխականներ են հայտարարվում.

```
var myIncome;
var MyIncome;
```

Մենք կարող ենք միանգամից մի քանի փոփոխականներ հայտարարել, դրանք իրարից անջատելով ստորակետներով:

```
var myIncome, procent, sum;
let a, b, c;
```

Օգտագործելով հավասարման նշանը (որը նաև կոչվում է վերագրման օպերատոր), կարող ենք փոխականին արժեք փոխանցել:

```
var income = 300;
let price = 76;
```

Սկզբնական արժեքի վերագրման գործընթացը կոչվում է սկզբնարժեքավորում: Այժմ income փոփոխականին վերագրվում է 300 թիվը, իսկ price փոփոխականին 76 թիվը:

Փոփոխականներն ունեն իրենց առանձնահատկությունները: Օրինակ՝ ստորև ներկայացվող օրինակում մենք փոխում ենք արդեն իսկ հայտարարված փոփոխականների արժեքները:

```
var income = 300;
income = 400;
console.log(income);

let price = 76;
price = 54;
console.log(price);
```

Հաստատուններ

Հաստատունը նույնպես հիշողության մեջ տիրույթ է, որի մեջ կարող ենք պահել ինչ-որ արժեքներ, բայց ի տարբերություն փոփոխականի այս դեպքում մենք չենք կարող հետագայում փոխել այդ արժեքները: Հաստատունը հայտարարվում է **const** բանալինային բառի միջոցով:

```
const rate = 10;
```

Եթե մենք փորձենք փոխել հաստատունի արժեքը, մենք կկանգնենք սխալի առջև.

```
const rate = 10;
rate = 23;  // սխալ, rate-ը հաստատուն է, մենք չենք կարող փոխել դրա արժեքը
```

Հարկ է նաև նշել, որ քանի որ մենք չենք կարող փոխել հաստատունի արժեքը, այն պետք է անպայման սկզբնավորվի, այսինքն՝ սահմանելով հաստատուն, մենք պետք է նրան վերագրենք նախնական արժեք: Եթե մենք չկատարենք սկզբնարժեքավորում, ապա նորից կկանգնենք սխալի առջև:

```
const rate; // սխալ, rate-ը սկզբնարժեքավորված չէ
```

Տվյալների տիպերը

Javascript-ում օգտագործվող բոլոր տվյալները հատուկ տիպի են: JavaScript-ում գոյություն ունեն տվյալների հինգ պարզ տիպեր, դրանք են՝

- String: իրենից ներկայացնում է տող
- Number: իրենից ներկայացնում է թվային արժեք
- Boolean: իրենից ներկայացնում է ճշմարիտ(true) կամ կեղծ(false) տրամաբանական արժեք
- undefined: նշում է, որ արժեքը չի սահմանվել
- null: ցույց է տալիս անորոշ արժեք

Բոլոր այն տվյալները, որոնք չեն պատկանում վերոնշյալ հինգ տիպերից որևէ մեկին, ապա դրանք **object**-ի տիպեր են:

Թվային տիպեր

Թվերը JavaScript-ում կարող են լինել երկու տիպի՝

- **ամբողջ թվեր**, օրինակ 35: Մենք կարող ենք օգտագործել դրական և բացասական թվեր: Օգտագործված թվերը պատկանում են՝ -2^{53}-ից մինչև 2^{53}-ը ընկած միջակայքին:
- **կոտորակային թվեր**, օրինակ 3.557: Կրկին կարող ենք օգտվել ինչպես դրական, այնպես էլ բացասական թվերից: Այս տիպի թվերի համար օգտագործվում է նույն տիրույթը՝ -2^{53}-ից մինչև 2^{53}:

Օրինակ՝

```
var x = 45;
var y = 23.897;
```

Որպես առանձնացնող՝ ամբողջ թվերի և մասերի միջև, ինչպես մյուս ծրագրավորման լեզուներում, այնպես էլ JavaScript-ում օգտագործվում է կետը:

Տողեր

String տիպը իրենից ներկայացնում է տող(եր), այսինքն, այնպիսի տվյալներ, որոնք գրվում են չակերտների ներսում: Օրինակ, "Привет мир":

Նշենք, որ մենք կարող ենք օգտագործել, ինչպես մեկական այնպես էլ կրկնակի չակերտներ՝ "Привет мир" և 'Привет мир': Զգուշացում՝ փակման չակերտի տիպը պետք է լինի նույնը, ինչ բացման չակերտինն է, այսինքն, երկուսն էլ կրկնակի կամ երկուսն էլ մեկական:

Եթե տողի ներսում կան մեջբերումներ, ապա մենք պետք է խուսափենք դրանցից: Օրինակ, ենթադրենք, մենք ունենք "Бюро "Рога и копыта"" տեքստը: Այժմ ցուցադրենք մեջբերումը:

```
var companyName = "Бюро \"Рога и копыта\"";
```

Բացի այդ, մենք կարող ենք չակերտների ներսում օգտագործել այլ տիպի մեջբերումներ.

```
var companyName1 = "Бюро 'Рога и копыта'";
var companyName2 = 'Бюро "Рога и копыта"';
```

Boolean տիպ

Boolean տիպը իրենից ներկայացնու է բուլյան կամ տրամաբանական գործողություններ որոնք ընդունում են true կամ **false** (այսինքն, այո կամ ոչ) արժեքները:

```
var isAlive = true;
var isDead = false;
```

null և undefined

Հաճախ null և undefined տիպերը մեզ շփոթության մեջ են գցում: Այսպիսով, երբ մենք փոփոխական ենք սահմանում առանց սկզբնարժեքի, ապա այն իրենից ներկայացնում է undefined տիպ:

```
var isAlive;
console.log(isAlive); // կցուցադրվի undefined
```

null նշանակում է, որ տվյալ փոփոխականը ունի ինչ-որ անորոշ իմաստ (ոչ թվային, ոչ տողային, ոչ էլ Boolean/տրամաբանական/), բայց և այնպես null-ը նշանակություն ունի, քանի որ undefined նշանակում է, որ փոփոխականը չունի արժեք:

```
var isAlive;
console.log(isAlive); // undefined
isAlive = null;
console.log(isAlive); // null
isAlive = undefined;  // նորից սահմանվել է undefined տիպը
console.log(isAlive); // undefined
```

object-ի տիպեր

object-ի տիպը իրենից ներկայացնում է բարդ **object**: **object**-ի ամենապարզ սահմանումը ներկայացվում է ձևավոր փակագծերում:

```
var user = { };
```

object-ը կարող է ունենալ տարբեր հատկություններ և մեթոդներ.

```
var user = {name: "Tom", age: 24};
console.log(user.name);
```

Այս դեպքում object-ը կոչվում է օգտագործող(**user**), որը ունի երկու հատկություններ՝ անուն(**name**) և տարիք(**age**):

Թույլ տիպավորում

JavaScript-ը թույլ տիպավորմամբ լեզու է: Սա նշանակում է, որ փոփոխականները կարող են դինամիկ կերպով փոխել իրենց տիպը: Օրինակ՝

```
var xNumber;  // undefined տիպ
console.log(xNumber);
xNumber = 45;  // թվային(number) տիպ
console.log(xNumber);
xNumber = "45"; // տողային(string) տիպ
console.log(xNumber);
```

Չնայած այն հանգամանքին, որ երկրորդ և երրորդ դեպքերում բրաուզերը մեզ կբերի 45-ը, բայց երկրորդ դեպքում **xNumber** փոփոխական իրենից կներկայացնի թիվ, իսկ երրորդ դեպքում՝ տող:

Սա կարևոր կետ է, որը պետք է հաշվի առնել: Տվյալ դեպքում փոփոխության վարքագիծը կախված է ծրագրից: Որպեսզի ամեն բան հասկանալի լինի դիտարկենք հետևյալ օրինակը:

```
var xNumber = 45;  // number տիպ
var yNumber = xNumber + 5;
console.log(yNumber); // արդյունքը՝ 50

xNumber = "45"; // string տիպ
var zNumber = xNumber + 5
console.log(zNumber); // արդյունքը՝ 455
```

Վերոնշված ծրագրում՝ երկու դեպքում էլ **xNumber** փոփոխականի նկատմամբ կիրառվում է գումարման(+) գործողությունը:

.Քանի որ առաջին դեպքում **xNumber**-ը իրենից ներկայացնում է number տիպ, հետևաբար **xNumber + 5** գործողության արդյունքը ստացվում է 50 թիվը:
Երկրորդ դեպքում **xNumber**-ը իրենից ներկայացնում է string տիպ: Սակայն տողի և թիվ 5-ի միջև գումարման գործողությունը հնարավոր չէ իրականացնել: Հետևաբար, **xNumber + 5** գործողության արդյունքում 5 թիվը կվերափոխվի տողի և տեղի կունենա տողային միաձուլում, այսինքն 5-ը կկցվի 45-ին, և որպես արդյունք կստանանք "455":

Typeof օպերատորը

Օգտագործելով typeof օպերատորը, մենք կարող ենք ստանալ փոփոխականի տիպը:

```
var name = "Tom";
console.log(typeof name); // string

var income = 45.8;
console.log(typeof income); // number

var isEnabled = true;
console.log(typeof isEnabled); // boolean

var undefVariable;
console.log(typeof undefVariable); // undefined
```

Գործողություններ փոփոխականներով

Մաթեմատիկական գործողություններ(օպերատորներ)

JavaScript-ում կիրառվում են բոլոր հիմնական մաթեմատիկական գործողությունները:

Գումարում՝

```
var x = 10;
var y = x + 50;
```

Հանում՝

```
var x = 100;
var y = x - 50;
```

Բազմապատկում՝

```
var x = 4;
var y = 5;
var z = x * y;
```

Բաժանում՝

```
var x = 40;
var y = 5;
var z = x / y;
```

Մոդուլով բաժանումը (օպերատորը) վերադարձնում է բաժանման արդյունքում մնացած մասը.

```
var x = 40;
var y = 7;
var z = x % y;
console.log(z); // 5
```

Արդյունքը կլինի 5, քանի որ 40-ից փոքր և 40-ին մոտ ամենամեծ թվիվը, որը բաժանվում է 7-ի 35 թիվն է, իսկ 40-ի և 35-ի տարբերությունը 5 է, այսինքն մնացորդը 5 է, ինչն էլ ստացանք վերջնարդյունքում:

Ինքրեմենտ

```
var x = 5;
x++; // x = 6
```

Փոփոխականի աճը 1-ով անվանում են ինքրիմենտ: Ինքրիմենտի օպերատորն է՝ (++):

Օրինակ՝ **x++;** // Համարժեք է **x=x+1;** արտահայտությանը

Ինքրեմենտի օպերատորը կարող ենք դնել ինչպես փոփոխականներից առաջ /այդ դեպքում կասենք որ ունենք պրեֆիքսի օպերատոր, օրինակ ++ x /, այնպես էլ նրանցից հետո /այս դեպքում կասենք, որ ունենք պոստֆիքսի օպերատոր, օրինակ x ++/: Պրեֆիքսի օպերատորը հաշվարկվում է նախքան վերագրումը, պոստֆիքսինը՝ վերագրումից հետո:

```
// պրեֆիքսով ինքրեմենտ
var x = 5;
var z = ++x;
console.log(x); // 6
console.log(z); // 6

// պրոֆիքսով ինքրեմենտ
var a = 5;
var b = a++;
console.log(a); // 6
console.log(b); // 5
```

Դեքրեմենտ

Փոփոխականի նվազումը 1-ով անվանում են դեքրեմենտ: Դեքրեմենտի օպերատորն է՝ (--):

Օրինակ՝ **x--;** // Համարժեք է **x=x-1;** արտահայտությանը

Դեքրեմենտի օպերատորը նույնպես կարող ենք դնել ինչպես փոփոխականներից առաջ, այնպես էլ նրանցից հետո:

```
// префиксный декремент
var x = 5;
```

```
var z = --x;
console.log(x); // 4
console.log(z); // 4

// постфиксный декремент
var a = 5;
var b = a--;
console.log(a); // 4
console.log(b); // 5
```

Ինչպես մաթեմատիկայում, այնպես էլ այստեղ բոլոր գործողությունները կատարվում են ձախից աջ և տարբերվում են առաջնահերթություններով. Նախ, ինքրեմենտի և դեքրեմենտի գործողություններ, ապա բազմապատկում և բաժանում, իսկ հետո գումարում և հանում: Գործողությունների ստանդարտ ընթացքը փոխելու համար արտահայտությունների մի մասը կարող ենք գրել փակագծերում:

```
var x = 10;
var y = 5 + (6 - 2) * --x;
console.log(y); //41
```

Վերագրման օպերատորներ

- **=** փոփոխականին վերագրում է հատուկ արժեք: var x = 5;

- **+ =** ավելացվում է արժեքը, ապա փոփոխականին վերագրում նոր ստացված արժեքը: Օրինակ `

```
var x = 23;
x += 5; // սա նույնն է ինչ x = x + 5
console.log(x); // 28
```

- **- =** փոքրացնում է արժեքը, ապա փոփոխականին վերագրում նոր ստացված արժեքը: Օրինակ `

```
var x = 28;
x -= 10; // սա նույնն է ինչ x = x - 10
console.log(x); // 18
```

- *** =** բազմապատկում է արժեքը տրված թվով, ապա փոփոխականին վերագրում նոր ստացված արժեքը: Օրինակ `

```
var x = 20;
x *= 2; // սա նույնն է ինչ x = x * 2
console.log(x); // 40
```

- / = բաժանում է արժեքը տրված թվով, ապա փոփոխականին վերագրում նոր ստացված արժեքը: Օրինակ `

```
var x = 40;
x /= 4; // սա նույնն է ինչ x = x / 4
console.log(x); // 10
```

- % = Բաժանում է արժեքը տրված թվով, ապա փոփոխականին վերագրում բաժանման արդյունքում ստացված մնացորդը: Օրինակ `

```
var x = 10;
x %= 3; // սա նույնն է ինչ x = x % 3
console.log(x); // 1, քանի որ 10 - 3*3 = 1
```

Համեմատության օպերատորները

Որպես կանոն, համեմատության օպերատորները օգտագործվում են պայմանի ստուգման համար: Համեմատության օպերատորները համեմատում են երկու արժեք և վերադարձնում են **true** կամ **false** արժեքը՝

