

# ԱՐՄԵՆ ՀԱԿՈԲԻ ՄԱՐԶԻՆՅԱՆ

Նպիրվում եմ ինֆորմատիկայի ուսուցչութուն՝  
ՄԱՐԴԱ ՀԱՐՈՒԹՅՈՒՆՅԱՆԻՆ

{}

# JavaScript

## ԵՐԱԳՐԱԿՈՐՄԱՆ ԼԵՇՎԻ ՏԱՐՐԵՐԸ

Խմբագրությամբ՝ մանկավարժական գիտությունների  
թեկնածու, դոցենտ

ՍԱՄՎԵԼ ՄԻՍԱԿԻ ԱՍԱՏՐՅԱՆԻ



ՈՍԿԱՆ ԵՐԵՎԱՆՑԻ  
ՏՊԱԳՐԱՏՈՒՆ - ՀՐԱՏԱՐԱԿՉՈՒԹՅՈՒՆ

Երևան - 2018

**Երաշխավորվում է Profit Development Company  
ծրագրավորման ընկերությանը կից՝ Պրոֆայթի  
ուսումնական կենտրոնի կողմից:**

## **ՏԵԽՆԻԿԱԿԱՆ ԽՄԲԱԳԻՐ՝ ԱՆԱՀԻՄ ԱՐԱՄԺԱՆ**

### **ԱՐՄԵՆ ՀԱԿՈԲԻ ՄԱՐԶԻՆՅԱՆ JAVA SCRIPT ԾՐԱԳՐԱՎՈՐՄԱՆ ԼԵԶՎԻ ՏԱՐՐԵՐԸ**

Ուսումնական ձեռնարկը նվիրված է ծրագրավորման ոլորտում տարածված լեզուներից մեկի՝ JavaScript ծրագրավորման լեզվի տարրերին: Ներկայացված են՝ գաղափար ալգորիթմների մասին, JavaScript լեզվի տարրերը, jQuery գրադարանը, օբյեկտ կողմնորոշված ծրագրավորման սկզբունքները (OOP), ինչպես նաև, ասինքրոն և գրաֆիկական ծրագրավորման մեթոդները: Տրված են գործնական առաջադրանքներ և դրանց կատարման մեթոդները: Ուսումնական ձեռնարկը հասցեագրված է ծրագրավորողներին, մանկավարժներին, ուսանողներին և բոլոր նրանց, ովքեր հետաքրքրվում են ծրագրավորմամբ:

Ձեռնարկը հրատարակվել է Profit Development Company ընկերության աշակեցությամբ:

---

© Մարգինյան Ա.Հ., 2018

© Profit Development Company

# ԲՈՎԱՆԴԱԿՈՒԹՅՈՒՆ

ԴԱՍ 1 - ԱԼԳՈՐԻԹՄՆԵՐ.....	6
ԴԱՍ 2 - JAVASCRIPT ԾՐԱԳՐԱԿՈՐՍԱՆ ԼԵԶՎԻ ՏԱՐՐԵՐԸ.....	18
ԴԱՍ 3 - DOCUMENT OBJECT MODEL .....	35
ԴԱՍ 4 - ՊԱՅՄԱՆԻ ՕՊԵՐԱՏՈՐ .....	44
ԴԱՍ 5 - ԼՈԿԱԼ ԵՎ ԳԼՈԲԱԼ ՓՈՓՈԽԱԿԱՆՆԵՐ .....	54
ԴԱՍ 6 - ՑԻԿԼԻ ՕՊԵՐԱՏՈՐՆԵՐ .....	61
ԴԱՍ 7 - ՉԱՇԿԱՑՆԵՐ .....	75
ԴԱՍ 8 - ՉԱՇԿԱՑՆԵՐԸ DOM - ՈՒՄ .....	85
ԴԱՍ 9 - ՏՈՂԱՅԻՆ ՏԻՊ .....	94
ԴԱՍ 10 - ՖՈՒՆԿՑԻՈՆԱԼ ԾՐԱԳՐԱԿՈՐՈՒՄ .....	102
ԴԱՍ 11 - ՌԵԿՈՒՐՍԻԱ .....	111
ԴԱՍ 12 - ԱՆԱԼՈՒ ՖՈՒՆԿՑԻԱՆԵՐ .....	123
ԴԱՍ 13 - ԴԻՆԱՄԻԿ ԾՐԱԳՐԱԿՈՐՈՒՄ .....	137
ԴԱՍ 14: JQUERY ՆԵՐԱԾՈՒԹՅՈՒՆ .....	148
ԴԱՍ 15: JQUERY ԿԱՌԱՎԱՐՈՒՄ .....	156
ԴԱՍ 16: CALCULATOR -ի ՄՇԱԿՈՒՄ .....	166
ԴԱՍ 17 - ԴԻՆԱՄԻԿ ԾՐԱԳՐԱԿՈՐՈՒՄ JQUERY .....	173
ԴԱՍ 18 - JQUERY ԵՎ DOM .....	178
ԴԱՍ 19 - EVENT ԾՐԱԳՐԱԿՈՐՈՒՄ .....	187
ԴԱՍ 20 - ԽԱՂԵՐԻ ԾՐԱԳՐԱԿՈՐՈՒՄ .....	196
ԴԱՍ 21 - ՕԲՅԵԿՏ-ԿՈՂՄՆՈՐՈՇՎԱԾ ԾՐԱԳՐԱԿՈՐՈՒՄ .....	209
ԴԱՍ 22 - ԿԼԱՍՆԵՐ .....	215

ԴԱՍ 23 - ԿԼԱՍՆԵՐԻ ՀԱՏԿՈՒԹՅՈՒՆՆԵՐԸ (ՄԱՍ 1) .....	234
ԴԱՍ 24 - ԿԼԱՍՆԵՐԻ ՀԱՏԿՈՒԹՅՈՒՆՆԵՐԸ (ՄԱՍ 2) .....	234
ԴԱՍ 25 - ԿԼԱՍՆԵՐՆ  ECMASCRIPT6 - ՈՒՄ .....	245
ԴԱՍ 26 - ԱՄԱՍԹՎԱՅԻՆ ՖՈՒՆԿՑԻԱՆԵՐ  LOCATION.....	251
ԴԱՍ 27 - ԺԱՌԱՆԳՈՒՄ.....	257
ԴԱՍ 28 - ԱՄԻՆԵՐՈՒ ԾՐԱԳՐԱԿՈՂՈՒՄ .....	268
ԴԱՍ 29 - WEB STORAGE JSON ԵՎ AJAX.....	279
ԴԱՍ 30 - ԳՐԱՖԻԿԱ ԵՎ CANVAS .....	293
ԴԱՍ 31 - ՓՂԻ ՀԱՐՎԱԾԸ ԶԻՆԿՈՐԻՆ .....	310
ԴԱՍ 32 - ԽԱՂԵՐԻ ԾՐԱԳՐԱԿՈՂՈՒՄ 1.....	319
ԴԱՍ 33 - ԽԱՂԵՐԻ ԾՐԱԳՐԱԿՈՂՈՒՄ 2.....	336
ԴԱՍ 34 - ԿՈԴԻ ՎԵՐԼՈՒԾՈՒԹՅՈՒՆ (DEBUGGING).....	346
ԴԱՍ 35 - ՏՎՅԱԼՆԵՐԻ ԿԱՌՈՒՑՎԱԹՔՆԵՐ ԵՎ ՀԵՐՁԵՐԻ ՏԵՍՈՒԹՅՈՒՆ .....	363
ԴԱՍ 36 - ԻՏԵՐԱՏՈՐՆԵՐ ԵՎ ԳԵՆԵՐԱՏՈՐՆԵՐ .....	372
ԴԱՍ 37 - SCOPE ԵՎ CLOSURE.....	378
ԴԱՍ 38 - ՀԱՐՑԱՋՐՈՒՅՑՆԵՐԻ 20 ՀԱՐՑԵՐ ԵՎ ՊԱՏԱՍԽԱՆՆԵՐ .....	392

# ՆԱԽԱԲԱՆԻ ՓՈԽԱՐԵՆ

## ԻՆՉՈ՞ F ՍՈՎՈՐԵԼ ՅԱՎԱՏՐԻ

Ծրագրավորման ոլորտում հայտնի StackOverflow կայքը տարիներ շարունակ իրականացնում է հարցումներ աշխարհի տարբեր ծայրերում գտնվող ծրագրավորողների շրջանում՝ պարզելու համար՝ ո՞րն է ներկայումս ամենապահանջված ծրագրավորման լեզուն: Ամփոփելով վերջին տարիների արդյունքները՝ նկատում ենք, որ հարցմանը մասնակցած ծրագրավորողների մեծ մասը նշել են JavaScript ծրագրավորման լեզուն: Այստեղից էլ կարելի է ենթադրել, որ JavaScript -ն աշխարհում ամենալայն տարածում ունեցող ծրագրավորման լեզուներից մեկն է:

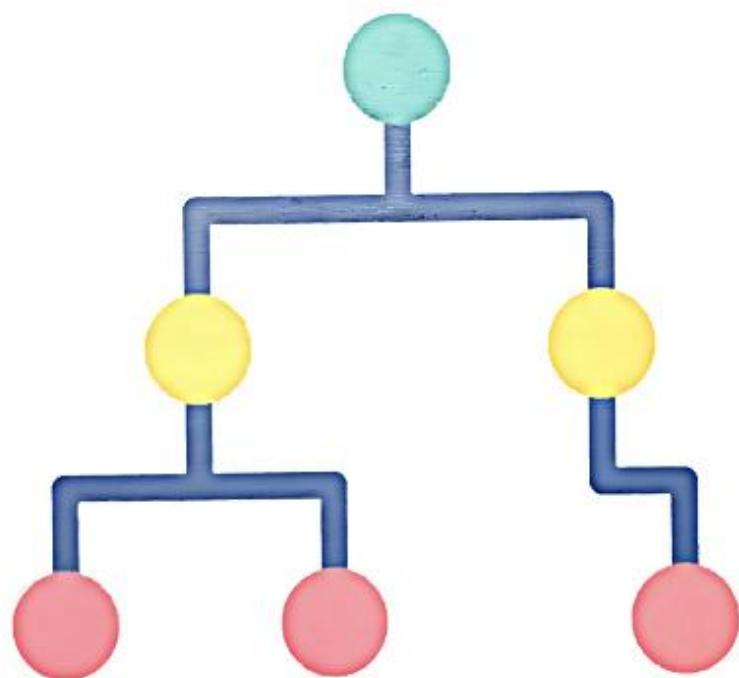
Վաղուց անցել են այն ժամանակները, երբ JavaScript -ը հանդիսանում էր պարզ սկրիպտավորման լեզու: Այսօր JavaScript -ը վեր ծրագրավորման գործընթացում կիրառվող ամենաառանցքային լեզուներից մեկն է, որի գրադարանները և framework-ները (օր.՝ *jQuery, Angular, React.js, Vue.js, Node.js* և այլն) թույլ են տալիս մշակել ամբողջական կայքեր՝ առանց որևէ այլ տեխնոլոգիայի կիրառման: Ի ուրախություն ծրագրավորողների՝ JavaScript -ը չի սահմանափակվում միայն վեր տիրույթով: JavaScript-ով նախագծված ժամանակակից համակարգերը հեշտությամբ կարելի է վերածել mobile application-ների:

Այսպիսով՝ սովորելով միայն JavaScript ծրագրավորման լեզուն, հնարավոր է ստեղծել ոչ միայն ամբողջական կայքեր, այլև խաղեր, mobile հավելվածներ և այլն:

JavaScript-ը ապագայի ծրագրավորման լեզու է...

# ԴԱՍ 1

## ԱԼԳՈՐԻԹՄՆԵՐ



# ԴԱՍ 1. ԳԱՂԱՓԱՐ ԱԼԳՈՐԻԹՄԻ ԵՎ ԾՐԱԳՐԱՎՈՐՄԱՆ ԼԵԶՈՒՆԵՐԻ ՄԱՍԻՆ

## 1.1 Ալգորիթմի սահմանումը

Տեղեկատվական տեխնոլոգիաների արդի դարաշրջանում անհնար է պատկերացնել կյանքն առանց համակարգիչների և ինտերնետի: Ժամանակակից թվային սարքերը (համակարգիչներ, հեռախոսներ և այլն) աչքի են ընկնում իրենց հզորությամբ, արագագործությամբ և տեխնոլոգիական հիմքով: Բացի գերազանց շախմատ խաղալուց, բազմաթիվ վեբ կայքերի հետ աշխատելուց, համակարգիչները կարողանում են կատարել տարատեսակ բարդության հաշվարկներ՝ նվազագույն ժամանակահատվածում: Ուշագրավ է այն, որ ժամանակակից աշխարհում այսօր լրջորեն ուսումնասիրվում են արհեստական բանականության հիմնախնդիրները, որոնք թույլ են տալիս, որ, օրինակ, շախմատ խաղալու ընթացքում, կախված իր հետ խաղացողի կատարած քայլերից, համակարգիչը կարող է անընդհատ զարգացնել իր խաղային տեխնիկան: Չնայած այս ամենին, համակարգիչներն իրենցից ներկայացնում են սարքավորումներ, որոնք չունեն սեփական ինտելեկտ, մտածելու և տրամաբանելու կարողություն: Համակարգիչները կատարում են միայն այն գործողությունները, որոնք ծրագրավորողներն են «սովորեցրել» իրենց: Հենց այդ նպատակով ստեղծվել են այսպես կոչված **համակարգչային լեզուներ**, որոնք նախատեսված են համակարգչին իր կատարելիք գործողությունները սովորեցնելու համար: Արհեստական այդ լեզուները նման են բնական լեզուներին, ունեն որոշակի սիմվոլներ, բառեր, որոնք կիրառվում են այդ լեզվում, ինչպես նաև, որոշակի քերականական առանձնահատկություններ, որոնք ձևավորում են լեզվի նախատիպը: Ընդունված է այդպիսի լեզուներին անվանել ծրագրավորման **լեզուներ**, որոշ դեպքերում նաև

**ալգորիթմական լեզուներ:** Ալգորիթմական լեզու ասելով՝ կարող ենք հասկանալ արհեստական լեզու, որը թույլ է տալիս համակարգչին ծանոթացնել գործողությունների այն բազմության հետ, որոնք նա պետք է կատարի: Ընդհանրապես տվյալ խնդրի լուծման համար կատարվող գործողությունների հաջորդականությունն ընդունված է անվանել **ալգորիթմ**: Քանի որ թվային սարքերը չունեն տրամաբանություն, ապա բնական է, որ գործողությունները, որոնք պետք է կատարի համակարգչը, նրան անհրաժեշտ է ներկայացնել առավելագույնս պարզ քայլերի հաջորդականության տեսքով, որտեղ քայլերը կլինեն հստակ նկարագրված, ճիշտ հերթականությամբ, իսկ երկիմաստ և ավելորդ քայլերը կրացանվեն:

**ՍԱՀՄԱՆՈՒՄ:** Գործողությունների կարգավորված հաջորդականությունը կոչվում է ալգորիթմ, եթե վերջավոր քայլերի արդյունքում այն հանգեցնում է տրված խնդրի լուծմանը:

## 1.2. JavaScript ծրագրավորման լեզուն:

Մենք արդեն գիտենք, որ համակարգում տեղի ունեցող բոլոր գործընթացները մշակված են ծրագրավորման կամ մեքենայական լեզուների օգնությամբ: Այժմ ծանոթանանք ծրագրավորման լեզուներից մեկին՝ JavaScript -ին: Գույք դուք նախկինում չեք լսել JavaScript -ի մասին, բայց վստահաբար այն կիրառել եք, կամ օգտվել եք նրա աշխատանքի արդյունքներից, քանի որ JavaScript - ն այն ծրագրավորման լեզուն է, որը կառավարում է ժամանակակից վեբ կայքերը: JavaScript -ը կարող է կառավարել կայքի տեսքը (html, css), ինչպես նաև որոշել՝ ինչ պետք է տեղի ունենա, երբ օգտատերը սեղմում է որոշակի դաշտերում, տեղաշարժում մկնիկը, մուտքագրում որոշակի տեքստ և այլն: JavaScript -ի հնարավորություններից են օգտվում կայքերի մեծ մասը, ինչպիսիք են՝ google, facebook, instagram, նրանք օգտագործում են JavaScript -ը՝ կայքի արագագործությունն ապահովելու նպատակով: Facebook -ում

դուք նամակ եք գրում ձեր ընկերոջը, like -ում (հավանել) տարրեր գրառումներ, instagram -ում scroll -ի արդյունքում դուք տեսնում եք ավելի շատ նկարներ, կայքերում գրանցվելիս պարտադրվում է մուտքագրել ճիշտ էլ-փոստի հասցե, կամ առավել երկար ծածկագիր: Բոլոր այս գործընթացները կառավարվում են JavaScript -ի կողմից:

Ծրագրավորման JavaScript լեզուն աշխարհում ամենալայն տարածում ունեցող լեզուներից մեկն է: Այն հեշտ է յուրացնել, ունի ճկուն կառուցվածք, ինչի արդյունքում հնարավոր է լուծել տարրեր խնդիրներ՝ նվազագույն ծավալով հրամաններ մուտքագրելով:

JavaScript ծրագրավորման լեզվի ստեղծողը համարվում է Բրենդան Այկը, ով ներկայումս հանդիսանում է նաև Mozilla Corporation ընկերության գործադիր տնօրենը:

Ժամանակակից JavaScript -ն իր մեջ ներդրված ֆունկցիոնալ և օբյեկտ



Բրենդան Այկ

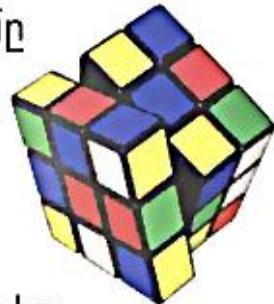
կողմնորոշված ծրագրավորման մեխանիզմների օգնությամբ հնարավորություն է տալիս գրել բարձր մակարդակի կոդ, ստեղծել տարատեսակ խաղեր և ապահովել կայքի ֆունկցիոնալությունը: JavaScript -ով կոդ գրելու համար անհրաժեշտ է ունենալ տեքստային խմբագրիչ (օր.' SublimeText, Notepad++ և այլն), ինչպես նաև Browser (օր.' Google Chrome, Mozilla Firefox, Opera և այլն): Յուրաքանչյուր Browser կարողանում է JavaScript -ով գրված կոդը կարդալ տող առ տող և այն վերածել անհրաժեշտ արդյունքի: Ժամանակակից ծրագրավորման մեջ լրջորեն զարգանում է JavaScript -ի դերը: Այսօր արդեն, JavaScript -ը համարվում է ոչ միայն FrontEnd ծրագրավորման լեզու, այլ նաև Backend' հաշվի առնելով Node.js տեխնոլոգիայի գոյությունը: JavaScript -ը կիրառվում է նաև Mobile ծրագրավորման մեջ: IONIC Framework -ի, cordova.js -ի կամ React Native -ի կիրառմամբ, հնարավոր է նախագծել համակարգեր, որոնք հեշտությամբ կարելի է վերածել Android,

iOS, Windows Phone և մի շարք այլ օպերացիոն համակարգերում կիրառելի ծրագրերի: Մինչ JavaScript -ին առավել մանրամասն ծանոթանալը, շարունակենք ուսումնասիրել ալգորիթմ գաղափարը՝ ծանոթանալով նրա մի շարք կարևոր հատկություններին:

### 1.3 Ալգորիթմների հատկությունները

Արդեն գիտենք, որ ալգորիթմը գործողությունների կարգավորված հաջորդականություն է, որը վերջավոր քայլերի արդյունքում կարող է հանգեցնել սպասված արդյունքի:

Ալգորիթմների տեսությունը ժամանակակից ծրագրավորման ամենաառանցքային ուղղություններից մեկն է: Ընդհանրապես ալգորիթմները բնութագրվում են ստորև բերված հատկություններով.



- Վերջավորություն՝** ալգորիթմները պետք է կազմված լինեն վերջավոր քայլերից,
- Դիսկրետություն՝** ալգորիթմի յուրաքանչյուր քայլ պետք է ունենա իր հստակ նշանակությունը և չինի երկիմաստ,
- Կարգավորվածություն՝** մեծ նշանակություն ունի քայլերի կատարման հերթականությունը, քանի որ այս հատկության խախտման դեպքում հնարավոր է, որ անհրաժեշտ խնդիրը չլուծվի,
- Արդյունավետություն՝** ալգորիթմը պետք է կարողանա լուծել դրված խնդիրը ամենաարդյունավետ տարրերակով:

Ալգորիթմների օրինակների հանդիպում ենք նաև առօրյայում:  
**Օրինակ 1: Դիտարկենք սուրճ պատրաստելու պարզագույն խնդիրը**

Այս խնդրի լուծման համար մեզ անհրաժեշտ գործիքներն են՝ սրճեփ, ջուր, սուրճ, շաքարավազ և գազօջախ: Խնդրի լուծման ալգորիթմը կարելի է ներկայացնել հետևյալ պարզ քայլերի հաջորդականության միջոցով

1. Վերցնել սրճեփը
2. ավելացնել անհրաժեշտ քանակով  
ջուր
3. ավելացնել սուրճ
4. ավելացնել շաքարավազ
5. միացնել գազօջախը
6. սրճեփը դնել գազօջախի կրակի  
վրա
7. որոշ ժամանակ անց վերցնել սրճեփը գազօջախի  
վրայից



Ինչպես նկատեցինք, սուրճ պատրաստելու համար կարելի է կատարել 7 քայլ, այսինքն՝ առկա է ալգորիթմի վերջավորության հատկությունը: Ալգորիթմի յուրաքանչյուր քայլ ունի իր հստակ նշանակությունը և երկիմաստ չէ, հետևաբար առկա է ալգորիթմի դիսկրետությունը: Ալգորիթմի քայլերի հաջորդականությունը ճիշտ է ընտրված, հետևաբար այն կարգավորված է: Ալգորիթմը նաև արդյունավետ է:

Այժմ փորձենք շեշտադրումը կատարել ալգորիթմի կարգավորվածության վրա:

Փորձենք տեղերով փոխել քայլ5-ը և քայլ 1 -ը:

1. Միացնել գազօջախը
2. Ավելացնել անհրաժեշտ քանակով ջուր
3. Ավելացնել սուրճ
4. Ավելացնել շաքարավազ
5. Վերցնել սրճեփը
6. Սրճեփը դնել գազօջախի կրակի վրա
7. որոշ ժամանակ անց վերցնել սրճեփը գազօջախի  
վրայից

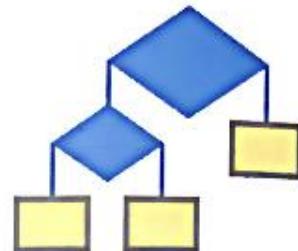
Դժվար չէ նկատել, որ քայլերի այս հաջորդականության դեպքում, երբ կարգավորվածությունը խախտված է, առաջադրված խնդիրը երբեք չի լուծվի: Հետևաբար ցանկացած ալգորիթմի կառուցման փուլում անհրաժեշտ է համոզվել, որ միաժամանակ առկա են բոլոր հատկությունները:



#### **1.4 Ալգորիթմների ներկայացման եղանակները**

Ժամանակակից ծրագրավորման ոլորտում հայտնի են ալգորիթմների ներկայացման մի շարք եղանակներ, սակայն դրանցից համեմատաբար ավելի պարզ կառուցվածք և լայն տարածում ունեն երկուար.

- 1. բառաբանաձևային եղանակ**
- 2. բլոկ սխեմաներ**



Մենք արդեն առիթ ունեցել ենք կիրառել ալգորիթմների ներկայացման բառաբանաձևային եղանակը: Այս մեթոդի հիմքում ընկած է ալգորիթմի աշխատանքի ներկայացումը տարբեր նախադասությունների միջոցով: Սովորաբար այս մեթոդը կիրառվում է տարբեր համակարգերի տեխնիկական բնութագրերում, սակայն կարող է կիրառվել նաև մի շարք այլ հանգամանքներում:

**Օրինակ 1. Ենթադրենք ունենք a,b թվեր և ցանկանում ենք գտնել՝ ո՞րն է դրանցից մեծագույնը:**

Դիտարկենք խնդրի լուծման ալգորիթմի ներկայացումը բառաբանաձևային մեթոդով

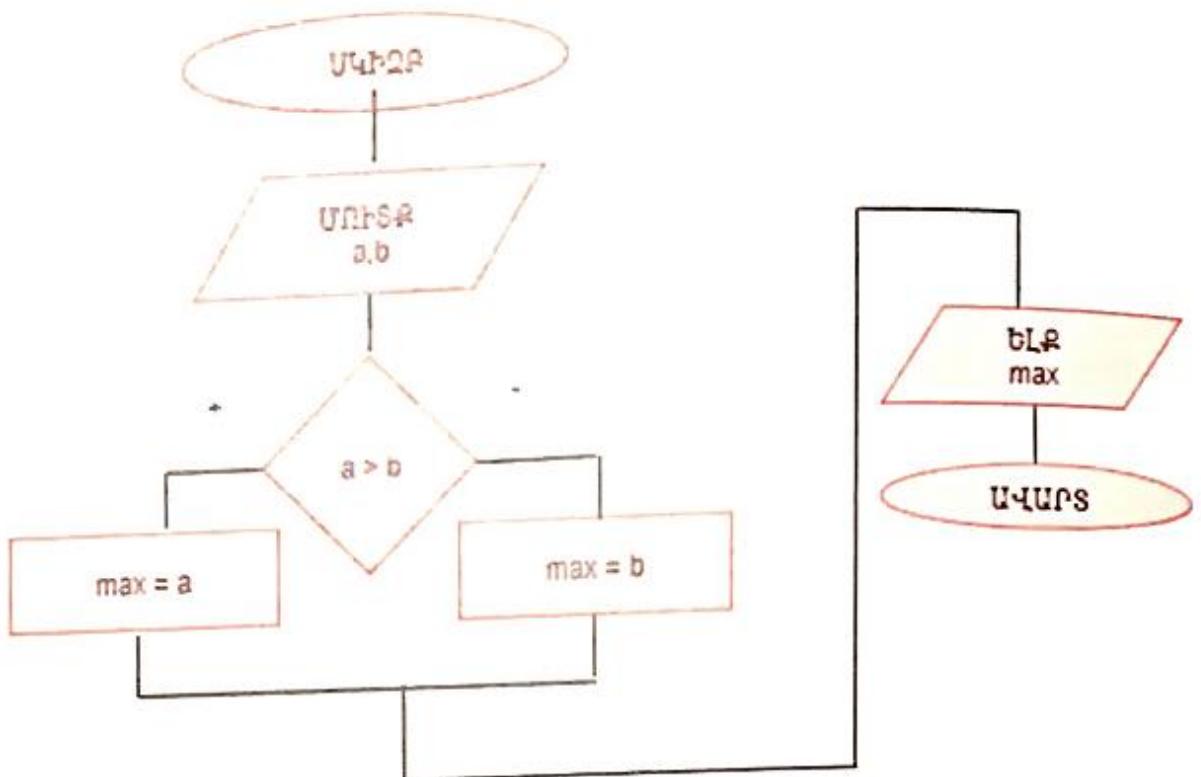
- 1. մուտքագրել a,b տարրերը**
- 2. ստուգել՝ արդյո՞ք  $a > b$ , եթե այո, անցնել քայլ 3-ին, եթե ոչ, անցնել քայլ 4 - ին**
- 3. մեծագույն տարրը a -ն է: Անցնել քայլ 5 -ին**
- 4. մեծագույն տարրը b -ն է**
- 5. ավարտ**

Այսպիսով՝ առաջին քայլում ցանկացած երկու ա և բ թվերը ընտրելով կհամոզվենք, որ 3-րդ կամ 4-րդ քայլում ստանում ենք խնդրի պատասխանը: Ըստ որում, եթե  $a = 8$ ,  $b = 4$ , ապա 2-րդ քայլում պարզում ենք, որ  $a > b$ , ուստի պետք է կատարվի քայլ 3-ը: Այս դեպքում արդեն ստանում ենք խնդրի լուծումը. մեծագույն տարրը  $a$ -ն է:

Ազորիթմսերի ներկայացման երկրորդ եղանակը բլոկ սխեման Ազորիթմսերի սխեման իրենից ներկայացնում է տարբեր է: Բլոկ սխեման իրենից ներկայացնում է տարբեր երկրաչափական պատկերներից (բլոկներ) կազմված գծանկար, որը թույլ է տալիս թվային ալգորիթմները ներկայացնել առավել պարզ, հասկանալի եղանակով: Ըստունված է բլոկ սխեմաներում պարզ, հասկանալի եղանակով: Կիրառել հետևյալ հիմնական երկրաչափական պատկերները.

	Էլիպսաձև բլոկ	Կիրառվում է ալգորիթմի սկիզբը և վերջը ցույց տալու նպատակով
	Գուգահեռագծի բլոկ	Կիրառվում է ալգորիթմում տվյալների մուտքեր, ինչպես նաև, պատասխանի ելքագրում իրականացնելու համար
	Չեղանկյան բլոկ	Կիրառվում է ալգորիթմներում տարբեր պայմաններ ստուգելու նպատակով
	Ուղղանկյան բլոկ	Կիրառվում է ալգորիթմներում պարզագույն գործողություններ կատարելու նպատակով

Այսպիսով՝ ներկայացնենք երկու թվերից մեծագույնը գտնելու խնդիրը բլոկ սխեմայի օգնությամբ: Ենթադրվում է, որ ա և ն հավասար չեն -ի:

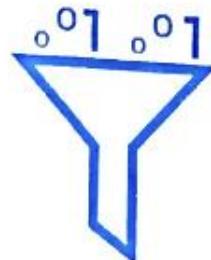


Ինչպես նկատեցինք՝ բլոկ սխեմայի սկիզբը էլիպսաձև բլոկն է, որը խորհրդանշում է ալգորիթմի աշխատանքի սկիզբը: Հաջորդ քայլում զուգահեռագծի բլոկում մուտքագրված են  $a, b$  թվերը: Մրանք այն թվերն են, որոնք պետք է համեմատել: Հաջորդ քայլին, շեղանկյան բլոկի օգնությամբ ստուգվում է պայմանական արտահայտություն՝  $a > b$ : Կախված  $a > b$  պայմանական արտահայտության արժեքից՝ շեղանկյան բլոկն առաջարկում է երկու ելք: Դրական ելքը (+) ցույց է տալիս այն գործողությունը, որը պետք է կատարվի, եթե  $a > b$  արտահայտությունը ճշմարիտ է: Բացասական ելքը (-) ցույց է տալիս հակառակ դեպքը: Բնական է, որ եթե օրինակ  $a = 8, b = 4$ , ապա  $a > b$  պայմանից ենելով՝ ալգորիթմը «կշարունակի աշխատել» դրական ուղղությամբ և կմտնի ուղղանկյան բլոկ, որում գրված է  $max = a$  արտահայտությունը: Ուղղանկյան բլոկից հետո, եթե  $max$  փոփոխականի մեջ պահպում է անհրաժեշտ արդյունքը բլոկ

սխեման իր «աշխատանքը» շարունակում է դեպի զուգահեռագծի բլոկ, որտեղ ելքագրվում է պատասխանը, այսինքն՝ a,b թվերից մեծագույնը՝ max -ը. Խնդիրը լուծված է, ուստի անհրաժեշտ է ավարտել ալգորիթմի աշխատանքը էլիպսաձև բլոկի օգնությամբ:

### 1.5 Որո՞շ հայտնի ալգորիթմների մասին

Ալգորիթմների տեսությունը ծրագրավորման ամենաառանցքային բաժիններից մեկն է: Ալգորիթմների տեսության շրջանակներում ցանկացած խնդիր կարելի է վերածել գործողությունների պարզ հաջորդականության: Գոյություն ունեն, խնդիրների լայն տարածում ունեցող խմբեր, որոնց համար, բոլոր ժամանակներում մշակվել են տարրեր ալգորիթմներ: Այդպիսի խնդիրներ են որոնման, տեսակավորման, պատահական թվերի գեներացման խնդիրները և այլն: Վեր կայքերում այսօր էլ կիրառվում են տարատեսակ որոնումների և տեսակավորման հետ կապված տարրեր ֆունկցիաներ: Օրինակ՝ օնլայն խանութներում երբեմն անհրաժեշտություն է առաջանում դասավորել ապրանքներն ըստ գների, ըստ արտադրման տարեթվերի և այլն, կամ որոնել նախընտրած ապրանքը: Առաջին հայացքից տեսակավորման խնդիրը պարզ է, սակայն, եթե գործ ունենք մեծաքանակ տվյալների հետ, անհրաժեշտություն է առաջանում ընտրել այնպիսի ալգորիթմ, որը լավագույնս կիրականացնի տեսակավորման գործընթացը:



Հայտնի տեսակավորման ալգորիթմներ են պղպջակների մեթոդը (bubble sort), ներմուծումներով տեսակավորման ալգորիթմը (insertion sort), ընտրություններով տեսակավորման ալգորիթմը (selection sort) և այլն:



Գոյություն ունեն այնպիսի ալգորիթմներ, որոնք կիրառվում են տվյալների բազմության մեջ որոնում իրականացնելիս: Օրինակ՝

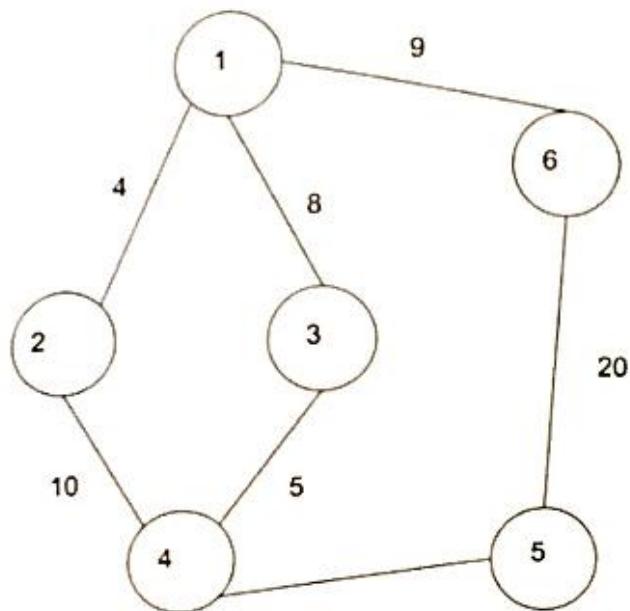
գծային որոնման մեթոդ (linear search), կիսման մեթոդ (binary search), էքսպոնենցիալ մեթոդ (exponential search), թռիչքային մեթոդ (jump search) և այլն:

Մեծ է ալգորիթմների նշանակությունը նաև տարատեսակ օպտիմալացման խնդիրներում: Օրինակ՝ Google Map -ը հնարավորություն է տալիս գտնել երկու աշխարհագրական դիրքերի միջև ամենակարճ ճանապարհը: Այս խնդրի փոքր-ինչ ավելի լայն տարբերակը հանդիսանում է **տրանսպորտային խնդիրը**, երբ հայտնի են  $N$  քաղաքների միջև հեռավորությունները և մատակարարը պետք է գտնի ամենակարճ և նվազագույն ծախսերով ճանապարհը, որով կարող է ապրանքները մատակարարել բոլոր  $N$  քաղաքները: Այսպիսով՝ անսահման է ալգորիթմների աշխարհը: Ալգորիթմներ, որոնք կիրառվում են ինչպես կենցաղում, գիտության մեջ, այնպես էլ ծրագրավորման ոլորտում:

## ՀԱՐՑԵՐ ԵՎ ԱՌԱՋԱԴՐԱՆՔՆԵՐ

1. Ի՞նչ է ալգորիթմը:
2. Որո՞նք են ալգորիթմի հատկությունները:
3. Ի՞նչ եղանակներով է հնարավոր ներկայացնել ալգորիթմները:
4. Ի՞նչ է ծրագրավորման լեզուն:
5. Ի՞նչ գիտեք JavaScript ծրագրավորման լեզվի մասին:
6. Ալգորիթմների տեսության ի՞նչ հայտնի խնդիրների եք ծանոթ:
7. Բերեք կենցաղային խնդրի օրինակ, որի լուծումը հնարավոր է ներկայացնել ալգորիթմի միջոցով:
8. Ստեղծեք ալգորիթմ և բլոկ սխեմա, որը a,b,c թվերից կարող է գտնել մեծագույնը:

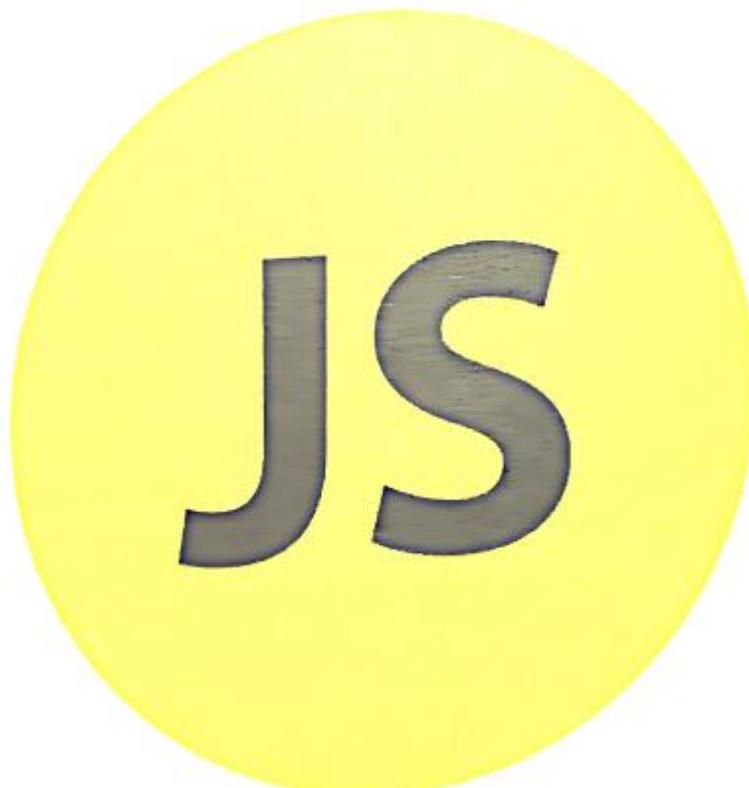
9. Ստորև բերված պատկերը ներկայացնում է հիվանդանոցում առկա 6 սենյակները և նրանց միացնող միջանցքների հեռավորությունը: Անհրաժեշտ է հասկանալ՝ որո՞նք են այն միջանցքները, որոնք փակելու դեպքում, հիվանդանոցի աշխատանքը չի դադարի, իսկ մնացյալ սենյակների միջև ընդհանուր հեռավորությունը կլինի նվազագույնը: Ի՞նչ ալգորիթմ կառաջարկեք առաջադրված խնդրի լուծման համար:



10. Ձեր տրամադրության տակ 3 և 5 լիտրանոց անոթներ կան և դուք գետից ջուր լցնելու և դատարկելու հնարավորություն ունեք: Նկարագրեք այն ալգորիթմը, որի կատարման արդյունքում գետից հնարավոր կլինի վերցնել 4 լիտր ջուր:

**ԴԱՍ 2**

**JAVASCRIPT  
ԵՐԱԳՐԱՎՈՐՄԱՆ  
ԼԵԶՎԻ SURFACE**



## ԴԱՍ 2 : JAVASCRIPT ԾՐԱԳՐԱՎՈՐՄԱՆ ԼԵԶՎԻ ԿԱՌՈՒՑՎԱԾՔԸ: ՓՈՓՈԽԱԿԱՆՆԵՐ ԵՎ ՏՎՅԱԼՆԵՐԻ ՏԻՊԵՐ

**2.1 JavaScript ծրագրավորման լեզվի միջավայրը**  
Ինչպես արդեն գիտենք, JavaScript ծրագրավորման լեզուն (այսուհետ JS) լայն կիրառություն ունի վեր ծրագրավորման մեջ: JavaScript ծրագրավորման լեզվի հրամանները html<sup>1</sup> ֆայլին ավելացնում ենք <script></script> տեղի օգնությամբ:

Օրինակ՝

```
<!DOCTYPE html>
<html>
<head>
    <title>JavaScript First Lesson</title>
</head>
<body>
<script>
    alert("Welcome to JavaScript");
</script>
</body>
</html>
```

Այս դեպքում, html ֆայլի բացմանը զուգընթաց, Էկրանին կհայտնվի փոքրիկ պատուհան, որում գրված կլինի Welcome to JavaScript տեքստը:

Հաճախ ծրագրավորման գործընթացում JS -ի կողերի ծավալը զգալիորեն մեծանում է: Նման իրավիճակներում JS -ը HTML ֆայլի ներսում ամբողջությամբ գրելը նպատակահարմար չէ: Այս իմաստով, անհրաժեշտություն է առաջանում JavaScript լեզվով գրված հրամանները պահպանել առանձին ֆայլերում: JavaScript

<sup>1</sup> HTML նշագրման լեզու է, որի օգնությամբ կառուցվում են վեր կայքերի մեծ մասը: HTML -ով գրված հրամանները browser -ը վերածում է մարդու կողմից առավել հեշտ ընթեռնելի փաստաթղթի:

ծրագրավորման լեզվով ստեղծվող ֆայլերն ունեն **js** ընդայնում: Մեր առաջին JS կոդը գոելու համար ստեղծենք թղթապանակ, որտեղ կլինի index.html ֆայլը, այնուհետև ստեղծենք js թղթապանակը, որտեղ կպահպանվեն JavaScript -ի ֆայլերը: Այդ թղթապանակի ներսում ստեղծենք script.js ֆայլը: Ըստունված է, որ JavaScript -ի ֆայլերը html -ին կցվում են body -ի վերջում, օրինակ՝

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



Այստեղ src հատկությունը ցոյց է տալիս js ֆայլի գտնվելու հասցեն: Պատահում է նաև, եթե script -ը ավելացվում է body -ից հետո, սակայն browser -ի կողմից այն կրկին ավտոմատ տեղափոխվում է body -ի վերջ:

Իսկ ինչո՞ւ ենք ավելացնում script -ը body -ի վերջում: JavaScript ծրագրավորման լեզուն, ինչպես գիտենք, կարող է կառավարել կայքի ամբողջ HTML հատվածը, ոհմել տարրեր HTML տեգերի (օր.' h1, p, input), փոխել դրանց պարունակությունը և այլն: Եթե JavaScript -ը էջին կցվում է ավելի շուտ, քան HTML էլեմենտներն են, ապա այն չի ունենում հասանելիություն այդ էլեմենտների նկատմամբ: Այս իմաստով HTML5 -ում կարելի է կիրառել **defer** հատկությունը, որը, անկախ ամեն ինչից, թույլ է տալիս,

որպեսզի script -ը աշխատի վերջում, երբ էջն ամբողջությամբ պատրաստ է.

```
<!DOCTYPE html>
<html>
<head>    <title></title>
</head>
<body>
<script src="js/script.js" defer></script>
<h1>Some Text</h1>
</body>
</html>
```

Այս դեպքում, չնայած նրան, որ h1 -ը գտնվում է script -ից հետո, միևնույն է script -ը կկարողանա ճանաչել և դիմել այդ տեգին: defer ատրիբուտի կիրառման փոխարեն խորհուրդ է տրվում մշտապես script -ը էջին կցել body -ի վերջում կամ body -ից հետո:

Այդպիսի հատկություն է նաև **async** -ը, որը թույլ է տալիս, որպեսզի script -ն աշխատի ասինքրոն, այսինքն՝ էջի բացմանը ընթացքում:

```
<script src="js/script.js" async></script>
```

Այսպիսով՝ defer -ի դեպքում script -ը աշխատում է ամենավերջում, երբ էջն արդեն պատրաստ է գործարկման, իսկ async -ի դեպքում script -ը սկսում է աշխատել էջի բեռնավորմանը (loading) գուգընթաց:

## 2.2 Ելքագրման օպերատորներ, մեկնաբանություններ

JavaScript ծրագրավորման լեզուն թույլ է տալիս էկրանին որոշակի տվյալներ արտածել: Արտածումը կարող է տեղի ունենալ ինչպես body -ում, այնպես էլ **console** պատուհանում, որը տեսնելու համար browser -ում անհրաժեշտ է սեղմել F12 կամ **ctrl+shift+c**: Արտածումը կարող է տեղի ունենալ նաև էկրանին բացված պատուհանի տեսքով:

Դիտարկենք script.js ֆայլում գրված հետևյալ ծրագիրը.

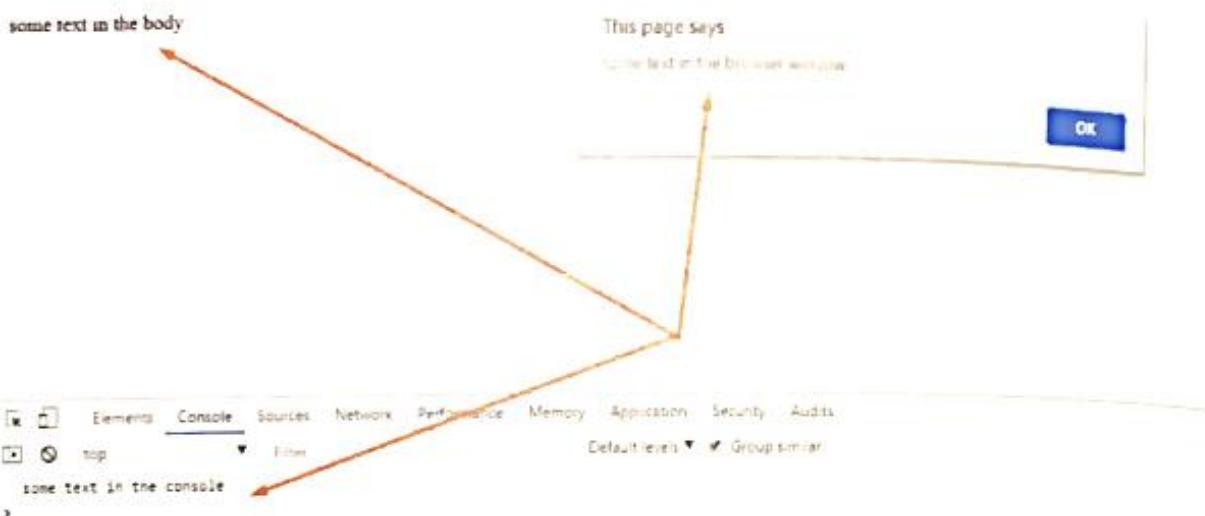
```

// արտածել տեքստ body -ում
document.write("some text in the body");

// արտածել տեքստ console պատուհանում
console.log("some text in the console");

// արտածել տեքստ alert պատուհանի միջոցով
alert("some text in the browser window");

```



Ինչպես նկատեցինք, JavaScript -ով գրված յուրաքանչյուր իրամանի վերևում առկա է երկու շեղագծով սկսվող արտահայտություն: Ընդունված է նման արտահայտություններին անվանել **մեկնաբանություն** (comment), դրանք նախատեսված են կոդում որոշ հատվածներ մեկնաբանելու համար, սակայն browser -ի կողմից դրանք անտեսվում են և չեն հայտնվում էլերանին: Մեկնաբանությունները կարող են լինել ինչպես տողային, այնպես էլ բլոկային: Նախատեսված են ծրագրավորողի համար: Երկու շեղագծով սկսվող մեկնաբանությունը կոչվում է **տողային**, քանի որ այն վերաբերվում է միայն տվյալ տողին: Բլոկային մեկնաբանությունը կարող է նկարագրվել մի քանի

տողով, այն սկսվում է /\* սիմվոլներով և ավարտվում \*/:

տարրերակով:

Մեր օրինակում `document.write` գործիքի օգնությամբ կարողացանք տվյալներ ավելացնել էջին, `console.log` -ի օգնությամբ՝ `console` պատուհանում, իսկ `alert` -ի դեպքում, հնարավորություն ունեցանք արտածումը իրականացնել պատուհանի միջոցով: Նկատենք, որ `document.write` գործիքը `body` -ին ավելացնում է ցանացած `html` պարունակություն, որը կփոխանցվի իրեն՝ ապաթարցերով:

Այսահուն՝ `document.write("<h1> Hello </h1>")`; իրամանի դեպքում էկրանին կհայտնվի `h1` տեք, որում գրված կլինի `Hello` տեքստը:

### 2.3 Փոփոխականներ և տվյալների տիպեր

Փոփոխականները համակարգի հիշողության մեջ որոշակի տեղ զբաղեցնող տարրեր են, որոնք բնութագրվում են հստակ անունով, տիպով և արժեքով: Փոփոխականներն օգտագործվում են ինֆորմացիա պահելու նպատակով:

Փոփոխականի գաղափարին ծանոթանում ենք դպրոցական մաթեմատիկայի և ֆիզիկայի դասընթացներից, որտեղ, օրինակ, ճանապարհի երկարությունը նշանակում են `s`-ով, մեքենայի արագությունը՝ `v`-ով և այլն:



JavaScript -ում փոփոխականների գլխավոր տիպերն են.

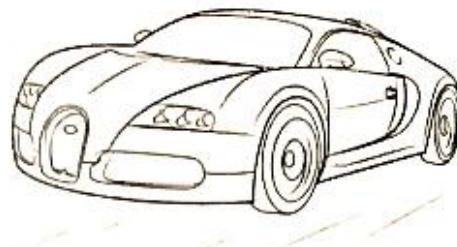
1. **Number** – թվային տիպ
2. **String** – տողային տիպ
3. **Boolean** – տրամաբանական տիպ

**Number** տիպը նախատեսված է թվային արժեքներ պահպանելու համար: Այստեղ կարող ենք պահել որևէ ապրանքի գինը, ինչ-որ մեկի տարիքը, մեքենայի անցած ճանապարհի երկարությունը և այլն:

**String** տիպը թույլ է տալիս պահպանել տեքստային տվյալներ: `String` -ի մեջ կարող ենք պահպանել որևէ ապրանքի

Նկարագրությունը, ինչոր մեկի անունը, ազգանունը,  
կենսագրությունը և այլն:

Boolean տիպը թույլ է տախս պահպանել միայն երկու  
հնարավոր արժեքներ՝ **true** և  
**false**: Համապատասխանաբար  
**true** -ն համարվում է ճշմարիտ  
կամ դրական արժեք, **false** -ը՝  
կեղծ կամ բացասական: Boolean  
տիպի ներսում, օրինակ, կարող  
ենք պահել՝ արդյո՞ք օգտատերը  
սիրում է մեքենա վարել, թե՛ ոչ: Ըսդհանրապես, բոլոր այն  
հարցերը, որոնց պատասխանը երկու հնարավոր տարրերակ է  
ենթադրում, կարելի է նկարագրել Boolean տիպով: Օրինակ՝  
արդյո՞ք օգտատերը ամուսնացած է, ծառայե՞լ է բանակում և  
այլն:



Այսպիսով՝ մենք իմացանք, որ փոփոխականները  
բնութագրվում են անունով, արժեքով և տիպով: Իսկ ինչպես ս  
կարող ենք հայտարարել փոփոխականներ: JavaScript -ում  
փոփոխականներ կարելի է հայտարարել **var** հրամանի  
օգնությամբ:

```
var x;  
console.log(x); //undefined
```

Մեր օրինակում հայտարարվել է փոփոխական, որի անունը **x** է,  
հաջորդ տողում մենք փորձել ենք **console** պատուհանում  
արտածել նրա արժեքը: Արդյունքում կտեսնենք, որ **console**  
պատուհանում հայտնվել է **undefined** արտահայտությունը:  
Պատճառն այն է, որ մենք հայտարարել ենք փոփոխական,  
սակայն չենք տվել նրան արժեք: Բոլոր այն փոփոխականները,  
որոնք չունեն արժեք, JS -ի կողմից ստանում են **undefined**  
արժեքը: Այսպիսով **undefined** -ը գոյություն կամ արժեք չունեցող  
փոփոխականների տիպը և արժեքն է:

Փոփոխականների անունը պետք է լինի **իդենտիֆիկատոր**:

Այսինքն.

- չի կարող սկսվել թվով
- չի կարող պարունակել բացատ (space)
- չի կարող պարունակել միջին գծիկ, շեղագիծ և այլ սիմվոլներ

Փոփոխականների սխալ անուններ են՝

**name\_1, 4x, first-name**

Ճիշտ փոփոխականների անուններ կլինեն

**name\_1, x4, first\_name**

Արդեն նշեցինք, որ փոփոխականների հիմնական 3 տիպերը JS -ում string, number և boolean տիպերն են: Դիտարկենք դրանց կիրառման օրինակներ.

```
var name = "Hayk"; //string  
var age = 25; //number  
var isMarried = false; //boolean
```

Ինչպես նկատեցինք, բոլոր այն դեպքերում, երբ կիրառվում է string տիպը նրա արժեքը գրվում է ապաթարցերի մեջ:

Դիտարկենք հետևյալ կոդը.

```
var x = 4;  
var y = 8;  
var s = x + y;  
alert(s); //12
```

Բերված օրինակում նախ հայտարարվել է x անունն ունեցող փոփոխական, ապա՝ y, որոնք համապատասխանաբար ունեն 4 և 8 արժեքները, այնուհետև հայտարարվել է s փոփոխականը, որի մեջ արդեն պահպել է x և y փոփոխականների գումարը: alert -ի արդյունքում էկրանին կհայտնվի 12 -ը:

JavaScript -ում փոփոխականի տիպը որոշվում է արժեքավորումից հետո: Այսպիսով՝ var x = 4 հրամանի դեպքում browser -ը հասկանում է, որ x փոփոխականը թվային է:

Թվային փոփոխականների հետ կատարվող հիմնական գործողություններն են.

- x + y - գումարում
- x - y - հանում

- $x * y$  - բազմապատկում
- $x / y$  - բաժանում
- $x \% y$  -  $x/y$  հարաբերությունից առաջացած մնացորդի անջատում

Կրկին նշենք, որ տեքստային ինֆորմացիան փոփոխականի ներսում պահվում է ապաթարցների օգնությամբ: Օրինակ.

```
var anun = "Hayk";
console.log("barev", anun); // barev Hayk
```

Նկատենք, որ տողային տիպի հետ նույնպես հնարավոր է կատարել որոշակի գործողություններ: Օրինակ՝ տողային տիպերի դեպքում գումարումը (concatination) իրենից ներկայացնում է երկու տողերի միավորում:

Դիտարկենք օրինակը.

```
var anun = "Hayk";
var language = "JavaScript";
var result = anun + " loves " + language + " so much!";
// Hayk loves JavaScript so much!
```

Ինչպես կարող ենք նկատել `result` փոփոխականի արժեքը ձևավորվում է 4 տարբեր տողեր միմյանց գումարելու արդյունքում: Արտահայտության մեջ `anun` -ը փոխարինվում է “`Hayk`” -ով, `language` -ը՝ “`JavaScript`” -ով, արդյունքում, եթե `alert` անենք `result` -ի արժեքը կտեսնենք, որ էկրանին հայտնվում է “`Hayk loves JavaScript so much !`” տեքստը:

Հիմնվելով ասվածի վրա՝ ծանոթանանք `JavaScript` -ի `prompt` գործիքի հետ, որը թույլ է տալիս հարց ուղարկել օգտատիրոջը և կիրառել նրա պատասխանը մեկ այլ գործողության մեջ:

Օրինակ՝

```
var name = prompt("what is your name?");
document.write("hello " + name);
```

Այս դեպքում, եթե էջը բացվում է էկրանին հայտնվում է պատուիան, որում գրված է մեր կողմից տրված հարցը, այսինքն՝ `what is your name`, ինչպես նաև մուտքային դաշտ, որտեղ օգտատերը կարող է մուտքագրել պատասխանը: Մուտքագրված պատասխանը պահպանվում է `name`

Վիստարականի մեջ, որն էլ իր հերթին հաջորդ տողում կիրառվում է `document.write` - ում:



String տիպի ներսում հնարավոր է պահպանել նաև թվային արժեքներ, սակայն այս դեպքում տեղի է ունենում տիպերի որոշակի ծևափոխություններ:

Դիտարկենք օրինակը.

```
var x = "5";
var y = "4";
var s = x + y;
alert(s) //54
```

Ինչպես նկատեցինք x փոփոխականի ներսում պահպանված է տողային տիպի 5 -ը, իսկ y -ի մեջ՝ 4 -ը: Դրանց գումարը ստացվում է 54, քանի որ տողային տիպերի դեպքում գումարել նշանակում է միացնել իրար:

Այստեղ մեծ նշանակություն ունի գումարվող պարամետրերի հերթականությունը: Դիտարկենք մի քանի պարզ օրինակներ.

```
console.log(5+4 + "1"); //91
console.log("5" + 4 + 1); // 541
```

Առաջին օրինակում ստացվում 91, քանի որ առաջին երկու արժեքները թվային են, ուստի սկզբից կատարվում է թվային գործողությունը՝  $5+4$ , ապա ստացված 9 -ին ավելանում է 1 -ը, որն արդեն տողային տիպի է, հետևաբար ստացվում է 91: Երկրորդ օրինակում առաջին պարամետրը տողային է, այս դեպքում արդեն ողջ արտահայտությունը համարվում է

տողային՝ անկախ շարունակությունից: Արդյունքում “5”, 4 և 1 պարամետրերը միանում են իրար և դառնում 541:

Այսպիսով՝ տարբեր տիպեր ունեցող արժեքների գումարման ժամանակ, եթե առաջին պարամետրը տողային է, ապա ամբողջ արտահայտությունն ենթարկվում է տողային տիպերի գումարման կանոնին: Եթե որևէ գումարելի կրկին տողային է, ապա, այդ կետից սկսած, ողջ արտահայտությունը համարվում է տողային:

Մենք արդեն գիտենք, որ տվյալների Boolean կամ տրամաբանական տիպը կարող է ունենալ միայն երկու արժեք՝ true և false: Boolean տիպը JS -ի հիմնային տիպերից է, ուստի այն կարող է հանդիսանալ տարբեր արտահայտությունների արժեքներ:

Դիտարկենք օրինակը.

```
console.log( 5 > 4 ) //true  
console.log( 4 < 2 ) //false  
console.log( 2*5 > 4*6 ) //false
```

Ըստհանրապես JavaScript -ում true արժեքին համապատասխանության մեջ է դրված 1 թիվը, իսկ false -ին՝ 0 -ն:

```
console.log( true + 1 ) // 2  
console.log( 2 * false ) // 0  
console.log(true / false) //Infinity
```

Առաջին դեպքում ստացվում է 2, քանի որ true = 1: Երկրորդ օրինակում 2\*false -ը համարժեք է 2 \* 0 արտահայտությանը, որի արժեքը 0 -է: Երրորդ օրինակում true/false հարաբերությունը համարժեք է 1/0 հարաբերությանը, որը, ինչպես հայտնի է մաթեմատիկայի դասընթացից, ծգում է անսահմանության:

Ամփոփելով տրամաբանական տիպը՝ փորձենք հասկանալ ինչ արդյունք կլինի էկրանին, եթե փորձենք ստուգել՝ արդյո՞ք 5 -ը հավասար է 5 -ի:

```
console.log( 5 = 5 ); //error
```

Այս հրամանի կատարման արդյունքում **console** պատուհանում  
կունենանք Error<sup>2</sup>:

☞ **Uncaught ReferenceError: Invalid left-hand side in assignment**

Խնդիրն այն է, որ JavaScript -ում  $=$  նշանը կիրառվում է խրագրման ժամանակ: Օրինակ՝  $x = 4$ ; արտահայտությունը կարող է նշանակել  $x$  փոփոխականի մեջ պահել 4 արժեքը: Սակայն վերագրում հնարավոր է իրականացնել միայն փոփոխականների նկատմամբ, իսկ թվի մեջ պահել մեկ այլ թվի արժեք՝ անհնար է, ահա թե ինչ է ասվում Error -ի միջոցով: Իսկ ինչպես կարող ենք ստուգել՝ արդյո՞ք երկու փոփոխականներ ունեն նույն արժեքը, թե ոչ: Ծրագրավորման լեզուների մեջ մասում, այդ թվում, JavaScript -ում, հավասարությունը ստուգելու համար կիրառվում է  $==$ , իսկ անհավասարության դեպքում՝  $!=$

նշանը:

```
console.log(5 == 5); //true
console.log(6 == 4); //false
console.log(5 != 4) //true
console.log(5 != 5) //false
```

Իսկ ի՞նչ տեղի կունենա, եթե համեմատենք տվյալների տարրեր տիպեր ունեցող արժեքներ: Օրինակ՝  $5 == "5"$ : Առաջին հայացքից թվում է, թե պատասխանը պետք է լինի `false`, քանի որ առաջին տարրը թվային է, երկրորդը՝ տողային, սակայն JavaScript -ը կրկնակի հավասարության նշանի դեպքում ուշադրություն է դարձնում միայն արժեքներին: Քանի որ երկու դեպքում էլ արժեքը 5 է, ուստի այն մեզ վերադարձնում է `true`: Եթե մենք ցանկանում ենք, որպեսզի արժեքներից բացի համեմատվեն նաև տվյալների տիպերը, կարող ենք կիրառել խիստ հավասարության նշանը՝  $==$ :

```
console.log(5 == "5"); //true
console.log(5 === "5"); //false
```



<sup>2</sup>Համակարգի աշխատանքի ընթացքում առաջացած խափանում

## 2.4 Տիպերի ձևափոխություններ

Մենք արդեն սովորեցինք, որ JavaScript -ում յուրաքանչյուր փոփոխական բնութագրվում է անունով, արժեքով և տիպով: Իսկ հնարավո՞ր է փոխել տվյալների տիպը՝ անցնելով մի տիպից: Մյուսին: Նախապես նշենք, որ JavaScript -ում առկա է **typeof** օպերատորը, որը թույլ է տալիս որոշել փոփոխականի տիպը: Դիտարկենք օրինակը.

```
var x = "text", y = 4, z = true;  
console.log(typeof x); //string  
console.log(typeof y); //number  
console.log(typeof z); //boolean  
console.log(typeof m); //undefined, քանի որ նման փոփոխական չկա հայտարարված
```

Այժմ ուսումնասիրենք, թե ինչպես կարող ենք անցում կատարել տվյալների մի տիպից մյուսին:

**Անցում թվային տիպից տողայինի.**

```
var x = 45;  
var y = x.toString() // "45"
```

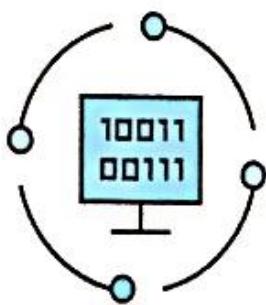
**Անցում տողային տիպից թվայինի.**

```
var x = "45";  
var y = Number(x); //45
```

Նկատենք, որ `toString` ֆունկցիան թույլ է տալիս ոչ միայն թվային, այլ նաև ցանկացած տիպից անցում կատարել տողային տիպի:

Սուանձնակի հետաքրքրություն է ներկայացնում այն, որ `toString` ֆունկցիան կարող է ստանալ պարամետր, որի օգնությամբ կարող ենք տրված թիվը ներկայացնել թվերի գրառման մեջ այլ համակարգում:

Օրինակ՝  
var x = 45;  
var y = x.toString(2); // 101101  
Մեր օրինակում 45 թիվը ներկայացվեց  
թվերի գրառման երկուական  
համակարգում<sup>3</sup>:



JavaScript -ի վերջին տարբերակներում այլևս հնարավոր է նաև կատարել գործողություններ երկուական թվերի հետ: Երկուական համակարգի թվերը JS -ում սկսվում են **0b** (զռու և b) սկարագրությամբ: Օրինակ՝ 45 թիվը երկուական համակարգում նշ. 101101 արժեքը, 10 թիվը՝ 1010: Դրանց գումարը JS -ում էլեկտրոնայի արտահայտությամբ.

կարող ենք գտնել համայական  
console.log(**ob101101** + **ob1010**); //55  
ենք որ այս դեպքում վեր

կարող սեր կատեր, որ այս դեպքում վերջնական արդյունքը, այսուհետեւ՝ `console.log(ob101101 + ob1010)`, կլինի այսպիսի ներկայացվում՝ 10-ական համակարգում:

Եթե ցանկանում ենք վերջնական արդյունքում կրկին ստանալ երկուական համակարգի թիվ, ապա կօգտվենք `toString` ֆունկցիայից:

```
// 110111
console.log((ob101101 + ob1010).toString("2"));
```

Երբուակ  
ֆունկցիայից:  
`console.log((ob101101 + ob1010).toString("2")); // 110111`  
եթե Ob -ի փոխարեն թիվը պարզապես սկսվի 0 -ով, ապա այն կղիտարկվի որպես 8-ական համակարգի թիվ, Օx դեպքում՝ 16-ական:

```
Օրինակ՝  
console.log(0165 + 0xF); // 132
```

<sup>3</sup> Հաշվարկման համակարգ է, որտեղ կիրառվում են միայն 0 և 1 թվանշանները: Ունի լայն կիրառություն ժամանակակից համակարգիչներում և թվային սարքերում:

Օրինակ՝

```
var x = "45.84256";
var y = parseInt(x); //45
var m = parseFloat(x); // 45.84256
var n = parseFloat(x).toFixed(2); // "45.84"
```

Այսպիսով՝ parseInt ֆունկցիան տվյալը ձևափոխում է թվի՝ վերցնելով ամբողջ մասը, parseFloat -ը ձևափոխում է տվյալը իրական թվի: toFixed ֆունկցիան որոշում է, թե քանի նիշ պետք է ցուցադրվի ստորակետից հետո, ստացված արդյունքը վերածում է տողային տիպի, ուստի մենք կարող ենք օգտագործել Number ֆունկցիան, որպեսզի արդյունքը կրկին լինի թվային:

Օրինակ՝

```
var n = Number(parseFloat(x).toFixed(2)); //45.84
```

Ուշագրավ է այն, որ parseInt կամ parseFloat ֆունկցիաները չեն կլորացնում թիվը: Թիվը դեպի վերև, կամ դեպի ներքև մոտարկելու համար կարող ենք օգտվել Math կլասի round, floor և ceil ֆունկցիաներից:

- **round** - մոտարկում է թիվը դեպի մոտակա ամբողջը
- **floor** - մոտարկում է թիվը դեպի վերև գտնվող մոտակա ամբողջը
- **ceil** - մոտարկում է թիվը դեպի ներքև գտնվող մոտակա ամբողջը

Ասվածը դիտարկենք օրինակում.

```
Math.round(4.5); // 5
Math.round(4.7); // 5
Math.round(4.3); // 4
Math.floor(4.9); // 4
Math.floor(4.3); // 4
Math.ceil(4.1); // 5
Math.ceil(4.9); // 5
```



Math կլասի այլ հետաքրքիր ֆունկցիաներ են.