- **==** օպերատորը համեմատում է երկու արժեք, և եթե դրանք հավասար են լինում, վերադարձնում է **true**, հակառակ դեպքում՝ **false** արժեքը: x == 5
- **===** օպերատորը համեմատում է երկու արժեքները և դրանց տիպերը, և եթե դրանք համընկնում են, վերադարձնում է **true**, հակառակ դեպքում՝ **false** արժեքը: x === 5
- **! =** օպերատորը համեմատում է երկու արժեք, և եթե դրանք հավասար չեն , վերադարձնում է **true**, հակառակ դեպքում՝ **false** արժեքը: x != 5
- **! ==** օպերատորը համեմատում է երկու արժեքները և դրանց տեսակները, և եթե դրանք չեն համընկնում, վերադարձնում է **true**, հակառակ դեպքում՝ **false** արժեքը: x !== 5
- > օպերատորը համեմատում է երկու արժեք, և եթե առաջինն ավելի մեծ է, քան երկրորդը, ապա այն վերադարձնում է **true**, հակառակ դեպքում՝ **false** արժեքը: x > 5
- < օպերատորը համեմատում է երկու արժեք, և եթե առաջինը երկրորդից փոքր է, ապա վերադարձնում է **true**, հակառակ դեպքում՝ **false** արժեքը: x < 5
- >= օպերատորը համեմատում է երկու արժեք, և եթե առաջինն ավելի մեծ է կամ հավասար է երկրորդին, ապա այն վերադարձնում է **true**, հակառակ դեպքում՝ **false** արժեքը: x >= 5

- <= օպերատորը համեմատում է երկու արժեք, և եթե առաջինը երկրորդից փոքր է կամ հավասար է, ապա այն վերադարձնում է **true**, հակառակ դեպքում՝ **false** արժեքը: x <= 5

Բոլոր օպերատորները բավականին պարզ են, հավանաբար, բացառությամբ հավասարության օպերատորի(==) և ինքնության օպերատորի(===): Նրանք երկուսն էլ համեմատում են երկու արժեք, բայց ինքնության օպերատորը հաշվի է առնում նաև արժեքի տիպը: Օրինակ `

```
var income = 100;
var strIncome = "100";
var result = income == strIncome;
console.log(result); //true
```

Համեմատության արդյունքում result փոփոխականը կունենա true արժեքը, քանի որ ըստ էության income և **strIncome** փոփոխականները ունեն 100 արժեքը:
Սակայն ինքնության օպերատորը այս դեպքում կվերադարձնի **false**, քանի որ **income** և **strIncome** փոփոխականներն ունեն տարբեր տիպեր.

```
var income = 100;
var strIncome = "100";
var result = income == strIncome;
console.log(result); // false
```

Ժխտման օպերատորները՝ != և !==, նույն սկզբունքով են աշխատում:

Տրամաբանական գործողություններ

Տրամաբանական գործողությունները օգտագործվում են երկու համեմատական գործողությունների արդյունքները համադրելու համար: JavaScript-ը ունի հետևյալ տրամաբանական գործողությունները.

- **&&** (տրամաբանական և): Վերադարձնում է **true**, եթե երկու համեմատական գործողությունները վերադարձնեն **true**, հակառակ դեպքում վերադարձնում է **false**:

```
var income = 100;
var percent = 10;
var result = income > 50 && percent < 12;
console.log(result); //true
```

- || (տրամաբանական կամ)։ Վերադարձնում է **true**, եթե առնվազն մեկ համեմատության գործողությունը վերադարձնի **true**, հակառակ դեպքում վերադարձնում է **false**։

```
var income = 100;
var isDeposit = true;
var result = income > 50 || isDeposit == true;
console.log(result); //true
```

- ! (տրամաբանական ոչ)։ Վերադարձնում է **true**, եթե համեմատության արդյունքը **false** է։

```
var income = 100;
var result1 = !(income > 50);
console.log(result1); // false, քանի որ income > 50 վերադարձնում է true

var isDeposit = false;
var result2 = !isDeposit;
console.log(result2); // true
```

String գործողություններ

String տիպի փոփոխականները, կամ այլ կերպ ասած տողերը կարող են օգտագործել + կապիչը կապակցման համար։ Օրինակ `

```
var name = "Том";
var surname = "Сойер"
var fullname = name + " " + surname;
console.log(fullname); //Том Сойер
```

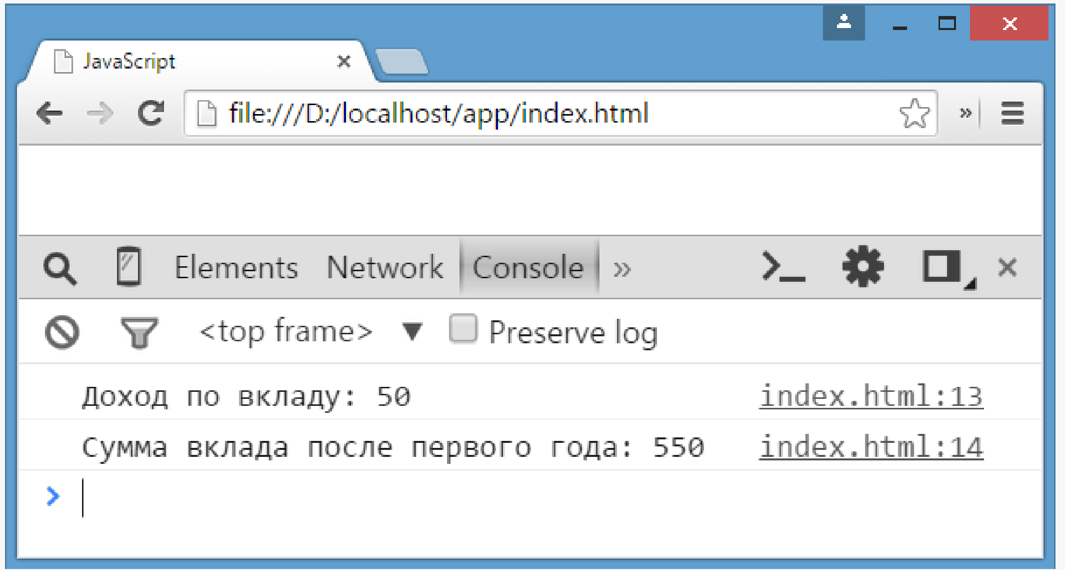
Եթե արտահայտություններից մեկը իրենից ներկայացնում է տող, իսկ մյուսը մի թիվ է, ապա համարը վերափոխվում է մի տողի, այլ կերպ ասած Convert է արվում, և կատարվում է տողերի միացման գործողությունը.

```
var name = "Том";
var fullname = name + 256;
console.log(fullname); //Том256
```

Վերջում մենք կգրենք փոքր ծրագիր, որը ցույց կտա փոփոխականների վրա գործողությունների կատարման սկզբունքը: Դա անելու համար մենք սահմանում ենք հետևյալ ծրագրային կոդը index.html ֆայլում:

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

    <meta charset="utf-8" />
    <title>JavaScript</title>

</head>
<body>
<script>

    var sum = 500; // Գումարի մուտքագրում
    var = 10;  //Տոկոսի սահմանում
    var income = sum * percent / 100;  // Գումարի տրված տոկոսի հաշվում, «եկամուտ»
    sum = sum + income; // Գումարին տոկոսի ավելացում
    console.log("Доход по вкладу: " + income);
    console.log("Сумма вклада после первого года: " + sum);

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

Ծրագրային կոդում հայտարարում ենք երեք փոփոխական՝ **sum**(գումար), **percent** (տոկոս) և income(եկամուտ): Եկամուտը հաշվարկվում է մնացած երկու փոփոխականների միջոցով, օգտագործելով բազմապատկման և բաժանման գործողությունները: Իսկ դրա արժեքի վերջում ավելացվում է փոփոխական գումարի արժեքը: Իսկ մեր բրաուզերում կցուցադրվի հտևյալը.



Տվյալների փոխակերպումները

Հաճախ անհրաժեշտություն կա մի տվյալը փոխարինել մեկ ուրիշով: Օրինակ՝

```
var number1 = "46";
var number2 = "4";
var result = number1 + number2;
console.log(result); //464
```

Երկու փոփոխականները իրենցից ներկայացնում են տողեր, որոնք ունեն թվային ներկայացումներ: Այդ իսկ պատճառով մենք վերջում ստանում ենք ոչ թե 50-ը, այլ 464-ը: Բայց լավ կլիներ, եթե կարողանայինք դրանք նաև գումարել, հանել և ընդհանրապես կատարել այն բոլոր գործողությունները, որոնք կատարում ենք սովորական թվերով աշխատելիս:

Այս դեպքում մեզ օգնության են գալիս փոխակերպման ֆունկցիաները: Պարամետրը փոփոխելու համար օգտագործեք parseInt() Ֆունկցիան.

```
var number1 = "46";
var number2 = "4";
var result = parseInt(number1) + parseInt(number2);
console.log(result); // 50
```

Կոտորակային թվեր պարունակող տողերի փոխակերպումը կատարում ենք օգտագործելով parseFloat() ֆունկցիան:

```
var number1 = "46.07";
```

```
var number2 = "4.98";
var result = parseFloat(number1) + parseFloat(number2);
console.log(result); //51.05
```

Այստեղ հարց է առաջանում, իսկ եթե տողը խառը բովանդակություն ունի, այսինք պարունակում է և թվեր և նիշեր: Օրինակ, **«123hello»**: Ամեն դեպքում **parseInt()** մեթոդը փորձում է կատարել փոխակերպումը:

```
var num1 = "123hello";
var num2 = parseInt(num1);
console.log(num2); // 123
```

Եթե մեթոդը չի կատարում փոխակերպումը, այն վերադարձնում է **NaN** (ոչ մի թիվ) արժեքը, որը ցույց է տալիս, որ տողը չի պարունակում ոչ մի թիվ, և փոխակերպում տեղի ունենալ չի կարող:

Օգտագործելով **NaN()** ֆունկցիան, մենք կարող ենք ստուգել, թե արդյոք տողը իր մեջ պարունակում է որևէ թիվ, թե ոչ: Եթե տողում չկա ոչ մի թիվ, ապա ֆունկցիան վերադարձնում է **true**, հակառակ դեպքում **false**:

```
var num1 = "javascript";
var num2 = "22";
var result = isNaN(num1);
console.log(result);  // true - num1-ը չի պարունակում ոչ մի թիվ

result = isNaN(num2);
console.log(result);  // false - num2 -  սա թիվ է
```

Վերևում մենք դիտարկեցինք տասնորդական թվեր պարունակող տողերի փոխակերպումները: Սակայն մենք կարող ենք փոխակերպել թվերը ցանկացած համակարգում:

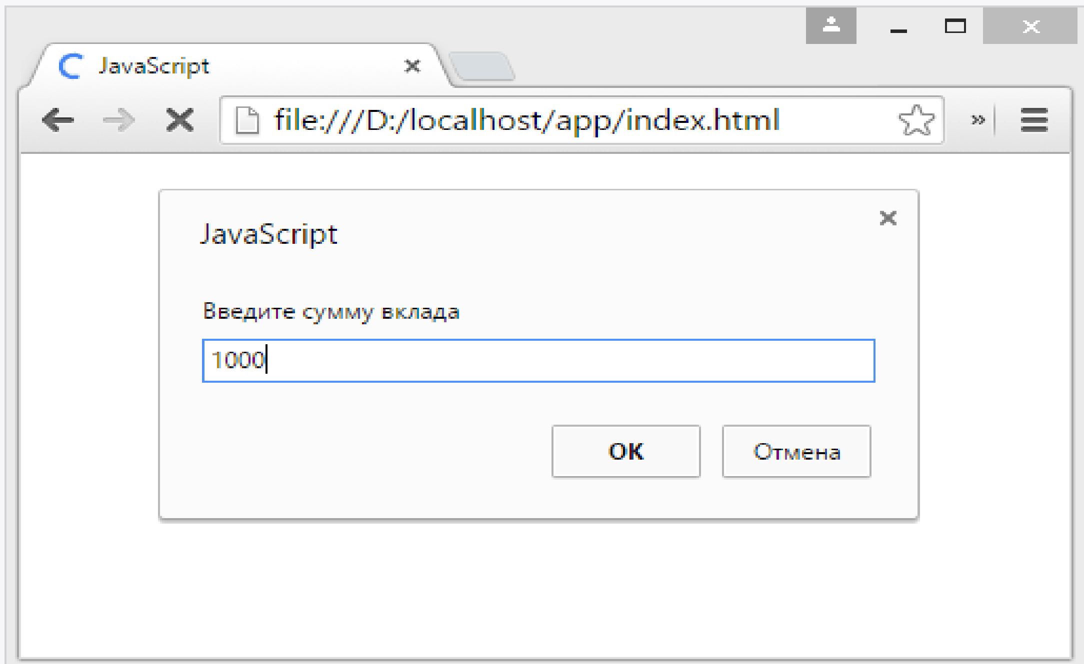
Լռելյայնորեն, JavaScript-ի թարգմանիչը ինքնին գուշակում է այն համակարգը, որի մենք ուզում ենք փոխակերպել տողը, և որը թվային համակարգ է (որպես կանոն, ընտրված է տասնորդական համակարգը): Բայց մենք կարող ենք ակնհայտորեն օգտագործել երկրորդ պարամետրը՝ նշելու համար, որ մենք ուզում ենք մի տողը փոխարկել որոշակի համակարգում: Օրինակ՝ եկեք երկուական համակարգով մի տող փոխակերպենք՝

```
var num1 = "110";
var num2 = parseInt(num1, 2);
console.log(num2); // 6
```

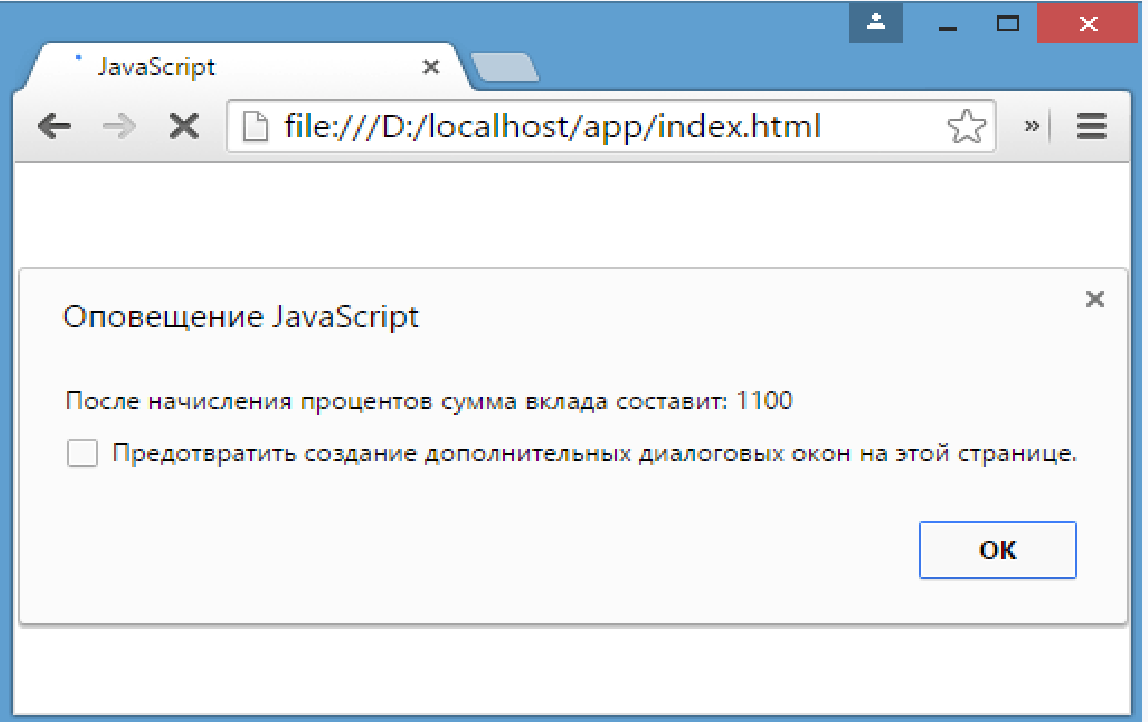
Արդյունքը կլինի 6-ը, քանի որ 110 թիվի տասական ներկայացումը 6-ն է:
Այժմ մենք կստեղծենք մի փոքրիկ ծրագիր, որտեղ կօգտագործենք ըվյալների փոխակերպման ֆունկցիաներ:

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

    <meta charset="utf-8" />
    <title>JavaScript</title>

</head>
<body>
<script>

    var strSum = prompt("Введите сумму вклада", 1000);
    var strPercent = prompt("Введите процентную ставку", 10);
    var sum = parseInt(strSum);
    var procent = parseInt(strPercent);
    sum = sum + sum * procent / 100;
    alert("После начисления процентов сумма вклада составит: " + sum);

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

Օգտագործելով **prompt()** ֆունկցիան բրաուզերը ցուցադրում է երկխոսության պատուհան՝ առաջարկելով մուտքագրել ինչ-որ արժեք: Այս ֆունկցիայի երկրորդ պարամետրը մատնանշում է այն արժեքը, որը կկիրառվի լռելյայն:
Այնուամենայնիվ, **prompt()** ֆունկցիան վերադարձնում է տող: Հետևաբար, մենք պետք է այս տողը փոխակերպենք թվի, որպեսզի դրա հետ կարողանանք գործողություն կատարել:
Բրաուզերում էջը բացելուց հետո կտեսնենք, այն երկխոսության պատուհանը, որտեղ պետք է մուտքագրենք գումարը:



Այնուհետև նման երկխոսության պատուհան կհայտնվի նաև տոկոսը մուտքագրելու համար: Իսկ վերջում ծրագիրը կստանա տվյալները, դրանք կփոխակերպի թվերի և հաշվարկ կկատարի:



Զանգվածներ

Զանգվածները նախատեսված են տվյալների հավաքածուների հետ աշխատելու համար: Ստեղծեք զանգված, օգտագործելով **new Array()** արտահայտությունը:

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

Զանգված ստեղծելու համար կան նաև ավելի կարճ ճանապարհ՝

```
var myArray = [];
```

Այս դեպքում մենք ստեղծում ենք դատարկ զանգված: Բայց մենք կարող ենք նաև նախնական տվյալներ ավելացնել զանգվածում: Ինչես օրինակ՝

```
var people = ["Tom", "Alice", "Sam"];
console.log(people);
```

Այս դեպքում people զանգվածում երեք տարրեր կլինեն: Գրաֆիկորեն այն կարող է ներկայացվել հետևյալ կերպ.

Ինդեքս	Տարր
0	Tom
1	Alice
2	Sam

Ինդեքսները օգտագործվում են զանգվածի առանձին տարրերի համար: Ինդեքսների հաշվարկը սկսվում է զրոյից, այսինքն, առաջին տարրը կունենա 0 ինդեքս, իսկ վերջինը՝ 2:

```
var people = ["Tom", "Alice", "Sam"];
console.log(people[0]); // Tom
var person3 = people[2]; // Sam
console.log(person3); // Sam
```

Եթե մենք փորձենք մուտքագրել զանգվածի տարրերի քանակից ավելի մեծ թիվ, ապա մենք կստանանք անորոշ արժեք՝ **undefined**:

```
var people = ["Tom", "Alice", "Sam"];
console.log(people[7]); // undefined
```

Նույնիսկ ըստ ինդեքսի արժեքներ է սահմանում զանգվածի տարրերի համար.

```
var people = ["Tom", "Alice", "Sam"];
console.log(people[0]); // Tom
```

```
people[0] = "Bob";
console.log(people[0]); // Bob
```

Ավելին, ի տարբերություն այլ լեզուների, ինչպիսիք են C # կամ Java, այստեղ կարող ենք սահմանել տարր, որը նախապես սահմանված չի եղել:

```
var people = ["Tom", "Alice", "Sam"];
console.log(people[7]); // undefined - զանգվածում ընդամենը երեք տարր կա
people[7] = "Bob";
console.log(people[7]); // Bob
```

Հարկ է նշել նաև, որ ի տարբերություն մի շարք ծրագրավորման լեզուների JavaScript-ի զանգվածները խիստ տիպավորրված չեն, այսինքն մենք մեկ զանգվածում կարող ենք պահել տվյալների տարբեր տիպեր:

```
var objects = ["Tom", 12, true, 3.14, false];
console.log(objects);
```

spread-օպերատորը

spread օպերատորը ... թույլ է տալիս զանգվածից առանձին արժեքներ վերցնել:

```
let numbers = [1, 2, 3, 4];
console.log(...numbers);    // 1 2 3 4
console.log(numbers);       // [1, 2, 3, 4]
```

spread օպերատորը նշվում է զանգվածի առաջևում: Արդյունքում արտահայտությունը **...numbers** կվերադարձնի թվերի շարք, բայց դա չի կարող լինել զանգված, այլ առանձին արժեքներ:

Բազմաչափ զանգվածներ

Զանգվածները կարող են լինել միաչափ և բազմաչափ: Յուրաքանչյուր տարր բազմաչափ զանգվածում կարող է լինել առանձին միաչափ զանգված: Վերևում մենք դիտարկեցինք միաչափ զանգվածը, հիմա մենք կստեղծենք բազմաչափ զանգված:

```
var numbers1 = [0, 1, 2, 3, 4, 5 ]; // միաչափ զանգված
var numbers2 = [[0, 1, 2], [3, 4, 5] ]; // երկչափ զանգված
```

Գրաֆիկորեն սա կարող ենք ներկայացնել հետևյալ կերպ.

numbers1 միաչափ զանգվածը՝

0	1	2	3	4	5

numbers2 երկչափ զանգվածը՝

0	1	2

3	4	5

Քանի որ numbers2 զանգվածը երկչափ է զանգված է, ապա այն կարող ենք անվանել նաև մատրից: Մատրիցի յուրաքանչյուր տարր կարող է ներկայացվել որպես առանձին զանգված:

```
var people = [
    ["Tom", 25, false],
    ["Bill", 38, true],
    ["Alice", 21, false]
];

console.log(people[0]); // ["Tom", 25, false]
console.log(people[1]); // ["Bill", 38, true]
```

people զանգվածը կարող է ներկայացվել հետևյալ աղյուսակի տեքով.

Tom	25	false
Bill	38	true
Alice	21	false

Զանգվածի էլեմենտները ստանալու համար օգտագործվում են ինդեքսները:

```
var tomInfo = people[0];
```

Միայն հիմա tomInfo փոփոխականը կներկայացնի զանգված: Ներկառուցված զանգվածի մեջ տարր ստանալու համար, մենք պետք է օգտագործենք նրա երկրորդ չափողականությունը.

```
console.log("Имя: " + people[0][0]); // Tom
console.log("Возраст: " + people[0][1]); // 25
```

Այսինքն, եթե մենք պատկերացնել երկչափ զանգվածը որպես մատրիցա, ապա people[0][1] տարրը կլինի մատրիցի առաջին տողի և երկրորդ սյան հանգույցը (առաջին տողի ինդեքսը 0 է, երկրորդ սյան ինդեքսը՝ 1):

Մենք կարող ենք նաև փոփոխություններ կատարել զանգվածում՝

```
var people = [
    ["Tom", 25, false],
    ["Bill", 38, true],
    ["Alice", 21, false]
```

```
];
people[0][1] = 56; // նոր արժեքի վերագրում
console.log(people[0][1]); // 56

people[1] = ["Bob", 29, false]; // զանգվածի սահմանում
console.log(people[1][0]); // Bob
```

Բազմաչափ զանգվածներ ասելով, մենք չենք սահմանափակում միայն երկչափ զանգվածով: Գոյություն ունի մեծ չափումենր ունեցող այլ զանգվածներ, որոնք մենք կարող ենք օգտագործել:

```
var numbers = [];
numbers[0] = []; // numbers – երկչափ զանգված
numbers[0][0]=[]; // numbers – եռաչափ զանգված
numbers[0][0][0] = 5; // եռաչափ զանգվածի առաջին տարրը 5 է
console.log(numbers[0][0][0]);
```

Պայմանական կառույցներ

Պայմանական կառույցները թույլ են տալիս կատարել որոշակի գործողություններ՝ կախված որոշակի պայմաններից:

If բլոկը

if կառույցը ստուգում է որոշակի պայմաններ, և եթե պայմանները տեղի են ունենում, այսինքն՝ ճիշտ են լինում, ապա կատարվում է դրան հաջորդող գործողությունները:

if կառույցը ունի հետևյալ գրելաձևը՝

```
if(условие) действия;
```

Օրինակ ՝

```
var income = 100;
if(income > 50) alert("доход больше 50");
```

Այստեղ, if կառույցում օգտագործել ենք հետևյալ պայմանները՝ income > 50: Եթե այս պայմանը վերադարձնի **true** արժեքը, այսինքն, income փոփոխականը լինի ավելի քան 50 արժեքը, ապա բրաուզերը ցուցադրում է հաղորդագրություն: Եթե income փոփոխականի արժեքը 50-ից պակաս է, այսինքն **false** է, ապա որևէ հաղորդագրություն չի ցուցադրվում:

Եթե անհրաժեշտ է իրականացնել մի շարք պայմանական զործողություններ, ապա գործողությունները տեղադրվում ենք ձևավոր փակագծերի ներսում:

```
var income = 100;
```

```
if(income > 50){

    var message = "доход больше 50";
    alert(message);
}
```

Ավելին, պայմանները կարող են լինել մեկից ավելին և դժվար.

```
var income = 100;
var age = 19;
if(income < 150 && age > 18){

    var message = "доход больше 50";
    alert(message);
}
```

if կառույցը թույլ է տալիս ստուգել փոփոխականի արժեքի առկայությունը: Օրինակ՝

```
var myVar = 89;
if(myVar){
    // գործողություններ
}
```

Եթե myVar փոփոխականն ունի արժեք, ապա այն կվերադարձնի պայմանական կառույցի **true** արժեքը:

Բայց հաճախ, փոփոխականի արժեքը ստուգելու համար օգտագործում ենք այլընտրանքային տարբերակ, որի ժամանակ ստուգում են, արդյոք փոփոխականի արժեքը **undefined**(չճանաչված) չէ:

```
if (typeof myVar != "undefined") {
    // գործողություններ
}
}
```

if կառույցում մենք կարող ենք նաև օգտագործել **elce** բլոկը: Այս բլոկը պարունակում է բոլոր այն հրահանգները, որոնք կատարվում են, if պայմանի կեղծ լինելու դեպքում, այսինքն երբ այն հավասար է **false**:

```
var age = 17;
if(age >= 18){
    alert("Вы допущены к программе кредитования");
}
```

```
else{
   alert("Вы не можете участвовать в программе, так как возраст меньше 18");
}
```

Օգտագործելով **else if** կառույցը, մենք կարող ենք ավելացնել այլընտրանքային պայման **if** բլոկի նկատմամբ.

```
var income = 300;
if(income < 200){
   alert("Доход ниже среднего");
}
else if(income >= 200 && income <= 400){
   alert("Средний доход");
}
else{
   alert("Доход выше среднего");
}
```

Այս դեպքում կատարվում է **else if** բլոկը: Անհրաժեշտության դեպքում մենք կարող ենք օգտագործել տարբեր **else if** բլոկներ տարբեր պայմաններով.

```
if(income < 200){
   alert("Доход ниже среднего");
}
else if(income>=200 && income<300){
   alert("Чуть ниже среднего");
}
else if(income>=300 && income<400){
   alert("Средний доход");
}
else{
   alert("Доход выше среднего");
}
```

True կամ false

JavaScript-ում ցանկացած փոփոխական կարող է օգտագործվել պայմանական արտահայտություններում, բայց ոչ մի փոփոխական իրենից boolean(բուլյան) տիպ չի ներկայացնում:

Եվ այս առումով հարց է ծագում, թե ինչու է այս կամ այն փոփոխականը վերադարձնում **true** կամ **false**: Եվ արդյունքում շատ բան է կախված լինում փոփոխականի տիպի տվյալներից:

- **undefined**- Վերադարձնում է **false**:
- **null**- Վերադարձնում է **false**:
- **Boolean**- Եթե փոփոխականը կեղծ է, ապա վերադարձվում է **false**: Հետևաբար, եթե փոփոխականը ճիշմարիտ է, ապա վերադարձվում է **true**:
- **Number**- Վերադարձնում է **false**, եթե արժեքը 0 է, կամ NaN (ոչ մի համար), հակառակ դեպքում վերադարձնում է **true**:

Օրինակ, հետևյալ փոփոխականը **false** (կեղծ) է:

```
var x = NaN;
if(x){  // false
}
```

- **String**- Վերադարձնում է **false**, եթե փոփոխականը հավասար է դատարկ տողին, այսինքն, դրա երկարությունը 0 է, հակառակ դեպքում այն վերադարձնում է **true**:

```
var y = ""; // false – քանի որ դատարկ տող է
var z = "javascript";   // true – քանի որ տողը դատարկ չէ
```

- **Object**- Միշտ վերադարձնում է **true**:

```
var user = {name:"Tom"};    // true
var isEnabled = new Boolean(false)  // true
var car = { }    // true
```

switch..case բլոկը

switch..case կառույցը այլընտրանքային տարբերակ է, որը կարող ենք օգտագործել if..else բլոկների փոխարեն, քանի որ այն թույլ է տալիս միաժամանակ մի քանի պայմաններ մշակել:

```
var income = 300;
switch(income){
   case 100 :
      console.log("Доход равен 100");
      break;
```

```
    case 200 :
        console.log("Доход равен 200");
        break;
    case 300 :
        console.log("Доход равен 300");
        break;
}
```

switch բանալինային բառից հետո, փակագծերում, գրվում է համեմատության ենթակա արտահայտությունը: Այս արտահայտության արժեքը հերթականորեն համեմատվում է case օպերատորի կողմից տրված արժեքների հետ: Եվ եթե համընկնում է լինում, այսինքն պայմանը բավարարվում է, ապա կատարվում է տվյալ case բլոկը:

Յուրաքանչյուր բլոկի վերջում տեղադրվում է ընդհատման break հայտարարությունը՝ մյուս բլոկների անիմաստ կատարումից խուսափելու համար:

Անախարժությունից խուսափելու համար մենք նախապես պետք է դիտարկենք այն դեպքը, երբ չկա ոչ մի համընկնում: Այդ խնդիրը լուծելու համար մենք կարող ենք ավելացնել **default** բլոկը:

```
var income = 300;
switch(income){
    case 100 :
        console.log("Доход равен 100");
        break;
    case 200 :
        console.log("Доход равен 200");
        break;
    case 300 :
        console.log("Доход равен 300");
        break;
    default:
        console.log("Доход неизвестной величины");
        break;
}
```

Ternary օպերատորը

Ternary օպերատորը իր մեջ ներառում է երեք օպերանդներ և ունի հետևյալ գրելաձևը՝

[առաջին օպերանդ - պայման] ? [երկրորդ օպերանդ] : [երրորդ օպերանդ];

Կախված պայմանից, **Ternary** օպերատորը վերադարձնում է երկրորդ կամ երրորդ օպերանդը. Եթե պայմանը ճիշտ է, ապա վերադարձվում է երկրորդ օպերանդը. եթե պայմանը սխալ է, ապա երրորդը: Օրինակ՝

```
var a = 1;
var b = 2;
var result = a < b ? a + b : a - b;
console.log(result); // 3
```

Եթե **a** փոփոխականի արժեքը փոքր է **b** փոփոխականի արժեքից, ապա **result** փոփոխականի արժեքը հավասար կլինի **a+b**-ին: Հակառակ դեպքում **result** փոփոխականի արժեքը հավասար կլինի **a - b**-ին:

Ցիկլեր

Ցիկլերը թույլ են տալիս, որոշակի պայմաններից ելնելով, պարբերաբար որոշակի գործողություններ կատարել: JavaScript-ն ունի հետևյալ ցիկլերի տեսակները.

- **for**
- **for..in**
- **for..of**
- **while**
- **do..while**

for ցիկլը

for ցիկլը ունի հետևյալ սահմանումը՝

```
for ([սկզբնարժեքավորում]; [պայման]; [արժեքի փոփոխություն]){
    // գործողություններ
}
```

Օրինակ, եկեք օգտագործենք for ցիկլը, զանգվածի տարրերը ստանալու համար:

```
var people = ["Tom", "Alice", "Bob", "Sam"];
for(var i = 0; i<people.length; i++){
        console.log(people[i]);
}
```

Ցիկլի առաջին մասում հայտարարվում է var տիպի i փոփոխական, որի սկզբնական արժեքը 0 է՝ i = 0:

Երկրորդ մասը այն պայմանն է, որի տեղի ունենալու դեպքում բլոկը կկատարվի: Այս դեպքում գործողությունները կկատարվեն այնքան ժամանակ, մինչև I փոփոխականի արժեքը հավասար լինի մարդկանց զանգվածի երկարությանը: Զանգվածի երկարությունը կարող ենք ստանալ, օգտագործելով people.length երկարության հատկությունը:

Երրորդ մասը փոփոխականի արժեքի ավելացումն է մեկով:

Եվ քանի որ զանգվածում առկա են 4 տարրեր, ցիկլի բլոկը կաշխատի 4 անգամ, մինչև i արժեքը դառնում է հավասար people զանգվածի երկարությանը (length), այսինքն` 4-ի, քանի որ ամեն անգամ այդ արժեքը ավելանում է 1-ով: Ցիկլի յուրաքանչյուր անհատական կրկնությունը կոչվում է իտերացիա: Այսպիսով, այս դեպքում կաշխատեն 4 իտերացիաները:

Իսկ people[i] արտահայտության օգնությամբ մենք բրաուզերում ստանում ենք զանգվածի համապատասխան տարրը:

Մենք կարող ենք այս գործողությունները կատարել նաև հետևյալ կերպ՝

```
var people = ["Tom", "Alice", "Bob", "Sam"];
for(var i = people.length - 1; i >= 0; i--){
        console.log(people[i]);
}
```

Սակայն այս դեպքում զանգվածի տարրերը դուրս են գալիս հակառակ դասավորությամբ՝ այսինքն վերջից սկիզբ: Այս դեպքում ևս կաշխատեն 4 իտերացիաները, զանգվածի կրկնությունը սկսվում է i = 3-ից մինչև i = 0:

Ցիկլ for..in

for..in ցիկլը ունի հետևյալ սահմանումը՝

```
for (ինդեքս in զանգված) {
   // գործողություններ
}
```

Օրինակ, եկեք դիտենք զանգվածի տարրերը.

```
var people = ["Tom", "Alice", "Bob", "Sam"];
for(var index in people){
        console.log(people[index]);
}
```

Ցիկլ for...of

Ցիկլ for...of-ը նման է ցիկլ for..in-ին: Օրինակ՝

```
let users = ["Tom", "Bob", "Sam"];
```

```
for(let val of users)
   console.log(val);
```

Հավաքածուի ընթացիկ թվարկված տարրը տեղադրվում է val փոփոխականում, որի արժեքը ներկայացվում է վահանակում:

Ցիկլ while

While ցիկլը կատարվում է այնքան ժամանակ, քանի դեռ որոշակի պայման ճշմարիտ է: Այն սահմանվում է հետևյալ կերպ.

```
while(պայման){
    // գործողություններ
}
```

Կրկին ներկայացնենք զանգվածի տարրերը:

```
var people = ["Tom", "Alice", "Bob", "Sam"];
var index = 0;
while(index < people.length){

   console.log(people[index]);
   index++;
}
```

While ցիկլը այստեղ կկատարվի այնքան ժամանակ, մինչև որ ինդեքսի արժեքը հավասարվի զանգվածի երկարությանը:

do..while ցիկլը

do ցիկլում առաջին գործողությունը կատարվում է առանց պայմանը ստուգելու, ապա ամեն հաջորդ գործողության համար ստուգվում է while-ում գրված պայմանը: Ցիկլը կրկնվում է այնքան ժամանակ, քանի դեռ պայմանը ճիշտ է: Օրինակ՝

```
var x = 1;
do{
   console.log(x * x);
   x++;
}while(x < 10)
```

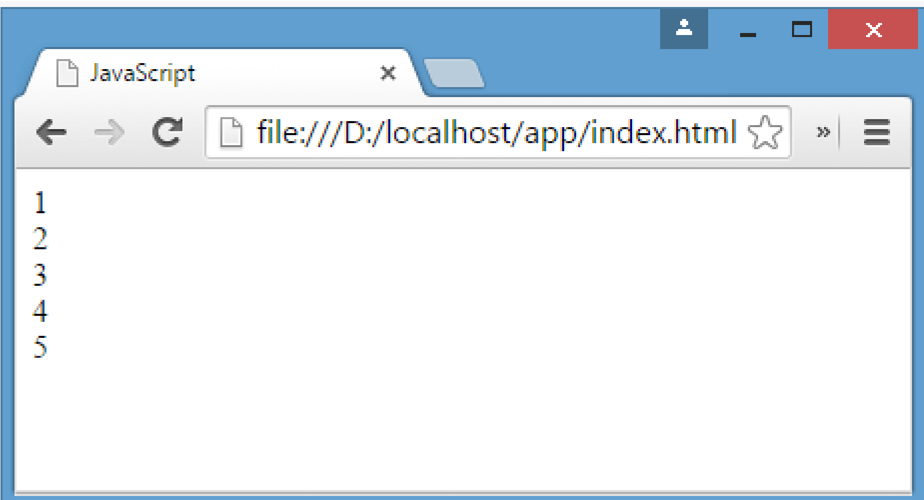
Այստեղ ցիկլի կոդը 9 անգամ կաշխատի, քանի որ x-ը չպետք է գորազանցի 10-ը: Միևնույն ժամանակ, do ցիկլը երաշխավորում է գործողությունների առնվազն մեկ անգամ կատարումը, նույնիսկ եթե while-ում գրված պայմանը ճիշտ չէ:

continue և break օպերատորները

Երբեմն անհրաժեշտություն է լինում դուրս գալ ցիկլից մինչև դրա ավարտը: Այս դեպքում, մենք կարող ենք օգտագործել break օպերատորը.

```
var array = [ 1, 2, 3, 4, 5, 12, 17, 6, 7 ];
for (var i = 0; i < array.length; i++)
{
   if (array[i] > 10)
      break;
   document.write(array[i] + "</br>");
}
```

Այս ցիկլը թվարկում է զանգվածի բոլոր տարրերը, սակայն վերջին չորս տարրերը չեն ցուցադրվի բրաուզերում, քանի որ if (array[i] > 10) պայմանի ստուգումը կդադարեցնի ցիկլի կատարումը, երբ հաշվարկը հասնի զանգվածի 12 տարրին: Այստեղ մենք օգտագործելով break օպերատորը դուրս ենք գալիս ցիկլից:

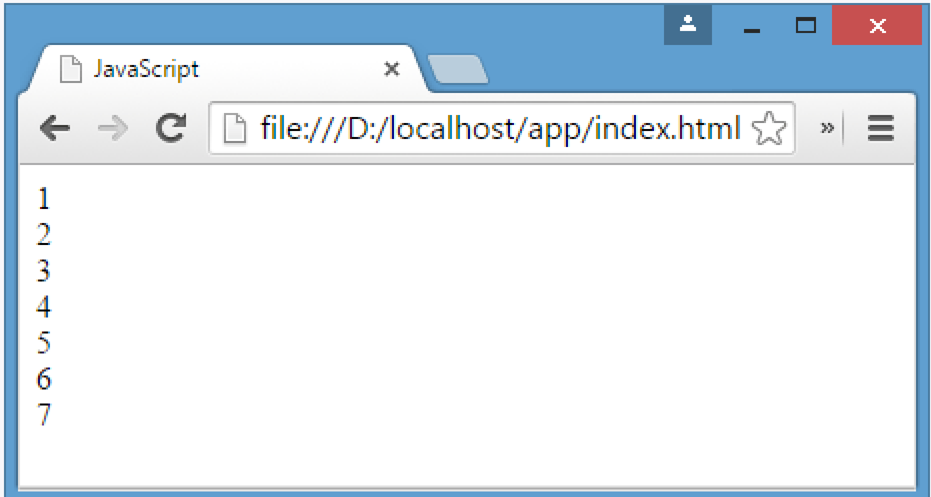


Եթե մենք պարզապես ցանկանում ենք բաց թողնել ցիկլի որոշակի հատված, բայց ոչ ամբողջ ցիկլը, մենք կարող ենք օգտագործել continue օպերատորը.

```
var array = [ 1, 2, 3, 4, 5, 12, 17, 6, 7 ];
for (var i = 0; i < array.length; i++)
{
   if (array[i] > 10)
      continue;
   document.write(array[i] + "</br>");
```

}

Այս դեպքում, եթե ծրագիրը հանդիպում է զանգվածի այնպիսի տարրի, որը 10-ից ավելի մեծ է, ապա այդ թիվը չի ցուցադրվում բրաուզերում:



Գլուխ 3. Ֆունկցիոնալ ծրագրավորում

Ֆունկցիաներ

Ֆունկցիաները հրահանգների շարք են, որոնք կատարում են որոշակի գործողություն կամ հաշվում են որևէ արժեք: Ֆունկցիան սահմանվում է հետևյալ գրելաձևով.

```
function ֆունկցիայի անվանումը([պարամետր [, ...]]){
   // Հրահանգներ
}
```

Ֆունկցիայի սահմանումը սկսվում է function բանալի բառով, այնուհետև գրվում է ֆունկցիայի անունը: Ֆունկցիայի անունը գրելիս պետք է հետևել այն նույն կանոններին, ինչ փոփոխական անունը սահմանելիս էինք հետևում, այն կարող է պարունակել միայն թվեր, տառեր, գծեր, դոլար ($) նշան և պետք է սկսվի տառով:

Ֆունկցիայի անունից հետո՝ փակագծերում թվարկվում են պարամետրերը: Նույնիսկ եթե ֆունկցիան չունի պարամետրեր, ապա պարզապես դրվում են դատարկ փակագծեր: Այնուհետև, ձևավոր փակագծերում, որը կոչվում է ֆունկցիայի մարմին, գրվում են համապատասխան հրահանգները:

Այժմ սահմանենք մի պարզ ֆունկցիա.

```
function display(){
   document.write("функция в JavaScript");
}
```

Այս օրինակում ֆունկցիայի անվանումը **display()** է: Այն ոչ մի պարամետր չի վերցնում, և այն ամենը, ինչ գրում ենք ֆոնկցիայի մարմնում, տեսնում ենք վեբ էջում: Այնուամենայնիվ, ֆունկցիայի պարզ սահմանումը բավարար չէ, որպեսզի այն աշխատի: Մենք պետք է կանչենք այն:

```
<!DOCTYPE html>
<html>
<head>
   <meta charset="utf-8" />
</head>
<body>
<script>
function display(){
   document.write("функция в JavaScript");
}
display();
</script>
</body>
</html>
```

Անհրաժեշտ չէ ֆունկցիաներին տալ կոնկրետ անուններ: Մենք կարող ենք օգտագործել նաև անանուն ֆունկցիաներ:

```
var display = function(){  // определение функции
   document.write("функция в JavaScript");
}
display();
```

Փաստորեն, մենք սահմանում ենք display անունով փոփոխական և նրան ենք վերագրում ֆունկցիայի հղումը: Եվ հետո, ֆունկցիան կանչվում է ըստ փոփոխականի անվան:

Մենք կարող ենք նաև դինամիկ կերպով նշանակել փոփոխական ֆունկցիաները.

```
function goodMorning(){
   document.write("Доброе утро");
}
function goodEvening(){
   document.write("Добрый вечер");
}
var message = goodMorning;
message(); // Բարի առավոտ
message = goodEvening;
message(); // Բարի երեկո
```