- `pow(x,y)` - հաշվում է  $x^{-y}$  աստիճանը նկատենք, որ  $x^{-y}$  աստիճանը կարելի է հաշվարկել նաև էքսպոնենցիալ օպերատորի (\*\*\*) օգնությամբ, որը ներդրվել է ECMAScript7 -ում: Օրինակ՝  $7^{3/2}$ -ը JS -ում այլսա կարելի է գրել  $7**3$  տեսքով:
- `sqrt(x)` - հաշվում է  $x$  թվի քառակուսի արմատը
- `sin(x)` - հաշվում է  $x$  թվի սինուսը
- `cos(x)` - հաշվում է  $x$  թվի կոսինուսը
- `random()` - վերադարձնում է պատահական թիվ 0-ից 1 հատվածից
- `max(x1,x2,...xn)` - վերադարձնում է իրեն փոխանցված թվերից մեծագույնի արժեքը
- `min(x1,x2,...xn)` - վերադարձնում է իրեն փոխանցված թվերից փոքրագույնի արժեքը

Օրինակ՝ Math կլասի օգնությամբ հաշվենք  $2^8 + \cos(0) + \sqrt{64}$

`var t = Math.pow(2,8) + Math.cos(0) + Math.sqrt(64);`

`console.log(t); //265`

## 2.5 Հաստատուներ

Հաստատունը ծրագրային օբյեկտ է, որը համակարգի աշխատանքի ընթացքում չի փոխում իր արժեքը, իսկ նրան մեկ այլ արժեք վերագրելու դեպքում առաջանում է error. Դիտարկենք օրինակը՝

`const X = 4;`

`X = 8; //error`

Ընդունված է, որ հաստատուների անունները պետք է գրել մեծատառ:

Նկատենք, որ հաստատունը պետք է նախապես արժեքավորվի հայտարարման փուլում, հակառակ դեպքում այն հանգեցնում է Error -ի: Օրինակ՝

`const X;`

`X = 2; //error`

Պատճառն այն է, որ const X իրամանի դեպքում X հաստատունը արժեքավորվում է undefined -ով, իսկ հաջորդ քայլում արդեն նրան 2 վերագրելով խախտում ենք հաստատունների կանոնը ինչն էլ հանգեցնում է Error -ի :

## ՀԱՐՏԵՐ ԵՎ ԱՌԱՋԱԴՐԱՆՔՆԵՐ

1. Ի՞նչ եղանակներով է հնարավոր HTML էջին ավելացնել script:
2. Ինչո՞ւ է խորհուրդ տրվում <script> -ը գոել <body> -ի վերջում
3. Բացատրել defer և async հատկությունների նշանակությունը:
4. Որո՞նք են փոփոխականների իմանական տիպերը JS -ում:
5. Ինչպե՞ս կարելի է հայտարարել փոփոխականներ:
6. Ելքագրման ինչպիսի՞ օպերատորներ գիտեք:
7. Ի՞նչ է undefined -ը:
8. Ի՞նչ նպատակով է կիրառվում prompt ֆունկցիան:
9. Ի՞նչ արժեք կստացվի true+false արտահայտության դեպքում:
10. Ի՞նչ կվերադարձնի true == “true” արտահայտությունը և ինչո՞ւ:
11. Ինչո՞ւ է առաջանում Error' 5=5 իրամանի կատարման արդյունքում:
12. Բացատրել toString(), parseInt(), parseFloat() և Number() ֆունկցիաների դերը:
13. Ինչպե՞ս կարող ենք երկուական համակարգի թվերի հետ գործողություններ կատարել JS -ում:
14. Բացատրել round, ceil, և floor ֆունկցիաների տարրերությունները:
15. Ի՞նչ է հաստատունը և ինչպե՞ս են այն հայտարարում:

ԴԱՍ 3

# DOCUMENT OBJECT MODEL



## ԴԱՍ 3 : DOCUMENT OBJECT MODEL (DOM)

### 3.1 Գաղափար DOM -ի մասին

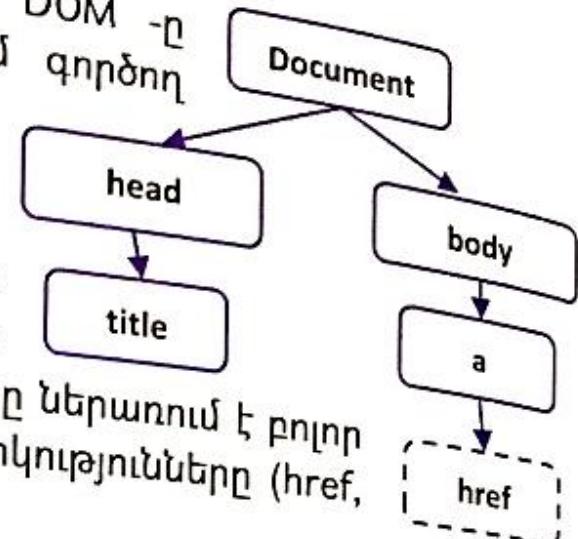
Document Object Model -ը՝ DOM -ը  
JavaScript - ի միջավայրում գործող գործող  
գործիքների համախումբ է,  
որով կարող ենք կառավարել  
վեր էջը:

Տանկացած վեր էջ բացվելիս  
browser -ը ստեղծում է այդ էջի

Document Object Model -ը, որը ներառում է բոլոր  
առկա տեգերը և նրանց հատկությունները (href,  
id, class , style և այլն):

Ընդհանուր առմամբ, DOM - ի կառուցվածքում գլխավոր դերը  
պատկանում է **document** -ին, որն էլ իրենից ներկայացնում է մեր  
ընթացիկ HTML փաստաթուղթը:

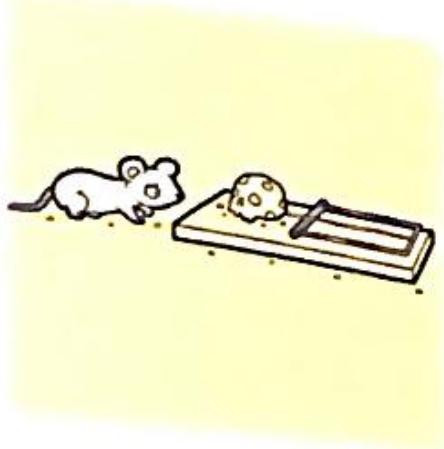
Document -ի մեջ են գտնվում head -ը և body -ին, իսկ դրանց մեջ  
արդեն յուրաքանչյուրի ժառանգ տեգերը: JavaScript -ում DOM -ը  
մեզ տրամադրում է ստանդարտ ֆունկցիաներ, որոնց  
կիրառմամբ մենք կարող ենք.



- փոփոխության ենթարկել HTML տեգերի  
պարունակությունը,
- փոփոխության ենթարկել յուրաքանչյուր HTML տեգի  
հատկությունները (id, class...),
- փոփոխության ենթարկել HTML տեգերի style -երը,
- էջից հեռացնել տարբեր HTML տեգեր, ինչպես նաև,  
դրանց հատկություններ (attribute),
- ստեղծել և ավելացնել նոր HTML տեգեր (ոինամիկ  
ծրագրավորման մեթոդ),
- ստեղծել և կառավարել HTML Event -ներ (կը ննարկենք  
հաջորդ ենթակեդում)

**3.2 Գաղափար Event -ների մասին**  
JavaScript -ում մեր կողմից դիտարկված բոլոր հրամանները կատարվում էին հերթականությամբ՝ մեկը մյուսի ետևից: Վեր կայքերում, մեկը մյուսի ետևից: Վեր կայքերում, երբեմն անհրաժեշտ է սակայն, երբեմն գործողությունների լինում, որպեսզի գործողությունների որոշ խմբեր կատարվեն միայն հրամանից հետո: Ծրագրավորման այս տարրերակն անվանում են ինտերակտիվ ծրագրավորում:

Այս մեթոդի օգնությամբ, ծրագրավորողը կարող է որոշել՝ ի՞նչ պետք է տեղի ունենա, եթե օգտատերը սեղմում է որևէ դաշտ, տեղաշարժում մկնիկը, մուտքագրում տեքստ և այլն: Այս գործողություններն անվանում են **Event** -ներ: Այսպիսով՝ event կարող է լինել button -ի սեղմումը, մկնիկի շարժումը, որևէ ստեղնի սեղմումը և այլն:



Իսկ ինչպե՞ս կարող ենք ստեղծել Event -ներ:

Ենթադրենք էջում ունենք button (կոճակ), որի սեղման ժամանակ ցանկանում ենք alert (հաղորդագրություն) անել էկրանին որևէ արտահայտություն:

HTML տեգերի մեջ մասը տրամադրում է հատկությունների խումբ, որոնց օգնությամբ կարող ենք տարբեր event -ներ կապել տվյալ տեգին:

Այդպիսի օրինակ է **onclick** հատկությունը, որի օգնությամբ կարող ենք կառավարել ցանկացած տեգի վրա մկնիկի սեղման գործընթացը:

Դիտարկենք օրինակ.

**<button onclick="myFunction()">**

Click Here

**</button>**

Կոդը կարելի է բացատրել հետևյալ կերպ. button -ի սեղման ժամանակ պետք է աշխատի JavaScript -ում գտնվող գործիք, որի անունն է myFunction(): Ըստ էության, JavaScript -ի միջավայրում չկա նման գործիք, ուստի մենք ստիպված ենք ստեղծել այն: Այս

իմաստով մեր կողմից ստեղծված script.js ֆայլում անհրաժեշտ է ավելացնել հետևյալը.

```
function myFunction()
```

```
{
```

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

```
}
```

Այստեղ function բառը ցույց է տալիս, որ ստեղծվում է ֆունկցիա: myFunction -ը հանդիսանում է ֆունկցիայի անունը, իսկ ծևավոր փակագծերի ներսում գրվածը՝ ֆունկցիայի մարմինը, ցույց է տալիս գործողությունը, որը պետք է ֆունկցիան իրականացնի:

Այսպիսով՝ յուրաքանչյուր անզամ Button -ի սեղման ժամանակ կաշխատի myFunction -ը, որի ծևավոր փակագծերը ցույց են տալիս տվյալ ֆունկցիայի սկիզբը և վերջը: Ինչպես նշվեց, ծևավոր փակագծերի միջև ընկած արտահայտությունը կոչվում է ֆունկցիայի մարմին, որտեղ գրված կոդը, մեր օրինակում, կատարվում է այն ժամանակ, երբ է սեղմվում է Button -ը: Այս դեպքում, button -ի սեղման ժամանակ բացվում է alert պատուհան, որի մեջ գրված է Hello տեքստը:

Բացի onclick -ից, event-ների համար հաճախ կիրառվող հատկություններ են.

- **onkeydown** - երբ սեղմվում է ստեղնաշարի ստեղներից որևէ մեկը,
- **ondblclick** - երբ կատարվում է մկնիկի կրկնակի սեղմում (double click),
- **onmousemove** - երբ տեղաշարժվում է մկնիկը,
- **onmouseup** - երբ սեղմելուց հետո բաց է թողնվում մկնիկի կոճակը,
- **onkeyup** - երբ սեղմված վիճակից ազատ ենք թողնում ստեղներից որևէ մեկը
- **onmousedown** - երբ սեղմվում է մկնիկի կոճակը և Նկատենք, որ էական տարրերություն կա onclick -ի և onmousedown -ի միջև, քանի որ click նշանակում է սեղմել

Ակտիվ կոճակը և բաց թռինել, մինչդեռ օպուսէծան -ը նշանակում է միայն սեղմել մկնիկի կոճակը:

**3.3 Ինտերակտիվ ծրագրավորման տարրերը**  
Այժմ փորձենք ստեղծել պարզագույն գործողություններ իրականացնող հաշվիչ: Դիցուք ունենք երկու input դաշտեր, button և h1: Button -ի սեղման ժամանակ պետք է գումարել input դաշտերում մուտքագրված թվերի արժեքները, որոնց արդյունքը ցույց տալ h1 -ի ներսում:

Դիտարկենք HTML կոդը.

```
<h1 id="pataxan"></h1>
<input type="text" id="tiv1">
<input type="text" id="tiv2">
<button onclick="gumarel()">Hashvel</button>
```

Արդյունքում էկրանին կունենանք նկարում բերված պատկերը:

Քանի որ button -ի սեղման ժամանակ աշխատում է gumarel կոչվող ֆունկցիան, ապա script.js -ում կարիք կլինի նկարագրելու այն: Ֆունկցիայի նպատակն է ստանալ input դաշտերում մուտքագրված թվերը և գումարել իրար: Այժմ ծանոթանանք JavaScript -ի **getElementById()** ֆունկցիայի հետ, որը մեզ վերադարձնում է տրված id -ով տեղը՝ ներառելով նրա բոլոր հատկությունները:

Օրինակ՝

```
var x = document.getElementById("tiv1");
```

Իրամանը x փոփոխականի մեջ պահում է #tiv1 տեղի հետ կապված ամբողջ ինֆորմացիան:

Նկատենք, որ մուտքային դաշտերի (օր.' input, textarea... ) ներսում գրված արժեքը վերցնում են value -ով, իսկ զույգ տեղերի դեպքում (օր.' div, h1...)' innerHTML-ով:

Այսպիսով՝ եթե մենք ցանկանում ենք վերցնել #tiv1 input դաշտի ներսում գրված արժեքը, ապա կարող ենք կատարել հետևյալը.

HASHVEL

```
tiv2 = Number(tiv2);
var s = tiv1 + tiv2;
alert(s);
```

} Ուշագրավ է այն հանգամանքը, որ JavaScript ծրագրավորման լեզուն անմիջական կապի մեջ է գտնվում Browser -ի հետ, ուստի #tiv1 էլեմենտի ներսում գրված արժեքը կարելի է ստանալ ոչ միայն document.getElementById -ի միջոցով, այլ պարզապես tiv1 -ին որպես փոփոխական դիմելով: Իսկ տողային տիպից թվայինի անցումը կարելի է կազմակերպել ոչ միայն Number -ի օգնությամբ, այլ՝ + նշանը փոփոխականից առաջ դնելով, որին անվանում են **ունար օպերատոր**:

Հիմնվելով ասվածի վրա՝ ֆունկցիան կստանա հետևյալ տեսքը.

```
function gumarel(){
    var t1 = +tiv1.value;
    var t2 = +tiv2.value;
    var s = t1 + t2;
    alert(s);
```

}

Կրկին ստուգելով՝ կհամոզվենք, որ գումարումն արդեն տեղի է ունենում թվային մեթոդով: Նկատենք, որ ավելի կարճ գրելածն կարելի էր ապահովել, եթե տիպերի ձևափոխությունը կատարվեր փոփոխականի հայտարարման փուլում.

```
var tiv1 = Number(document.getElementById("tiv1").value);
```

Մեր սկզբնական խնդրում պահանջվում էր, որպեսզի վերջնական արդյունքը ցուցադրվեր ոչ թե alert -ի օգնությամբ այլ h1 -ի մեջ: Ինչպես արդեն նշվեց, ընդունված է, որ բոլոր տրված տեքերն ունեն value, իսկ մնացյալ տեքերի պարունակությունը JavaScript -ի DOM -ում դեկավարվում է innerHTML հատկությամբ: Արդյունքում՝

```
function gumarel(){
    var tiv1 = document.getElementById("tiv1").value;
    var tiv2 = document.getElementById("tiv2").value;
    tiv1 = Number(tiv1);
```

```

    tiv2 = Number(tiv2);
    var s = tiv1 + tiv2;
    var p = "gumar=" + s;
    patasxan.innerHTML = p;
}

```

GUMAR = 15

Ամփոփելով նշենք՝ եթե հաշվարկի արդյունքում s - ի համար ստացվել էր 15, ապա h1 -ի մեջ կհայտնվի gumar=15 տեքստը:

10
5
<b>HASHVEL</b>

### 3.4 DOM -ի աշխատանքը CSS -ի հետ

JavaScript ծրագրավորման լեզուն թույլ է տալիս կառավարել նաև HTML տեքտերի CSS հատկությունները։ Այդ իմաստով է կիրառվում DOM -ի style հատկությունը։

Ենթադրենք ունենք հետևյալ էլեմենտը։

<h1 id="patasxan"> some text here </h1>

Անհրաժեշտ է նրա տառերի գույնը դարձնել կարմիր։

Այդ նպատակով նախ DOM -ի օգնությամբ այդ տեքի տվյալները պահպանենք ելմ կոչվող փոփոխականի ներսում։

var elm = document.getElementById("patasxan");

Այնուհետև արժեք տանք ելմ -ի style հատկության color դաշտին։

elm.style.color = "red";

Այս դեպքում տեքստի գույնը կդառնա կարմիր։ Նկատենք, որ այս ամենը կարելի էր գրել ընդամենը մեկ տողով՝ առանց փոփոխականի հայտարարման։

document.getElementById("patasxan").style.color = "red";

JavaScript -ում հնարավոր է կիրառել բոլոր այն հատկությունները, որոնք առկա են CSS -ում։ Ուշագրավ է այստեղ այն, որ եթե որևէ CSS հատկություն նկարագրվում է մեկից ավելի բառերով, անհրաժեշտ է այդ բառերը միացնել իրար, և բացի առաջին բառից մնացյալ բոլորի դեպքում առաջին տառը դարձնել մեծատառ։

Օրինակ՝ `background-color` հատկությունը CSS -ում, ինչպես գիտենք գրվում է միջին գծիկով, մինչդեռ JavaScript -ում դրան փոխարինողը կլինի `backgroundColor` -ը:

Դիտարկենք ասվածը օրինակներով.

`elm.style.backgroundColor = "red";`  
`elm.style.animationTimingFunction = "linear";`

Ուշագրավ է նաև այն, որ JavaScript -ում կարելի է արժեք տալ ոչ միայն կոնկրետ CSS հատկությունների, այլ գրել ամբողջական CSS հրաման:

`document.body.style = "height:100vh; background:red;"`

Այս դեպքում, սակայն, տվյալ տեղի `html` նկարագրության մեջ, `style` ատրիբուտում գրվածը անտեսվում է:

## ՀԱՐՑԵՐ ԵՎ ԱՌԱՋԱԴՐԱՆՔՆԵՐ

1. Ի՞նչ է DOM -ը և ո՞րն է նրա նշանակությունը:

2. Ո՞րն է `getElementById` ֆունկցիայի նշանակությունը:

3. Ինչպես կարելի է իմանալ, թե ի՞նչ է գրված `input` -ի ներսում:

4. Ինչպես կարելի է փոխել `input` -ի պարունակությունը:

5. Ե՞րբ պետք է կիրառել `value` հատկությունը և ե՞րբ `innerHTML` -ը:

6. Ինչպես է աշխատում DOM -ը սթայլերի հետ

7. Ստեղծել HTML փաստաթուղթ, որի ներսում կա `input`

դաշտ, `h1` և `button`: `Input` դաշտում օգտատերը

մուտքագրում է վայրկյանների X քանակը, `button` -ի

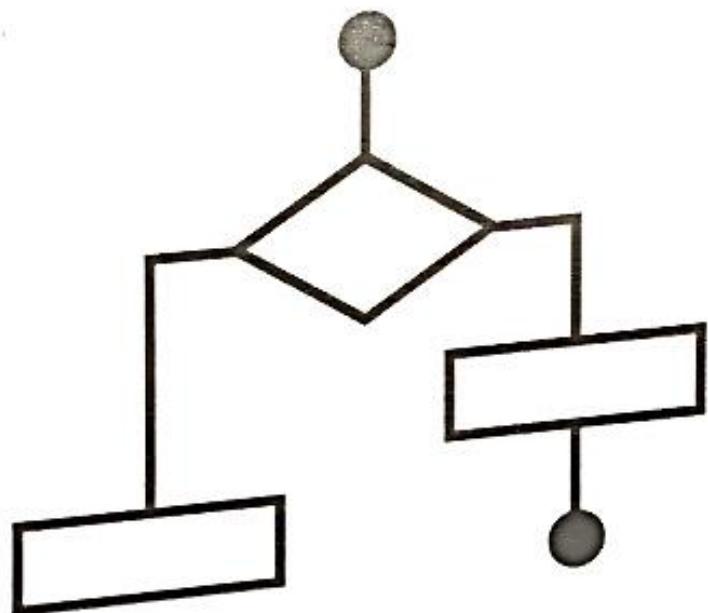
սեղման ժամանակ վերածել այդ վայրկյանները օրերի,

ժամերի և րոպեների: Օրինակ՝ X = 180600 մուտքի

դեպքում `h1` -ի մեջ պետք է գրել 2 օր, 2 ժամ, 10 րոպե:

# ԴԱՍ 4

## ՊԱՅՄԱՆԻ ՕՊԵՐԱՏՈՐ



## ԴԱՍ 4 : ՊԱՅՄԱՆԻ ՕՊԵՐԱՏՈՐ

### 4.1 Ըսդհանուր պատկերացում պայմանի օպերատորի

մասին

Պայմանի օպերատորը JavaScript ծրագրավորման լեզվի և առհասարակ ծրագրավորման լեզուների ամենակիրառվող օպերատորներից մեկն է: Այն մեզ թույլ է տալիս կատարել գործողություններ որոշակի պայմանի կամ պայմանների առկայության դեպքում: Բոլոր այն արտահայտությունները, որոնց արժեքը JS -ում true է կամ false, ընդունված է անվանել տրամաբանական կամ պայմանական արտահայտություն: Ըստ կազմության՝ պայմանական արտահայտությունները լինում են պարզ և բաղադրյալ: Այժմ ծանոթանանք պայմանի օպերատորի հետ:

### 5.2 Պարզ պայմանական արտահայտություն

Պայմանի օպերատորը JS -ում ունի հետևյալ տեսքը.

**if(պայման){**

//գործողություններ, որոնք կկատարվեն եթե պայմանը բավարարվում է

**}else{**

//գործողություններ, որոնք կկատարվեն, եթե պայմանը չի բավարարվում

**}**

Ծանոթանանք պայմանի օպերատորի աշխատանքին կոնկրետ օրինակով:

Դիտարկենք հետևյալ HTML կառուցվածքը.

**<h1 id="result"></h1>**

**<input type="text" id="age" onblur="checkAge()">**

Ունենք input և h1: input դաշտի նկատմամբ կիրառված է blur event -ը: Վերջինս աշխատում է այն ժամանակ, երբ կատարվում

Է կամայական գործողություն իրut դաշտից դուրս<sup>4</sup>: Այսպիսով եթե իրut դաշտում որևէ տվյալ մուտքագրենք և սեղմենք իրut դաշտից դուրս որևէ այլ տեղ, ապա կաշխատի նրա blur event-ը:

Օգտագործողը իրut դաշտում պետք է մուտքագրի իր տարիքը: Մեր նպատակն է պարզել՝ արդյո՞ք մուտքագրվածը թիվ է, թե ոչ:

Դիտարկենք checkAge ֆունկցիան.

```
function checkAge(){
    var age = document.getElementById("age");
    age = Number(age.value);
    var res;
    if(Number.isNaN(age) == false){
        res = "The age is correct";
    }else{
        res = "Please enter a correct number";
    }
    document.getElementById("result").innerHTML = res;
}
```

Ֆունկցիայի առաջին տողում հայտարարված է age փոփոխականը, որի մեջ պահպանվել է #age տեղը: Հաջորդ տողում արդեն age -ի մեջ պահպանվում է #age տեղի value -ն: Հաշվի առնելով, որ HTML -ից եկած արժեքները JS -ի համար դիտարկվում են որպես տողային տիպի տվյալ, ուստի Number ֆունկցիայի օգնությամբ կիրականացնենք անցում տողային տիպից դեպի թվային: Հաջորդիվ հայտարարվել է res փոփոխականը, որի մեջ պետք է պահենք վերջնական արդյունքը: Այսինքն՝ արդյո՞ք ճիշտ է մուտքագրված տարիքը, թե՛ ոչ: Number ֆունկցիայի աշխատանքի սկզբունքը հետևյալն է. Եթե Number(x) արտահայտության մեջ x -ը իրենից ներկայացնում է թվային սիմվոլ, ապա Number ֆունկցիան վերադարձնում է այդ սիմվոլը վերածած թվի: Բոլոր մնացյալ

---

<sup>4</sup> Ենթադրվում է, որ մինչ այդ գործողությունը իրut դաշտը եղել է ակտիվ:

Դեպքերում ֆունկցիան վերադարձնում է NaN (Not a Number)<sup>5</sup> արժեք, ինչը ցույց է տալիս, որ x -ը թվային չէ: Այսիսով՝ պայմանական արտահայտության մեջ Number.isNaN(age)==false արտահայտությունը ստուգում է՝ արդյո՞ք age -ը թվային է: Այս դեպքում res -ի մեջ պահպանվում է “The age is correct” արտահայտությունը, ինչը նշանակում է, որ տարիքը ճիշտ է մուտքագրված, հակառակ դեպքում res -ի մեջ պահպանվում է “Please enter a correct number” տեքստը, ինչը նշանակում է, որ անհրաժեշտ է մուտքագրել ճիշտ թվային արժեք:

Այսիսով, մուտքագրելով “Tigran” կստանանք բացասական արժեք:

**Please enter a correct number**

Tigran

Մուտքագրելով 18՝ կստանանք դրական արժեք:

**The age is correct**

18

Նույնը չի կարեի ասել Infinity մուտքագրելու դեպքում.

**The age is correct**

Infinity

Այս մասին, սակայն, կխոսենք հաջորդ ենթաբաժնում:

---

<sup>5</sup>Տե՛ս, դաս 38, հարց 6. Ի՞նչ է իրենից ներկայացնում NaN -ը

Պայմանի օպերատորը կարող է ունենալ նաև շղթայական կառուցվածք:

Օրինակ.

```
if(պայման){
```

```
//A
```

```
}else if( պայման 2){
```

```
//B
```

```
}else if( պայման 3){
```

```
//C
```

```
}else{
```

```
//D
```

```
}
```

Գրվածք կարելի է բացատրել այսպես. Եթե պայմանը ճշմարիտ է, կկատարվի գործողությունների Ա խումբը: Հակառակ դեպքում, եթե ճշմարիտ է պայման2 -ը, ապա կկատարվի գործողությունների Բ խումբը: Հակառակ դեպքում, եթե ճշմարիտ է պայման3 -ը, ապա կկատարվի Ծ խումբը: Բոլոր մնացյալ դեպքերում կկատարվի գործողությունների Ծ խումբը. Մնացյալ դեպքերում կկատարվի գործողությունների Ծ խումբը. Պայմանի օպերատորն ավելի լավ հասկանալու համար դիտարկենք հետևյալ խնդիրը: Ենթադրենք ունենք 3 input դաշտեր, որոնցում պետք է մուտքագրվեն թվեր: Button -ի սեղման ժամանակ անհրաժեշտ է գտնել այդ երեք թվերից մեծագույնը:

```
<input type="text" id="tiv1">
<input type="text" id="tiv2">
<input type="text" id="tiv3">
<button onclick="mecaguyn()">Maximum</button>
```

Խնդիրը կարելի է լուծել հետևյալ կերպ.

1. համեմատել ա, բ թվերը և մեծագույնը պահել max փոփոխականի մեջ,
2. համեմատել c թիվը max -ի հետ. Եթե c > max կատարել 3-րդ կետը, հակառակ դեպքում 4 -րդը,
3. max -ի մեջ պահել c թիվը,
4. արդյունքը տպել էկրանին

Այժմ տեսնենք mecaguyn() ֆունկցիայի նկարագրությունը.

```
function mecaguyn(){  
    with(document){  
        var a = Number(getElementById("tiv1").value);  
        var b = Number(getElementById("tiv2").value);  
        var c = Number(getElementById("tiv3").value);  
        var max;  
        if(a > b){  
            max = a;  
        }else{  
            max = b;  
        }  
        if(c > max){  
            max = c;  
        }  
        write("<h1>Maximum = " + max + "</h1>");  
    }  
}
```

Այստեղ **with** օպերատորը թույլ է տալիս խուսափել ամեն անզամ **document** հրամանը գրելուց: Այսպիսով՝ **with(document)** բլոկի ներսում **document** բառն այլևս չի շեշտվում:

256

45

132

MAXIMUM

Maximum = 256

#### 4.3 Բաղադրյալ պայմաններ

Եթեմն պայմանական արտահայտությունը կարող է կազմված լինել մի քանի մասերից: Օրինակ՝ Եթե թիվը մեծ է 9 -ից և փոքր է 100 -ից, ապա թիվը երկնիշ է : Ֆորմալիզացնելով տրված

պայմանը՝ կստանանք  $x > 9$  և  $x < 100$  արտահայտությունը: Այս օրինակում մեր երկու ենթապայմանները միմյանց կապված են և - ի օգնությամբ: Ծրագրավորման JavaScript լեզվում և կապին փոխարինում է **&&** նշանը:

```
if( x > 9 && x < 100){  
    console.log("erknish e");  
}else{  
    console.log("erknish che");  
}
```

**&&** -ի օգնությամբ կառուցված պայմանական արտահայտությանն անվանում են կոնյունկտիվ պայման: Կոնյունկտիվ պայմանը true արժեք ընդունում է միայն այն ժամանակ, երբ բոլոր ենթապայմանները ստանում են true արժեք:

Բաղադրյալ պայմանը կարող է հանդես գալ նաև **դիվյունկտիվ** տեսքով: Դիվյունկտիվ պայմանական արտահայտությունն առաջանում է այն ժամանակ, երբ ենթապայմանները միմյանց հետ կապվում են կամ - ի օգնությամբ, որը ծրագրավորման լեզվում նշվում է || սիմվոլի միջոցով: Ենթադրենք ունենք 3 թվեր և պետք է ստուգենք՝ արդյո՞ք այդ թվերից որևէ մեկը հավասար է 0 -ի:

```
if( a == 0 || b == 0 || c == 0){  
    console.log("tveric mek@ zro e");  
}else{  
    console.log("o -n bacakayum e");  
}
```

Նախորդ օրինակներից մեկում, օգտատերը պետք է մուտքագրեր իր տարիքը և ստուգվեր՝ արդյո՞ք այն թվային է, թե՝ ոչ: Մուտքագրելով Infinity ստացանք ճշմարիտ արժեք: Պատճառն այն է, որ JS -ում Infinity -ին նշանակում է անսահմանություն, այսինքն՝ շատ մեծ թվային արժեք, իսկ typeof Infinity == "number" արտահայտությունն ունի ճշմարիտ արժեք: Նման արժեք հնարավոր է ստանալ այն դեպքում, եթե

որևէ արտահայտություն վերադարձնի զգալի չափով ավելի մեծ արժեքը, քան ինարավոր է պահպանել JS -ում (**Number.MAX\_VALUE**), ինչպես նաև բոլոր այն դեպքերում, երբ տեղի է ունենում բաժանում 0 -ի վրա: Number կլասը տրամադրում է նաև **isFinite** ֆունկցիան, որը ստուգում է՝ արդյո՞ք տրված թվային արտահայտությունը վերջավոր է, ուստի այդ խնդրում պայմանական արտահայտության վերջնական տեսքը կլիներ.

```
if(Number.isNaN(age) == false && Number.isFinite(age)){}  
Հաշվի առնելով, որ տարիքը չի կարող լինել իրական թիվ,  
օրինակ 57.8՝ անհրաժեշտ կլինի ավելացնել նաև ամբողջ թվի  
ստուգման պայման:
```

Կիրառելով **Math.isInteger** ֆունկցիան՝ վերջնական տեսքում կունենանք.

```
if(Number.isNaN(age) == false && Number.isFinite(age) &&  
Number.isInteger(age)){}  
Այսպիսով՝ կարող է լուծվել նաև այլ մեթոդով:
```

Նախորդ ենթաբաժնում դիտարկվեց 3 թվերից մեծագույնը որոշելու խնդիրը, որի լուծման մեջ սկզբում համեմատվեցին առաջին երկու թվերը և ապա երրորդը: Կիրառելով բաղադրյալ պայման՝ խնդիրը կարող է լուծվել նաև այլ մեթոդով:

Դիտարկենք ալգորիթմը.

1. Եթե  $a > b$  և  $a > c$ , ապա  $\max = a$ ;
2. Իականակ դեպքում, եթե  $b > a$  և  $b > c$ , ապա  $\max = b$ ;
3. Իականակ դեպքում,  $\max = c$ ;

Այսպիսով՝ կոդը կլինի.

```
function mecaguyn(){
```

```
    with(document){
```

```
        var a = Number(getElementById("tiv1").value);
        var b = Number(getElementById("tiv2").value);
        var c = Number(getElementById("tiv3").value);
        var max;
        if(a > b && a > c){
            max = a;
```

```

}else if(b > a && b > c){
    max = b;
}else{
    max = c;
}
write("maximum = " + max);
}

```

#### 4.4 Տերնար օպերատոր

Ծրագրավորման մի շարք լեզուներ, այդ թվում, JavaScript -ը, թույլ են տալիս, որպեսզի պայմանական արտահայտությունը գրվի առավել կարճ ձևով, որն ընդունված է անվանել **տերնար օպերատոր**:

Դիտարկենք երկու թվերից մեծագույնի որոշման կողմը.

```

if( a > b){
    max = a;
}else{
    max = b;
}

```

Այժմ դիտարկենք նոյն խնդրի լուծումը տերնար օպերատորի կիրառմամբ.

```
var max = a > b ? a : b;
```

Դժվար չէ նկատել, որ կոդի ծավալը զգալիորեն կրճատվեց:

Այստեղ **հարցական նշանը** ցույց է տալիս այն արժեքը, որը պետք է պահել max -ի մեջ, եթե նախապես գրված պայմանն ընդունի ճշմարիտ արժեք: **Վերջակետը** ցույց է տալիս այն արժեքը, որը պետք է պահել max -ի մեջ՝ հակառակ դեպքում:

## ՀԱՐՑԵՐ ԵՎ ԱՌԱՋԱԴՐԱՆՔՆԵՐ

1. Ի՞նչ նպատակով է կիրառվում պայմանի օպերատորը:
2. Ի՞նչո՞վ է տարբերվում պարզ պայմանական արտահայտությունը բաղադրյալից:
3. Ի՞նչ է տերնար օպերատորը:
4. Բացատրել with օպերատորի դերը:
5. Բերել այնպիսի օրինակ, որտեղ անհրաժեշտ է կիրառել շղթայական պայմանի օպերատոր:
6. Բացատրել              Number.isFinite              ֆունկցիայի նշանակությունը:
7. Ինչպես կարող ենք որոշել՝ արդյո՞ք մուտքագրված թիվն ամբողջ է, թե՞ ոչ:
8. Ի՞նչ տարբերություն կա || և && օպերատորների միջև
9. Ի՞նչ է իրենից ներկայացնում Number.MAX\_VALUE -ն:
10. HTML ֆայլում ունենք 3 input -ներ և button: input դաշտերում մուտքագրվելու են 3 թվեր: Անհրաժեշտ է button -ի սեղման ժամանակ դասավորել այդ երեք թվերը աճման կարգով և վերջնական արդյունքը ցույց տալ h1 -ի մեջ:

**ԴԱՍ 5**

# **ԼՈԿԱԼ ԵՎ ԳԼՈԲԱԼ ՓՈՓՈԽԱԿԱՆԵՐ**



## ԴԱՄ 5 : ԼՈԿԱԼ ԵՎ ԳԼՈԲԱԼ ՓՈՓՈԽԱԿԱՆՆԵՐ, ՓՈՓՈԽԱԿԱՆՆԵՐԻ ՏԵՍԱՆԵԼԻՈՒԹՅՈՒՆ

### 5.1 Լոկալ և գլոբալ փոփոխականներ

Ենթադրենք անհրաժեշտ է մշակել այնպիսի ծրագիր, որը հաշվում է, թե քանի՞ անգամ է սեղմվել button -ի վրա: Յուրաքանչյուր սեղման ժամանակ h1 -ի մեջ պետք է գրել “սեղմվել է x անգամ”, որտեղ x -ը ցույց կտա անհրաժեշտ քանակը:

Դիտարկենք հետևյալ HTML կառուցվածքը

```
<h1 id="patasxan"></h1>  
<button onclick="hashvel()">Count</button>
```

Ինչպես կարող ենք նկատել, hashvel ֆունկցիան պետք է աշխատի այն ժամանակ, երբ սեղմվում է button -ը:

Մեր խնդիրն է ունենալ փոփոխական, որին նախապես կտրվի 0 արժեքը, իսկ յուրաքանչյուր սեղման ժամանակ կավելացնենք փոփոխականի արժեքը մեկով՝ ստացված արդյունքը ցույց տալով h1 -ի մեջ:

Դիտարկենք կոդը.

```
function hashvel(){  
    var qanak = 0;  
    qanak = qanak + 1; //նախկին արժեքին գումարել մեկ  
    var text = "Մկնիկը սեղմվել է " + qanak + " անգամ";  
    patasxan.innerHTML = text;  
}
```

Յուրաքանչյուր անգամ սեղմելով button -ի վրա՝ դուք կնկատեք, որ Էկրանին հայտնվում է նույն արտահայտությունը՝ «Մկնիկը սեղմվել է 1 անգամ»:

Իսկ ո՞րն է դրա պատճառը:

Ինչպես գիտենք, hashvel ֆունկցիան աշխատում է յուրաքանչյուր անգամ, եթե սեղմվում է բուտոն -ը: Ֆունկցիայի առաջին տողում գրված է var qanak = 0 արտահայտությունը, այսինքն՝ յուրաքանչյուր անգամ սեղմելիս ֆունկցիայի ներսում հայտարարվում է փոփոխական, որը ստանում է 0 արժեք: Հաջորդ տողում այդ արժեքն ավելանում է մեկով և հենց այդ 1 -ն է, որը մենք տեսնում ենք բուտոն -ը սեղմելիս:

Նշանակում է՝ յուրաքանչյուր անգամ սեղմելով, մենք վերահայտարարում ենք փոփոխականը 0 արժեքով՝ թույլ չտալով նրան աճել: Խնդիրն անմիջապես կլուծվեր, եթե փոփոխականը հայտարարվեր միայն մեկ անգամ, իսկ ֆունկցիայի ներսում փոխվեր նրա արժեքը:

Փոփոխականը, որը հայտարարված է ֆունկցիայի ներսում անվանում են լոկալ փոփոխական: Փոփոխականը, որը հայտարարված է նախքան ֆունկցիայի նկարագրությունը անվանում են գլոբալ փոփոխական տվյալ ֆունկցիայի նկատմամբ :

Մեր խնդիրն է qanak լոկալ փոփոխականը դարձնել գլոբալ այսինքն՝ այն նկարագրել նախքան ֆունկցիան սահմանելը: Այս դեպքում արդեն փոփոխականը կստանա 0 արժեք, եթե HTML էջը կրացվի առաջին անգամ, իսկ յուրաքանչյուր սեղման դեպքում նրա արժեքը կավելանա մեկով:

Դիտարկենք կոդը.

```
var qanak = 0; //գլոբալ փոփոխական
```

```
function hashvel(){
```

```
    qanak = qanak + 1;
```

```
    var text = "Մկնիկը սեղմվել է " + qanak + " անգամ";
```

```
    patasxan.innerHTML = text;
```

```
}
```



Այս դեպքում, աշխատեցնելով համակարգը, կնկատենք, որ շնորհիվ գլոբալ փոփոխականի՝ մեր խնդիրը լուծվում է, և յուրաքանչյուր սեղման ժամանակ իսկապես զանակ փոփոխականի արժեքն ավելանում է մեկով:

## ՄԿՆԻԿԸ ՍԵՂՄՎԵԼ Է 56 ԱՆԳԱՄ

COUNT

### 5.2 Փոփոխականների տեսանելիության տիրույթներ

JavaScript -ում փոփոխականները բացի լոկալ կամ գլոբալ լինելուց, բնութագրվում են նաև տեսանելիության որոշակի տիրույթներով:

Ասվածը փորձենք ներկայացնել կոնկրետ օրինակում:

```
if( 5 > 4){  
    var x = 5;  
}  
console.log(x); //5
```

Առաջին հայացքից թվում է, թե **console.log(x)** հրամանը պետք է տպի `undefined` արժեքը, քանի որ `x` փոփոխականը հայտարարված է `if` -ի բլոկի ներսում և պետք է տեսանելի լիներ միայն այնտեղ: Քանի որ պայմանական արտահայտությունը վերադարձնում է ճշմարիտ արժեք, ապա, հայտնվելով `If` -ի ներսում JS -ի ինտերպրետատորը<sup>6</sup>, հայտարարում է այդ փոփոխականը: Այսպիսով՝ `var` -ի դեպքում հայտարարված փոփոխականը տեսանելի է նաև բլոկից դուրս:

ECMAScript -ում, սկսած 6-րդ տարբերակից, JS -ը թույլ է տալիս կիրառել `let` հրամանը:

<sup>6</sup>Համակարգ, որը կարդում, թարգմանում և իրացնում է JavaScript -ի կողը Browser – ում:

Let հրամանը փոխարինում է var -ին, այս դեպքում, սակայն, հայտարարված փոփոխականը տեսանելի է միայն բլոկի ներսում:

```
if( 5 > 4){  
    let x = 5;  
}  
console.log(x); //error - 'x' is not defined
```

Այսպիսով՝ var -ով նկարագրված փոփոխականներին անվանում են ֆունկցիոնալ մակարդակի փոփոխական (function level), իսկ լեռ -ի դեպքում՝ բլոկային մակարդակի (block level):

Let -ի և var -ի մեկ այլ տարրերություն է այն, որ let -ով հայտարարված փոփոխականը այլևս չի կարող վերահայտարարվել:

```
let x = 4;  
let x = 8; //error - Identifier 'x' has already been declared
```

**5.3 Գաղափար ինկրեմենտի և դեկրեմենտի մասին**  
Ծրագրավորման լեզուներում վերագրման գործողությունը զգալիորեն տարբերվում է մաթեմատիկական վերագրումից: Օրինակ՝  $x = 2 * x$  արտահայտությունը մաթեմատիկայում գծային հավասարում է, որի լուծումն է  $x = 0$  -ն, մինչդեռ ծրագրավորման մեջ այն նշանակում է  $x$  փոփոխականի ներսում պահել դրա կրկնակի արժեքը:

```
var x = 4;  
x = 2 * x; //8
```

Ինկրեմենտ անվանում են փոփոխականի արժեքը մեկով ավելացնելու գործողությունը, իսկ դեկրեմենտ՝ 1 -ով պակասեցնելու:

```
var x = 4;  
x = x + 1; // x=5, ինկրեմենտ  
x = x - 1; // x=4, դեկրեմենտ
```

JavaScript ծրագրավորման լեզուն թույլ է տալիս ինկրեմենտի և դեկրեմենտի համար կիրառել տարբեր մեթոդներ.

```

    x = x + 1;
    x++;
    ++x;
    x += 1;
    x = x - 1;
    x--;
    --x;
    x -= 1;

```

Մասնավոր դեպքերում նաև.

$x^* = 4; x += 8; x /= 7; x \% = 2;$

Օրինակ. Եթե  $x = 4$ , ապա  $x^* = 2$  -ից հետո այն կդառնա 8, քանի որ  $x^* = 2$  գործողությունը համարժեք է  $x = x * 2$  -ին, որը նշանակում է 'x

-ի կրկնակի արժեքը պահել  $x$  -ի մեջ:

Առանձնակի հետաքրքրություն են ներկայացնում  $x++$  և  $++x$  գործողությունները: Երկուսն են ավելացնում են տրված փոփոխականի արժեքը մեկով, սակայն ունեն որոշակի տարրերություն, որը հնարավոր է նկատել միայն արտահայտության կազմության մեջ:

`var x = 4;`

`var y = x++;`  $\Rightarrow y = 4$

Այս դեպքում  $y = x++$  արտահայտությունը կազմված է երկու մասից՝  $y = x$  և  $x++$ : Նշանակում է, նախ՝  $y$  -ի մեջ կպահպանվի  $x$  -ի արժեքը՝ 4 -ը, այնուհետև  $x++$  -ով  $x$  -ը կդառնա 5:

Դիտարկենք այլ դեպք.

`var x = 4;`

`var y = ++x;`  $\Rightarrow y = 5$

Այս օրինակում իսկաքանչյան կատարվում է  $++x$  գործողությունը, որը  $x$  -ի արժեքը դարձնում է 5, ապա  $y$  -ի մեջ պահպում է  $x$  -ի արժեքը դարձնում է 5, ապա  $y$  -ի մեջ պահպում է  $x$  -ի արժեքը:

Ամփոփելով նշենք՝  $x++$  գործողությունը փոխում է  $x$  -ի արժեքը միայն հաջորդ քայլի համար, իսկ  $++x$  -ը փոխում է  $x$  -ի արժեքը ընթացիկ քայլում:

```
var x = 4;  
var y = x++ + ++x + x--; // y = 16; x = 5
```

Ինչպես նկատեցինք արտահայտության առաջին մասն ունի հետևյալ տեսքը

**y = x++**

Արդեն գիտենք, որ տվյալ դեպքում y -ի մեջ պահպում է 4, իսկ x -ը դառնում է 5: Բնական է, որ հաջորդ քայլում x=5, իսկ ++x -ի արդյունքում այն դառնում է 6:

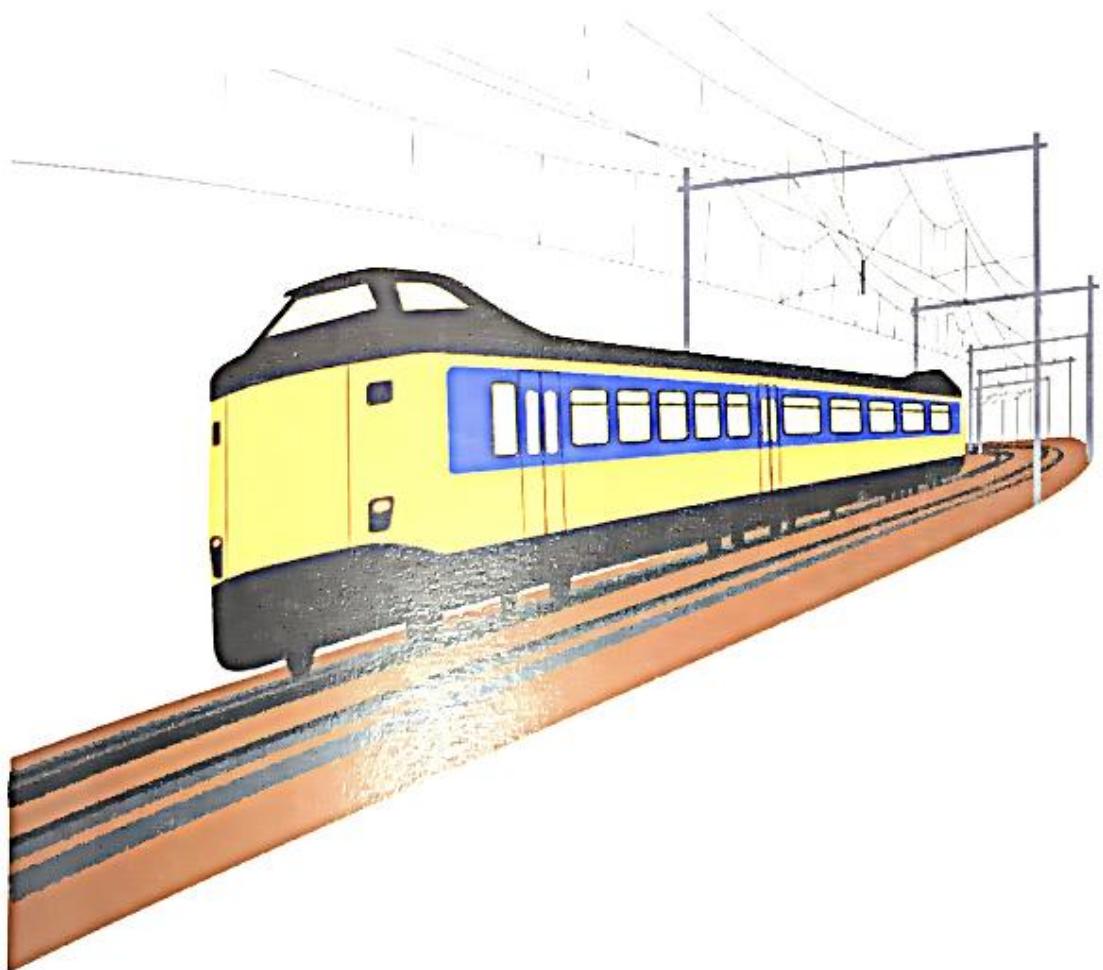
Նշանակում է՝ մեր առաջին երկու գումարելիներն են 4 -ը և 6 -ը: Վերջին գումարելին ունի x-- տեսքը: Այստեղից հետևություն՝ գործ ունենք x -ի ընթացիկ արժեքի հետ, որը 6 -ն է, իսկ արժեքի նվազումը տեղի է ունենում գործողությունից հետո: Այսպիսով՝ մեր երեք գումարելիներն են՝ 4, 6, 6 թվերը: Գումարման արդյունքը կլինի 16:

## ՀԱՐՑԵՐ ԵՎ ԱՌԱՋԱԴՐԱՆՔՆԵՐ

1. Ինչո՞վ են տարբերվում լոկալ և գլոբալ փոփոխականները միմյանցից:
2. Ի՞նչ տարբերություններ կան var -ի և let -ի միջև:
3. Ի՞նչ են իրենցից ներկայացնում ինկրեմենտը և դեկրեմենտը:
4. Ո՞րն է x++ և ++x գործողությունների միջև տարբերությունը:
5. Էկրանի վրա ավելացնել նկար, որը 200x200 չափսի է: Նկարի սեղման ժամանակ, եթե այն փոքր է (200x200), ապա նրա չափսերը դարձնել 400x400, իսկառակ դեպքում՝ կրկին 200x200:

# ԴԱՍ 6

## ՑԻԿԼԻ ՕՊԵՐԱՏՈՐՆԵՐ



## ԴԱՏ 6 : ՑԻԿԼԵՐ ԵՎ ԴՐԱՆՑ ՏԵՍԱԿՆԵՐԸ

### 6.1 Ցիկլի գաղափարը

Նախորդ ենթագլուխներում ծանոթացանք պայմանի օպերատորին և սովորեցինք, որ պայմանական արտահայտությունը թույլ է տալիս կատարել որևէ գործողություն որոշակի պայմանի առկայության դեպքում: Ծրագրավորման լեզվի մեկ այլ կարևոր գաղափար է ցիկլը: Ցիկլը կազմակերպում է մեկ կամ մի քանի գործողությունների անընդհատ կրկնություն, քանի դեռ տրված պայմանն ընդունում է ճշմարիտ արժեք:

JavaScript -ում առկա են ցիկլերի տարրեր տեսակներ, որոնցից առաջին երեքին կծանոթանանք այս դասի ընթացքում: Այդ տեսակներն են՝ նախապայմանով, հետպայմանով և պարամետրով ցիկլի օպերատորները:

### 6.2 Նախապայմանով և հետպայմանով ցիկլեր

Նախապայմանով ցիկլի օպերատորը կազմակերպում է մեկ կամ մի քանի գործողությունների կրկնություն, եթե նախապես տրված պայմանն ունի ճշմարիտ արժեք:

Ընդհանուր առմամբ օպերատորն ունի հետևյալ տեսքը.

**while(պայման){}**

//գործողություններ, որոնք կատարվեն անընդհատ, քանի դեռ պայմանը ճիշտ է

}

Ենթադրենք անհրաժեշտ է առաջի բոլոր երկնիշ թվերը: Դիտարկենք ալգորիթմը:

1. հայտարարենք  $x$  փոփոխական, որին վերագրենք առաջին երկնիշ թիվը՝ 10,
2. գրենք նախապայմանով ցիկլ, որը կկատարվի քանի դեռ  $x < 100$  -ից պայմանն ունի ճշմարիտ արժեք,
3. ցիկլի ներսում տպենք  $x$  -ի արժեքը,
4. ավելացնենք  $x$  -ի արժեքը մեկով

```

var x = 10;
while(x < 100){
    console.log(x);
    x++;
}

```

Մեր օրինակում x փոփոխականին է վերագրվել 10 արժեքը: Քանի որ  $x < 100$  պայմանը բավարարվել է, ապա աշխատել է `console.log(x)` հրամանը, այնուհետև ավելացել x արժեքը մեկով: Այս դեպքում արդեն x -ի արժեքը դառնում է 11, որն իր հերթին կրկին փոքր է 100 -ից, հետևաբար տեղի է ունենում նոյն գործընթացը՝ `console.log(x)` և `x++`: Այս դեպքում x < 100 նախապայմանն ունի ճշմարիտ արժեք: Եթե  $x = 100$  արժեքը, ինչը հանգեցնում է նրան, որ  $x < 100$  ենք պայմանն այլևս ճշմարիտ չեն, հետևաբար ցիկլը դադարում է: Հետպայմանով ցիկլի օպերատորը կազմակերպում է մեկ կամ մի քանի գործողությունների կրկնություն՝ սկզբում կատարում գործողությունը, ապա ստուգում պայմանը: Հետպայմանով ցիկլի օպերատորի ընդհանուր տեսքը հետևյալն է.

```

do{
    //գործողություններ, որոնք կկատարվեն, քանի ոեն
    պայմանը ճիշտ է
}
while(պայման);

```

ԽՆԴԻՐ - գտնել բոլոր երկնիշ թվերի գումարը՝ օգտագործելով հետպայմանով ցիկլի օպերատորը:

Նախ դիտարկենք խնդրի լուծման ալգորիթմը.

1. հայտարել ցումար փոփոխական, որին վերագրել 0,
2. հայտարել x փոփոխական, որին վերագրել առաջին երկնիշ թիվը՝ 10,

3. հետպայմանով ցիկլի ներսում կատարել 4 և 5 գործողությունները՝  $x < 100$  հետպայմանի առկայության դեպքում,
4. ավելացնել `gumar` փոփոխականին  $x$ -ի արժեքը,
5. ավելացնել  $x$ -ի արժեքը մեկով

```
let x = 10; let gumar = 0;
do{
    gumar += x;
    x++;
}
while( x < 100);
console.log(gumar); // 4905
```

Դիտարկենք նոյն խնդրի լուծումը նախապայմանով ցիկլի դեպքում.

```
let x = 10; let gumar = 0;
while(x < 100){
    gumar += x;
    x++;
}
console.log (gumar); // 4905
```

Դժվար չէ նկատել, որ երկու դեպքում էլ պատասխանը նոյնն է: Իսկ ո՞րն է նախապայմանով և հետպայմանով ցիկլերի միջև տարբերությունը:

Ինչպես գիտենք, նախապայմանով ցիկլի դեպքում պայմանը ստուգվում է նախապես, ապա կատարվում գործողությունները, մինչդեռ հետպայմանով ցիկլը կատարում է գործողություններ, ապա ստուգում պայմանը: Այստեղից հետևում է, որ եթե նոյնիսկ պայմանը սխալ է, հետպայմանով ցիկլը առնվազն մեկ անգամ կկատարի տվյալ գործողությունը:

Օրինակ:

```

let x = 4;
do{
    x--;
}
while(x > 5);
console.log(x); //3

```

Այստեղ գրված  $x > 5$  պայմանը վերադարձնում է բացասական արժեք, քանի որ 4 -ը մեծ չէ 5 – ից, սակայն հետպայմանով ցիկլի դեպքում գործողություններն առնվազն մեկ անգամ կատարվում են, ուստի տեղի է ունենում  $x--$  գործողությունը, որի արդյունքում  $x$  փոփոխականը ստանում է 3 արժեքը:

### 6.3 Պարամետրով ցիկլի օպերատոր

Պարամետրով ցիկլի օպերատորն ամենահաճախ կիրառվող օպերատորն է: Ընդհանուր կառուցվածքը շատ պարզ է:

```

for(սկզբնարժեք; պայման; քայլ){
    //գործողություններ
}

```

**Սկզբնարժեք** բաժնում հայտարարվում է ցիկլի պարամետրը և տրվում է նրա նախնական արժեքը: **Պայման** բաժնում գրվում է այն պայմանը, որի առկայության դեպքում ցիկլը պետք է կատարվի, իսկ **քայլ** բաժնում ինկրեմենտի կամ դեկրեմենտի օգնությամբ որոշում ենք ցիկլի պարամետրի հերթական քայլը: Ասկան ավելի պարզ ներկայացնելու համար փորձենք էկրանին տպել 0-20 հատվածի բոլոր թվերը.

```

for(let i = 0; i <= 20; i++){
    document.write(i + "<br>");
}

```

Այստեղ ցիկլի առաջին քայլում հայտարարվում է  $i$  փոփոխականը, որին տրվում է 0 արժեքը: Քանի որ  $i <= 20$  պայմանը բավարարվում է, ապա, հայտնվելով ցիկլի մարմնում, կատարվում է `document.write` իրամանը, որն էկրանին տպում է

Օթիվը: Այսուհետև տեղի է ունենում  $i++$  գործողությունը, որն ավելացնում է ի պարամետրի արժեքը մեկով, և ցիկլ կատարվում է կրկին:

Գործողությունների այս բազմությունը կատարվում է, քանի դեռ  $i \leq 20$  պայմանն ունի ճշմարիտ արժեք:

Ենթադրենք անհրաժեշտ է տպել 0-20 հատվածի միայն զույգ թվերը:

```
for(let i = 0; i <= 20; i+=2){  
    document.write(i + "<br>");  
}
```

Համեմատելով նախկին օրինակի հետ՝ կհամոզվենք, որ միակ տարբերությունը ցիկլի քայլում գրված  $i+=2$  արտահայտությունն է, որը յուրաքանչյուր անգամ ցիկլի քայլը ավելացնում է 2 - ով, ինչի արդյունքում է կրանին հայտնվում են 0,2,4 ... 20 թվերի շարքը:

Այժմ փորձենք պարամետրով ցիկլը կիրառել հայտնի FizzBuzz խնդրի լուծման նպատակով:

**ԽՆԴՐԻՑ: Տպել 1-99 բոլոր**

**թվերը:** Եթե հերթական թիվը անմնացորդ բաժանվում է 5 -ի, ապա նրա փոխարեն տպել Fizz, հակառակ դեպքում, եթե այն բաժանվում է 3 -ի անմնացորդ, ապա նրա փոխարեն գրել Buzz, իսկ, եթե թիվը միաժամանակ բաժանվում է և 3 -ի և 5 -ի անմնացորդ, ապա նրա փոխարեն գրել FizzBuzz տեքստը, բոլոր մնացած դեպքերում պարզապես տպել տվյալ թիվը:



```

for(let i = 1; i <= 99; i++){
    if(i % 3 == 0 && i % 5 == 0){
        document.write("fizbuzz <br>");
    }else if(i % 5 == 0){
        document.write("fiz <br>");
    }else if(i % 3 == 0){
        document.write("buzz <br>");
    }
    else{
        document.write( i + "<br>");
    }
}

```

1  
 2  
 buzz  
 4  
 fiz  
 buzz  
 7  
 8  
 buzz  
 fiz  
 11  
 buzz  
 13  
 14  
 fizbuzz  
 16  
 17  
 buzz

Վերլուծենք կոդը մեկնաբանող ալգորիթմը.

1. Դիտարկել 1-99 թվերը յուրաքանչյուր թվի համար կատարել քայլ 2 -ը,
2. Եթե թիվը բաժանվում է 3-ի և 5 -ի անմնացորդ, ապա տպել fizbuzz, հակառակ դեպքում անցնել քայլ 3 -ին,
3. Եթե թիվը բաժանվում է 5 -ի անմնացորդ, ապա տպել fiz, հակառակ դեպքում անցնել քայլ 4 -ին,
4. Եթե թիվը բաժանվում է 3 -ի անմնացորդ, ապա տպել buzz, հակառակ դեպքում անցնել քայլ 5 -ին,
5. Տպել թվի արժեքը

## 6.4 Անվերջ ցիկլեր, ցիկլի ընդհատում

Ինչպես գիտենք, նախապայմանով, հետպայմանով և պարամետրով ցիկլի օպերատորները կազմակերպում են մեկ կամ մի քանի գործողությունների կրկնություն՝ որոշակի պայմանների առկայության դեպքում: Իսկ ի՞նչ տեղի կունենա, եթե նախապես սահմանված ցիկլի պայմանը մշտապես ունենա ճշմարիտ արժեք:



```

Օրինակ.
while(1 == 1){
    console.log("ok");
}

```

Այս օրինակում նախապես գրված է `1==1` պայմանը, որը միշտ վերադարձնում է ճշմարիտ արժեք, իետևաբար բնական է, որ անընդհատ կկատարվի `console.log` հրամանը: Ցիկլի այս տեսակն ընդունված է անվանել **անվերջ ցիկլ**:

Անվերջ ցիկլեր կառաջանան նաև հետևյալ դեպքերում.

```
while(true){
```

գործողություններ

```
}
```

```
for(; ;){
```

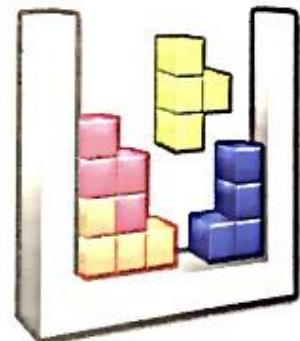
գործողություններ

```
}
```

Իսկ ո՞ր իրավիճակներում է անհրաժեշտ կիրառել անվերջ ցիկլեր:

**Համակարգչային**                    **ամենահայտնի**  
 խաղերից է Tetris -ը, որում խաղաքարերն անվերջ իջնում են վերևից քանի դեռ խաղը չի ավարտվել: Նման իրավիճակում աշխատում է անվերջ ցիկլը, իսկ երբ տեղի է ունենում այնպիսի գործընթաց, որուած պետք է ավարտվի, այդ ժամանակ աշխատում են ցիկլի ընդհատման օպերատորները, որոնց օգնությամբ ցիկլը դադարում է: JavaScript -ում, ընդհանուր առմամբ, գոյություն ունեն ցիկլի ընդհատման երկու եղանակներ: Առաջինը՝ մասնակի ընդհատում, երկրորդը՝ լրիվ ընդհատում: Մասնակի ընդհատման դեպքում բաց է թողնվում ցիկլի միայն տվյալ քայլը, սակայն մյուս քայլերը ցիկլում շարունակվում են: Լրիվ ընդհատման դեպքում ցիկլը դադարում է ամբողջությամբ:

Ցիկլի մասնակի ընդհատումը տեղի է ունենում `continue` օպերատորի օգնությամբ, լրիվ ընդհատումը՝ `break`:



ԽՆԴԻՐ: Տպել 30 -ից 50 և 60-70 հատվածի թվերը մեկ ցիկլի  
կիրառմամբ:

```
for(let i = 30; i <= 70; i++){
    if((i >= 30 && i <= 50) || (i >= 60 && i <= 70)){
        console.log(i);
    }else{
        continue;
    }
}
```

Մեր օրինակում ցիկլի ներսում դիտարկվում են 30-70 հատվածի թվերը: Պայմանական արտահայտությունը ստուգում է՝ արդյո՞ք հերթական թիվը պատկանում է 30-ից 50 կամ 60-70 հատվածին: Պայմանի բավարարման դեպքում console պատուհանում ելքագրվում է տրված թվի արժեքը, հակառակ դեպքում ցիկլի մասնակիորեն ընդհատվում է, այսինքն բոլոր այն քայլերը, որոնք չեն բավարարում պայմանին (50-60 հատվածի թվերը) ավտոմատ բաց են թողնվում ցիկլի կողմից:

ԽՆԴԻՐ: Տպել առաջին երկնիշ թիվը, որը 17 -ով բազմապատկելիս ստացվում է 300-ից մեծ թիվ:

```
for(let i = 10; i <= 99; i++){
    if(17 * i >= 300){
        console.log(i); //18
        break;
    }
}
```

Տվյալ օրինակում պարամետրով ցիկլի օգնությամբ դիտարկվել են բոլոր երկնիշ թվերը: Ցիկլի ներսում ստուգվում է պայման՝ արդյո՞ք տրված թիվը 17-ով բազմապատկելիս, ստացված արդյունքը կգերազանցի 300 -ը: Եթե պայմանը բավարարվում է console պատուհանում ելքագրվում է տրված թվի արժեքը, անմիջապես դրանից հետո կատարվում է break -ը, որը դադարեցնում է ցիկլը: Այս դեպքում ելքագրվում է միայն 18 թիվը: Եթե break -ը բացակայեր էկրանին կհայտնվեր 18 -ից

սկսած բոլոր երկնիշ թվերը, քանի որ 18 -ից հետո ցանկացած  
թիվ բազմապատկելով 17-ով կստանանք 300 -ից մեծ թիվ:

**6.5 Ներկառուցված ցիկլ**  
Եթե ցիկլի մեջ գրենք մեկ այլ ցիկլ, ապա կստանանք ցիկլերի  
համակարգ, որն ընդունված է անվանել **ներկառուցված ցիկլ**:  
Ներկառուցված ցիկլի յուրաքանչյուր քայլում ամբողջությամբ  
կատարվում է ներքին ցիկլը: Այսպիսով, եթե առաջին ցիկլն ունի  
n քայլ, երկրորդը՝ m, ապա յուրաքանչյուր i -րդ քայլում ( $i=0, n$ )  
կատարվում է  $j = 0, m$  քայլերն ամբողջությամբ:  
`for(var i = 0; i < n; i++){  
 for(var j = 0; j < m; j++){  
 //գործողություններ  
 }  
}`

Հետևաբար վերջնական արդյունքում կատարվող քայլերի  
քանակը կլինի  $n \times m$ :  
Ներկառուցված ցիկլի օրինակ է ստորև բերվածը.

```
for(var i = 0; i < 3; i++){  
    for(var j = 0; j < 3; j++){  
        console.log("i=" + i, "j=" + j);  
    }  
}
```

Այս դեպքում console պատուհանում կունենանք  
նկարում ցույց տրված արդյունքը: Դժվար չէ  
նկատել, որ i ցիկլի յուրաքանչյուր քայլում  
կատարվում է  $j$  ցիկլն ամբողջությամբ, և միայն  
ներքին ցիկլի ավարտից հետո է փոխվում  
գլխավոր ցիկլի արժեքը:

i=0 j=0
i=0 j=1
i=0 j=2
i=1 j=0
i=1 j=1
i=1 j=2
i=2 j=0
i=2 j=1
i=2 j=2

Ենթադրենք անհրաժեշտ է աստղանիշների օգնությամբ էկրանին պատկերել ստորև բերված ուղղանկյուն եռանկյունը:

\*  
\* \*  
\* \* \*  
\* \* \* \*  
\* \* \* \* \*

Ինչպես կարող ենք նկատել, եռանկյունը կազմված է տողերից, որի 0-րդ տողում առկա է 1 աստղանիշ, 1-ում՝ 2 և այն: Կատարելով ընդհանրացում՝ կնկատենք որ i-րդ տողում աստղանիշների յ քանակը հավասար է i+1: Օգտագործելով ստացված առնչությունը՝ կարող ենք հեշտությամբ ստանալ եռանկյան պատկերը՝ ներկառուցված ցիկլի միջոցով: Ցիկլը կատարվում է 5 անգամ և յուրաքանչյուր i -րդ քայլում այն տեղադրում է i+1 հատ աստղանիշ:

Դիտարկենք կոդը.

```
for(let i = 0; i <= 5; i++){
    for(let j = 0; j < i+1; j++){
        document.write("* ");
    }
    document.write("<br>");
}
```

Ցիկլի առաջին քայլում, եթե i = 0, ներքին ցիկլը կատարվում է միայն մեկ անգամ (0, i+1), արդյունքում՝ էկրանին հայտնվում է միայն մեկ աստղանիշ: Աստղանիշների հաջորդ խումբը էկրանին է հայտնվում է, եթե i = 1, այդ դեպքում յ ցիկլը կատարվում է 2 անգամ:

Այսպիսով՝ i ցիկլի ամեն քայլում յ ցիկլը կատարվում է i+1 անգամ: Նկատենք, որ աստղանիշների խմբերն իրարից առանձնացված են <br> տեգի օգնությամբ, որպեսզի յուրաքանչյուր խումբ լինի հաջորդ տողում և պատկերը նման լինի եռանկյան:

Առանձնակի հետաքրքրություն է ներկայացնում ցիկլի  
ընդհատման գործընթացը ներկառուցված ցիկլերում:

Ենթադրենք ուսե՞ք 5 ցիկլերից կազմված համակարգ:  
Հետաքրքրիր է, թե ի՞նչ տեղ կունենա, եթե օրինակ որոշակի  
պայմանի առկայության դեպքում break հրամանը կիրառենք  
ներքին ցիկլում: Պարզվում է՝ ընդհատումը տեղի կունենա միայն  
վերջին ցիկի համար, իսկ մնացյալը կշարունակեն իրենց  
գործողությունները:

Այսպիսով՝ ներկառուցված ցիկլերի դեպքում որևէ ցիկլի ներսից  
մեկ այլ ցիկլ ընդհատելու համար օգտվում ենք label -ներից: label  
-ը հնարավորություն է տալիս որոշակի անուն վերագրել ցիկլին:

Օրինակ՝

a:

for(...){

b:

for(...){

c:

for(...){

if(որևէ պայման) {

break b;

}

}

Մեր օրինակում ցիկլերի անուններն են՝ a,b,c: Վերջին ցիկլում  
որոշակի պայմանի առկայության դեպքում ընդհատվում է ոչ թե  
c ցիկլը, այլ b -ն՝ break b; հրամանով:

## ՀԱՐՑԵՐ ԵՎ ԱՌԱՋԱԴՐԱՆՔՆԵՐ

- Ի՞նչ նպատակով են կիրառվում ցիկլերը:
1. Տիկի ի՞նչ օպերատորներ գիտեք:
  2. Ինչո՞վ են միմյանցից տարբերվում նախապայմանով և հետպայմանով ցիկլի օպերատորները:
  3. Ի՞նչ եղանակներով է հնարավոր ընդհատել ցիկլը:
  4. Ի՞նչ է նշանակում անվերջ ցիկլ:
  5. Ի՞նչ է իրենից ներկայացնում ներկառուցված ցիկլը:
  6. Ի՞նչպե՞ս կարելի է ընդհատել ներկառուցված ցիկլը:
  7. Input դաշտում մուտքագրվում է թիվ, button -ի սեղման ժամանակ, ցիկլի միջոցով գտնել տրված թվի թվանշանների գումարը և արտածել այն:
  8. Input դաշտում մուտքագրվում է թիվ, button -ի սեղման ժամանակ, ցիկլի օգնությամբ պարզել՝ արդյո՞ք տրված թիվը պարզ է, թե ոչ: Թիվը կոչվում է պարզ, եթե այն անմնացորդ բաժանվում է միայն իր և 1-ի վրա:
  9. Input դաշտում մուտքագրվում է թիվ, button -ի սեղման ժամանակ, ցիկլի օգնությամբ պարզել՝ արդյո՞ք տրված թիվը պարզ է, թե ոչ: Թիվը կոչվում է պարզ, եթե այն անմնացորդ բաժանվում է միայն իր և 1-ի վրա:
  10. a,b,c թվերը կանվանենք այութագորասյան եռյակ, եթե  $c^2=a^2+b^2$

Ներկառուցված ցիկլի օգնությամբ տպել 1-99 հատվածի բոլոր այութագորասյան եռյակները՝ առանց կրկնվող տվյալների:

11. Input դաշտերում մուտքագրվում են x,y թվերը: Տիկի օգնությամբ հաշվել, թե բանկում ներդրված x գումարը, քանի ամիս հետո կգերազանցի 200.000 դրամը, եթե յուրաքանչյուր ամիս գումարն ավելանում է y% -ով:
12. Ֆիբոնաչչի հաջորդականություն կոչվում է թվերի շարքը, որի առաջին երկու անդամները 1 են, իսկ յուրաքանչյուր հաջորդը՝ նախորդ երկուսի գումարը: Հաջորդականության առաջին անդամներն են՝ 1,1,2,3,5,8....: Տիկի օգնությամբ տպել առաջին 50 ֆիբոնաչչի թվերը:

13. Ներկառուցված ցիկլի օգնությամբ նկարել ստորև  
բերված պատկերները

I.