Ֆունկցիայի պարամետրերը

Դիտարկենք պարամետրի փոխանցումը.

```
function display(x){  // Ֆունկցիայի սահմանում
   var z = x * x;
   document.write(x + " в квадрате равно " + z);
}
display(5); // Ֆունկցիայի կանչ
```

display ֆունկցիան մեկ պարամետր է վերցնում,որի այս x-ն է: Հետևաբար, ֆունկցիան կանչելու դեպքում մենք կարող ենք նրան որևէ արժեք փոխանցել, օրինակ՝ 5 թիվը, ինչպես այս դեպքում:
Եթե ֆունկցիան վերցնում է մի քանի պարամետր, ապա օգտագործելով **spread** օպերատորը **...** մենք կարող ենք զանգվածից փոխանցել մի շարք արժեքներ այդ պարամետրերի համար:

```
function sum(a, b, c){
   let d = a + b + c;
   console.log(d);
}
sum(1, 2, 3); //երկրորդ դեպք
let nums = [4, 5, 6];
sum(...nums);
```

Երկրորդ դեպքում **nums** զանգվածի թվերը փոխանցվում են ֆունկցիային: Բայց որպեսզի ամբողջ զանգվածը չանցնի որպես մեկ արժեք, այլ անցնեն տվյալ զանգվածի թվերը մենք օգտագործում ենք **spread** օպերատորը (բազմակետեր **...**):

Լրացուցիչ պարամետրեր

Ֆունկցիան կարող է շատ պարամետրեր վերցնել, սակայն պարամետրերի մի մասը կամ նույնիսկ բոլորը կարող են լրացուցիչ լինել: Եթե պարամետրերի համար ոչ մի արժեք չի անցնում, ապա ըստ default-ի՝ դրանք սահմանվում են «undefined»:

```
function display(x, y){
   if(y === undefined) y = 5;
   if(x === undefined) x = 8;
   let z = x * y;
   console.log(z);
}
display();  // 40
display(6); // 30
display(6, 4) // 24
```

Այստեղ **display** ֆունկցիան երկու պարամետր է վերցնում: Ֆունկցիան կանչելիս մենք կարող ենք ստուգել ֆունկցիայի արժեքները: Այս դեպքում, ֆունկցիան կանչելիս, անհրաժեշտ չէ այդ պարամետրերին որևէ արժեքներ փոխանցել: Պարամետրային արժեքի ստուգման համար օգտագործվում է համեմատություն օպերատոր, որը համեմատում է պարամետրի արժեքը **undefined** արժեքի հետ:
Պարամետրերի արժեքները որոշելու համար գոյություն ունի ևս մեկ եղանակ՝

```
function display(x = 5, y = 10){
   let z = x * y;
   console.log(z);
}
display();     // 50
display(6);    // 60
display(6, 4)  // 24
```

Եթե x և y պարամետրերին որևէ արժեքներ չեն փոխանցվում, ապա դրանք լռելյայնորեն վերցնում են համապատասխանաբար 5 և 10 համարների արժեքները:

Այս մեթոդը ավելի հակիրճ և ինտուիտիվ է, քան նախորդ համեմատման տարբերակը:
Պարամետրի լռելյայն արժեքը կարող է հանդիսանալ նույնիսկ արտահայտությունը.

```
function display(x = 5, y = 10 + x){
   let z = x * y;
   console.log(z);
}
display();      // 75
display(6);     // 96
display(6, 4)   // 24
```

Այս դեպքում y պարամետրի արժեքը կախված է x պարամետրի արժեքից:
Անհրաժեշտության դեպքում, մենք կարող ենք ստանալ բոլոր պարամետրերը գլոբալ հասանելի arguments զանգվածի միջոցով:

```
function display(){
   var z = 1;
   for(var i=0; i<arguments.length; i++)
      z *= arguments[i];
   console.log(z);
}
display(6); // 6
display(6, 4) // 24
display(6, 4, 5) // 120
```

Նույնիսկ նշանակություն չունի, որ ֆունկցիան սահմանելիս մենք չենք սահմանում որևէ պարամետր, մենք դեռ կարող ենք դրանք փոխանցել և ստանալ դրանց արժեքները arguments զանգվածի միջոցով:

Անորոշ թվով պարամետրեր

spread օպերատորի օգնությամբ, մենք կարող ենք փոփոխականին տարբեր քանակությամբ արժեքներ տալ:

```
function display(season, ...temps){
```

```
    console.log(season);
    for(index in temps){
        console.log(temps[index]);}
}
display("Весна", -2, -3, 4, 2, 5);
display("Лето", 20, 23, 31);
```

Տվյալ դեպքում երկրորդ պարամետրը՝ ...temps-ը ցույց է տալիս, որ նրա փոխարեն կարող ենք տալ արժեքների կամայական ժողովածու.
Փաստորեն հիմնական ֆունկցիան temps-ը ներկայացնում է որպես մի զանգված, որի տարրերը մեր իսկ փոխանցած արժեքներն են և որոնք մենք կարող ենք ստանալ. Ընդ որում, չնայած այդ հանգամանքի, կանչի ֆունկցիան, այն փոխանցում է ոչ թե որպես զանգված, այլ որպես հենց առանձին արժեք.
Այժմ թողարկենք փահանակում:

```
Весна
-2
-3
4
2
5
Лето
20
23
31
```

Ֆունկցիան կարող է վերադարձնել արդյունք: Դրա համար օգտագործվում է return օպերատորը:

```
var y = 5;
var z = square(y);
document.write(y + " в квадрате равно " + z);
function square(x) {
  return x * x;
```

```
}
```

return օպերատորից հետո գրվում է այն արժեքը, որը պետք է վերադարձվի ըստ մեթոդի: Այս դեպքում դա x-ի քառակուսին է:
Ֆունկցիայի արդյունքը ստանալուց հետո մենք կարող ենք այդ ֆունկցիային տալ ցանկացած այլ փոփոխական:

```
var z = square(y);
```

Ֆունկցիան որպես պարամետր

Ֆունկցիաները կարող են հանդես գալ որպես պարամետրեր այլ ֆունկցիաներում:

```
function sum(x, y){
   return x + y;
}
function subtract(x, y){
   return x - y;
}
function operation(x, y, func){
   var result = func(x, y);
   console.log(result);
}
console.log("Sum");
operation(10, 6, sum);  // 16
console.log("Subtract");
operation(10, 6, subtract); // 4
```

operation ֆունկցիան վերցնում է երեք պարամետրեր՝ x, y և func: func-ը ևս իրենից ներկայացնում է ֆունկցիա, և operation-ի որոշման ժամանակ կարևոր չէ, թե դա ինչ ֆունկցիա կլինի: Միակ բանը, որ հայտնի է, այն է որ func ֆունկցիան կարող է երկու պարամետր վերցնել և վերադարձնել այն արժեքը, որը դրսևորվում է բրաուզերի վահանակում: Հետևաբար, մենք կարող ենք սահմանել տարբեր ֆունկցիաներ (օրինակ, գումարման(sum) և հանման(subtract) ֆունկցիաները) և դրանք փոխանցվում են կանչի ֆունկցիա:

Վերադարձի ֆունկցիան ֆունկցիայում

Մեկ ֆունկցիան կարող է վերադարձնել այլ ֆունկցիա:

```
function menu(n){
   if(n==1) return function(x, y){ return x+y;}
   else if(n==2) return function(x, y){ return x - y;}
   else if(n==3) return function(x, y){ return x * y;}
   return undefined;
}
for(var i=1; i < 5; i++){
   var action = menu(i);
   if(action!==undefined){
      var result = action(5, 4);
      console.log(result);
   }
}
```

Այս դեպքում menu ֆունկցիան, կախված է իր մեջ ներառված ֆունկցիաներից ստացված արժեքից, և ըստ այդ արժեքների վերադարձնում է երեք ֆունկցիաներից մեկը, հակառակ դեպքում վերադարձնում է անորոշ(undefined):

Տարբեր տեսանելիությամբ փոփոխականներ

JavaScript-ի բոլոր փոփոխականներն ունեն հատուկ տեսանելիություն, որի շրջանակներում նրանք կարող են գործել:

Գլոբալ փոփոխականներ

Ֆունկցիաներից դուրս հայտարարված բոլոր փոփոխականները գլոբալ են.

```
<!DOCTYPE html>
<html>
<head>
   <meta charset="utf-8" />
</head>
<body>
<script>
var x = 5;
let d = 8;
```

```
function displaySquare(){
     var z = x * x;
    console.log(z);
}
</script>
</body>
</html>
```

Այստեղ, x և d փոփոխականները գլոբալ են: Նրանք հասանելի են ծրագրի ցանկացած վայրից: Սակայն z փոփոխականը գլոբալ չէ, քանի որ այն սահմանվում է ֆունկցիայի ներսում:

Լոկալ փոփոխականներ

Ֆունկցիայի մեջ սահմանված փոփոխականը կոչվում է լոկալ.

```
function displaySquare(){
    var z = 10;
    console.log(z);
    let b = 8;
    console.log(b);
}
```

z և b փոփոխականները լոկալ են, դրանք հասանելի են միայն տվյալ ֆունկցիայի շրջանակներում: Մյուս Ֆունկցիաներում դրանք չեն կարող օգտագործվել.

```
function displaySquare(){
    var z = 10;
    console.log(z);
}
console.log(z); // սխալ, քանի որ z- ը չի սահմանվել
```

Ֆունկցիայի աշխատանքի ավարտից հետո, նրանում սահմանված փոփոխականները «ոչնչացվում» են, այսինքն, դադարում են նշանակություն ունենալ:

Թաքնված փոփոխական

Այժմ դիտարկենք այն դեպքը, երբ նույն անունով ունենք երկու փոփոխական, մեկ գլոբալ իսկ մյուսը լոկալ:

```
var z = 89;
function displaySquare(){
```

```
    var z = 10;
    console.log(z); // 10
  }
  displaySquare(); // 10
```

Այս դեպքում ֆունկցիան կօգտագործի այն z փոփոխականը, որն ուղղակիորեն սահմանվում է իր մեջ: Այսինքն, լոկալ փոփոխականը թաքցնում է գլոբալից:

var և let փոփոխականներ

let օպերատորի օգտագործման ժամանակ յուրաքանչյուր բլոկի կոդը սահմանում է տեսանելիության նոր տիրույթ, որտեղ առկա է փոփոխական:
Օրինակ, մենք կարող ենք միաժամանակ սահմանել փոփոխական և՛ բլոկի մակարդակում և՛ ֆունկցիոնալ մակարդակում.

```
let z = 10;
function displayZ(){
  let z = 20;
  {
    let z = 30;
    console.log("Block:", z);
  }
  console.log("Function:", z);
}
displayZ();
console.log("Global:", z);
```

Այստեղ, displayZ ֆունկցիայի ներսում, սահմանվում է կոդի բլոկ, որի մեջ փոփոխականը z-ն է: Այն թաքցնում է ինչպես գլոբալ z փոփոխականից, այնպես էլ ֆունկցիայի մակարդակում սահմանված z փոփոխականից:
Իրական ծրագրային բլոկում կարող են ներկայացված լինել ֆունկցիայի բլոկ, for ցիկլի բլոկ կամ if կառույց(կոնստրուկցիա): Սակայն ցանկացած դեպքում նման բլոկները սահմանում են նոր տեսանելության տիրույթներ, որոնցից դուրս փոփոխականները գոյություն չեն ունենում:
Վերը նշված գրագրի գործարկմամբ մենք ստանում ենք հետևյալ արդյունքը.

Block: 30 Function: 20 Global: 10

Օգտագործելով var օպերատորը, մենք չենք կարող միաժամանակ սահմանել նույն անունով փոփոխական ֆունկցիայում և առանձին բլոկում, ինչպես մեր օրինակում՝

```
function displaySquare(){
    var z = 20;
    {
        var z = 30;  // Սխալ. z փոփոխականն արդեն իսկ  սահմանվել է
        console.log("Block:", z);
    }
    console.log("Function:", z);
}
```

Այսինքն, օգտագործելով **var** օպերատորը, մենք կարող ենք սահմանել տվյալ անունով փոփոխականը կամ ֆունկցիայի մակարդակում կամ էլ կոդի բլոկի մակարդակում:

Հաստատուն(constant)

Այն ամենը ինչ ասվեց let օպերատորի մասին, վերաբերում է նաև const օպերատորին: Այն թույլ է տալիս մեզ սահմանել հաստատուններ:

Կոդի բլոկներում սահմանում են հաստատունների տեսանելիության տիրույթները: Բլոկներում սահմանված հաստատունները թաքնվում են նույն անունն ունեցող արտաքին հաստատուններից:

```
const z = 10;
function displayZ(){
    const z = 20;
    {
        const z = 30;
        console.log("Block:", z);   // 30
    }
    console.log("Function:", z);    // 20
}
displayZ();
console.log("Global:", z);  // 10
```

Չհայտարարված փոփոխականներ

Եթե մենք ֆունկցիայում փոփոխական սահմանելիս չօգտագործենք վերը նշված բանալինային բառերը, կամ այսպես կոչված օպերատորները (var, let, const), ապա այդպիսի փոփոխականը կլինի գլոբալ: Օրինակ `

```
function bar(){
   foo = "25";
}
bar();
console.log(foo);   // 25
```

Թեև foo փոփոխականը չի սահմանվում որևէ ֆունկցիայից դուրս, այնուամենայնիվ այն հասանելի(տեսանելի) է նաև ֆունկցիայի սահմաներից դուրս: Որպեսզի ամեն բան պարզ լինի, եկեք դիտարկենք նաև հակառակ դեպքը, այսինքն այն դեպքը, երբ օգտագործվել է օպերատորներից որևէ մեկը:

```
function bar(){
   var foo = "25";
}
bar();
console.log(foo);   // Սխալ
```

strict mode

Ֆունկցիաների գլոբալ փոփոխականներ սահմանելը կարող է հանգեցնել պոտենցիալ սխալների: Այդպիսի սխալներից խուսափելու համար օգտագործենք խիստ ռեժիմ՝ այսպես կոչված strict mode:

```
"use strict";
function bar(){
   foo = "25";
}
bar();
console.log(foo);
```

Այս դեպքում որպես արդյունք ստանում ենք հետևյա արտահայությունը՝

SyntaxError: Unexpected identifier,

ինչը նշանակում է, որ foo փոփոխականը չի սահմանվել:

strict mode սահմանելու երկու ճանապարհ կա.

- ավելացնենք "use strict" արտահայտությունը JavaScript կոդի սկզբում, ապա strict mode-ը կկիրառվի ամբողջ ծրագրային կոդի համար:

- ավելացնենք "use strict" արտահայտությունը ֆունկցիայի մարմնի սկզբին, ապա strict mode-ը կկիռարվի միայն այդ ֆունկցիայի համար:

Կապակցում և IIFE ֆունկցիան

Կապակցումը(closure) իրենցից ներկայացնում են կառույցներ(կոնստրուկցիաներ), երբ ֆունկցիան ստեղծում է տեսանելիության տիրույթը, հաշվի առնելով իր բանալինային բառերը, նույնիս այն դեպքում, երբ ինքն իրագործվում է իր տեսանելության տիրույթից դուրս, այսինքն արտաքին տեսանելիության տիրույթում:

Կապակցումը տեխնիկապես ներառում է երեք բաղադրիչ.

- արտաքին ֆունկցիա, որը որոշակի տեսանելիության տիրույթ է սահմանում և որոշակի փոփոխականներ սահմանվում են հենց այդ տիրույթում:
- փոփոխականներ (արտաքին տեսանելիության տիրույթ), որոնք սահմանվում են արտաքին ֆունկցիայում
- ներկառուցված ֆունկցիա, որն օգտագործում է այս փոփոխականները

```
function outer(){      // արտաքին ֆունկցի
   var n;              // անորոշ փոփոխական
   return inner(){     // ներածման ֆունկցիա
      // գործողություններ n  փոփոխականով    }
}
```

Մտածեք ամենապարզ օրինակների մասին,

```
function outer(){
   let x = 5;
   function inner(){
      x++;
      console.log(x);
   };
   return inner;
}
let fn = outer();    // fn = inner, քանի որ արտաքին ֆունկցիան վերադարձնում է inner ֆունկցիան
// կանչվում է inner ներքին ֆունկցիան
fn();   // 6
```

```
fn();   // 7
fn();   // 8
```

Այստեղ outer ֆունկցիան սահմանում է այն շրջանակները, որոնցում սահմանվում են inner ներքին ֆունկցիան և x փոփոխականը: x փոփոխականը inner ֆունկցիայի համար տեսանելիության տիրույթ: inner ֆունկցիայի մեջ մենք ավելացնում ենք x փոփոխականը և թողարկում դրա արժեքը վահանակին: Վերջում, outer ֆունկցիան վերադարձնում է inner ֆունկցիան:

Այնուհետև, կանչենք outer ֆունկցիան:

```
let fn = outer();
```

Քանի որ outer ֆունկցիան վերադարձնում է inner ֆունկցիան, ապա fn-ի փոփոխականը հղումով կանցնի inner ֆունկցիայ: Միևնույն ժամանակ, այս ֆունկցիան հիշում է իր միջավայրը, այսինքն արտաքին փոփոխական x-ը:

Արդյունքում մենք իրականում կանչում ենք Inner ֆունկցիան երեք անգամ, և մենք տեսնում ենք, որ x-ը, որը սահմանվում է inner ֆունկցիայի սահմաններից դուրս, ավելացվում է.

```
fn();   // 6
fn();   // 7
fn();   // 8
```

Այսինքն, չնայած այն հանգամանքին, որ x փոփոխականը սահմանվում է inner ֆունկցիայի սահմաններից դուրս, այս ֆունկցիան իր տեսանելիության տեսնում է և կարող է օգտագործել այն, չնայած այն հանգամանքին, որ այն կանչվում է outer ֆունկցիայից դուրս, որտեղ որ սահմանվել է այն: Սրանում էլ կայանում է կապակցման էությունը:

Դիտարկենք մեկ այլ օրինակ.

```
function multiply(n){
   var x = n;
   return function(m){ return x * m;};
}
var fn1 = multiply(5);
var result1 = fn1(6);  // 30
console.log(result1);  // 30

var fn2= multiply(4);
var result2 = fn2(6);  // 24
console.log(result2);  // 24
```

Այսպիսով, այստեղ multiply() ֆունկցիայի արժեքը կանչվում է ներքին ֆունկցիայի կողմից: Ներքին ֆունկցիան՝

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

տեսնում է այն տիրույթը որտեղ սահմանվել է x փոփոխականի արժեքը:
Արդյունքում, երբ կանչվում է բազմապատկման ֆունկցիան, սահմանվում է fn1 փոփոխական և տեղի է ունենում կապակցում, այսինքն, այն միավորում է երկու բան՝ ֆունկցիա և միջավայր(տիրույթ), որի սահմաններում գործում է տվյալ ֆունկցիան: Տեսանելիության տիրույթը բաղկացած է ցանկացած լոկալ փոփոխականից, որը multiply ֆունկցիայի շրջանակում էր կապի ստեղծման ընթացքում:
Այսինքն, fn1 կապակցումը պարունակում է ներքին ֆունկցիա՝
function(m){ return x * m;}; և կապի ստեղծման ժամանակ գոյություն ունեցող x փոփոխականը:
Երբ ստեղծում ենք երկու փոխկապակցում. fn1 և fn2, յուրաքանչյուր կապակցման համար ստեղծվում է իր առանձին միջավայրը:
Կարևոր չէ, որ շփոթված լինեն պարամետրերով, փոխկապակցման սահմանման ժամանակ:

```
var fn1 = multiply(5);
```

multiply ֆունկցիայի n պարամետրին տրվում է 5 արժեքը:
Ներքին ֆունկցիա կանչելով՝

```
var result1 = fn1(6);
```

Ներքին function(m){ return x * m;}; ֆունկցիայի m պարամետրին տրվում է 6 արժեքը:
Մենք նաև կարող ենք օգտագործել մեկ այլ տարբերակ՝ փոխկապակցման համար.

```
function multiply(n){
   var x = n;
   return function(m){ return x * m;};
}
var result = multiply(5)(6); // 30
console.log(result);
```

Ինքնականչվող ֆունկցիաներ

Սովորաբար ֆունկցիայի սահմանումը առանձնացվում է իր կանչից. Նախ՝ սահմանում ենք ֆունկցիան, հետո կանչում այն: Բայց դա անհրաժեշտ չէ: Մենք կարող ենք նաև ստեղծել այնպիսի ֆունկցիաներ, որոնք կանչվում են ինքնաբերաբար: Նմանատիպ ֆունկցիաները կոչվում են Անմիջապես Կանչված Ֆունկցիոնալ Արտահայտություն (IIFE):

```
(function(){
   console.log("Привет мир");
}());
(function (n){
   var result = 1;
   for(var i=1; i<=n; i++)
      result *=i;
   console.log("Факториал числа " + n + " равен " + result);
}(4));
```

Նմանատիպ ֆունկցիաները ավարտվում են փակագծերով, և ֆունկցիան սահմանվելուց հետո պարամետրերը փոխանցվում են հենց այդ փակագծեր:

Մոդուլի կառուցվածքը

Մոդուլի կառուցվածքի հիմքում ընկած է փոխկապակցումը, և բաղկացած է երկու բաղադրիչներից. Արտաքին ֆունկցիա, որը սահմանում է բառապաշարային տիրույթ, այսինքն օգտագործված փոփոխականների ցանկը, և վերադարձվող ներքին ֆունկցիաների հավաքածու, որոնք հասանելի են այդ միջավայրում:
Այժմ սահմանենք ամենապարզ մոդուլը:

```
let foo = (function(){
   let obj = {greeting: "hello"};
   return {
      display: function(){
         console.log(obj.greeting);
      }
   }
})();
foo.display();  // hello
```

Այստեղ սահմանվում է foo փոփոխական, որը ներկայացնում է անանուն ֆունկցիայի արդյունքը: Այս ֆունկցիայի մեջ որոշվում է obj օբյեկտի որոշ տվյալներ:
Անանուն ֆունկցիան ինքնին վերադարձնում է այն օբյեկտը, որը սահմանում է display ֆունկցիան: Վերադարձված օբյեկտը սահմանում է հանրային API, որի միջոցով մենք կարող ենք մուտք գործել մոդուլ և օգտագործել նրանում առկա տվյալները:

```
    return {
        display: function(){
            console.log(obj.greeting);
        }
```

Այս կառուցվածքը հնարավորություն է ընձեռում համախմբել որոշակի տվյալներ ֆունկցիա- մոդուլներում, իսկ տվյալ մոդուլ մուտք ունենալու համար ստեղծել հատուկ API վերադարձվող ներքին ֆունկցիայի միջոցով:
Դիտարկենք մի փոքր ավելի բարդ օրինակ.

```
let calculator = (function(){
    let data = { number: 0};
    return {
        sum: function(n){
            data.number += n;
        },
        subtract: function(n){
            data.number -= n;
        },
        display: function(){
            console.log("Result: ", data.number);
        }
    }
})();
calculator.sum(10);
calculator.sum(3);
calculator.display();   // Result: 13
calculator.subtract(4);
calculator.display();   // Result: 9
```

Այս մոդուլը պարզունակ հաշվիչ է, որն իրականացնում է երեք գործողություններ՝ արդյունքի ավելացում, նվազեցում և արդյունքի վերադարձ:
Բոլոր տվյալները ներկառուցված են data օբյեկտում, որը պահում է գործողության արդյունքը.

Բոլոր գործողությունները ներկայացված են երեք վերադարձի ֆունկցիաներով՝ sum(գումարում), subtract(հանում) և display(ներկայացում): Այս ֆունկցիաների միջոցով մենք կարող ենք վերահսկել հաշվիչի արդյունքը դրսից:

Ռեկուրսիվ ֆունկցիաներ

Ֆունկցիաների մեջ առանձին-առանձին կարելի է տեսնել ռեկուրսիվ ֆունկցիաներ: Այս ֆունկցիաների էությունը այն է, որ ֆունկցիան իր ներսում կանչում է ինքն իրեն: Այժմ եկեք դիտարկենք մի ֆունկցիա, որը կհաշվի տրված թվի ֆակտորիալը:

```
function getFactorial(n){
   if (n === 1){
      return 1;
   }
   else{
            return n * getFactorial(n - 1);
   } }
var result = getFactorial(4);
console.log(result); // 24
```

getFactorial() ֆունկցիան վերադարձնում է 1 արժեքը, եթե n փոփոխականի արժեքը 1 է, հակառակ դեպքում արդյունքը կրկին վերադարձնում է getFactorial ֆունկցիային, ապա բազմապատկում այն n-1 արժեքով: Օրինակ, երբ փոխանցենք 4 թիվը, մենք կունենանք հետևյալ կանչերի շղթան.

```
var result = 4 * getFactorial(3);
var result = 4 * 3 * getFactorial(2);
var result = 4 * 3 * 2 * getFactorial(1);
var result = 4 * 3 * 2 * 1; // 24
```

Այժմ դիտարկենք մեկ այլ օրինակ՝ Ֆիբոնաչիի թվերի սահմանումը.

```
function getFibonachi(n)
{
   if (n == 0){
      return 0;
   }
   if (n == 1){
      return 1;
   }
```

```
    else{
        return getFibonachi(n - 1) + getFibonachi(n - 2);
    }
}
var result = getFibonachi(8); //21
console.log(result); // 21
```

Վերափոխվող ֆունկցիաներ

Ֆունկցիաները ունակ են փոխելու իրենց վարքագիծը: Վերափոխումը տեղի է ունենում անանուն ֆունկցիա նշանակելու միջոցով, որը կանչվում է այնպես, ինչպես վերափոխված ֆունկցիան:

```
function display(){
    console.log("Доброе утро");
    display = function(){
        console.log("Добрый день");
    }
}
display(); // Доброе утро
display(); // Добрый день
```

Երբ առաջին անգամ ակտիվացվում է ֆունկցիան, գործում է ֆունկցիայի օպերատորների հիմնական բլոկը, մասնավորապես այս դեպքում ցուցադրվում է "Доброе утро" հաղորդագրությունը: Առաջին անգամ աշխատում է display ֆունկցիան, իսկ հետո տեղի է ունենում նրա վերափոխումը: Հետևաբար, ֆունկցիայի հետագա բոլոր կանչերի ժամանակ կաշխատի ֆունկցիայի վերանայված նոր տարբերակը, իսկ վահանակի վրա կցուցադրվի "Добрый день" հաղորդագրությունը:
Բայց ֆունկցիան վերաիմաստավորելու ժամանակ անհրաժեշտ է հաշվի առնել որոշ նրբություններ: Մասնավորապես, մենք կփորձենք հղում տալ ֆունկցիայի փոփոխականին և այս փոփոխության միջոցով կանչել ֆունկցիան:

```
function display(){
    console.log("Доброе утро");
    display = function(){
        console.log("Добрый день");
```

```
    }
}
// Ֆունկցիայի հղումը տալ նախքանի վերափոխումը
var displayMessage = display;
display(); // Доброе утро
display(); // Добрый день
displayMessage(); // Доброе утро
displayMessage(); // Доброе утро
```

Այստեղ, displayMessage փոփոխականը հղում է կատարում display ֆունկցիային՝ նախքան այն կվերահիմաստավորրվի: Հետևաբար, կանչելով displayMessage()-ը, կանչվում է display ֆունկցիայի չվերափոխված տարբերակը:
Բայց ենթադրենք, մենք սահմանում ենք displayMessage փոփոխական display ֆունկցիայի վերափոխումից հետո:

```
display(); // Доброе утро
display(); // Добрый день
var displayMessage = display;
displayMessage(); // Добрый день
displayMessage(); // Добрый день
```

Այս դեպքում displayMessage փոփոխականը կցուցադրի display ֆունկցիայի վերափոխված տարբերակը:

Hoisting

Hoisting-ը իրենից ներկայացնում է փոփոխականներ մուտքագրման գործընթաց, նախքան փոփոխականների սահմանումը: Թերևս այս հայեցակարգը մի փոքր տարօրինակ է թվում, բայց այն կապված է JavaScript-ի կոմպիլյատորի աշխատանքի հետ:
Կոդի կոմպիլիացիան տեղի է ունենում երկու մուտքերով: Առաջին մուտքում կոմպիլիատորը ստանում է բոլոր հայտարարված փոփոխականները, և բոլորը ID-ները: Այնուամենայնիվ, ոչ մի կոդը չի կատարվում և մեթոդներ չեն կանչվում:
Այդ ամենը կատարվում է երկրորդ մուտքի ժամանակ: Նույնիսկ եթե փոփոխականը սահմանվում է նախքան օգտագործումը, ոչ մի սխալ չի առաջանում, քանի որ արդեն

իսկ առաջին մուտքի ժամանակ կամպիլիատորը ճանաչում է բոլոր փոփոխականները:
Այսինքն, կարծես թե կոդը հասկանում է ծրագրի վերնամասում արդեն իսկ սահմանված փոփոխականները և ֆունկցիաները, նախքան դրանց օգտագործումը:
Փոփոխականները որոնք ընկնում են Hoisting-ի տակ ունեն undefined(անորոշ) արժեքներ:
Այժմ եկեք դիտարկենք հետևյալ պարզագույն օրինակը՝

```
console.log(foo);
```

Սրա իրագործումը կհանգեցնի սխալի՝ **ReferenceError: foo is not defined**
Ավելացնենք փոփոխականի սահմանումը:

```
console.log(foo);   // undefined
var foo = "Tom";
```

Այս դեպքում վահանակի վրա ցուցադրվում է undefined արժեքը: Առաջին մուտքի ընթացքում կոմպիլյատորը ճանաչում է foo փոփոխականի գոյությունը: Այն սահմանվում է undefined(անորոշ): Երկրորդ մուտքի ժամանակ կանչվում է console.log(foo) մեթոդը:
Դիտարկենք մեկ այլ օրինակ՝

```
var c = a * b;
var a = 7;
var b = 3;
console.log(c); // NaN
```

Այստեղ նույն իրավիճակն է: a և b փոփոխականները օգտագործվում են մինչև դրանց սահմանումը: Լռելյայն, դրանց վերագրվում է undefined(անորոշ) արժեքներ: Եվ եթե մենք բազմապատկենք undefined(անորոշ)-ը undefined(անորոշ)-ով, մենք չենք ստանա և ոչ մի թիվ, այսինքն՝ (NaN):
Նույնը վերաբերում է ֆունկցիաների օգտագործմանը: Մենք կարող ենք նախ կանչել ֆունկցիան, այնուհետև սահմանել այն.

```
display();
 function display(){
    console.log("Hello Hoisting");
}
```

Այստեղ display ֆունկցիան սահուն կաշխատի, չնայած այն հանգամանքին, որ այն սահմանվում է կանչի ավարտից հետո: Սակայն այստեղ կա նրբություն՝ սա չպետք է շփոթել այն դեպքի հետ երբ ֆունկցիան որպես փոփոխական է սահմանվում՝

```
display();
var display = function (){
   console.log("Hello Hoisting");
}
```

Այս դեպքում մենք ստանում ենք սխալ. TypeError: display is not a function.