\*\*\*\*\*

\*\*\*\*

\*\*\*

\*\*

\*

II.

\*

\*\*

\*\*\*

\*\*\*\*

\*\*\*\*\*

\*\*\*\*\*

III.

1

22

333

4444

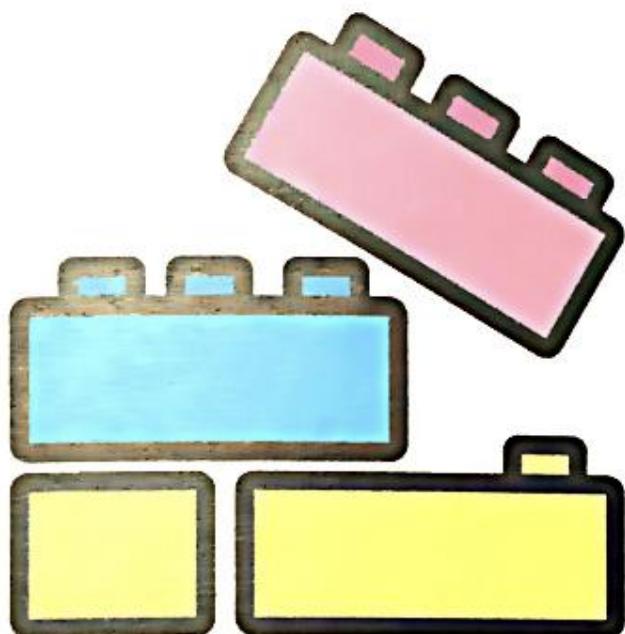
55555

IV.

\* \* \* \* \* \* \* \*  
\* \* \* \* \* \* \* \*  
\* \* \* \* \* \* \* \*  
\* \* \* \* \* \* \* \*  
\* \* \* \* \* \* \* \*

# ԴԱՍ 7

## ՉԱՆԳՎԱՑՆԵՐ



## ԴԱՍ 7: ԶԱՆԳՎԱԾՆԵՐ

**7.1 Զանգվածի գաղափարը**  
Զանգվածնշանակում է տվյալների խումբ, որը բնութագրվում է հստակ անունով. և, որի յուրաքանչյուր տարր պետք է տարրերվի մյուսներից հերթական համարով: Ընդհանրապես զանգվածների հետ կատարվող աշխատանքների վրա: Կայքում գտնվող ցանկացած տվյալների խումբ իրենից ներկայացնում է զանգված: Լրատվական կայքում կարող է լինել նորությունների նկարների զանգված: Բնական է, որ զանգվածն ունի առանցքային դեր ծրագրավորման գործընթացում:  
Զանգվածի օրինակ կարող է հանդիսանալ անունների հետևյալ խումբը.  
["Anna", "Hayk", "David", "Miqayel", "Narek"]

0      1      2      3      4

Ինչպես կարող ենք նկատել զանգվածի յուրաքանչյուր տարր տարրերվում է մյուսից հերթական համարով:  
Ծրագրավորման JavaScript լեզվում զանգվածի տարրերի համարակալումը սկսվում է 0 -ից:  
Եվ չնայած նրան, որ մեր զանգվածում առկա է 5 տարր, այնուամենայնիվ մենք չունենք 5-րդ տարրը, իսկ վերջին տարրի հերթական համարն է 4 -ը:

**7.2 Զանգվածի հայտարարումը**  
JavaScript ծրագրավորման լեզվում զանգված կարելի է հայտարարել [ ] սիմվոլների օգնությամբ:

Օրինակ՝  
`var x = [];`

Այս դեպքում x -ն իրենից ներկայացնում է դատարկ զանգված:

Զանգված կարելի է նկարագրել նաև արժեքավորելով նրա  
տարրերը.  
Օրինակ.  
`var x = ["Hayk", "Lusine", "Narek", "Arman"];`

Մեր օրինակում զանգվածն ունի 4 տարր: Զանգվածի  
տարրերին դիմելու համար անհրաժեշտ է գրել զանգվածի  
անունը և քառակուսի փակագծերում նշել տարրի ինդեքսը,  
օրինակ `x[0]` արտահայտությունը կնշանակի `x` զանգվածի 0-րդ  
տարր:

Ենթադրենք ցանկանում ենք զանգվածի բոլոր տարրերը տպել  
էկրանին: Այս դեպքում կարող ենք կատարել հետևյալը:

```
console.log(x[0]);
console.log(x[1]);
console.log(x[2]);
console.log(x[3]);
```

Խնդիրը զգալիորեն կբարդանար, եթե զանգվածը պարունակեր  
ոչ թե 4 այլ օրինակ՝ 4000 տարր:

Դժվար չէ նկատել, որ մեր դիտարկած կոդում փոփոխվող  
տարրը միայն ինդեքսն է, մնացյալ արտահայտությունն  
ամբողջությամբ կրկնվում է, ուստի կարող ենք կիրառել  
պարամետրով ցիկլ, որի ի պարամետրը ցույց կտա զանգվածի  
հերթական տարրի համարը:

```
for(let i = 0; i < 4; i++){
    console.log(x[i]);
}
```

Մեր օրինակում ցիկլը կատարվում է `i = 0, 1, 2, 3` քայլերի համար,  
որտեղ 4 -ը ցույց է տալիս զանգվածի տարրերի քանակը:  
Հաճախ կարող են լինել իրավիճակներ, երբ հայտնի չէ  
զանգվածի տարրերի քանակը: Այս դեպքում կարող ենք ցիկլը  
գրել հետևյալ կերպ.

```
for(let i = 0; i < x.length; i++){
    console.log(x[i]);
}
```

Այստեղ `x.length` հատկությունը վերադարձնում է `x` զանգվածուառկա տարրերի քանակը, որը մեր օրինակում 4 է, հետևաբար `i < x.length` արտահայտությունը համարժեք է `i < 4` արտահայտությանը:

Իսկ ինչո՞ւ է պետք կիրառել զանգվածներ:

Ենթադրենք ծրագրավորման JavaScript լեզվում բացակայում է զանգվածի գաղափարը, այս դեպքում անունների տվյալները պահելու համար կարիք կլիներ հայտարարել փոփոխականներ:



```
var name1 = "Hayk";
var name2 = "Lusine";
var name3 = "Narek";
var name4 = "Arman";
```

Կարծես թե ամեն ինչ կարգին է, մինչդեռ այս դեպքում այլև անհնար է կիրառել ցիկլի գաղափարը և կատարել գործողություններ տվյալների հետ: Բացի այդ, զանգվածի դեպքում մենք տարրեր տարրերի դիմում ենք մեկ անունով, ինչը զգալիորեն հեշտացնում է ծրագրավորման գործընթացը:

### 7.3 Զանգվածին տարրերի ավելացումը և հեռացումը

Զանգվածին տարրեր ավելացնելու համար կիրառվում է `push` ֆունկցիան, որը ավելացնում է իրեն փոխանցված տարրը կամ տարրերը զանգվածին՝ սկսած վերջին դիրքից:

Օրինակ.

```
var x = ["Hayk", "Narek"];
x.push("Aram"); // ["Hayk", "Narek", "Aram"];
x.push("Armine", "Anna"); // ["Hayk", "Narek", "Aram", "Armine", "Anna"];
```

Զանգվածին տարրեր կարելի է ավելացնել նաև `unshift` ֆունկցիայի օգնությամբ, վերջինս տարրերի ավելացումը կազմակերպում է սկզբնական դիրքից:

Օրինակ՝

```
var x = ["Hayk", "Narek"];
x.unshift("Aram"); // ["Aram", "Hayk", "Narek"];
x.unshift("Armine", "Anna"); // ["Armine", "Anna", "Aram", "Hayk",
"Narek"];
```

Զանգվածից տարրերի հեռացումը կարելի է կազմակերպել `pop` և `shift` ֆունկցիաների կիրառմամբ: `pop` ֆունկցիան հեռացնում է զանգվածի վերջին տարրը, իսկ `shift` -ը առաջին:

Օրինակ.

```
var x = ["Armine", "Anna", "Aram", "Hayk", "Narek"];
x.pop(); // ["Armine", "Anna", "Aram", "Hayk"];
x.shift(); // ["Anna", "Aram", "Hayk"];
```

Զանգվածի բոլոր տարրերը միաժամանակ ջնջելու համար կարող ենք կիրառել 3 մեթոդ.

```
var x = [3,4,5,6,2,3,4,8];
// մեթոդ 1
x = []
//մեթոդ 2
x.length = 0;
//մեթոդ 3
while(typeof x[0] != "undefined"){
    x.shift();
}
```



Առանձնակի հետաքրքրություն է

ներկայացնում `splice` ֆունկցիան, որի օգնությամբ կարող ենք տարրեր ավելացնել կամ հեռացնել զանգվածի կամայական դիրքում:

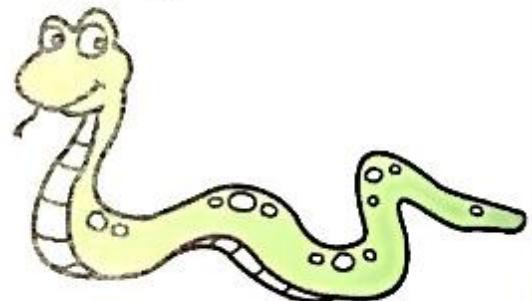
`splice(a,b,...c)` դեպքում `a` – ն ցույց է տալիս ինդեքսը, որտեղից սկսված պետք է ավելացվի կամ հեռացվի տարրեր: `b` –ն ցույց է տալիս ջնջվող տարրերի քանակը, իսկ `c` –ն, որը պարտադիր

պարամետր չէ, իրենից ներկայացնում է ավելացվող տարրերի թվարկումը:

Օրինակ.

```
var animals = ["Dog", "Cat", "Mouse", "Dolphin"];
animals.splice(1,2); // ջնջել animals[1] դիրքից 2 տարր
console.log(animals); // ["Dog", "Dolphin"]
```

Սկսած 3-րդ պարամետրից splice ֆունկցիան ավելացնում է բոլոր իրեն փոխանցված տարրերը զանգվածի անհրաժեշտ դիրքում: Ենթադրենք անհրաժեշտ է մեր զանգվածում Mouse -ից հետո ավելացնել նաև Snake և Eagle տարրերը:



```
var animals = ["Dog", "Cat", "Mouse", "Dolphin"];
animals.splice(3,0,"Snake", "Eagle");
console.log(animals); // ["Dog", "Cat", "Mouse", "Snake",
"Eagle", "Dolphin"]
```

Զանգվածին տարրերի ավելացումը կարող է տեղի ունենալ նաև ինդեքսին դիմելու եղանակով:

Օրինակ՝

```
var x = [];
x[0] = 8;
x[1] = 4;
```

Այս դեպքում, սակայն, ինդեքսավորման հերթականությունը պարտադիր չէ ապահովել:

Օրինակ՝

```
var x = [];
x[54] = 8;
```

Ինչպես կարող ենք նկատել, մենք արժեքավորեցինք զանգվածի միանգամից 54 -րդ տարրը: Այս դեպքում, զանգվածում առկա տարրերի քանակը, իհարկե, հավասար է 1 -ի, սակայն length հատկությունը, այնուամենայնիվ մեզ վերադարձնում է 55:

```
console.log(x.length); //55
```

Պատճառ այն է, որ JavaScript -ը մինչ `x[54]` -ն ընկած տարրերը համարում է դատարկ՝ empty տարրեր և նրանց արժեքը հանդիսանում է `undefined` -ը:

```
console.log(x[28]); // undefined
```

#### 7.4 Զանգվածում տարրերի որոնումը

Հանգվածում տարրերի որոնման համար կիրառվում են `indexOf` և `lastIndexOf` ֆունկցիաները:

Երկու ֆունկցիաներն ել վերադարձնում են զանգվածում տվյալ տարրի հերթական համարը: Իսկ եթե տվյալ տարրը զանգվածում բացակայում է, ապա ֆունկցիաները վերադարձնում են `-1` արժեքը:

`IndexOf` ֆունկցիան զանգվածում տարրերի որոնումը կազմակերպում է սկզբնական դիրքից, `lastIndexOf` -ը՝ վերջնական:

Օրինակ՝

```
var x = [4,3,2,3,3,8];
var y1 = x.indexOf(3);
var y2 = x.lastIndexOf(3);
console.log(y1); // 1
console.log(y2); // 4
```

```
var y3 = x.indexOf(58);
```

```
console.log(y3); // -1 քանի որ 58 տարրը բացակայում է
```

#### 7.5 Պարզագույն թվային ալգորիթմներ

Ենթադրենք ունենք զանգված, որի տարրերը այն ապրանքների գներն են, որը գնել է X մարդը: Անհրաժեշտ է գտնել զանգվածի տարրերի գումարը, որը ցույց կտա, թե որքան գումար պետք է վճարի գնորդը:

Խնդիրը լուծենք հետևյալ ալգորիթմի համաձայն.

1. հայտարարենք `gumar` փոփոխական, որին տանք 0 արժեք,

2. ցիկլի օգնությամբ դիտարկենք զանգվածի տարրերը,
3. ավելացնենք յուրաքանչյուր տարրի արժեքը `gumar`-ին,
4. ցիկլից դուրս տպենք `gumar` -ի արժեքը

```
var zangvac = [120, 360, 450, 280, 190, 200];
var gumar = 0;
for(var i = 0; i < zangvac.length; i++){
    gumar += zangvac[i];
}
document.write(gumar); //1600
```

Ինչպես կարող ենք տեսնել մենք ունենք `gumar` փոփոխականը, որի արժեքը 0 է: Առաջին քայլում, եթե  $i = 0$ , `gumar` -ին ավելանում է `zangvac[0]` -ն, մեր դեպքում՝ 120 -ը, որի արդյունքում ունենում ենք `gumar = 120` արդյունքը: Հաջորդ քայլում առկա 120 -ին ավելանում է `zangvac[1]` -ը, այսինքն՝ 360 -ը: Ցիկլի ավարտին բոլոր տարրերի արժեքներն ավելացված են լինում `gumar` փոփոխականին, ինչի արդյունքում ունենում ենք `gumar = 1600` արժեքը, որը ցույց է տալիս զանգվածի տարրերի գումարը:

Ենթադրենք ունենք մեքենաների զանգված և ցանկանում ենք գտնել ամենահեշտան մեքենայի տվյալները:

Խնդրի լուծումը հանգեցնում է զանգվածում փոքրագույն տարրի որոնմանը: Մեզ տրված է մեքենաների գների զանգվածը:

Ստորև բերված է խնդրի լուծման ալգորիթմը.



1. Հայտարարել ուն փոփոխական նրան վերագրել 0-ոդ մեքենայի գինը
2. Ցիկլի օգնությամբ դիտարկենք 1-ից N մեքենաները, որտեղ N -ը մեքենաների քանակն է

3. Յուրաքանչյուր օրդ մեքենայի գինը համեմատել տես -ի հետ, եթե մեքենայի գինն ավելի փոքր է, քան տես -ը, ապա անցնել քայլ 4 -ին
4. Min -ի մեջ պահել ընթացիկ մեքենայի գինը
5. տպել տես -ի արժեքը

```

var zangvac = [2500, 400, 700, 1800, 200, 7000];
var min = zangvac[0];
for(var i = 1; i < zangvac.length; i++){
  if( zangvac[i] < min){
    min = zangvac[i];
  }
}
document.write(min); //200

```

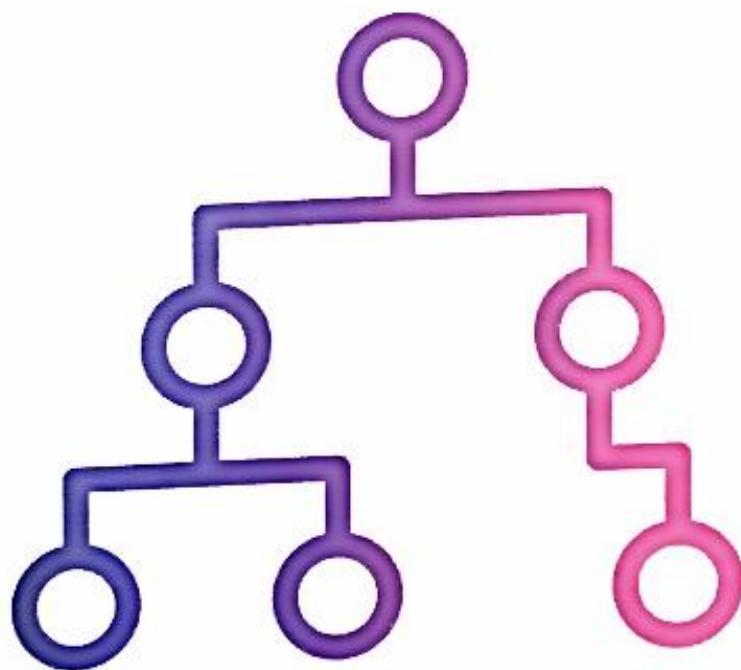
Գրված կոդի արդյունքում, min փոփոխականի մեջ պահպանվում է zangvac[0] -ն, այսինքն՝ 2500: Ցիկլի առաջին քայլին ստուգվում է՝ արդյո՞ք zangvac[1] < min: Քանի որ  $400 < 2500$ -ից, ապա min -ի մեջ պահպանվում է 400 -ը: Նույն գործողությունը կրկնվում է մնացյալ քայլերի դեպքում: Արդյունքում min -ի մեջ ունենում ենք 200 -ը:

## ՀԱՐՑԵՐ ԵՎ ԱՌԱՋԱԴՐԱՆՔՆԵՐ

1. Ի՞նչ նպատակով են կիրառվում զանգվածները:
2. Ինչպես կարելի է հայտարարել զանգված:
3. Բացատրել `pop`, `shift`, `unshift`, `push`, `indexOf` և `lastIndexOf` ֆունկցիաների աշխատանքի սկզբունքները:
4. Հայտարարել զանգված և արտածել դրա երկրորդ մեծագույն տարրը:
5. Հայտարարել զանգված և գտնել այդ զանգվածում զույգ տարրերի քանակը:
6. Հայտարարել զանգված և դրա կրկնվող տարրերից ստանալ մեկ այլ զանգված:
7. Հայտարարել զանգված և տեղերով փոխել դրա մեծագույն և փոքրագույն տարրերը:
8. Հայտարարել զանգված և հաշվել, թե քանի՞ տարր կա զանգվածում, որի արժեքի և հերթական համարի գումարը զույգ է:
9. Հայտարարել զանգված և գտնել դրա փոքրագույն տարրի հերթական համարը:
10. Հայտարարել զանգված և ցիկլի օգնությամբ ստուգել՝ արդյո՞ք այն դասավորված է աճման կարգով, թե՝ ոչ:

# ԴԱՍ 8

## ՀԱՆԳՎԱՅԻՆԵՐԸ DOM - ՈՒՄ



## ԴԱՏ 8: ԶԱՆԳՎԱԾՆԵՐԸ DOM -ՈՒՄ

### 8.1 DOM - ի զանգված վերադարձնող ֆունկցիաները

JavaScript -ի Document Object Model -ից մեզ արդեն ծանոթ է getElementById ֆունկցիան, որը վերադարձնում է տրված id -ով տեղը: Կան իրավիճակներ, որոնցում մենք գործ ունենք ոչ թե մեկ տեղի, այլ տեղերի խմբի հետ:

Բացի getElementById -ից JavaScript -ը մեզ տրամադրում է մի շարք այլ ֆունկցիաներ, որոնցից են.

1. **getElementsByName** - վերադարձնում է տրված տեղի անունով բոլոր տեղերը զանգվածի տեսքով,
2. **getElementsByClassName** - վերադարձնում է տրված կլասն ունեցող բոլոր տեղերը զանգվածի տեսքով,
3. **querySelectorAll** - վերադարձնում է տրված CSS selector -ին բավարարող բոլոր տեղերը զանգվածի տեսքով:

Նշված ֆունկցիաներից լայն կիրառում ունի հատկապես querySelectorAll -ը, քանի որ վերջինս ունիվերսալ (համապիտանի) է և ներառում է նախորդ երկուսի ֆունկցիոնալությունը: Զանգվածների և DOM -ի կապը փորձենք բացատրել կոնկրետ օրինակում: Դիցուք էջի վրա ունենք 16 տարբեր div<sup>7</sup> -եր, յուրաքանչյուր div -ի սեղման ժամանակ անհրաժեշտ է սեղմված div -ի ֆոնի գույնը դարձնել կանաչ: Խնդիրը լուծելու համար կարող ենք յուրաքանչյուր div -ին ավելացնել onclick հատկությունը, սակայն դա լավագույն տարբերակը չէ, քանի որ div-երի քանակը կարող էր լինել օրինակ 400, իսկ այդ դեպքում onclick ավելացնելը կիներ բավական ժամանակատար գործընթաց: Հենց այս իրավիճակում մեզ կարող է օգնել querySelectorAll ֆունկցիան:

<sup>7</sup> Տեք HTML -ում, որը նախատեսված է էջում տարբեր հատվածներ առանձնացնելու համար:

```
var elements = document.querySelectorAll("div");
```

Այս դեպքում elements -ի մեջ կպահպանվեն բոլոր div-երի տվյալները՝ որպես զանգված:

Քանի որ elements -ը զանգված է, ապա elements[0] -ն կլինի այդ զանգվածի առաջին տարրը, նման դեպքում կարող ենք գունավորել օրինակ առաջին վանդակը.



```
elements[0].style.background = "green";
```

Կստանանք նկարում բերված արդյունքը.

Եթե elements[0] -ն իրենից ներկայացնում է DOM -ի տարր, ապա նրան հնարավոր է ավելացնել event-ներ JS -ի ստանդարտ հատկությունների (onclick, onchange և այլն), ինչպես նաև **addEventListener** ֆունկցիայի օգնությամբ<sup>8</sup>:

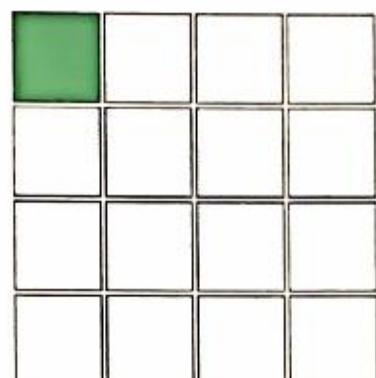
Այժմ փորձենք կիրառել addEventListener և onclick մեթոդները, որպեսզի event ավելացնենք div -ին:

```
//տարրերակ 1
```

```
elements[0].onclick = F;
```

```
//տարրերակ 2
```

```
elements[0].addEventListener("click", F);
```



Երկու դեպքում էլ տեղի սեղման ժամանակ աշխատում է F ֆունկցիան: Սակայն խորհուրդ է տրվում մշտապես կիրառել

---

Այս մոտեցումը, եթե JS -ի հետ կապված կողերն այլև չեն գրվում HTML -ում և նշանակում են ամբողջությամբ առանձնացվում են js այլում, ընդունված է անվանել unobtrusive JavaScript:

addEventListener -ը, քանի որ վերջինս ունի առավելություն onclick -ի նկատմամբ:  
Դիտարկենք օրինակը.  
**elements[1].onclick = F;**  
**elements[1].onclick = F2;**  
Բերված օրինակում elements[0] -ի սեղման ժամանակ կաշխատի միայն F2 ֆունկցիան, քանի որ onclick -ի դեպքում կատարվում է վերջին հրամանը: addEventListener ֆունկցիայի դեպքում միևնույն տեղի կամայական event կարող է կապված լինել տարբեր ֆունկցիաների հետ:

Օրինակ՝

```
elements[0].addEventListener("click", F);
elements[0].addEventListener("click", F2);
```

Այս դեպքում elements[0] -ի սեղման ժամանակ կաշխատեն և F, և F2 ֆունկցիաները: Նկատենք, որ նոյն տարբերակով, կարելի է հեռացնել տեղին կապված event-ը օգտագործելով removeEventListener ֆունկցիան:

```
elements[0].removeEventListener("click", F);
```

Այս դեպքում, տեղի սեղման ժամանակ կաշխատի միայն F2 ֆունկցիան, քանի որ F -ն այլևս հեռացված է: Մեր նպատակն է յուրաքանչյուր div -ի սեղման ժամանակ գունավորել այդ div -ը կանաչ գույնով: Քանի որ խոսքը վերաբերվում է ոչ թե մեկ div -ի, այլ div -երի զանգվածի, ուստի event -ի ավելացումը կազմակերպենք ցիկլի օգնությամբ:

```
var elements = document.querySelectorAll("div");
for(let i = 0; i < elements.length; i++){
    elements[i].addEventListener("click", F);
}
```

Այս դեպքում ցանկացած div -ի սեղման ժամանակ կաշխատի F ֆունկցիան, որն անհրաժեշտ է նկարագրել: Իսկ ինչպես F ֆունկցիան կարող է իմանալ, թե առկա 16 div-երից հատկապես ո՞ր մեկն է սեղմվել: Պարզվում է, որ event-ներին կապված ֆունկցիաներում կարելի է օգտագործել **this** հրամանը, որը ցույց

Է տալիս այն տեղը, որի նկատմամբ տեղի է ունեցել event -ը:  
Այսպիսով F ֆունկցիան կունենա հետևյալ տեսքը.

function F(){

```
this.style.backgroundColor = "green";
```

}

Սեղմելով յուրաքանչյուր div -ի վրա՝ կարող ենք համոզվել, որ  
այն կստանա կանաչ գույն:

## 8.2 Պատահական գործնթացների ծրագրավորում

Համակարգչային ծրագրավորման ոլորտում առանցքային դեր  
ունեն պատահական գործնթացները: Հաճախ կարող ենք  
տեսնել վեր կայքեր, որտեղ առկա նկարները կամ գրառումները,  
յուրաքանչյուր անգամ, պատահականության սկզբունքով են  
դասավորվում: Նույնը կարող ենք նկատել ժամանակակից  
խաղերում, որտեղ հաճախ գործ ունենք պատահական  
օբյեկտների հետ: Օրինակ՝ հայտնի Գուշակիր մեղեղին խաղում  
յուրաքանչյուր անգամ ընտրվում են պատահական երգեր,  
որոնք խաղացողը պետք է փորձի գուշակել: Տեսրիս  
համակարգչային խաղում յուրաքանչյուր անգամ  
խաղադաշտում հայտնվող օբյեկտը կրկին ընտրվում է  
պատահականության սկզբունքով:

Պատահական գործնթացների մեծ մասի հիմքում ընկած են  
պատահական թվերի հետ կատարվող գործողությունները:

Պատահականությունների աշխարհում, անշուշտ, անհնար  
կիներ գտնել օրինաչափություններ, ուստի, բնական է, որ  
ծրագրավորման լեզուները մեզ թույլ են տալիս ստանալ  
պատահական թվեր՝ ենելով որոշակի օրինաչափություններից:

Այլ կերպ ասած՝ պատահական թվերը պատահական չեն  
ստացվում ծրագրավորման միջավայրում: Ծրագրավորման  
լեզուներում և ընդհանրապես համակարգիչներում  
պատահական թվեր գեներացնող համակարգերին անվանում  
ենք պատահական թվերի գեներատորներ: Գոյություն ունեն  
պատահական թվերի իրական և կեղծ գեներատորներ: Կեղծ  
գեներատորների հիմքում ընկած են հատուկ մաթեմատիկական

ալգորիթմներ, որոնք գեներացնում են պատահական թվեր: Իրական գեներատորների հիմքում կարող են ընկած լինել: Համակարգչային համակարգերի ներգործությունը: Ենթադրենք համակարգչի ընթացիկ ժամի հետ կատարվող գործողություն, JavaScript ծրագրավորման լեզուն մեզ տրամադրում է Math պատահական թվեր 0-ից 1 հատվածում: Օրինակ.

```
var tiv = Math.random(); //0.8541151052432024
```

0 և 1 թվերը չեն ներառվում Math.random() -ի կողմից ստացված թվերի բազմության մեջ, ուստի բնական է, որ  $0.99(9)$  -ը կլինի  $(0;1)$  հատվածի մեջագույն իրական թիվը: Եթե ստացված թիվը բազմապատկենք  $x$  -ով, ապա կունենանք պատահական իրական թվեր  $(0; x-1]$  հատվածից: Եթե մեզ անհրաժեշտ են ամբողջ թվեր, ապա ստացվածը կարող ենք ձևափոխել parseInt ֆունկցիայի օգնությամբ:

Նախորդ օրինակում div-ի սեղման ժամանակ այն ներկում էինք կանաչ գույնով: Այժմ անհրաժեշտ է, որպեսզի յուրաքանչյուր div -ի սեղման ժամանակ ստանանք պատահական գույն:

Պատահական գույներ ստանալու հարցում շեշտադրումը պետք է կատարել CSS -ի գունային մոդելներից rgb() -ին: Ինչպես գիտենք, rgb -ն գունային մոդել է, որը ստանում է 3 պարամետրեր x,y,z, որոնք համապատասխանաբար ցույց են տալիս կարմիրի, կանաչի և կապույտի քանակը՝ արտահայտված 0-255 հատվածի արժեքներով:

Օրինակ՝

1. **rgb(255,0,0)** - կարմիր
2. **rgb(0,255,0)** - կանաչ
3. **rgb(0,0,255)** - կապույտ

---

<sup>9</sup> Ստատիկ կլասն իրենից ներկայացնում է հիշողության տիրույթ, որտեղ կարող են առկա լինել տարբեր ֆունկցիաներ, որոնց անվանում են մեթոդներ: X ստատիկ կլասի F մեթոդը կարելի է կանչել X.F() եղանակով:

4. `rgb(0,0,0)` - սև
5. `rgb(255,255,255)` - սպիտակ
6. `rgb(172,65,48)` - գունային երանգ կարմիր գույնի գերակայությամբ

Բնական է, եթե տրված երեք գույների թվային արժեքները լինեն պատահական թվեր, ապա կստանանք պատահական գույն: Զեափոխենք նախորդ կետում գրված F ֆունկցիան՝ հիմնվելով բացատրված նյութի վրա:

`function F()`

```
let red = parseInt(Math.random() * 256);
let green = parseInt(Math.random() * 256);
let blue = parseInt(Math.random() * 256);
let guyn = "rgb(" + red + "," + green + "," + blue + ")";
this.style.background = guyn;
```

}

Արդյունքում տարբեր վանդակների սեղման ժամանակ կունենանք նկարում բերված արդյունքը:

Ինչպես նկատեցինք, կոդում ընտրվել են 0-255 հատվածում 3 պատահական թվեր և string-ների գումարման եղանակով կառուցվել է անհրաժեշտ գույնի կոդը, որն էլ իր հերթին փոխանցվել է `this.style.background` - ին:

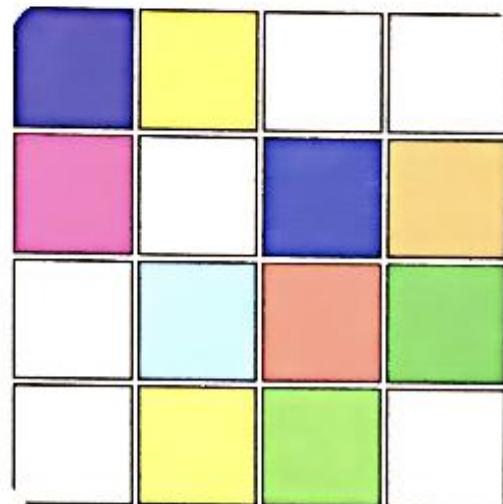
Ուշագրավ է այն, որ ECMAScript6

- ում արդեն string-ների գումարման եղանակով որոշակի արդյունք ստանալու գործընթացն առավել պարզեցված է նոր մեթոդի օգնությամբ:

Դիտարկենք ստորև բերված հրամանը:

```
var guyn = `rgb(${red},${green},${blue})`
```

`` նշանների ներսում գրված արտահայտությունն ամբողջությամբ համարվում է տողային տիպի: Բացառություն



Են կազմում այն դեպքերը, երբ տողային տիպում հանդիպուած `{} $()` նշանները: Այս դեպքում ձևավոր փակագծերում գրված արտահայտությունը համարվում է տրամաբանական կիրացվում է JS -ի կողմից:

Օրինակ՝

```
let s = `5 -ի քառակուսին կլինի ${5*5}`;  
console.log(s); //5 -ի քառակուսին կլինի 25
```

Նախորդ օրինակում ստեղծված F ֆունկցիայի ներսում կիրառեցինք տողային տիպերի գումարումը (concatination) պատահական գույնի կոդ ստանալու համար: Դիտարկենք այդ ֆունկցիան՝ իիմնվելով տողային տիպերի գումարման նոր մեթոդի վրա.

```
function F(){  
    var red = parseInt(Math.random() * 255);  
    var green = parseInt(Math.random() * 255);  
    var blue = parseInt(Math.random() * 255);  
    var guyn = `rgb(${red}, ${green}, ${blue})`;  
    this.style.background = guyn;  
}
```

Ենթադրենք անհրաժեշտ է յուրաքանչյուր սեղման ժամանակ գունավորել միայն սեղմված `div` -ը, իսկ մնացյալ վանդակները դարձնել սպիտակ: Մինչ ֆունկցիան մենք հայտարարել ենք գլոբալ փոփոխական, որին անվանել ենք `elements`: Փոփոխականի ներսում `querySelectorAll` -ի օգնությամբ պահել ենք առկա `div`-երի գանգվածը, ուստի մինչ տեղի կունենա սեղմված `div` -ի գույնի փոփոխությունը, կարող ենք ցիկլի օգնությամբ բոլոր `div`-երը դարձնել սպիտակ:

Դիտարկենք ասվածը.

```
function F(){  
    for(let i = 0; i < elements.length; i++){  
        elements[i].style.background = "white";  
    }  
    let red = parseInt(Math.random() * 255);
```

```
let green = parseInt(Math.random() * 255);
let blue = parseInt(Math.random() * 255);
this.style.background = `rgb(${red}, ${green}, ${blue})`;
}
```

Արդյունքում բոլոր div-երը դառնում են սպիտակ, իսկ սեղմվածը ստանում է պատահական գույն:

## ՀԱՐՑԵՐ ԵՎ ԱՌԱՋԱԴՐԱՆՔՆԵՐ

1. Բացատրել getElementsByTagName, getElementsByTagName ֆունկցիաների նշանակությունը:
2. Ի՞նչ առավելություն ունի querySelectorAll ֆունկցիան getElementsByTagName, getElementsByTagName ֆունկցիաների նկատմամբ:
3. Ի՞նչ նշանակություն ունի addEventListener ֆունկցիան և ո՞րն է նրա առավելությունը:
4. Ինչպես ստանալ պատահական թվեր JavaScript -ում:
5. Ունենք զանգված, որում գրված են մարդկանց անուններ: Եցում ունենք 10 div -եր և button: Button - ի սեղման ժամանակ յուրաքանչյուր div -ի ներսում գրել որևէ պատահական անուն՝ նախապես հայտարարված զանգվածից: Անունները div -ի ներսում պետք է ունենան պատահական գույն և տառաչափ:

ԴԱՍ 9

# ՏՈՂԱՅԻՆ ՏԻՊ



## ԴԱՍ 9: ՏՈՂԱՅԻՆ ՏԻՊ, REGULAR EXPRESSIONS

### 9.1 Տողային տիպը՝ որպես զանգված

Տողային տիպը՝ string -ը, օգտագործվում է տեքստային տվյալներ պահպանելու համար:  
Օրինակ.

```
var text = "I love drinking coffee so much";
```



JavaScript ծրագրավորման լեզվում տողային տիպը համարվում է զանգված տիպի մասնավոր դեպք, ընդ որում նրա յուրաքանչյուր սիմվոլը (այդ թվում բացատը) համարվում էն նրա հերթական տարրերը:  
Դիտարկենք օրինակը.

```
var text = "I love drinking coffee so much";
console.log(text[0]); // I
console.log(text[4]); // v
```

Տողային տիպը միաչափ զանգվածներից տարբերվում է նրանով, որ այն կայուն տիպ է: Նշանակում է՝ նրա տարրերը հնարավոր չեն փոփոխության ենթարկել՝ առանց տողը ձևափոխող ֆունկցիաների կիրառման:

Օրինակ.

```
var text = "Effect";
text[0] = "A";
console.log(text); // "Effect"
```

Հաշվի առնելով այս հանգամանքը՝ տողային տիպերում ձևափոխություններ իրականացնելու համար պետք է կիրառենք որոշ ֆունկցիաներ, որոնք մեզ տրամադրում է JavaScript -ը:  
Ծանոթանանք այդ ֆունկցիաներից մի քանիսի հետ:

**Split** - ֆունկցիան նախատեսված է տեքստն առանձին մասերի տրոհելու համար: Մասերի բաժանման պրոցեսը տեղի է ունենում՝ ըստ որոշակի սիմվոլի: Տրոհման արդյունքում

ստացվում է միաչափ զանգված: Որպես պարամետր՝ ֆունկցիան ստանում է այն սիմվոլը, ըստ որի բաժանումը պետք է տեղի ունենա:

Դիտարկենք օրինակը.

```
var text = "I/love/coffee";
text = text.split("/");
console.log(text); // ["I", "love", "coffee"];
console.log(text[0]); // 'I'
console.log(text[1]); // 'love'
console.log(text[2]); // 'coffee'
```

**Join**- ֆունկցիան նախատեսված է զանգվածի տարրերը տրված սիմվոլով միավորելու համար: Միավորման արդյունքում ստացվում է տողային տիպ: Որպես պարամետր՝ ֆունկցիան ստանում է այն սիմվոլը, ըստ որի պետք է կատարվի միավորումը:

Դիտարկենք օրինակը.

```
var text = ["I", "love", "drinking", "coffee"];
text = text.join("-");
console.log(text); // 'I-love-drinking-coffee'
```

**toUpperCase** - ֆունկցիան փոխում է տրված տեքստի տառերը մեծատառով:

Օրինակ.

```
var text = "JavaScript is cool";
text = text.toUpperCase();
console.log(text); // 'JAVASCRIPT IS COOL'
```

**toLowerCase**- ֆունկցիան ձևափոխում է տրված տեքստի տառերը փոքրատառերի:

Օրինակ.

```
var text = "JavaScript is cool";
text = text.toLowerCase();
console.log(text); // 'javascript is cool'
```

startsWith, endsWith - ֆունկցիաները ստուգում են՝ արդյո՞ք x տեքստը սկսվում կամ ավարտվում է յ ենթատեքստով:

Օրինակ՝

```
var text = "JavaScript is cool";
var p = text.startsWith("java"); //false
var p = text.startsWith("Java"); //true
var p2 = text.endsWith("java"); //false
var p2 = text.endsWith("ol"); //true
```

includes - ֆունկցիան ստուգում է՝ արդյո՞ք x տեքստի մեջ պարունակվում է յ ենթատեքստը:

Օրինակ՝

```
var text = "JavaScript is cool";
var p = text.includes("cool"); //true
var p = text.includes("html"); //false
```

indexOf- ֆունկցիան վերադարձնում է x տեքստում յ ենթատեքստի առաջին հատման ինդեքսը, եթե x տեքստը չի պարունակում յ ենթատեքստ, ապա ֆունկցիան վերադարձնում է -1:

Օրինակ՝

```
var text = "I'm learning javascript";
var p = text.indexOf("l"); // 4
var p2 = text.indexOf("java"); // 13
var p2 = text.indexOf("html"); // -1
```

lastIndexOf- ֆունկցիան վերադարձնում է x տեքստում յ ենթատեքստի վերջին հատման ինդեքսը, եթե x տեքստը չի պարունակում յ ենթատեքստ, ապա ֆունկցիան վերադարձնում է -1

Օրինակ՝

```
var text = "I'm learning javascript";
var p = text.lastIndexOf("a"); // 16
```

trim - ֆունկցիան հեռացնում է տվյալ տեքստի սկզբում և  
վերջում առկա բացատները.

```
var text = " what's up? ";
var p = text.trim(); // "what's up?"
```

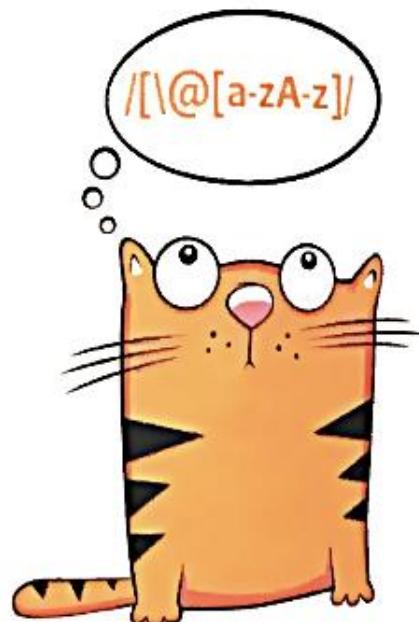
Առանձնակի հետաքրքրություն է ներկայացնում նաև substring  
ֆունկցիան:

substring(a,b) - x տեքստից անջատում է a -ից b հատվածին  
պատկանող ենթատեքստ: Այստեղ a -ն և b-ն թվեր են, որոնք  
ցույց են տալիս տեքստում որոշակի հերթական համար: Եթե b -  
ն չի նշվում, ապա ենթատեքստը սկսվում է a -ից և ավարտվում  
տեքստի վերջում:

```
var text = "I'm learning javascript";
var p = text.substring(4,12); // learning
var p = text.substring(13); // javascript
```

## 9.2 Regular Expressions

Regular Expression -ը խիստ  
որոշակի կառուցվածք ունեցող  
արտահայտություն է, որը կարող է  
կիրառվել տողային տիպում  
որոնում կամ փոխարինում  
կատարելու նպատակով:  
Ցանկացած regular expression ունի  
երկու մաս՝ **մոդել** և **մոդիֆիկատոր**:  
Մոդելը ցույց է տալիս որոնվող  
տեքստը, իսկ մոդիֆիկատորը  
ավելացնում է որոշակի  
հատկություններ



արդեսին:

Regular expression - ի օրինակ կարող է հանդիսանալ ստորև  
բերված արտահայտությունը.

/JavaScipt/i

Բերված օրինակում JavaScript -ը այն տեքստն է, որն որոնվում է, այսինքն՝ մոդել, իսկ և տառը մոդիֆիկատորն է, որը տվյալ օրինակում ցուց է տալիս, որ մեծատառ-փոքրատառ տարրերությունները պետք է անտեսվեն, այսինքն՝ տվյալ regular expression -ն որոնումը կհամարի հաջողված, եթե տեքստում հանդիպի ոչ միայն JavaScript, այլ նաև javascript, JAVAscript, javaSCRIPT և մի շարք այլ տարրերակներ:

Տողային տիպի ստանդարտ ֆունկցիաներից երկուսը՝ replace և search որպես պարամետր կարող են ստանալ regular expression:

Դիտարկենք պարզ օրինակ

```
let text = "I love PROGRAMMING";
let result = text.search(/programming/i); //7
```

Այս օրինակում կիրառվել է և մոդիֆիկատորը, որն ուշադրություն չի դարձնում տառերի մեծատառ փոքրատառ լինելուն և վերադարձնում է programming ենթատեքստի առաջին հանդիպման ինդեքսը, որը մեր դեպքում հավասար է 7 -ի:

Այժմ կիրառենք replace ֆունկցիան՝ տեքստում փոփոխություն կատարելու նպատակով:

```
var text = "JS stands for javascript. JavaScript is different
from Java.";
text = text.replace(/java/i, "Action");
console.log(text); // "JS stands for Actionscript. ActionScript is
different from Java.";
```

Ինչպես նկատեցինք, այս դեպքում փոփոխության ենթարկվեց միայն առաջին հանդիպած java -ն: Եթե ցանկանում ենք փոփոխության ենթարկել բոլոր հանդիպած տարրերը, ապա կարող ենք օգտվել ց մոդիֆիկատորից, որը թույլ է տալիս regular expression -ը կիրառել string -ի բոլոր տարրերի համար:

Օրինակ՝

```
var text = "JS stands for javascript. JavaScript is different
from Java.";
text = text.replace(/java/ig, "Action");
console.log(text); // "JS stands for Actionscript. ActionScript
is different from Action";
```

Դժվար չէ նկատել, որ այստեղ կիրառվել է երկու մոդիֆիկատոր՝ i և g: Արդյունքում՝ տողում հանդիպած բոլոր յա՞ն բառերը փոխարինվեցին Action բառով՝ առանց ուշադրություն դարձնելով տառերի մեջատառ կամ փոքրատառ լինելու: Այժմ ծանոթանանք Regular Expression -ների մշակման որոշ առանձնահատկությունների հետ:

Ենթադրենք անհրաժեշտ է տեքստի մեջ գտնվող բոլոր e, o տառերը փոխարինել X -ով: Այստեղ պետք է օգտագործենք Regular Expression -ների [xyz] մոդելը, որը նշանակում է որոնում քառակուսի փակագծերում տրված սիմվոլներից յուրաքանչյուրով:

```
let text = "Hello everyone";
text = text.replace(/leol/g, "X");
console.log(text); // "HXllX XvXryXnX"
```

[xyz] մոդելն ունի նաև ժխտական տարրերակ [^xyz], որը կնշանակի փնտրել տարրեր, որոնք նկարագրված չեն քառակուսի փակագծերում:

Առանձնակի հետաքրքրություն է ներկայացնում [a-b] մոդելը, որն առանձնացնում է a-ից b հատվածին պատկանող սիմվոլները, վերջինս ունի իր ժխտական մոդելը [^a-b]:

Ենթադրենք անհրաժեշտ է տվյալ տեքստի բոլոր թվերը փոխարինել աստղանիշներով:

```
let text = "Our phone number is 4456123";
text = text.replace(/[\d]/g, "*");
console.log(text); // "Our phone number is *****"
```

Գոյություն ունեն նաև այսպես կոչված մետամոդելներ, որոնց օգնությամբ կարող ենք գտել տեքստը՝ ըստ որոշակի խմբերի: Օրինակ՝ \d -ն թույլ է տալիս տեքստից առանձնացնել թվերը, իսկ \w -ն՝ բառերը:

```
let text = "Add $150 to $100 and you will get $250";
text = text.replace(/\d/g, "*");
console.log(text); // Add $*** to $*** and you will get $***
```

## ՀԱՐՑԵՐ ԵՎ ԱՌԱՋԱԴՐԱՆՔՆԵՐ

1. Ինչո՞ւ է տողային տիպը համարվում զանգվածի մասնավոր դեպք:
2. Ի՞նչ տարբերություն կա տողային տիպի և զանգվածի միջև:
3. Բացատրել `split` և `join` ֆունկցիաների նշանակությունը:
4. Բացատրել `substring` ֆունկցիայի դերը:
5. Բացատրել `toUpperCase` և `toLowerCase` ֆունկցիաների նշանակությունը:
6. Բացատրել `indexOf` և `lastIndexOf` ֆունկցիաների նշանակությունը:
7. Ի՞նչ է իրենից ներկայացնում `regular expression` -ը:
8. Ո՞րն է մոդելի և մոդիֆիկատորի տարբերությունը:
9. `input` դաշտում մուտքագրվել է տեքստ: `Button` -ի սեղման ժամանակ, մուտքագրված տեքստի յուրաքանչյուր բառի առաջին տառը դարձնել մեծատառ: Ստացված արդյունքը ցույց տալ `h1` -ում:

ԴԱՍ 10

# ՖՈՒՆԿՑԻՈՆԱԼ ԵՐԱԳՐԱՎՈՐՈՒՄ



## ԴԱՍ 10: ՖՈՒՆԿՑԻՈՆԱԼ ԾՐԱԳՐԱՎՈՐՈՒՄ

### 10.1 Գաղափար ֆունկցիայի մասին

Ֆունկցիան ծրագրավորողի կողմից մշակվող գործիք է, որը, նկարագրվելով մեկ անգամ կարող է կիրառվել բազմաթիվ անգամներ: Ֆունկցիաները թույլ են տալիս մեզ օգտագործել կոդի որոշակի հատվածներ տարբեր տեղերում՝ առանց ավելորդ անգամ վերանկարագրելու այն:

Մինչ այս պահը մենք ծանոթ ենք եղել ֆունկցիաներին, որոնք աշխատում են որոշակի event-ների ժամանակ: Այս բաժնում կուսումնասիրենք ֆունկցիոնալ ծրագրավորման մի շարք այլ հատկություններ:

Ֆունկցիա նկարագրելու համար օգտվում ենք հետևյալ կառուցվածքից:

```
function ֆունկցիայի_անուն(պարամետրեր){
```

```
    ... ֆունկցիայի մարմին
```

```
}
```

Ֆունկցիայի անունը իդենտիֆիկատոր է, այսինքն՝ այն չպետք է սկսվի թվային արժեքով, չի կարող պարունակել բացատներ, ինչպես նաև գծիկ: **Պարամետրերը** դրանք որոշակի փոփոխականներ են, որոնք փոխանցվում են ֆունկցիային, որպեսզի այն կատարի պահանջվող գործողությունը: Ֆունկցիան կարող է նաև չունենալ պարամետրեր: **Ֆունկցիայի բլոկը** կամ **ֆունկցիայի մարմինը** այն հատվածն է, որտեղ նկարագրվում է քայլերի այն բազմությունը, որը պետք է ֆունկցիան կատարի: Ընդհանրապես, ընդունված է ֆունկցիաները բաժանել երկու խմբի՝ **void** և **value-type**: Այն ֆունկցիաները, որոնք արժեք չեն վերադարձնում կոչվում են **void**, հակառակ դեպքում ֆունկցիաները կոչվում են **value-type**:

Այս մասին առավել մանրամասն կխոսենք հաջորդ ենթաբաժնում: Նշենք նաև, որ JavaScript -ում գտնվող բոլոր random() ֆունկցիան պատկանում են որոշակի կլասների<sup>10</sup>: Օրինակ՝ ֆունկցիան պատկանում է Math կլասին, log ֆունկցիան պատկանում է console կլասին և այլն: Այն պատկանում են window կլասին, Եթե, իհարկե, մենք ինքներս ֆունկցիան չնկարագրենք որոշակի կլասի ներսում: Կլասների մասին առավել մանրամասն կխոսենք ուսումնական ձեռնարկի վերջին հատվածում:

## 10.1 Ֆունկցիայի կանչ

Դիտարկենք հետևյալ օրինակը.

```
function sayHello(){  
    console.log("barev");  
}
```

Այս ֆունկցիան նախատեսված է “barev” տեքստը ելքագրելու համար, սակայն ֆունկցիայի նկարագրությունը դեռ բավարար չէ, որպեսզի այն կատարի իր մարմնում գրված գործողությունները: Որպեսզի ֆունկցիան աշխատի անհրաժեշտ է նրա նկատմամբ կիրառել ֆունկցիայի կանչ: Ֆունկցիայի կանչը արտահայտություն է, որը թույլ է տալիս «աշխատեցնել» ֆունկցիան: Ֆունկցիայի կանչ իրականացնելու համար անհրաժեշտ է գրել ֆունկցիայի անունը, որին պետք է հաջորդի փակագծեր:

```
Օրինակ՝ sayHello(); // "barev"
```

Ֆունկցիայի կանչ կարող է տեղի ունենալ նաև Event-ների ժամանակ.

```
document.querySelector("button").onclick = sayHello;
```

Այս դեպքում ֆունկցիայի կանչը տեղի կունենա button -ի սեղման ժամանակ:

---

<sup>10</sup> Կլասը իիշողության տարածք է, որն իր մեջ կարող է պարունակել տարբեր ֆունկցիաներ և փոփոխականներ:

## 10.2 Պարամետրով ֆունկցիաներ

Մեր կողմից մշակված sayHello ֆունկցիան ելքագրում է “barev” տեքստը, ինչը նրան դարձնում է պարզունակ:

Որպեսզի ֆունկցիան լինի համապիտանի անհրաժեշտ է, որպեսզի այն ստանա պարամետրեր և կատարի գործողություններ՝ հիմնվելով փոխանցված պարամետրերի վրա:

Օրինակ.

```
function sayHello(anun){  
    console.log("barev " + anun);  
}
```

Այս դեպքում ֆունկցիան՝ որպես պարամետր ստանում է anun փոփոխականը և էկրանին տպում է “barev” + anun արտահայտությունը:

Եթե anun = “Hayk”, ապա էկրանին կունենանք “barev Hayk” տեքստը:

Դիտարկենք ֆունկցիայի կանչի օրինակներ.

```
sayHello("Hayk"); // "barev Hayk"  
sayHello("Anna"); // "barev Anna"  
sayHello(); // "barev undefined"
```

Եթե ֆունկցիան ունի պարամետր, սակայն նրան պարամետր չի փոխանցվում, ապա այդ պարամետրի արժեքը դառնում է undefined: Դա է պատճառը, որ sayHello ֆունկցիան առանց պարամետրի կանչելու դեպքում ելքագրվում է “barev undefined” տեքստը: Ենթադրենք անհրաժեշտ է ֆունկցիան ծևափոխել այնպես, որ պարամետր չփոխանցելու դեպքում այն ելքագրի “barev bolorin” տեքստը:



Դիտարկենք օրինակը.

```
function sayHello(anun){  
    if(typeof anun == "undefined"){  
        anun = "bolorin";  
    }  
    console.log("barev " + anun);  
}
```

Ինչպես նկատեցինք ֆունկցիային ավելացվել է պայման, որը ստուգում է՝ արդյո՞ք առս փոփոխականն ունի undefined տիպ։ Այդ իրավիճակում առս -ի մեջ պահպում է “bolorin” տեքստը, ինչը թույլ է տալիս լուծել առաջադրված խնդիրը։

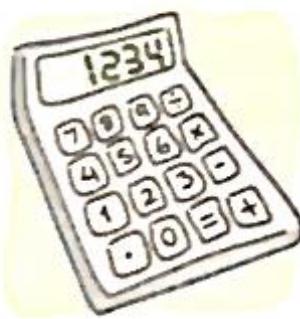
ECMAScript6 -ում ներդրվել է այսպես կոչված ֆունկցիայի default parameter - ի գաղափարը, որը թույլ է տալիս տրված խնդիրը լուծել՝ առանց պայմանական արտահայտության կիրառման։ Դիտարկենք օրինակը.

```
function sayHello(anun="bolorin"){  
    console.log("barev " + anun);  
}
```

Default պարամետրի դեպքում, ֆունկցիայի նկարագրության ժամանակ պարամետրը ստանում է որոշակի արժեք։ Այդ արժեքը JavaScript -ը կիրառում է այն ժամանակ, եթե ֆունկցիային պարամետր չի փոխանցվում, հակառակ դեպքում՝ ֆունկցիան օգտագործում է փոխանցված արժեքը։ Մեր օրինակում գրված է՝ **anun="bolorin"**, հետևաբար, եթե ֆունկցիային պարամետր չփոխանցվի, ապա այն կընդունի “bolorin” արժեքը, հակառակ դեպքում՝ այն արժեքը, որը նրան փոխանցվել է։

```
sayHello("Anna"); // "barev Anna"  
sayHello(); // "barev bolorin"
```

**10.3** Արժեք Վերադարձնող ֆունկցիաներ  
Դիտարկենք հետևյալ ֆունկցիան, որը  
ստանում է թիվ և հաշվում տրված թվի  
քառակուսին:



} Ֆունկցիայի օրինակ qarakusi(4) կանչի դեպքում **console**  
պատուհանում կելքագրվի 16 թիվը: Սակայն, եթե փորձենք  
ստացված արժեքը պահպանել փոփոխականի ներսում, ապա  
կստանանք այլ պատկեր:

Մեր դեպքում արտածվել է 16 թիվը, քանի որ կանչվել է ֆունկցիան, որի ներսում էլ գրված է console.log հրամանը: Էկրանին հայտնվել է նաև undefined արտահայտությունը, որը որում է տալիս արդեն m -ի արժեքը:

Եպիական հայութեան մասին արժեքը:  
ցույց է տալիս արդեն ո՞ -ի արժեքը:  
Չնայած նրան, որ խնդիրը լուծված է, այնուամենայնիվ  
ֆունկցիան սահմանափակում է ծրագրավորողի  
հնարավորությունները, քանի որ ֆունկցիայի ներսում  
հաշվարկել է որոշակի արտահայտության արժեք, որը  
հնարավոր չէ օգտագործել ֆունկցիայից դուրս:  
Ենթադրենք ցանկանում ենք հաշվել  $4^2+5^2$  արտահայտության  
արժեքը.

let s = qarakusi(4) + qarakusi(5);

Այս դեպքում կելքագրվեն 16 և 25 թվերը, սակայն եթե առտաձենք s -ը, կստանանք NaN:

Պատճառն այն է, որ մեր ֆունկցիան void ֆունկցիա է, որն արժեք չի վերադարձնում կամ, որ նույն է՝ վերադարձնում **undefined**, իսկ JavaScript -ում, երկու undefined -ների գումարը NaN է (Not A Number), այսինքն՝ ոչ թվային արժեք:

Արժեք վերադառնող կամ value-type ֆունկցիաները թույլ են տալիս, որպեսզի ֆունկցիայի ներսում հաշվարկված արժեքը կիրառվի նաև ֆունկցիայից դուրս: Ֆունկցիայից արժեքի

Վերադարձման պրոցեսը կազմակերպվում է `return` հրամանի  
օգնությամբ:

```
function qarakusi(x){  
    return x*x;
```

}

Այս դեպքում արդեն կարող ենք ֆունկցիայի կանչն իրականացնել հետևյալ կառուցվածքով.

```
let s = qarakusi(4) + qarakusi(5);  
console.log(s); //41
```

Այսպիսով՝ `return` հրամանի շնորհիվ ֆունկցիայի կանչը դառնում է այն արժեքը, որը ֆունկցիան վերադարձնում է:  
Օրինակ՝ `qarakusi(4)` -ը դառնում է 16, հետևաբար կարող ենք կիրառել օրինակ՝ `qarakusi(4) + 1` և մի շարք այլ արտահայտություններ:

Այժմ, եթե մեր ֆունկցիան վերադարձնում է արժեք, կարող ենք նրան կիրառել նաև բարդ արտահայտության մեջ:

Ենթադրենք անհրաժեշտ է հաշվել ստորև բերված արտահայտության արժեքը.

$$(5^2)^2 + 4^2$$

Ինչպես արդեն գիտենք,  $5^2$  արտահայտությունը կարող ենք գրել `qarakusi(5)` տեսքով: Հաշվի առնելով, որ ֆունկցիան արժեք է վերադարձնում, այսինքն՝ `qarakusi(5)` -ը համարժեք է 25 -ին, ուստի բնական է, որ  $(5^2)^2$  արտահայտությունը արդեն հնարավոր կլինի ներկայացնել `qarakusi(qarakusi(5))` տեսքով:

Դիտարկենք վերջնական արդյունքը.

```
let s = qarakusi(qarakusi(5)) + qarakusi(4);  
console.log(s); //641
```

Նշենք, որ արժեք վերադարձնող ֆունկցիաների դեպքում, ֆունկցիայի կանչը դառնում է այն արժեքը, որը վերադարձվել է, այսինքն՝ qarakusi(4) հրամանը համարժեք է պարզապես 16-ի: Բնական է, որ արժեք վերադարձնող ֆունկցիաները կարող են հանդիսավոր արտահայտության մեջ: Օրինակ՝ qarakusi(4) + 1 արտահայտությունը համարժեք է  $16 + 1$ -ին: Ասվածից կարելի է հետևողություն անել, որ ֆունկցիայի կանչը կարելի է կիրառել նույն կամ մեկ այլ ֆունկցիայի կիրառմամբ:

Օրինակ՝ qarakusi (qarakusi(4)) արտահայտության դեպքում գործում է qarakusi ֆունկցիային որպես պարամետր փոխանցվել է qarakusi(4)-ը, գրվածը համարժեք է qarakusi(16)-ին, որի արդյունքը կստացվի արդեն 256.

Նկատենք, որ return հրամանն անհրաժեշտ է գրել ֆունկցիայի վերջում, քանի որ նրանից հետո գրված այլ կող չի աշխատում: Այսպիսով՝ ֆունկցիայի աշխատանքը ցանկացած պահին կարելի է դադարեցնել՝ օգտագործելով return հրամանը: Կարելի է դադարեցնել անհրաժեշտ է գրել ֆունկիա, որը վերադարձնում է նթաղրենք անհրաժեշտ է գրել ֆունկիա, որը վերադարձնում է true, եթե իրեն փոխանցված թիվը զոյգ է և false՝ հակառակ դեպքում:

```
function zuyg(x){
    if(x % 2 == 0){
        return true;
    }else{
        return false;
    }
}
```

Հաշվի առնելով, որ return -ից հետո գրված հրամանները չեն կատարվում, կարելի էր ֆունկցիան գրել ավելի կարճ ձևով.

```

function zuyg(x){
    if(x % 2 == 0){
        return true;
    }
    return false;
}

```

Մեր օրինակում տրված `x` թվի համար ստուգվում է՝ արդյո՞ք երկուսի բաժանվելուց մնացորդում ստացվում է 0, եթե այս ապա ֆունկցիան վերադարձնում է `true` և դադարեցնում իր աշխատանքը, հակառակ դեպքում՝ եթե `if`-ի ներսում գրվածքը չի կատարվում, ապա բնական է, որ կատարվում է `if`-ին հաջորդող հրամանը, որը տվյալ դեպքում `return false` -ն է:

## ՀԱՐՑԵՐ ԵՎ ԱՌԱՋԱԴՐԱՆՔՆԵՐ

- Ի՞նչ նպատակով են կիրառվում ֆունկցիաները:
- Ի՞նչ մասերից է կազմված ցանկացած ֆունկիա:
- Ի՞նչ է նշանակում ֆունկցիայի կանչ:
- Ի՞նչ տարրերություն կա `void` և `value-type` ֆունկցիաների միջև:
- Ի՞նչ նպատակով է կիրառվում `return` հրամանը:
- Ի՞նչ է իրենից ներկայացնում ֆունկցիայի `default` պարամետրը:
- Գրել ֆունկիա, որը գտնում է իրեն փոխանցած զանգվածի մեծագույն տարրը և վերադարձնում այն:
- Գրել ֆունկիա, որը վերադարձնում է `true`, եթե իրեն փոխանցված տողը աշից և ձախից կարդացվում է նույն կերպ, այսինքն՝ պայինդրում է, և `false`՝ հակառակ դեպքում:
- Գրել ֆունկիա, որն իրեն փոխանցված տեքստի բոլոր մեծատառերը փոխարինում է փոքրատառերով և հակառակը:

# ԴԱՍ 11

## ՈԵԿՈՒՐՍԻԱ



## ԴԱՍ 11: ՌԵԿՈՐԴԻԱԼ: ՌԵԿՈՐԴԻՎ ՖՈՒՆԿՑԻԱՆԵՐ

11.1 ՌԵԿՈՐԴԻՎ ԳԱղափարը  
Ռեկորդիան ծրագրավորման գործընթաց է, որի ժամանակ ֆունկցիայի կանչը տեղի է ունենում նույն ֆունկցիայի մարմնում: Ֆունկցիան, որն իրականացնում է ռեկորդիա, անվանում են ռեկորդիվ ֆունկցիա:

Դիտարկենք հետևյալ օրինակը.

```
function fact(x){  
    if(x == 0 || x == 1){  
        return 1;  
    }  
    return x * fact(x-1);  
}
```

Ենթադրենք ֆունկցիային փոխանցվել է  $x = 4$  արժեքը.