Առաջին մուտքի ժամանակ կոմպիլյատորը կստանա նաև display փոփոխական և նրան կվերագրի undefined(անորոշ) արժեք: Երկրորդ մուտքի ժամանակ, երբ մենք պետք է կանչենք այն ֆունկցիան, որին հղված է այս փոփոխականը, կոմպիլիատորը կտեսնի, որ ոչինչ չկա կանչելու: display փոփոխականը դեռևս undefined(անորոշ) է:

Եվ որպես պատասխան ստանում ենք սխալ:

Հետևաբար փոփոխականներ և ֆունկցիաներ սահմանելիս պետք է ուշադրություն դարձնել այնպիսի հանգամանքների ինչպիսին է Hoisting-ը:

Փոխանցվող պարամետրերը ըստ արժեքի և հղման

Փոխանցվող պարամետրերը ըստ արժեքի

Տողերը, թվերը, տրամաբանական արժեքները ֆունկցիային փոխանցվում են արժեքով: Այլ խոսքերով, երբ ֆունկցիային արժեք է անցնում, այդ ֆունկցիան ստանում է տվյալ արժեքի պատճենը: Դիտարկենք այդ ամենը գործնական օրինակով՝

```
function change(x){
   x = 2 * x;
   console.log("x in change:", x);
}
var n = 10;
console.log("n before change:", n); // n before change: 10
change(n);                          // x in change: 20
console.log("n after change:", n);  // n after change: 10
```

change ֆունկցիան ստանում է որոշակի թիվ և կրկնապատկում է այն: Երբ կանչվում է change ֆունկցիան, նրան է փոխանցվում n թիվի արժեքը:

Այնուամենայնիվ, ֆունկցիան կանչելուց հետո մենք տեսնում ենք, որ n թիվը չի փոխվել, չնայած ֆունկցիայի մեջ պարամետրական արժեքի ավելացում է տեղի ունեցել: Քանի որ երբ կանչվում է change ֆունկցիան, այն ստանում է n փոփոխականի արժեքի պատճենը: Հետևաբար ցանկացած փոփոխություն, որը կատարվում է պատճենի հետ, ոչ մի կերպ չի ազդում n փոփոխականի արժեքի վրա:

Փոխանցվող պարամետրերը ըստ հղման

Օբյեկտները և զանգվածները փոխանցվում են ըստ հղման: Այսինքն, ֆունկցիան ստանում է օբյեկտը կամ զանգվածը, այլ ոչ թե դրանց պատճենը:

```
function change(user){
   user.name = "Tom";
}
var bob ={
   name: "Bob"
};
console.log("before change:", bob.name);    // Bob
change(bob);
console.log("after change:", bob.name);     // Tom
```

Այս դեպքում change ֆունկցիան ստանում է օբյեկտը և փոխում է նրա տվյալներից name-ն: Արդյունքում մենք կտեսնենք, որ ֆունկցիայի կանչից հետո փոխվել է bob օբյեկտի բնօրինակը, որը փոխանցվել է ֆունկցիային:

Այնուամենայնիվ, եթե փորձենք ամբողջովին վերականգնել օբյեկտը կամ զանգվածը, ապա սկզբնական արժեքը չի փոխվի:

```
function change(user){
   // օբյեկտի ամբողջական վերականգնում
   user= {
      name:"Tom"
   };
}
 var bob ={
   name: "Bob"
};
console.log("before change:", bob.name);    // Bob
change(bob);
console.log("after change:", bob.name);     // Bob
```

Նույնը վերաբերում է զանգվածներին՝

```
function change(array){
```

```
    array[0] = 8;
}
function changeFull(array){
    array = [9, 8, 7];
}
 var numbers = [1, 2, 3];
 console.log("before change:", numbers);     // [1, 2, 3]
change(numbers);
console.log("after change:", numbers);        // [8, 2, 3]
changeFull(numbers);
console.log("after changeFull:", numbers);  // [8, 2, 3]
```

Ուղորդված ֆունկցիաներ

Ուղորդված ֆունկցիաները իրենցից ներկայացնում են նորմալ ֆունկցիաների կրճատ տարբերակը: Ուղորդված ֆունկցիաները ձևավորվում են օգտագործելով arrow(=>) նշանը, որից հետո փակագծերի մեջ գրվում է ֆունկցիայի պարամետրերը, իսկ հետո ֆունկցիայի մարմինը: Օրինակ `

```
let sum = (x, y) => x + y;
let a = sum(4, 5);      // 9
let b = sum(10, 5);    // 15
```

Այս դեպքում (x, y) => x + y ֆունկցիան իրականացնում է երկու թվերի գումարում և սահմանում է sum փոփոխական: Ֆունկցիային տրվում է երկու պարամետր` x և y: Ֆունկցիայի մարմնում այս պարամետրերի արժեքները գումարվում են:
Եվ քանի որ (=>)- ից հետո ստացվում է կոնկրետ արժեք, որը թվերի գումարն է, որպես արդյունք ֆունկցիան վերադարձնում է այդ արժեքը:
Եվ մենք կարող ենք կանչել այս ֆունկցիան sum փոփոխականով և ստանալ դրա արդյունքները a և b փոփոխականների համար:
Դիտարկենք մեկ այլ դեպք, երբ (=>)-ից հետո կա որևէ գործողություն կամ արժեքը վերադարձնող արտահայտություն: Այդ արտահայտությունը, որը ոչինչ չի վերադարձնում և պարզապես կատարում է որոշ գործողություններ, կարող է օգտագործվել որպես ֆունկցիայի մարմին:

```
let sum = (x, y) => console.log(x + y);
sum(4, 5);      // 9
sum(10, 5);    // 15
```

Այս դեպքում console.log() ֆունկցիան ոչինչ չի վերադարձնում, և հետևաբար գումարի ֆունկցան նույնպես որևէ արդյունք չի վերադարձնում:
Եթե ֆունկցիան վերցնում է մեկ պարամետր, ապա փակագծեր կարող ենք չօգտագործել:

```
var square = n => n * n;
console.log(square(5));     // 25
console.log(square(6));     // 36
console.log(square(-7));    // 49
```

Եթե ֆունկցիայի մարմինը իրենից ներկայացնում է արտահայտությունների շարք, ապա այն գրվում է ձևավոր փակագծերում:

```
var square = n => {
   let result = n * n;
   return result;
}
console.log(square(5));     // 25
```

Այս դեպքում ֆունկցիայից արդյունքը վերադարձնելու համար օգտագործվում է **return** ստանդարտ օպերատորը:
Հատուկ ուշադրություն դարձնենք այն դեպքին, երբ ուղորդող ֆունկցիան վերադարձնում է օբյեկտ:

```
let user = (userName, userAge) => ({name: userName, age: userAge});
let tom = user("Tom", 34);
let bob = user("Bob", 25);
console.log(tom.name, tom.age);     // "Tom", 34
console.log(bob.name, bob.age);     // "Bob", 25
```

Օբյեկտը նույնպես գրվում է ձևավոր փակագծերում, բայց բացի այդ փակագծերից դրվում են նաև սովորական փակագծեր:
Եթե ուղորդող ֆունկցիան չի ստանում որևէ պարամետր, ապա դրվում են դատարկ փակագծեր:

```
var hello = ()=> console.log("Hello World");
hello();    // Hello World
hello();    // Hello World
```

Գլուխ 4. Օբյեկտ-կողմնորոշված ծրագրավորում

Օբյեկտ-կողմնորոշված ծրագրավորում (ՕԿԾ) (անգլ.՝ Object-oriented programming (OOP)), ծրագրավորման մոտեցում, որի գաղափարական հիմք են հանդիսանում Օբյեկտ և Դաս (class) հասկացությունները: OOP-ի բոլոր առավելություններից կարող ենք օգտվել JavaScript-ով աշխատելիս: JavaScript-ում, օբյեկտի վրա հիմնված ծրագրավորումն ունի որոշ առանձնահատկություններ: Ծանոթանանք դրանց հետ:

Օբեկտներ

Անցյալ թեմաներում մենք աշխատել ենք պարզունակ տվյալների հետ՝ թվեր, տողեր, բայց տվյալները միշտ չէ, որ ներկայացվում են պարզունակ տեսքով: Օրինակ, եթե մեր ծրագրում մենք պետք է նկարագրենք անունը, ազգանուն, տարիքը, հասակ և այլ տվյալներ ունեցող որևէ անձի, ուրեմն, բնականաբար, մենք չենք կարող այդ ամենը ներկայացնել որպես զանգված կամ տող: Մարդուն պատշաճ կերպով նկարագրելու համար մենք պետք է մի քանի տողեր կամ թվեր: Այս առումով մարդը հանդես կգա որպես մեկ ամբողջական համալիր կառույց, որը կունենա առանձին հատկություններ՝ տարիք, հասակ, անուն, ազգանուն և այլն:

JavaScript-ում նմանատիպ կառույցները կոչվում են օբյեկտներ: Յուրաքանչյուր օբյեկտ իր մեջ պահպանում է այնպիսի հատկություններ, ինչպիսիք են վարքը բնութագրող մեթոդները:

Նոր օբյեկտի ստեղծումը

Նոր օբյեկ ստեղծելու մի քանի եղանակներ կան:

Առաջին եղանակը Object բանալինային բառի օգտագործումն է:

```
var user = new Object();
```

Այս դեպքում օբյեկտը user-ն է: Այն սահմանվում է այնպես, ինչպես ցանկացած հերթական փոփոխական, օգտագործելով **var** բանալի բառ:

new Object() արտահայտությունը իրենից ներկայացնում է կանչ ուղված կոնստրուկտոր ֆունկցիանյին, որը ստեղծում է նոր օբյեկտ: Կոնստրուկտորին կանչելու համար օգտագործվում է **new** օպերատորը: Կոնստրուկտորի կանչը իրականում նման է սովորական ֆունկցիայի կանչին:

Օբյեկտի ստեղծման երկրորդ եղանակը հետևյալն է՝

```
var user = {};
```

Այսօր երկրորդ մեթոդը ավելի տարածված է:

Օբյեկտի հատկությունները

Օբյեկտը ստեղծելուց հետո մենք կարող ենք սահմանել նրա համար հատկություններ: Օբյեկտի հատկությունները սահմանելու համար պետք է նշեք օբյեկտի անունը, կետ (.), հետո օբյեկտի հատկության անվանումը, այնուհետև տալիս ենք տվյալ հատկության համապատասխան արժեքը:

```
var user = {};
user.name = "Tom";
user.age = 26;
```

Այս դեպքում հայտարարվում է երկու հատկություններ, name և age համապատասխան նշանակված արժեքներով: Դրանից հետո մենք կարող ենք օգտագործել այդ հատկությունները, օրինակ՝ ցույց տանք դրանց արժեքները վահանակին:

```
console.log(user.name);
console.log(user.age);
```

Մենք կարող ենք հատկությունները սահմանել նաև օբյեկտի սահմանման ժամանակ:

```
var user = {
    name: "Tom",
  age: 26
};
```

Ինչպես տեսնում ենք այս օրինակում հատկությանը արժեքը վերագրվում է (:)-ի օգնությամբ, իսկ հատկությունները թվարկելիս դրվում է (,) նախորդ օրինակի (;)-ի փոխարեն:

Բացի վերը նշված եղանակներից, գոյություն ունի հատկությունների որոշման առավել կարճ մեթոդ:

```
var name = "Tom";
var age = 34;
var user = {name, age};
console.log(user.name);    // Tom
console.log(user.age);     // 34
```

Այս դեպքում փոփոխականների անունները նաև օբյեկտի հատկությունների անուններն են: Եվ այս կերպ մենք կարող ենք ստեղծել ավելի բարդ կանստրուկցիաներ(կառույցներ).

```
var name = "Tom";
var age = 34;
```

```
var user = {name, age};
 var teacher = {user, course: "JavaScript"};
console.log(teacher.user);      // {name: "Tom", age: 34}
console.log(teacher.course);    // JavaScript
```

Օբյեկտի մեթոդները

Օբյեկտի մեթոդները որոշում են այն վարքագիծը կամ այն գործողությունները, որոնք սահմանվում են: Մեթոդները ֆունկցիաներ են: Օրինակ, եկեք դիտարկենք մի մեթոդ, որը ցույց կտա անձի անունը և տարիքը.

```
var user = {};
user.name = "Tom";
user.age = 26;
user.display = function(){
   console.log(user.name);
   console.log(user.age);
};
// մեթոդի կանչ
user.display();
```

Ինչպես ֆունկցիաներ դեպքում, այնպես էլ մեթոդները նախ սահմանվում են, հետո կանչվում:

Բացի այդ, մեթոդները կարող են որոշվել հենց օբյեկտի սահմանման ժամանակ.

```
var user = {
   name: "Tom",
   age: 26,
   display: function(){
           console.log(this.name);
      console.log(this.age);
   }
};
```

Ինչպես հատկությունների դեպքում, մեթոդները ևս ներկայացնում են հղումներ դեպի ֆունկցիա (:)-ի օգնությամբ:

Տվյալ օբյեկտի հատկություններին կամ մեթոդներին անդրադառնալու համար պետք է օգտագործել **this** բանալինային բառը: Դա նշանակում է հղում դեպի ընթացիկ օբյեկտ:

Վերը նշված գործողությունը կարող ենք կատարել նաև առավել կարճ եղանակով, երբ բացակայում են (:)-ը և **function** բանալինային բառը:

```
var user = {
     name: "Tom",
    age: 26,
    display(){
         console.log(this.name, this.age);
    },
    move(place){
        console.log(this.name, "goes to", place);
    }
};
user.display(); // Tom 26
user.move("the shop");  // Tom goes to the shop
```

Զանգվածների հայտարարում

Օբյեկտների հատկությունները և մեթոդները սահմանելու համար գոյություն ունի ևս մեկ այլընտրանքային եղանակ, դա զանգվածների օգտագործումն է:

```
var user = {};
user["name"] = "Tom";
user["age"] = 26;
user["display"] = function(){
    console.log(user.name);
    console.log(user.age);
};
 // մեթոդի կանչ
user["display"]();
```

Յուրաքանչյուր հատկության կամ մեթոդի անվանումը գրվում է քառակուսի փակագծերում, ապա տրվում է դրանց համապատասխան արժեքներ: Օրինակ **user["age"] = 26**:

Այս հատկություններն ու մեթոդները օգտագործել համար գոյություն ունի 2 գրելաձև՝ **(user.name)** կամ **user["name"]**:

Տողերը որպես հատկություններ և մեթոդներ

Պետք է նաև նշել, որ օբյեկտի հատկությունների և մեթոդների անունները մշտապես ներկայացնում են տողերի տեսքով: Այսինքն, մենք կարող ենք նախորդ ծրագիրի օբյեկտի սահմանումը վերաշարադրել հետևյալ կերպ.

```
var user = {
   "name": "Tom",
   "age": 26,
   "display": function(){
          console.log(user.name);
      console.log(user.age);
   }
};
// մեթոդի կանչ
user.display();
```

Մի կողմից, երկու սահմանումների միջև տարբերություն չկա: Մյուս կողմից, կան դեպքեր, երբ տողային տեսքով ներկայացումը կարող է օգնել: Օրինակ, եթե հատկության անվանումը բաղկացած է երկու բառից:

```
var user = {
   name: "Tom",
   age: 26,
   "full name": "Tom Johns",
   "display info": function(){
      console.log(user.name);
      console.log(user.age);
   }
};
console.log(user["full name"]);
user["display info"]();
```

Միայն այս դեպքում, հղում անելով նույն հատկություններին և մեթոդներին, մենք պետք է օգտագործենք զանգվածի գրելաձև:

Հատկությունների հեռացում

Մինչ այս մենք դիտարկում էինք, թե ինչպես կարող ենք դինամիկորեն ավելացնել օբյեկտի նոր հատկություններ: Այնուամենայնիվ, մենք կարող ենք նաև delete

օպերատորի օգտագործմամբ հեռացնել տվյալ օբյեկտի հատկությունները և մեթոդները: Եվ ինչպես ավելացնելու դեպքում, այս դեպքում ևս մենք կարող ենք հեռացնել հատկությունները երկու եղանակով:
Առաջին եղանակում օգտագործում ենք կետ (.)

```
delete объект.свойство
```

Երկրորդեղանակում օգտագործում ենք զանգվածի շարահյուսություն []:

```
delete объект["свойство"]
```

Դիտարկենք հատկության հեռացման մի օրինակ:

```
var user = {};
user.name = "Tom";
user.age = 26;
user.display = function(){
        console.log(user.name);
    console.log(user.age);
};
console.log(user.name); // Tom
delete user.name; // հատկության հեռացում
// այլընտրանքային գրելաձև
// delete user["name"];
console.log(user.name); // undefined(անորոշ)
```

Ջնջելուց հետո հատկությունը անորոշ կլինի: Հետևաբար, երբ փորձենք որևէ գործողություն կատարել հեռացված հատկության հետ, ծրագիրը մեզ կվերադարձնի undefined արժեքը:

Օբյեկտներում ներկառուցված օբյեկտներ և զանգվածներ

Որոշ օբյեկտներ կարող են պարունակել այլ օբյեկտներ: Օրինակ, կա մի օբյեկտ, որի մի շարք հատկություններ կարելի է առանձնացնել:
Ենթադրենք այդ հատկություններից մեկը մայրաքաղաքն է: Բայց մայրաքաղաքում մենք կարող ենք նաև առանձնացնել իր տվյալները, օրինակ՝ անունը, բնակչությունը, հիմնադրման տարեթիվը և այլն.

```
var country = {
    name: "Германия",
    language: "немецкий",
```

```
    capital:{
        name: "Берлин",
        population: 3375000,
        year: 1237
    }
};
console.log("Столица: " + country.capital.name); // Берлин
console.log("Население: " + country["capital"]["population"]); // 3375000
console.log("Год основания: " + country.capital["year"]); // 1237
```

Նման ներկառուցված օբյեկտների հատկություններին հասնելու համար մենք կարող ենք օգտագործել ստանդարտ կետային նշումը.

```
country.capital.name
```

Կամ պարզապես դրանք ներկայացնել որպես զանգվածի տարեր:

```
country["capital"]["population"]
```

Նույնիսկ գոյություն ունի խառը ներկայացման տեսակ:

```
country.capital["year"]
```

Հատկությունները նաև կարող են օգտագործվել զանգվածներում, ինչպես նաև ներկառուցված այլ օբյեկտների զանգվածներում:

```
var country = {
    name: "Швейцария",
    languages: ["немецкий", "французский", "итальянский"],
    capital:{
            name: "Берн",
        population: 126598
    },
    cities: [
        { name: "Цюрих", population: 378884},
        { name: "Женева", population: 188634},
        { name: "Базель", population: 164937}
    ]
};

// country.languages-ից բոլոր տարրերի թողարկում
document.write("<h3>Официальные языки Швейцарии</h3>");
```

```
for(var i=0; i < country.languages.length; i++)
   document.write(country.languages[i] + "<br/>");
// country.cities-ից բոլոր տարրերի թողարկում
document.write("<h3>Города Швейцарии</h3>");
for(var i=0; i < country.cities.length; i++)
   document.write(country.cities[i].name + "<br/>");
```

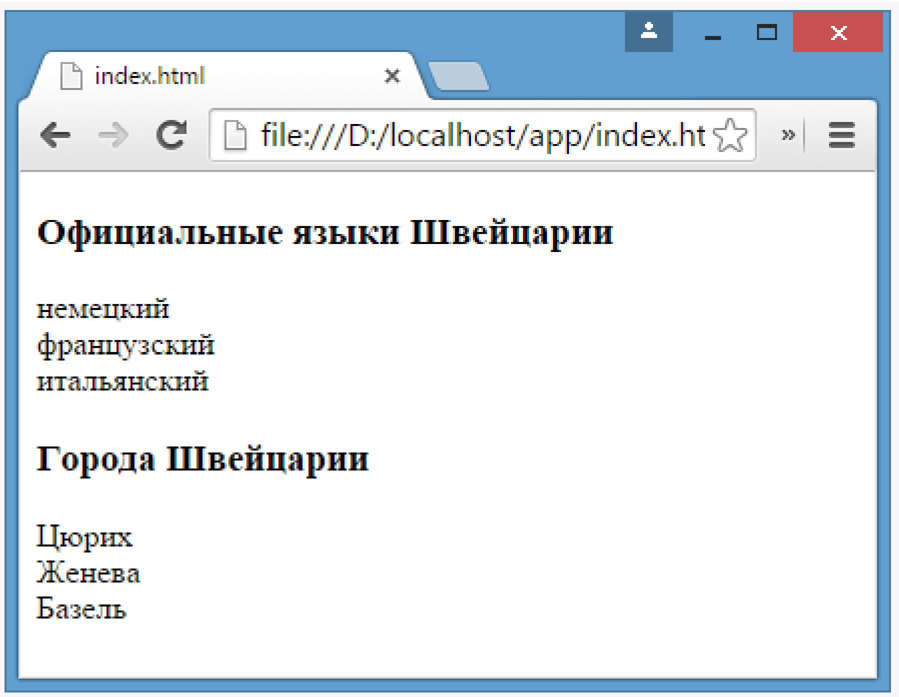
country օբյեկտը պարունակում է languages հատկությունը, որը ներկայացվում է միաչափ զանգվածի տեսքով, ինչպես նաև cities հատկությունը, որը իր մեջ ներառում է նմանատիպ օբյեկտների զանգված:

Այդ զանգվածների հետ մենք կարող ենք աշխատել այնպես, ինչպես մյուս բոլորի զանգվածների հետ, օրինակ՝ օգտագործել for ցիկլային օպերատորը:

Զանգվածի յուրաքանչյուր տարր իրենից ներկայացնում է առանձին օբյեկտ, որին մենք կարող ենք անդրադառնալ՝ ելնելով նրա հատկություններից և մեթոդներից:

```
country.cities[i].name
```

Արդյունքում մեր բրաուզերին կտեսնենք հետևյալ բովանդակությամբ զանգվածը՝



Հասանելիության ստուգում և մեթոդների ու հատկությունների որոնումը

Երբ դինամիկ կերպով որոշում է օբյեկտի նոր հատկությունները և մեթոդները, նախքան դրանց օգտագործումը կարևոր է ստուգել, արդյոք այդ մեթոդներն ու հատկությունները հասանել են այն տիրույթում որտեղ ցանկանում ենք դրանք օգտագործել: Դրա համար javascript-ում օգտագործվում է in օպերատորը:

```
var user = {};
user.name = "Tom";
user.age = 26;
user.display = function(){
        console.log(user.name);
    console.log(user.age);
};
var hasNameProp = "name" in user;
console.log(hasNameProp); // true – user-ում կա name հատկություն
var hasWeightProp = "weight" in user;
console.log(hasWeightProp); // false – user-ում չկա weight հատկություն կամ մեթոդ
```

in օպերատորը ունի հետևյալ գրելաձևը. *"հատկություն / մեթոդ" in օբյեկտ* :
("")-ի մեջ գրվում է հատկության կամ մեթոդի անունը, իսկ in օպերատորից հետո՝ օբյեկտի անվանումը: Եթե միևնույն անունով հատկություն կամ մեթոդ կա, ապա օպերատորը վերադարձնում է true(ճշմարիտ) արժեքը: Եթե ոչ, ապա false(կեղծ) արժեքը:
Այլընտրանքային եղանակն այն է, երբ վերադարձվող արժեքը undefined(անորոշ) է: Այսինքն հատկությունը կամ մեթոդը undefined(անորոշ) է:

```
var hasNameProp = user.name!==undefined;
console.log(hasNameProp); // true
var hasWeightProp = user.weight!==undefined;
console.log(hasWeightProp); // false
```

Քանի որ օբյեկտները իրենցից ներկայացնում են Object տիպը, ինչը նշանակում է, որ այդ բոլոր օբյեկտները կարող են օգտվել բոլոր մեթոդներից ու հատկություններից, ինչպես նաև hasOwnProperty() մեթոդից, որը սահմանվում է Object տիպում:

```
var hasNameProp = user.hasOwnProperty('name');
console.log(hasNameProp); // true
var hasDisplayProp = user.hasOwnProperty('display');
console.log(hasDisplayProp); // true
var hasWeightProp = user.hasOwnProperty('weight');
console.log(hasWeightProp); // false
```

Որոնելի հատկություն և մեթոդ

Ցիկլի օգնությամբ մենք կարող ենք թվարկեց օբյեկտները, ինչպես զանգվածի դեպքում արեցինք: Արդյունքում կստանանք բոլոր հատկություններն ու մեթոդները և դրանց համապատասխան արժեքները:

```
var user = {};
user.name = "Tom";
user.age = 26;
user.display = function(){

    console.log(user.name);
    console.log(user.age);
};
for(var key in user) {
    console.log(key + " : " + user[key]);
}
```

Մեր բրաուզերում կստանանք հետևյալ արդյունքը՝

```
name : Tom
age : 26
display : function (){

        console.log(user.name);
        console.log(user.age);
}
```

Օբյեկտները ֆունկցիաներում

Ֆունկցիաները կարող են վերադարձնել արժեքները: Բայց այդ արժեքները միշտ չէ, որ իրենցից ներկայացնում են պարզունակ տվյալների, թվեր, տողեր: Դրանք կարող են լինել նաև բարդ օբյեկտներ:
Օրինակ, ներկայացնենք **user** օբյեկտի ստեղծումը առանձին ֆունկցիայում:

```
function createUser(pName, pAge) {
    return {
        name: pName,
        age: pAge,
        displayInfo: function() {
            document.write("Имя: " + this.name + " возраст: " + this.age + "<br/>");
        }
    };
};
var tom = createUser("Tom", 26);
tom.displayInfo();
var alice = createUser("Alice", 24);
alice.displayInfo();
```

Այստեղ **createUser()** ֆունկցիան վերցնում է **pName** և **pAge** արժեքները, այնուհետև նրանց համար ստեղծում է նոր օբյեկտ, որն էլ հանդիսանում է վերադարձվող արդյունք:
Օբյեկտի ստեղծման առավելությունն այն է, որ հետագայում մենք կարող ենք ստեղծել տարբեր արժեքներով նույն տիպի մի քանի օբյեկտներ:
Բացի այդ օբյեկտը կարող է ներկայացվել որպես ֆունկցիայի պարամետր:

```
function createUser(pName, pAge) {
    return {
        name: pName,
        age: pAge,
        displayInfo: function() {
            document.write("Имя: " + this.name + " возраст: " + this.age + "<br/>");
        },
```

```
        driveCar: function(car){
            document.write(this.name + " ведет машину " + car.name + "<br/>");
        }
    };
};
function createCar(mName, mYear){
    return{
        name: mName,
        year: mYear
    };
};
var tom = createUser("Том", 26);
tom.displayInfo();
var bently = createCar("Бентли", 2004);
tom.driveCar(bently);
```

Այստեղ օգտագործվում է երկու ֆունկցիա՝ հաճախորդների և մեքենների օբյեկտը ստեղծելու համար: **user** օբյեկտի **driveCar()** մեթոդը որպես պարամետր է վերցնում car օբյեկտը:

Արդյունքում մեր բրաուզերում կցուցադրվի հետևյալը.

```
Имя: Том возраст: 26
Том ведет машину Бентли
```

Գրականության ցանկ

1. http://ww1.javascript.org/
2. https://www.codecademy.com/learn/introduction-to-javascript
3. https://metanit.com/web/javascript/4.5.php
4. https://learn.javascript.ru/