1. Քանի որ  $x \neq 0$ , ապա աշխատում է  $4 * \text{fact}(3)$  -ը, այսինքն՝ ֆունկցիան կրկին կանչվում է, սակայն նրան փոխանցվում է 3 արժեքը,
2.  $\text{fact}(3)$  -ի դեպքում, քանի որ  $x \neq 0$ , ապա աշխատում է  $4 * 3 * \text{fact}(2)$  -ը,
3.  $\text{fact}(2)$  -ի դեպքում,  $x \neq 0$ , հետևաբար ունենում ենք  $4 * 3 * 2 * \text{fact}(1)$  -ը,
4.  $\text{fact}(1)$  -ի դեպքում  $x == 1$ , հետևաբար ստանում ենք  $4 * 3 * 2 * 1$  արժեքը

Դժվար չէ նկատել, որ ռեկորդիայի օգնությամբ գրված ֆունկցիան հաշվում է իրեն փոխանցված թվի ֆակտորիալը: Համար թվի ֆակտորիալ ասելով՝ հասկանում ենք 1 -ից մինչև x թիվն ընկած թվերի արտադրյալը: Օրինակ 5 -ի ֆակտորիալը հավասար է 120: Այսպիսով՝ ռեկորդիան ֆունկցիոնալ

ծրագրավորման հատկություն է, որը ցիկլերի այլընտրանքային տարրերակ է:

Ռեկուրսիվ ֆունկցիա մշակելիս առաջին հերթին պետք է համոզվել, որ ֆունկցիան ինչ-որ պահի ավարտվում է:

Մաթեմատիկայի դասընթացից հայտնի է, որ 0 կամ 1 թվի ֆակտորիալը հավասար է 1 -ի: Այդ է պատճառը, որ ֆունկցիայի մեջ գրված պայմանը նախ ստուգում է՝ արդյո՞ք փոխանցված թիվը հավասար է 1 կամ 0 -ի:

Եթե նկատենք, որ ֆունկցիան կարելի էր գրել ավելի կարճ, եթե կիրառենք նաև տերնար օպերատոր:

```
function fact(x){  
    return x == 0 || x == 1 ? 1 : x * fact(x-1);  
}
```

Այսպիսով՝ ռեկուրսիայի կիրառումը կարող է հանգեցնել կորի ծավալի զգալի չափով նվազեցման:

## 11.2 Ցիկլ vs ռեկուրսիա

Ինչպես նշեցինք ռեկուրսիան ցիկլերի կիրառման այլընտրանքային տարրերակ է: Բոլոր այն խնդիրները, որոնք լուծվում են ռեկուրսիայի կիրառմամբ, կարող են լուծվել նաև մեզ ծանոթ ցիկլերի միջոցով:



Օրինակ՝ գրենք ֆունկցիա, որը հաշվում է  $x$  թվի  $y$  աստիճանը:

$x -ի y$  ասկղիճան նշանակում է բազմապատկել  $x$  թիվը ինքն իրենով յանգամ: Օր.՝  $2^3 = 2 \times 2 \times 2 = 8$ :

Սկզբում ֆունկցիան գրենք ցիկլի միջոցով (իտերացիոն եղանակ):

```

function ast(x,y){
    var p = 1;
    for(let i = 1; i < y; i++)
    {
        p*=x
    }
    return p;
}

```

Ինչպես նկատեցինք ցիկլի ներսում x թիվը բազմապատկվում է ինքն իրենով յ անգամ:

Ներկայացնենք նոյն ֆունկցիան ռեկուրսիվ եղանակով.

Ռեկուրսիվ ֆունկցիայի կառուցման ժամանակ հիմնվել ենք այն պնդմանը, որ ցանկացած թվի 0 աստիճանը հավասար է մեկի:

```

function ast2(x,y){
    if( y == 0){
        return 1;
    }else{
        return x * ast2(x, y-1);
    }
}

```

Այժմ ռեկուրսիվ ֆունկցիային ավելացնենք տերնար օպերատոր, որպեսզի կոդը բերենք առավել պարզ տեսքի:

Դիտարկենք արդյունքը.

```

function ast2(x,y){
    return y == 0 ? 1 : x * ast2(x, y-1);
}

```

Ինչպես նկատեցինք, ռեկուրսիայի միջոցով գրված կոդն ունի առավել պարզ և կարճ կառուցված, մինչդեռ նրա վերլուծությունն ավելի բարդ է: Լուծել տվյալ խնդիրը ռեկուրսիվ, թե իտերացիոն եղանակով կախված է տվյալ իրավիճակից:

Ուկուրսիայի առավելությունն է կոդի կարճությունը, իսկ թերությունների շարքում կարելի է առանձնացնել այն, որ կոդը դժվար է վերլուծել և աշխատանքի արագությունը ցածր է<sup>11</sup>:

Այժմ փորձենք շեշտադրումը կատարել վերջին կետի՝  
աշխատանքի արագության վրա:  
Ունենք հետևյալ ֆունկցիաները.

```
function ast1(x,y){  
    return y == 0 || y == 1 ? 1 : x * ast1(x,y-1);
```

}

```
function ast2(x,y){  
    var p = 1;  
    for(let i = 0; i < y; i++){  
        p *= x;  
    }  
    return p;
```

Ֆունկցիայի աշխատանքի տևողությունը հաշվենք console.time()  
և console.timeEnd() ֆունկցիաներով: time ֆունկցիան ստանում է  
պարամետր, որը կարող է լինել ֆունկցիայի անունը, և սկսում է  
հաշվարկել ֆունկցիայի աշխատանքի տևողությունը:  
Ֆունկցիայի կանչից հետո, անհրաժեշտ է կիրառել timeEnd  
ֆունկցիան, որին փոխանցված պարամետրը պետք է համընկնի  
time ֆունկցիայի պարամետրի հետ: Ֆունկցիան վերադարձնում  
միլիվայրկյանների քանակը, որն անհրաժեշտ է ֆունկցիայի  
կանչն իրականացնելու համար: Ֆունկցիաներին կփոխանցենք  
գործընթացն առավել բարդ լինի, իսկ արագությունների միջև  
տարբերությունը լինի տեսանելի:

```
console.time("ast1")
```

---

<sup>11</sup> Անշուշտ, սա կախված է ծրագրավորման լեզվից: JavaScript -ում  
ուկուրսիայի արագությունը դանդաղ է իտերացիաների նկատմամբ:

```
ast1(2564,5122);
```

```
console.timeEnd("ast1"); // 1.46 միլիվայրկյան
```

```
console.time("ast2")
```

```
ast2(2564,5122);
```

```
console.timeEnd("ast2"); // 0.46 միլիվայրկյան
```

Այսպիսով՝ ռեկուրսիվ եղանակի դեպքում տևողությունը կազմեց 1.46մվ, մինչդեռ իտերացիոն եղանակի դեպքում՝ 0.46: Նշանակում է, որ ռեկուրսիայի դեպքում ֆունկցիայի աշխատանքի ժամկետն ավելի երկար է, ինչը պայմանավորված է նրանով, որ ռեկուրսիվ ֆունկցիաների աշխատանքը կազմակերպվում է Stack (ստատիկ հիշողություն) - ում, իսկ իտերացիոն եղանակի դեպքում Heap (դինամիկ հիշողություն) - ում:

Թե՛ Stack -ը, թե՛ Heap -ը հանդիսանում են RAM -ի (օպերատիվ հիշողություն) առանձին մասեր:

Stack -ում գործընթացները հիմնված են հերթերի գաղափարի վրա: Այստեղ հիշողության մեջ բաշխվում են ռեկուրսիվ ֆունկցիայի բոլոր կանչերը, որոնք սկսում են կատարվել հերթականությամբ: Օրինակ՝ ast1(2,3) դեպքում հիշողության մեջ տեղակայվում են ast1(2,2), ast1(2,1), ast1(2,0) կանչերը, որն էլ հանգեցնում է ռեկուրսիայի դանդաղ աշխատանքին: Որոշ իրավիճակներում, եթե stack-ում տեղակայված ֆունկցիաների քանակը գերազանցում է սահմանված հիշողությունը, ստանում ենք Error, որում ասվում է՝ maximum call stack size exceeded:

Այս դեպքում, ո՞րն է ռեկուրսիայի կիրառման իմաստը:

Պարզվում է բացի կողի կրճատումից, կան մի շարք խնդիրներ, որոնց լուծումը ցիկլի օգնությամբ լուրջ բարդություն է ներկայացնում: Դիտարկենք այդպիսի մի օրինակ:

Կասենք X զանգվածը կոմպլեքս զանգված է, եթե նրա յուրաքանչյուր տարրը կարող է լինել մեկ այլ զանգված:

Օրինակ՝  $x = [4, 3, [2, 4, 6], 8]$

Ենթադրենք անհրաժեշտ է գտնել տրված կոմպլեքս զանգվածի տարրերի գումարը: Բերված օրինակում x զանգվածի տարրերի գումարը կարելի է գտնել 2 ցիկլով:  
 $[4, 3, [2, 4, [1, 3, 2], 6], 8]$  զանգվածի դեպքում խնդիրը լուծելու համար անհրաժեշտ կլինի արդեն 3 ցիկլ: Բնական է, որ N -րդ կարգի խորություն ունեցող զանգվածի դեպքում անհրաժեշտ մեծ արժեքների դեպքում: Ուկուրսիայի կիրառմամբ այս խնդիրը կարող է ստանալ անհամեմատ ավելի պարզ լուծում:

### 11.3 Որոնում կիսման մեթոդով

Ալգորիթմների տեսության ամենահայտնի խնդիրներից մեկն է զանգվածում տվյալների որոնման խնդիրը: Այսինքն՝ ունենք տվյալների չ զանգված, որտեղ որոնում ենք որևէ կ տարը: Հայտնի են այս խնդիրի լուծման մի շարք ալգորիթմներ, օրինակ՝ գծային որոնման ալգորիթմը, կիսման մեթոդի ալգորիթմը և այլն: Գծային որոնման ալգորիթմի հիմքում ընկած է զանգվածի տարրերը մեկ առ մեկ դիտարկելու սկզբունքը: Դիտարկենք հետևյալ ֆունկցիան, որը x զանգվածում փնտրում է կ տարրի հերթական համարը կամ վերադարձնում -1, եթե տարրը բացակայում է:

```
function linearSearch(x, k){  
    for(let i = 0; i < x.length; i++){  
        if(x[i] == k){  
            return i;  
        }  
    }  
    return -1;  
}
```

```
}  
linearSearch([4,3,2,8,6], 8); //3  
linearSearch([4,3,2,8,6], 888); // -1
```

Այս օրինակում, ցիկլի օգնությամբ դիտարկվում են զանգվածի բոլոր տարրերը, եթե այնտեղ հայտնաբերվում է կ տարը, ապա



ֆունկցիան վերադարձնում է այդ տարրի հերթական համարը, հակառակ դեպքում՝ ֆունկցիան վերադարձնում է -1: Գծային որոնման ալգորիթմի համար լավագույն դեպքը հանդիսանում է այն, եթե որոնվող տարրը զանգվածի  $i = 0$  տարրն է: Նույն այն, եթե որոնվող տարրը զանգվածի  $i = n$  տարրն է: Նույն այն դեպքում ալգորիթմի համար վատագույն դեպք տրամաբանությամբ ալգորիթմի համար վատագույն դեպք հանդիսանում է այն իրավիճակը, եթե որոնվող տարրը զանգվածի վերջին տարրն է, կամ տարրը բացակայում է, քանի որ այս դեպքում դիտարկվում են զանգվածի բոլոր տարրերը: Ենդիրը հատկապես կրարդանա այն ժամանակ, եթե իրավիճակը լինի շատ մեծ:

զանգվածում տարրերի քանակը լինի շատ մեծ: Յուրաքանչյուր անգամ, եթե որևէ տվյալ ենք որոնում Google - ի վերջինս բացի որոնման արդյունքներից մեզ ում, վերջինս բացի որոնման արդյունքներից մեզ պահանջվում է նաև այն ժամանակահատվածը, որը պահանջվել է որոնումն իրականացնելու համար:

# Google

Պատկերացնենք, եթե Google -ն օգտագործեր անհրաժեշտ տվյալը: Բնական է, որ կան մի շարք այլ ալգորիթմներ, որոնք տարբեր իրավիճակներում կարող են առավել օպտիմալ աշխատել որոնման խնդրում: Այդպիսին է կիսման մեթոդը որոնումը (Binary Search): Կիսման մեթոդի ալգորիթմը կարելի է կիրառել միայն այն զանգվածերի նկատմամբ, որտեղ տվյալները դասավորված են աճման կամ նվազման կարգով:

Դիտարկենք ալգորիթմի աշխատանքը կոնկրետ օրինակում: Ունենք 10 տարր պարունակող հետևյալ զանգվածը, որում փնտրում ենք  $k = 23$  տարրը:

2	5	8	12	16	23	38	56	72	91
---	---	---	----	----	----	----	----	----	----

Կիսման մեթոդի դեպքում, առաջին քայլով, զանգվածը տրոհում ենք երկու հավասար մասերի՝ առանձնացնելով այն տարրը, որը գտնվում է կիսման կենտրոնում:

2	5	8	12	16	23	38	56	72	91
---	---	---	----	----	----	----	----	----	----

Մեր դեպքում կիսման կետում գտնվող տարրը  $m = 16$  տարը է:  
 Համեմատելով  $k = 23$  և  $m = 16$  տարը՝ պարզում ենք, որ  $k > m$ :  
 Հիմնվելով այն հանգամանքի վրա, որ զանգվածում տարրերը  
 դասավորված են աճման կարգով, բնական է, որ մեր կողմից  
 որոնվող տարրը պետք է գտնվի կիսման կենտրոնից աջ ընկած  
 հատվածում, ուստի հաջորդ քայլում մենք կանտեսենք ձախ  
 հատվածի բոլոր տարրերը:

23	38	56	72	91
----	----	----	----	----

Այժմ անհրաժեշտ է իրականացնել ևս մեկ կիսման  
 գործողություն, սակայն արդեն աջ հատվածում գտնվող  
 ենթազանգվածում: Այս դեպքում կիսման կենտրոնը  
 կիանդիսանա  $m = 56$ -ը:

2	5	8	12	16	23	38	56	72	91
---	---	---	----	----	----	----	----	----	----

Համեմատելով  $k$  և  $m$  թվերը այս անգամ պարզում ենք, որ  $k < m$ :  
 Նշանակում է այս քայլում պետք է բաց թողնենք 56-ից աջ  
 հատվածում գտնվող տարրերը:

2	5	8	12	16	23	38	56	72	91
---	---	---	----	----	----	----	----	----	----

Այժմ կիսման մեթոդը իրականացնենք [23,38] զանգվածի  
 նկատմամբ: Այս իրավիճակում կիսման կենտրոնը  
 կիանդիսանա  $23$  -ը<sup>12</sup>, որը հենց հանդիսանում է որոնվող  
 տարրը:

2	5	8	12	16	23	38	56	72	91
---	---	---	----	----	----	----	----	----	----

Այսպիսով՝ կիսման մեթոդի օգնությամբ  $n = 10$  տարր  
 պարունակող զանգվածում  $k = 23$  տարրի որոնումը  
 իրականացրեցինք ընդամենը 3 քայլով: Նույն խնդրի լուծումը

<sup>12</sup> Կիսման մեթոդում, եթե հերթական ենթազանգվածն ունի 2 տարր,  
 ապա ընտրվում է ձախ կողմում տեղակայված տարրը:

գծային որոնման մեթոդով կհաջողվեր գտնել 6 քայլի գծային որոնման մեթոդով կհաջողվեր գտնել 6 քայլի արդյունքում: Նկատենք, որ եթե զանգվածը պարունակեր 1.000.000 տարր, ապա առաջին քայլից հետո, դիտարկման ենթակա կլիներ միայն 500.000 -ը: Այսպիսով՝ կիսման մեթոդի ալգորիթմը շատ հարմար է մեծ զանգվածներում որոնում իրականացնելիս:

Այժմ փորձենք JavaScript ծրագրավորման լեզվի օգնությամբ գրել ֆունկցիա, որը ունկուրսիայի միջոցով կիրականացնի կիսման մեթոդի ալգորիթմը:

Դիտարկենք կոդը.

```
function binarySearch(arr, x, l = 0, r = arr.length-1)
{
    if (r >= l)
    {
        let mid = parseInt(l + (r - l)/2);

        if (arr[mid] == x){
            return mid;
        }

        if (arr[mid] > x){
            return binarySearch(arr, x, l, mid-1);
        }

        return binarySearch(arr, x, mid+1, r);
    }

    return -1;
}
```

```
let p = binarySearch([2,5,8,12,16,23,38,56,72,91], 23)
console.log(p); //5
```

Ֆունկցիայի պարամետրերն են.

- **arr** - հանդիսանում է այն զանգվածը, որում պետք է իրականանա որոնումը
- **x** - այն տարրն է, որը որոնվում է զանգվածում

- l - ներկայացնում է այն հատվածի սկզբնական ինդեքսը, որից սկսած պետք է կիսել զանգվածը
- r - ներկայացնում է այն հատվածի վերջնական ինդեքսը, որում պետք է իրականանա կիսման մեթոդը

և  $r$  փոփոխականները ունեն նախնական արժեքներ.  $l = 0$  և  $r = arr.length - 1$ , այսինքն՝ առաջին անգամ կիսման մեթոդն իրականանալու է ողջ զանգվածի նկատմամբ: Ֆունկցիայի ներսում, ինչպես կարող ենք նկատել, ստուգվում է պայման՝  $arr[l] \geq r$ : Այսինքն որ հակառակ դեպքում կիսման մեթոդի կիրառումն անհնար է:

mid փոփոխականի ներսում պահում ենք կիսման կենտրոնի ինդեքսը:

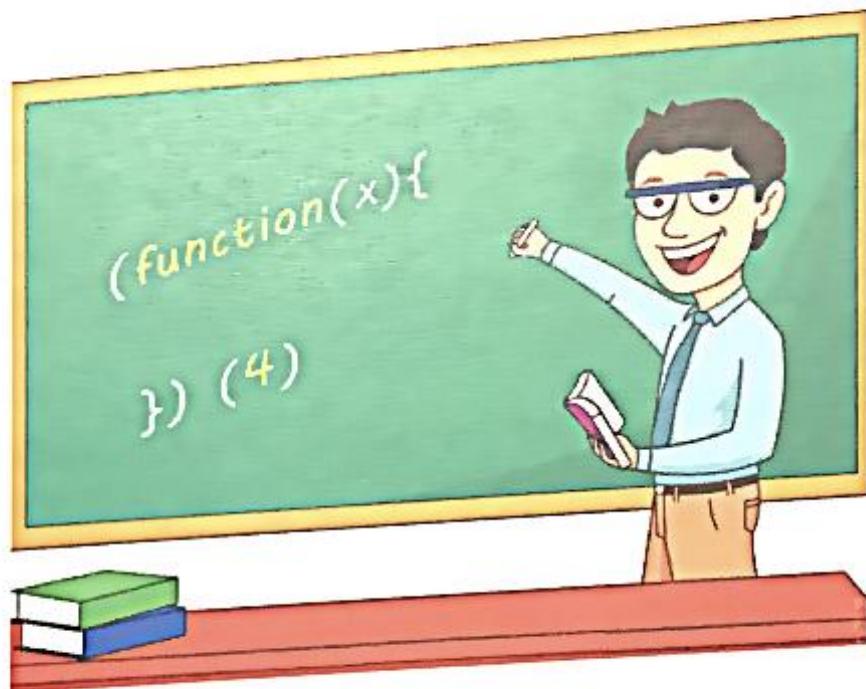
Առաջին քայլում, եթե  $l = 0$ , mid -ի համար կստանանք  $mid = 0 + (9-0)/2 = 4$ , այսինքն՝ կիսման կենտրոնն առաջին քայլում կլինի  $arr[4]$ -ը, որը հանդիսանում է 16 -ը: Հաջորդիվ ստուգվում է պայման՝  $arr[l] \geq mid$  կիսման կենտրոնում գտնվող տարրը հանդիսանում է մեր կողմից որոնվողը: Այս դեպքում անմիջապես վերադարձվում է տարրը: Մեր օրինակում  $arr[4]$ -ը չի հանդիսանում այդ տարրը, ուստի անհրաժեշտ է կիսման կենտրոնում գտնվող տարրը համեմատել որոնվողի հետ: Մեր օրինակում որոնվող տարրը՝ 23 -ը, ավելի մեծ է, քան կիսման կենտրոնում գտնվողը՝ 16-ը: Հետևաբար՝ որոնումն անհրաժեշտ է իրականացնել  $mid + 1$  հատվածում: Այդ իմաստով ֆունկցիան կանչում է ինքն իրեն, սակայն այս անգամ հատվածը սկսվում է  $l = mid + 1$ -ից, այսինքն՝ դիտարկվում է զանգվածի կիսման կենտրոնից աջ գտնվող հատվածը: Այս գործընթացը շարունակվում է այնքան, մինչ կհայտնաբերվի որոնվող տարրը, իսկ եթե տարրը բացակայում է, ապա  $r >= l$  պայմանը կընդունի կեղծ արժեք և ֆունկցիան կավարտի իր աշխատանքը՝ վերադարձնելով -1:

## ՀԱՐՑԵՐ ԵՎ ԱՌԱՋԱԴՐԱՆՔՆԵՐ

1. Ի՞նչ է ռեկորսիան:
2. Որո՞նք են ռեկորսիայի առավելությունները և թերությունները:
3. Ինչպե՞ս կարելի է որոշել ֆունկցիայի աշխատանքի տևողությունը:
4. Բացատրել կիսման մեթոդի ալգորիթմը:
5. Գրել ֆունկցիա, որը կիսման մեթոդի ալգորիթմը կիրականացնի ոչ ռեկորսիվ եղանակով:
6. Ստեղծել ռեկորսիվ ֆունկցիա, որը ստանում է ա և բ պարամետրեր, և հաշվում է  $[a;b]$  հատվածում բոլոր երկնիշ թվերի գումարը:
7. Ստեղծել ռեկորսիվ ֆունկցիա, որը ստանում է ա և բ պարամետրեր, և վերադարձնում է պատահական թիվ  $[a;b]$  հատվածից:
8. Ստեղծել ֆունկցիա, որն իրեն փոխանցված  $x$  թվի համար, ռեկորսիայի միջոցով, հաշվում է  $[0; x]$  հատվածում գտնվող թվերի գումարը:
9. Ստեղծել ֆունկցիա, որը ռեկորսիայի օգնությամբ կգտնի իրեն փոխանցված զանգվածի մեծագույն տարրը:

# ԴԱՍ 12

## ԱՆԱՆՈՒՆ ՏՈՒԿԱՑԻԱՆԵՐ



## ԴԱՍ 12: ԱՆԱՆՈՒՆ ՖՈՒՆԿՑԻԱՆԵՐ ԵՎ ԼՅԱՄԲԴԱ ԱՐՏԱՀԱՅՏՈՒԹՅՈՒՆ

### 12.1 Անանուն ֆունկցիաներ

JavaScript ծրագրավորման լեզվում ֆունկցիան կարող է դառնալ պարամետր մեկ այլ ֆունկցիայի համար: Ֆունկցիաների այս հատկությունը թույլ է տալիս ֆունկցիոնալ ծրագրավորումը դնել առավել բարձր մակարդակի վրա:

Մենք արդեն ծանոթ ենք իրավիճակների, որոնցում ֆունկցիան դառնում է մեկ այլ ֆունկցիայի համար պարամետր: Օրինակ՝ JavaScript -ից `addEventListener` ֆունկցիան որպես առաջին պարամետր ստանում է `event` -ի անվանումը (`click, dblclick...`), իսկ երկրորդ պարամետրում ստանում է այն ֆունկցիայի անունը, որը պետք է կանչվի `event` -ի աշխատանքի ժամանակ: `addEventListener` -ի դեպքում, ինչպես գիտենք, ֆունկցիան նկարագրվում էր առանձին:

Օրինակ՝

```
document.querySelector(".btn").addEventListener("click", F);
function F(){
    alert("hello");
}
```

Ընդհանրապես ֆունկցիաները մեզ հնարավորություն են տալիս կողի որոշակի հատված գրել մեկ անգամ և կիրառել բազմաթիվ անգամներ, սակայն երբեմն լինում են իրավիճակներ, երբ տվյալ ֆունկցիան կիրառվելու է միայն տվյալ իրավիճակում: Օրինակ՝ `addEventListener` -ի բերված օրինակում `F` ֆունկցիան կիրառվելու է միայն կոճակի սեղման ժամանակ: Նման դեպքերում, երբ ֆունկցիան պետք է կիրառվի միայն մեկ իրավիճակում, առավել հարմար է օգտագործել անանուն ֆունկցիաները: Անանուն ֆունկցիան ֆունկցիայի տեսակ է, որը չունի անուն:

Անանուն ֆունկցիան կարող ենք կիրառել այլ ֆունկցիաներում՝  
որպես պարամետր:

Օրինակ.

```
btn.addEventListener("click", function(){
    alert("JS is the coolest language, right?");
});
```

Ուշագրավ է այն, որ ֆունկցիան JavaScript -ում հանդիսանում է  
նաև տվյալների տիպ.

Դիտարկենք օրինակը.

```
function F(){
}
console.log(typeof F); // "function"
```

Քանի որ ֆունկցիան տվյալների տիպ է, ապա բնական է, որ այն  
հնարավոր է պահպանել որոշակի փոփոխականի մեջ և  
օգտագործել նրան ըստ այդ փոփոխականի:

```
var f = function(){
    alert("Hello");
}
f(); //Hello
```

Նման իրավիճակում շատ կարևոր է հասկանալ, թե ո՞րն է  
անանուն ֆունկցիան փոփոխականի մեջ պահելու կամ  
պարզապես ֆունկցիան սովորական ձևով նկարագրելու միջև  
տարրերությունը::

Բոլոր ոչ անանուն ֆունկցիաները, անկախ կողի ներսում իրենց  
նկարագրված դիրքից, JS -ի ինտերպրետատորի կողմից վերև  
են բարձրանում դեպի կողի սկիզբ: Հետևաբար նրանք  
հասանելի են նաև ավելի վաղ, քան իրենց նկարագրությունն է,  
ինչը չի վերաբերվում անանուն ֆունկցիաներին:

Դիտարկենք օրինակը.

```
f(); //error: f is not a function
```

```
var f = function(){
    alert("Hello");
}
```

Այսպիսով՝ անանուն ֆունկցիայի նկարագրության դեպքում, եթե Այսպիսով՝ անանուն ֆունկցիայի նկարագրության դեպքում, եթե ֆունկտիվ պահպանվում է փոփոխականի մեջ, նրան հնարավոր չէ կանչել մինչ ֆունկցիայի նկարագրությունը։ Այս պրոցեսն իրագործելի է միայն սովորական ֆունկցիաների դեպքում։

```
f(); //ok  
function f(){  
    alert("ok");  
}
```

**12.2 Անանուն ֆունկցիաների կիրառումը զանգվածներում**  
JavaScript -ում զանգվածների կառավարման համար կան մի շարք ֆունկցիաներ, որոնք որպես պարամետր ստանում են անանուն ֆունկցիաներ։

**Array.map** - թույլ է տալիս զանգվածի յուրաքանչյուր տարր փոփոխության ենթարկել՝ ըստ փոխանցված ֆունկցիայի։ Ենթադրենք ունենք թվերի զանգված և ցանկանում ենք յուրաքանչյուր տարրը բարձրացնել քառակուսի։

```
let x = [2,4,6,8,5,3];  
x = x.map(function(a){  
    return a * a  
});  
console.log(x); // [4,16,36,64,25,9]
```

Ինչպես նկատեցինք, որ ֆունկցիային փոխանցված անանուն ֆունկցիան, որպես պարամետր, ստանում է փոփոխական, տվյալ դեպքում a -ն, որը ցույց է տալիս զանգվածի ընթացիկ տարրը։ Այսպիսով որ ֆունկցիան ցիկլի միջոցով դիտարկում է զանգվածի յուրաքանչյուր ա տարրը և այդ տարրի փոխարեն պահում է անանուն ֆունկցիայի վերադարձրած արժեքը։

Array.filter - թույլ է տալիս որոնում իրականացնել զանգվածում ըստ փոխանցված բոլյան ֆունկցիայի: Բոլյան ֆունկցիա կանվանենք այն ֆունկցիան, որը վերադարձնում է տրամաբանական արժեք՝ true կամ false: Ենթադրենք ունենք թվերի x զանգված և ցանկանում ենք ստանալ նոր y զանգված, որի տարրերը կլինեն x զանգվածի զույգ տարրերը.

Դիտարկենք կոդը.

```
let x = [2,9,4,6,8,5,3];
let y = x.filter(function(a){
    if(a % 2 === 0){
        return true;
    }else{
        return false;
    }
});
console.log(y); // [2,4,6,8]
```

Ընդհանրապես a % 2 == 0 արտահայտությունը համարվում է տրամաբանական արտահայտություն, այսինքն՝ այն կարող է ունենալ երկու հնարավոր արժեք՝ true կամ false, բնական է, որ նման իրավիճակում կարող ենք խուսափել պայմանական օպերատորի կիրառումից և օգտագործել առավել կարճ գրելածեւ:

Դիտարկենք արդյունքը.

```
let x = [2,9,4,6,8,5,3];
let y = x.filter(function(a){
    return a % 2 == 0;
});
console.log(y); // [2,4,6,8]
```

Ակնհայտ է, որ անանուն ֆունկցիան կվերադարձնի true կամ false արժեք և վերջնական արդյունքը կլինի նույնը:

Array.reduce - թույլ է տալիս զանգվածի տարրերից ստանալ որոշակի ֆիքսված թիվ՝ ըստ փոխանցված անանուն ֆունկցիայի: Օրինակ՝ ունենք ամբողջ թվերից կազմված x զանգվածը և ցանկանում ենք գտնել նրա տարրերի գումարը.

```
let x = [2,9,4,6,8,5,3];
```

```
let y = x.reduce(function(a,b){  
    return a + b;  
});  
console.log(y); //37
```

Գոյություն ունի նաև `reduceRight` ֆունկցիան, որը գործողությունները կատարում է աջից ձախ:

**Array.every** - թույլ է տալիս ճշտել՝ արդյո՞ք զանգվածի բոլոր տարրերն են բավարարում փոխանցված ֆունկցիային: Ենթադրենք ցանկանում ենք ճշտել՝ արդյո՞ք զանգվածի բոլոր տարրերը զույգ են.

```
var x = [2,9,4,6,8,5,3];  
var y = x.every(function(a){  
    return a % 2 == 0;  
});  
console.log(y); //false
```

Ֆունկցիան վերադարձնում է `false`, քանի որ այնտեղ առկա են նաև կենտ տարրեր:

**Array.some** - թույլ է տալիս ճշտել՝ արդյո՞ք զանգվածում կա գոնե մեկ տարր, որը բավարարում է փոխանցված ֆունկցիային: Ենթադրենք ցանկանում ենք ճշտել՝ արդյո՞ք զանգվածում կա գոնե մեկ տարր, որը կենտ է:

```
var x = [2,9,4,6,8,5,3];  
var y = x.some(function(a){  
    return a % 2 != 0;  
});  
console.log(y); //true
```

**Array.forEach** - ցիկլի տեսակ է, որտեղ ցիկլի հերթական տարրը փոխանցված անանուն ֆունկցիայի առաջին պարամետրն է, ինդեքսը՝ երկրորդ: Օրինակ գտնենք զանգվածի տարրերի գումարը.

```
let x = [2,9,4,6,8,5,3];  
let s = 0;
```

```
let y = x.forEach(function(item,index){  
    s += item;  
});  
console.log(s); //37
```

### 12.3 Լյամբդա արտահայտություն (Arrow functions)

Լյամբդա արտահայտությունը ECMAScript6 -ում ներդրված մեխանիզմ է, որը թույլ է տալիս կրճատել ֆունկցիոնալ ծրագրավորման ժամանակ գրվող կողմ:

Դիտարկենք ֆունկցիա, որը ստանում է երկու պարամետր և վերադարձնում դրանց գումարը.

```
function gumar(a,b){  
    return a+b;
```

Նույն ֆունկցիան կարելի է գրել ընդամենը մեկ տողով՝ կիրառելով սամբռդա արտահայտություն:

Դիտարկենք օրինակը.

```
var gumar = (a, b) => a + b;
```

Այստեղ gumar - ը ֆունկցիայի անվանումն է, իսկ => նշանից հետո գրվածը այն արժեքը, որը պետք է ֆունկցիան վերադարձնի:

Լյամբդա արտահայտություն կարելի է կիրառել բոլոր այն ֆունկցիաներում, որտեղ որպես պարամետր սպասվում է մեկ այլ ֆունկցիա:

Նախորդ օրինակում դիտարկեցինք տար ֆունկցիայի օգնությամբ զանգվածի տարրերի քառակուսիները գտնելու խնդիրը: Այժմ դիտարկենք այդ խնդրի լուծումը՝ սամբռդա արտահայտության կիրառմամբ.

```
let x = [2,4,6,8,5,3];  
x = x.map( a => a * a);  
console.log(x); // [4,16,36,64,25,9]
```

Դժվար չէ նկատել, որ սամբռդա արտահայտությունը իսկապես զգալիորեն կրճատում է գրվող կողմ:

## 12.4 Տեսակավորման խնդիր

Հանգվածի տարրերի աճման կամ նվազման կարգով դասավորելու խնդիրն անվանում են տեսակավորման խնդիր: Տեսակավորման խնդրի լուծման համար հայտնի ալգորիթմներ են պղպջակների մեթոդը, ընտրություններով տեսակավորման ալգորիթմը և այլն:

Ծանոթանանք ընտրություններով տեսակավորման ալգորիթմի աշխատանքի սկզբունքին:

Այս ալգորիթմի հիմքում ընկած է զանգվածի  $i = 0$  -ից  $n-1$  տարրերի դիտարկումը, որտեղ  $n$  -ը հանդիսանում է զանգվածում առկա տարրերի քանակը: Յուրաքանչյուր  $i$  տարրի համար դիտարկվում է  $j = i+1$  -ից  $n-1$  հատվածի տարրերը, ընտրվում այդ հատվածի փոքրագույն տարրը և փոխարինում այն  $i$ -րդի հետ:

Դիտարկենք օրինակ.

64	25	12	22	11
----	----	----	----	----

Դիտարկենք  $i = 0$  տարրը՝ 64 -ը: Այս տարրի համար ընտրենք  $j=i+1$  -ից  $n-1$  հատվածի տարրերը և առանձնացնենք այդ հատվածի փոքրագույն տարրը:

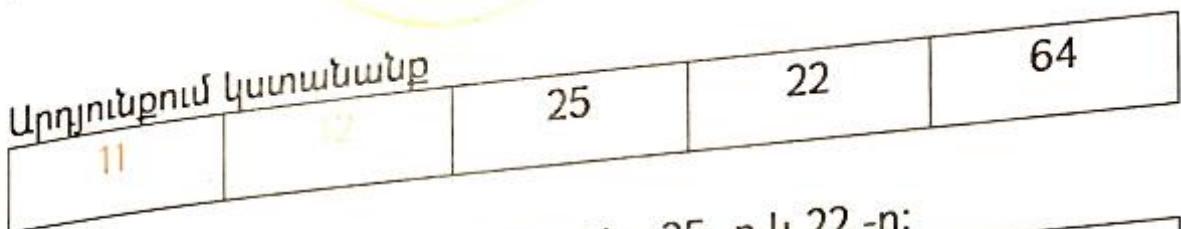
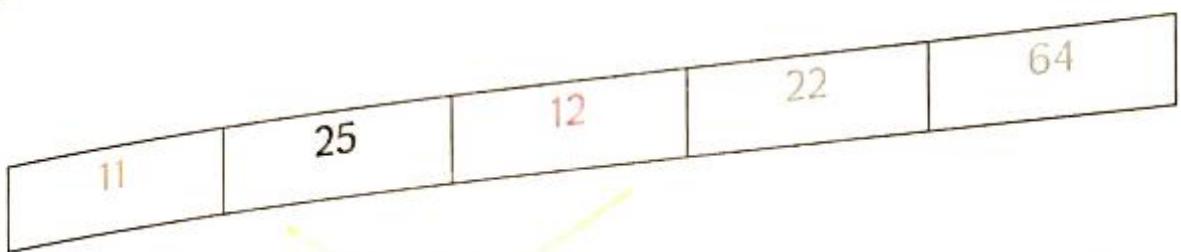
64	25	12	22	11
----	----	----	----	----



Փոխարինելով փոքրագույն տարրը  $i$ -րդի հետ՝ կստանանք

11	25	12	22	64
----	----	----	----	----

Այժմ անհրաժեշտ է դիտարկել  $i = 1$  -ից  $n-1$  հատվածի տարրերը: Այստեղ գլխավոր տարրը 25 -ն է, իսկ  $i + 1$  -ից  $n-1$  հատվածի փոքրագույն տարրը՝ 12 -ը.



Ինչպես նկատեցինք զանգվածն արդեն գտնվում է կարգավորված վիճակում:

կարգավորված վիճակում Array.sort JavaScript ծրագրավորման լեզվում առկա է Array.sort ֆունկցիան, որը համեմատում է զանգվածում տարրերի բոլոր հնարավոր  $a, b$  զույգերը, յուրաքանչյուրի համար դիտարկում է  $a - b$  տարրերությունը, եթե  $a - b < 0$ , ապա տարրերը տեղափոխում են դիրքերով, հակառակ դեպքում տարրերը մնում են իրենց դիրքում: Համեմատությունների ավարտին զանգվածն այլևս գտնվում է դասավորված վիճակում:

Դիտարկենք օրինակը.

```
let x = [2,4,6,8,5,3];
x = x.sort((a,b) => a-b);
console.log(x); // [2, 3, 4, 5, 6, 8]
```

Եթե ցանկանում ենք զանգվածը դասավորել նվազման կարգով, ապա sort ֆունկցիային փոխանցում ենք անանուն ֆունկցիա կամ յամբդա արտահայտություն, որն իր  $a, b$  պարամետրերի համար վերադարձնում է  $b - a$  տարրերությունը:

```
let x = [2,4,6,8,5,3];
x = x.sort( (a,b) => b-a );
console.log(x); // [8, 6, 5, 4, 3, 2]
```

Նկատենք, որ տողային տիպերի դեպքում տեսակավորման այս մեթոդը չի գործում, քանի որ երկու տողային տիպերի տարբերությունը NaN է, այսինքն՝ իրենից ներկայացնում է ոչ թվային արժեք: Այս դեպքում մենք կարող ենք օգտվել տողային տիպերի ոչ թե տարբերությունից, այլ համեմատության արդյունքից:

Օրինակ՝ JavaScript -ում, “a” < “b” արտահայտությունը վերադարձնում է true արժեք, քանի որ ‘a’ տառն այբբենական դիրքում առավել ցածր արժեք ունի, քան բ -ն: Հիմնվելով ասվածի վրա՝ կարող ենք sort ֆունկցիային փոխանցել անանուն կախված համեմատության արդյունքից՝ կվերադարձնի դրական կամ բացասական արժեք, որն էլ sort մեթոդին թույլ կտա որոշել՝ կամ բացասական արժեք է կատարել տարրերի դիրքերի փոփոխություն, թե՛ ոչ:

Դիտարկենք օրինակը.

```
let arr = ["Hayk", "Anna", "Armen", "Mary", "Levon"]
```

```
arr = arr.sort(function(a,b){
    if( a > b){
        return 1;
    }else{
        return -1;
    }
})
```

```
console.log(arr); // ["Anna", "Armen", "Hayk", "Levon", "Mary"]
```

Կոդն առավել կարճ տեսքով կարող ենք ներկայացնել տերնար օպերատորի և յամբդա արտահայտության կիրառմամբ.

```
let arr = ["Hayk", "Anna", "Armen", "Mary", "Levon"]
```

```
arr = arr.sort( (a,b) => a > b ? 1 : -1);
```

```
console.log(arr); // ["Anna", "Armen", "Hayk", "Levon", "Mary"]
```

## 12.5 Անորոշ թվով պարամետրեր

Ծրագրավորման գործընթացում երբեմն լինում են իրավիճակներ, երբ ֆունկցիային փոխանցվող պարամետրերի

քանակը հայտնի չէ: JavaScript -ը մեզ է տրամադրում arguments ստանդարտ օբյեկտը, որը կիրառվում է ֆունկցիայի ներսում և ինֆորմացիա է պարունակում ֆունկցիայի ընթացիկ կանչում փոխանցված պարամետրերի մասին: arguments -ն իրենից ներկայացնում է սովորական զանգված, որի տարրերը ֆունկցիայի պարամետրերն են:

Դիտարկենք օրինակը:

```
function F(){
    return arguments.length;

}
F(1,2,3); //3
F("A", "B"); //2
F([3,2,1,5]); //1
F(); //0
F(true); //1
```

Ինչպես նկատեցինք, մեր կողմից նկարագրված F ֆունկցիան էկրանին ելքագրում է arguments.length -ը, որը ցույց է տալիս ֆունկցիային փոխանցված պարամետրերի քանակը:  
Այժմ փորձենք լուծել հետևյալ խնդիրը. Անհրաժեշտ է գտնել զանգվածին փոխանցված պարամետրերի գումարը.

```
function F(){
    var s = 0;
    for(let i = 0; i < arguments.length; i++){
        s += arguments[i];
    }
    return s;
}
F(1,2,3); //6
F(1,1); //2
F(5,10,10, -5, -20, 4 ); // 4
F(2,5,-7,0,0,1,2,-3,0); // 0
```

Անկախ նրանից, թե քանի պարամետր է փոխանցվել ֆունկցիային, վերջինս հաշվում է իրեն փոխանցված պարամետրերի գումարը:

ECMAScript6 -ից սկսած arguments օբյեկտին կարող է փոխարինել երեք կետերով սահմանվող պարամետրը (rest operator):

Օրինակ.

```
function F(...x){
    if(! x.length)
        return 0;
    return x.reduce((a,b)=> a+b);
}

F(5,10,10, -5, -20, 4); // 4
F(2,5,-7,0,0,1,2,-3,0); // 0
```

Այստեղ ...x արտահայտությունը ցոյց է տալիս, որ ֆունկցիային մեր կողմից գրված ֆունկցիան համարժեք է նախորդ կետում գրվածին: Նկատենք, որ եթե if -ի բլոկում կատարվում է միայն մեկ գործողություն, ապա ձևավոր փակագծերի առկայությունը պարտադիր չէ: Այդ իմաստով ֆունկցիայի ներսում կատարվում է հետևյալը. եթե զանգվածը դատարկ է, ապա return է արվում 0, հակառակ դեպքում' return է արվում գումարը, որի հաշվարկը տեղի է ունենում reduce ֆունկցիայի միջոցով:

## 12.6 Ֆունկցիայի կանչը նկարագրման պահին

Ֆունկցիայի կանչը հնարավոր է իրականացնել նաև նրա ստեղծման պահին: Այս ֆունկցիաներին սովորաբար անվանում են **անմիջապես կանչվող ֆունկցիաներ (immediately invoking functions)**: Բնական է, որ այս ֆունկցիաները նախատեսված են ընդհամենը մեկ անգամ գործարկվելու համար:

Օրինակ՝

```
(function(x){  
    console.log(x*x);  
}) (8)
```

Մեր օրինակում նկարագրեցինք անանուն ֆունկցիա, որն իրեն փոխանցված  $x$  թվի համար ելքագրում է նրա քառակուսին: Եթե ֆունկցիայի նկարագրությունը վերցնում ենք կլոր փակագծերի մեջ, ապա կարող ենք () նշանի օգնությամբ իրականացնել նրա կանչը, ընդ որում՝ փոխանցված 8 -ն արդեն հանդիսանում է անանուն ֆունկցիայի պարամետրը:

Մեր օրինակում ֆունկցիայի աշխատանքի արդյունքում կելքագրվի 64 թիվը:

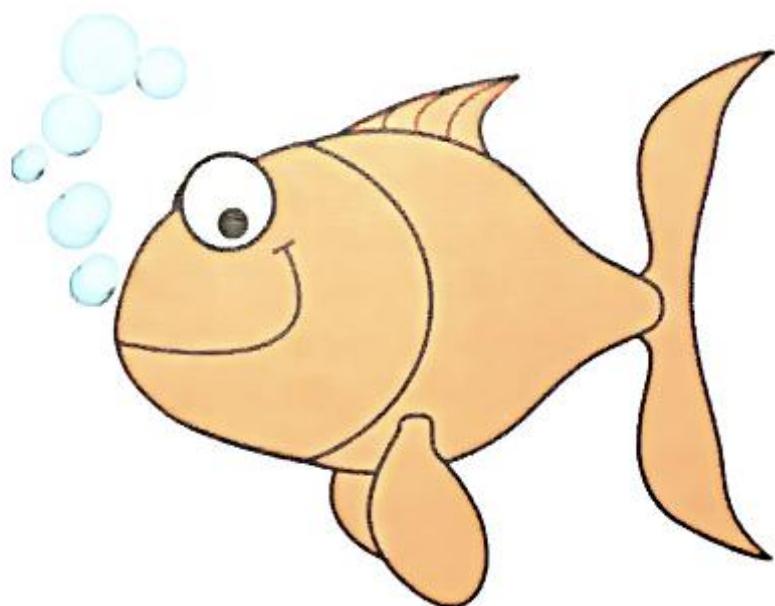
## ՀԱՐՑԵՐ ԵՎ ԱՌԱՋԱԴՐԱՆՔՆԵՐ

1. Ի՞նչ նպատակով են կիրառվում անանուն ֆունկիաները:
2. Ո՞րն է ֆունկցիայի սովորական նկարագրության և անանուն ֆունկցիան փոփոխականի մեջ պահելու միջև եղած տարբերությունը:
3. Բացատրել Array կլասի map, filter, reduce, forEach some, every ֆունկցիաների նշանակությունը:
4. Ի՞նչ է իրենից ներկայացնում յամբյա արտահայտությունը:
5. Տեսակավորման ի՞նչ ալգորիթմներ գիտեք:
6. Բացատրել ընտրություններով տեսակավորման ալգորիթմը:
7. Ո՞ր իրավիճակը կհանդիսանա լավագույն դեպք ընտրություններով տեսակավորման ալգորիթմում:

8. Ո՞ր իրավիճակը կհանդիսանա վատագույն դեպքը ընտրություններով տեսակավորման ալգորիթմում:
9. Բացատրել arguments օբյեկտի և rest օպերատորի դերը:
10. Ինչպե՞ս է հնարավոր կազմակերպել ֆունկցիայի կանչը նկարագրման պահին:
11. Գրել ֆունկցիա, որը, որպես պարամետր, ստանում է զանգված և վերադարձնում է այդ զանգվածը դասավորված աճման կարգով: Օգտագործել ընտրություններով տեսակավորման ալգորիթմը:

**ԴԱՍ 13**

**ԴԻՆԱՄԻԿ  
ԵՐԱԳՐԱՎՈՐՈՒՄ**



## ԴԱՍ 13: ԴԻՆԱՄԻԿ ԾՐԱԳՐԱՎՈՐՄԱՆ ՄԵԹՈԴԸ

---

### 13.1 Դինամիկ ծրագրավորման ներածություն

Վեք կայքերի պատրաստման գործընթացում, ծրագրավորողները մուտքագրում են կայքի ամբողջ պարունակությունը HTML տեգերի օգնությամբ: Ցանկացած էջում առկա է այն, ինչ նկարագրվել է համապատասխան html ֆայլում: Ընդունված է նման էջերին անվանել ստատիկ էջեր: Պատահում են այնպիսի իրավիճակներ, որտեղ ստատիկ էջերի կիրառումը նպատակահարմար չէ:

Ենթադրենք ցանկանում ենք ստեղծել լրատվական կայք, որտեղ պետք է ցուցադրվեն տվյալ օրվա նորությունները: Մեր նպատակն է այդ նորությունները հավաքագրել տարբեր կայքերից և ցուցադրել մեր էջում<sup>13</sup>: Բնական է, որ յուրաքանչյուր օրվա համար այդ տվյալները լինելու են տարբեր, ուստի ստատիկ էջերի կիրառման դեպքում ծրագրավորողն ամեն օր պետք է նորից մուտքագրի առկա նորությունները: Այս իրավիճակում մեզ օգնության է գալիս դինամիկ ծրագրավորման մեթոդը, որը թույլ է տալիս ավտոմատ կերպով ստեղծել տարբեր HTML տեգեր, ավելացնել դրանց պարունակություն և մի շարք հատկություններ: Ստանալով նորությունների զանգվածը՝ հեշտությամբ կկարողանանք ավելացնել այդ տվյալները մեր էջին դինամիկ մեթոդով՝ առանց որևէ տեսակի html կոդի խմբագրումների:

### 13.2 Դինամիկ ծրագրավորում JavaScript -ում

Դինամիկ ծրագրավորման մեթոդը JavaScript -ում, ընդհանուր առմամբ, կազմված է երեք փուլից:

1. տեգի ստեղծում
2. տարբեր հատկությունների ավելացում

---

<sup>13</sup> Տարբեր լրատվական կայքեր RSS -ի օգնությամբ թույլ են տալիս ստանալ այդ տվյալները XML ֆորմատով:

3. տեղի տեղադրումը էջում  
Տեղի ստեղծումը տեղի է ունենում createElement ֆունկցիայի  
օգնությամբ: Հատկությունների ավելացումը, օրինակ՝ id, class,  
գույն, պարունակություն և այլն, տեղի է ունենում DOM -ի  
ստանդարտ ֆունկցիաների օգնությամբ: Վերջնական  
արդյունքն էջին հնարավոր է ավելացնել օրինակ appendChild  
ֆունկցիայի միջոցով:

Ենթադրենք ունենք button (#btn), որի յուրաքանչյուր սեղման  
ժամանակ էջին պետք է դինամիկ կերպով ավելանա h1 տեղ,  
որի ներսում գրված կլինի dynamic element տեքստը կարմիր  
գույնով:

Օրինակ՝

```
let btn = document.querySelector("#btn");
btn.addEventListener("click", function(){
    //ստեղծվում է տեղը ելմ փոփոխականի ներսում
    let elm = document.createElement("h1");
    //Ավելացվում է տեղի պարունակությունը
    elm.innerHTML = "Dynamic Element";
    //փոփոխվում է տեղի գույնը
    elm.style.color = "red";
    //ստեղծվածը ավելանում է body -ին
    document.body.appendChild(elm);
})
```

Այսպիսով՝ createElement ֆունկցիային,  
որպես պարամետր, անհրաժեշտ է  
փոփոխանցել այն տեղի անվանումը, որը  
ցանկանում ենք դինամիկ կերպով  
ստեղծել: Ֆունկցիան վերադարձնում է  
ստեղծվող էլեմենտը, որին էլ հնարավոր է  
արդեն ավելացնել innerHTML, style և  
կամայական այլ հատկություն: Ստեղծված

Dynamic Element

Dynamic Element

Dynamic Element

Dynamic Element

Dynamic Element

տեղը ավելացվում է body -ին appendChild ֆունկցիայի օգնությամբ: Նկատենք, որ document.body -ի փոխարեն կարող էինք կիրառել querySelector ֆունկցիան, որի օգնությամբ ստեղծված դինամիկ տեղը կկարողանանք ավելացնել էջի կամայական կետում:

Վերջնական արդյունքում button -ի յուրաքանչյուր սեղման ժամանակ էկրանին ավելանում է դինամիկ տեղ, որի մեջ գրված է Dynamic Element տեքստը: Ավելացված տեղերը նախապես չեն եղել body -ի ներսում, դրանք ստեղծվել են JavaScript -ի կողմից, այդ պատճառով էլ կոչվում են դինամիկ տեղեր:

**ԽՆԴԻՐ:** Ունենք երկու input դաշտ, որտեղ համապատասխանաբար մուտքագրվում են տողերի և սյուների քանակ: create կոճակի սեղման ժամանակ անհրաժեշտ է դինամիկ ծրագրավորման մեթոդով ստեղծել html այլուսակ, որը կունենա մուտքագրված քանակով tr -ներ և td -ներ: Այլուսակի վանդակներից յուրաքանչյուրի ներսում գրված կլինի որևէ պատահական թիվ: Վանդակի double click (կրկնակի սեղման) -ի ժամանակ տվյալ դաշտում կհայտնվի input, որի մեջ գրված կլինի այն թիվը, որը գրված է վանդակում: Եթե input -ի մեջ կփոխվի թիվը և կսեղմվի կամայական այլ կետում, այդ ժամանակ անհրաժեշտ է td -ից հեռացնել input -ը՝ td -ի մեջ գրելով այն թիվը, որը մուտքագրված է input -ում: Այսպիսով՝ անհրաժեշտ է ստեղծել table -ի տվյալները տեղում փոփոխության ենթարկելու հնարավորություն:

տողերի քանակ

սյուների քանակ

Create

Խնդրի լուծման գործընթացը տրոհենք առանձին մասերի

1. դինամիկ մեթոդով table -ի ստեղծում,
2. յուրաքանչյուր td -ի մեջ պատահական թվի տեղադրում,
3. td -ի կրկնակի սեղման ժամանակ input -ի ավելացում
4. input -ի փոփոխության ժամանակ տվյալների պահպանում:

```

Օրինակ՝
let btn = document.querySelector("#btn");
btn.addEventListener("click", function(){
    let toxer = document.querySelector("#rows").value;
    let syuner = document.querySelector("#cols").value;
    // դիսամիկ մեթոդով աղյուսակի ստեղծման կողը կգրվի
    // այստեղ
})

```

Կողը կարելի է բացատրել հետևյալ կերպ. #btn -ի սեղման ժամանակ պետք է աշխատի անանուն ֆունկցիա, որում toxer,syuner փոփոխականների մեջ պահպանվել են input-ների համապատասխան արժեքները:

Դիտարկենք ներկառուցված ցիկլի միջոցով table -ի ստեղծման կողը.

```

// դատարկենք #main div -ի պարունակությունը, որպեսզի
// նրանում ավելացնենք աղյուսակը
document.querySelector("#main").innerHTML = "";
// ստեղծենք table տեղ
let table = document.createElement("table");
// կիրառենք ցիկլ, որը կկատարվի այնքան անգամ, որքան տող,
// որ անհրաժեշտ է ստեղծել
for(let i = 0; i < toxer; i++) {
    // ցիկլի յուրաքանչյուր քայլում ստեղծենք tr տեղ
    let tr = document.createElement("tr");
    // կիրառենք մեկ այլ ցիկլ, որը ստեղծված tr -ին
    // կավելացնի անհրաժեշտ բանակով td -ներ
    for(let j = 0; j < syuner; j++) {
        // ցիկլի յուրաքանչյուր քայլում ստեղծենք td տեղ
        let td = document.createElement("td");
        // ստանանք թիվը, որը պետք է գրված լինի td -ի մեջ
        let number = parseInt(Math.random() * 50);
    }
}

```

պատսիսկան թիվը ավելացնենք

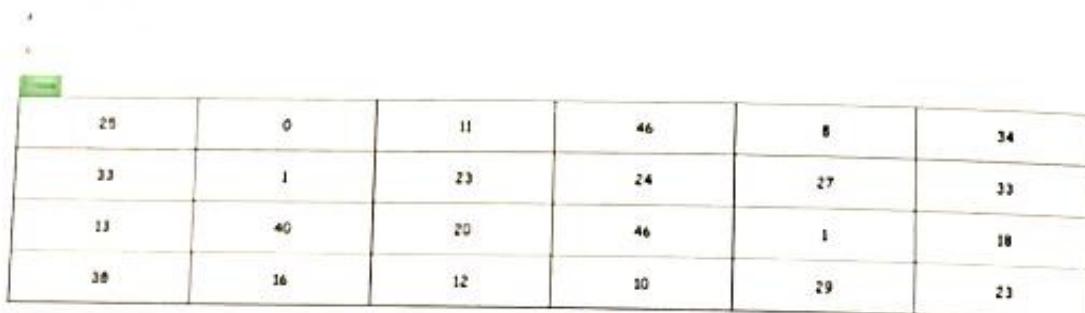
```

td -ին՝ որպես innerHTML
td.innerHTML = number;
// td տեղի double click -ի ժամանակ պետք է
// աշխատի edit ֆունկցիան, որը դեռ չենք
// նկարագրել
td.addEventListener("dblclick", edit)

// ավելացնենք td -ին tr -ին
tr.appendChild(td);
}

// ստեղծված tr տեղն ավելացնենք table -ին
table.appendChild(tr);
}

// ցիկլից դուրս. Եթե բոլոր տարրերն ավելացված են table -ին,
// table -ն ավելացնել #main տեղին
document.querySelector("#main").appendChild(table)
Այս պահին արդեն input դաշտերում մուտքագրելով օրինակ 4 և
6, կոճակի սեղման ժամանակ կստանանք հետևյալ արդյունքը:
```



25	0	11	46	8	34
33	1	23	24	27	33
13	40	20	46	1	18
38	16	12	10	29	23

Ինչպես նկատեցինք, կոդում #main տեղի innerHTML - ին վերագրվում է դատարկ արժեք: Պատճառն այն է, որ button -ի յուրաքանչյուր սեղման ժամանակ դինամիկ կերպով ստեղծվում է table: Եթե մինչ ցիկլը #main -ը չդատարկվի, ապա բոլոր ստեղծված table-ները կշարունակեն մնալ էկրանին:

Դինամիկ կերպով ստեղծված `td` -ներին `addEventListener` -ի  
միջոցով կցվել է `edit` ֆունկցիա, որը կաշխատի `td` -ի կրկնակի  
սեղման ժամանակ:

Դիտարկենք կոդը.

```
function edit(){
    // val -ի մեջ պահպանենք կրկնակի սեղմված td -ի ներսում
    // գրված թիվը
    let val = this.innerHTML;
    // item փոփոխականի մեջ պահենք this -ը
    let item = this;
    // դինամիկ մեթոդով ստեղծենք input տեղ
    let input = document.createElement("input");
    // input -ի ներսում գրել պահպանված թիվը
    input.value = val;
    // դատարկենք td -ն
    this.innerHTML = "";
    // td -ին ավելացնենք դինամիկ ստեղծված էլեմենտը
    this.appendChild(input)
    // Ակտիվացնենք input -ը, որպեսզի նրանում
    // անմիջապես հնարավոր լինի տվյալներ մուտքագրել
    input.focus();
    // input -ին ավելացնենք blur14 event -ը
    input.addEventListener("blur", function(){
        // this -ը այստեղ արդեն ցոյց է տալիս input -ը
        // վերցնենք input -ի արժեքը
        let newVal = this.value;
        // td -ի մեջ գրենք փոփոխված արժեքը
        item.innerHTML = newVal;
    })
}
```

---

<sup>14</sup> `blur` event -ն աշխատում է այն ժամանակ, երբ տվյալ տեղից դուրս որևէ գործողություն է կատարվում: Այլ կերպ ասած՝ երբ տվյալ տեղի ակտիվ վիճակը փոխվում է:

Այժմ կարող է առաջանալ հետևյալ հարցը. ո՞րն է  
let item = this գործողության նպատակը:



Ինչպես արդեն նշեցինք edit ֆունկցիայի ներսում  
this -ը ցույց է տալիս td -ին, սակայն edit -ի  
ներսում մենք blur event -ը կապեցինք input -ին,  
այս դեպքում անանուն ֆունկցիայի ներսում this -ը ցույց է տալիս  
տվյալ input -ը, իետևաբար td -ին կրկին դիմելու համար  
անհրաժեշտ էր օգտագործել փոփոխական, որն էլ  
հանդիսացավ item -ը<sup>15</sup>:

Չատ հաճախ նմանատիպ խնդիրների լուծման համար  
կիրառում են input -ի change event -ը, որը շատ նման է blur -ին:  
Այստեղ կարևոր այն է, որ change event -ն աշխատում է միայն  
այն դեպքում, եթե input -ում արվում է որևէ փոփոխություն,  
մնացյալ դեպքերում event -ը չի աշխատում, իսկ blur -ն  
աշխատում է ցանկացած դեպքում, եթե տվյալ դաշտի ակտիվ  
վիճակը փոխվում է: Արդյունքում, եթե մենք օգտագործենք  
change -ը, double click կատարենք td -ի վրա և ստեղծված input  
-ում ոչինչ չփոփոխենք, ապա input -ը կմնար նույնիսկ այն  
դեպքում, եթե մենք double click անենք այլ դաշտի վրա: Ուստի  
առավել նպատակահարմար է կիրառել blur event -ը:

Այժմ կիրառենք ստեղծվածը.

1. ստեղծենք դինամիկ table,
2. double click կատարենք td -ներից որևէ մեկի վրա
3. double click կատարենք նաև դինամիկ ստեղծված input  
-ի վրա

Արդյունքում կունենանք ստորև բերված պատկերը.

---

<sup>15</sup> Նկատենք, որ եթե this -ը ցույց է տալիս input տեղը, ապա  
this.parentNode -ի միջոցով կարող ենք դիմել input -ի ծնող տեղին՝ td-ին:

20	11
<input>	
4	47
31	10

Այսինքն՝ input -ի ներսում գրվել է <input> տերսոր:

Փորձենք հասկանալ,թե ինչո՞ւ է նման բան տեղի ունենում և շտկենք առկա սխալը:

Ընդհանրապես, կրկնակի սեղման event -ը կապված է td -ին, սակայն, քանի որ, input դաշտը հանդիսանում է ժառանգ տեղ td -ի համար, ապա նրա նկատմամբ double click -ը նույնպես աշխատում է: Ինչպես գիտենք, double click -ի ժամանակ մենք վերցնում ենք td -ի innerHTML -ը և այն ավելացնում ենք դինամիկ ստեղծված input -ին: Եթե input -ն արդեն ստեղծված է, ապա td -ի innerHTML -ը կլինի <input> -ը, որն էլ հենց ավելացել է դինամիկ input -ին՝ որպես value: Այսպիսով՝ տեղի է ունեցել event-ների ժառանգում, որը կարելի է բացառել stopPropagation ֆունկցիայի օգնությամբ՝ edit ֆունկցիայի վերջում ավելացնելով հետևյալը.

```
input.addEventListener("dblclick", function(e){
  e.stopPropagation();
```

})

Այսպիսով՝ stopPropagation ֆունկցիան կանխում է event-ների ժառանգումը:

## ՀԱՐՑԵՐ ԵՎ ԱՌԱՋԱԴՐԱՆՔՆԵՐ

1. Ո՞րն է դինամիկ ծրագրավորման մեթոդի կիրառման նշանակությունը:
2. Ի՞նչ է նշանակում ստատիկ html էջ արտահայտությունը:
3. Բացատրել createElement և appendChild ֆունկցիաների նշանակությունը:
4. Ո՞րն է stopPropagation ֆունկցիայի նշանակությունը:
5. Ի՞նչ է իրենից ներկայացնում parentNode -ը:
6. Դինամիկ մեթոդով էջի վրա ստեղծել պատահական դիրքերում գտնվող շրջանագծեր<sup>16</sup>, որոնցից յուրաքանչյուրն ունի պատահական գույն:
7. Ունենք զանգված, որի տարրերն են տարրեր նկարների ֆայլային կամ օնլայն հասցեներ: Զանգվածում առկա նկարներն ավելացնել body -ին՝ դինամիկ ծրագրավորման մեթոդով: Յուրաքանչյուր նկարի սեղման ժամանակ նրան ավելացնել active կլասը:  
**.active{  
border: 2px solid;  
}**

Եթե նկարն արդեն ունի active կլասը, ապա նրա սեղման ժամանակ հեռացնել այդ կլասը:

---

<sup>16</sup> Որպես շրջանագիծ անհրաժեշտ է կիրառել div տեղը, որի լայնությունը և բարձրությունը միմյանց հավասար են, իսկ border-radius պարամետրի արժեքն է 50%:

# ԴԱՍ 14

# jQUERY

# ԳՐԱԴԱՐԱՆԸ



## ԴԱՍ 14: JQUERY ՆԵՐԱԾՈՒԹՅՈՒՆ

---

### 14.1 Գաղափար Library-ի և Framework - ի մասին

Յուրաքանչյուր ծրագրավորման լեզու կարող է ունենալ իր հետ կապված որոշակի տեխնոլոգիաներ, որոնց օգնությամբ դյուրացվում է ծրագրավորման գործընթացը՝ հնարավորություն տալով գրել առավել կարճ կոդեր և օգտագործել մի շարք պատրաստի գործիքներ: JavaScript -ը նույնպես բացառություն չէ: Ընդունված է նմանատիպ հավելյալ տեխնոլոգիաները բաժանել երկու խմբի՝ **library** և **framework**:

Library-ն կամ գրադարանը, իրենից ներկայացնում է որոշակի գործիքների խումբ, որոնք միտված են հեշտացնելու ծրագրավորման գործընթացը:

Framework -ը մի փոքր ավելի բարդ կառուցվածք ունեցող համակարգ է, որի հետ աշխատանքը կազմակերպվում է խիստ որոշակի կանոններով:

JavaScript ծրագրավորման լեզվի համար Library է հանդիսանում, jQuery -ին, որին և մենք կանդրադառնաք: JS -ի ամենահայտնի framework-ներից են Angular -ը, React.js -ը և այլն:

Ինչպես նշվեց, library -ին (գրադարանը) ֆունկցիաների խումբ է, որը script տեգի օգնությամբ կարելի է կցել էջին և օգտվել նրանում առկա հնարավորություններից: Framework -ը՝ որպես համակարգ աբստրակտ կառուցվածք ունի: Բացի այդ, Framework -ն իր հերթին կարող է պարունակել մի շարք գրադարաններ: JavaScript -ի ամենահայտնի framework-ներից է **Angular** -ը: Վերջինս հիմնված է կոմպոնենտային ծրագրավորման վրա: Այստեղ ընթացիկ նախագիծը (project) բաժանվում է առանձին մասերի՝ կոմպոնենտների, որոնցից յուրաքանչյուրն ունի իր առանձին view-ն (այսինքն՝ html css կառուցվածքը): Օրինակ՝ եթե մենք մշակում ենք օնլայն խանութ, ապա Angular -ում անհրաժեշտ կլինի կիրառել Product, Basket կոմպոնենտներ, որտեղ Product -ը կներկայացնի վաճառվող

ապրանքների տվյալները, իսկ Basket -ը՝ զամբյուղի: Մշակվող նախագիծն առանձին մասերի տրոհելու իմաստն է կոդերի պարզեցումը: Այսպիսով՝ library և framework -ի տարրերությունն այն է, որ framework -ը ավելի բարդ կառուցվածք ունեցող համակարգ է, որտեղ ծրագրավորման գործընթացը տեղի է ունենում միայն տվյալ framework -ին բնորոշ կանոններով, իսկ գրադարանը պարզ համակարգ է, ֆունկցիաների խումբ, որից օգտվելու համար անհրաժեշտ է այն կցել էջին:

## 14.2 jQuery - ներածություն

JavaScript ծրագրավորման լեզվի ամենահայտնի գրադարանը jQuery -ին է, վերջինս պարզեցնում է HTML տեգերի կառավարման գործընթացը, վեր կայքերում անհմացիաների, event-ների և մի շարք այլ գործընթացների իրականացումը: jQuery -ին մեր կայքին միացնելու համար անհրաժեշտ է այն բեռնել [jquery.com](https://jquery.com) կայքից կամ օգտվել CDN<sup>17</sup> – ներից: CDN-ի կիրառման դեպքում մենք մեր կայքին ավելացնում ենք <script> տեգ, որի մեջ առկա է jQuery գրադարանի հղումը.

օրինակ՝

```
<script src="https://ajax.googleapis.com/ajax/libs/jquery/3.3.1/jquery.min.js"></script>
```

Հակառակ դեպքում, եթե script -ի համար նախատեսված ֆայլը բեռնել ենք և տեղադրել մեր կայքում, այն կարող ենք ավելացնել մեզ արդեն ծանոթ մեթոդով.

```
<script src="js/jquery.min.js"></script>
```

Ընդհանրապես CDN -ի կիրառումը մեզ տալիս է մի քանի առավելություններ. Նախ՝ CDN-ների բաշխման արդյունքում jQuery -ին տրամադրվում է այն սերվերից որն առավել մոտ է տվյալ օգտատիրոջը: Այս դեպքում, նաև, jQuery -ի ֆայլն ենթարկվում է browser caching -ի. այսինքն՝ եթե օգտատերը այցելել է որևէ կայք, որում առկա է եղել jQuery որևէ CDN -ից,

---

<sup>17</sup> CDN (content delivery network) – իրենից ներկայացնում է տարրեր աշխարհագրական դիրքերում գտնվող սերվերներ խումբ, որոնց միջոցով, մեծ արագությամբ հնարավոր է ստանալ տարրեր ֆայլեր:

ապա Browser -ն իր հիշողության մեջ է դնում ֆայլը, այնուհետև երբ օգտատերը այցելում է մեկ այլ կայք, որում առկա է jQuery -ին, ֆայլն այլևս չի բեռնվում և browser-ն օգտվում է cache<sup>18</sup>-ում առկա ֆայլից:

Հայտնի են jQuery ֆայլի երկու հիմնական տեսակները:

1. **jquery.js** – Այս ֆայլը կիրառվում է կայքի ծրագրավորման փուլում (development stage), այստեղ կոդը գրված է մեզ համար հասկանալի պարզ կառուցվածքով:
2. **jquery.min.js** – Այս ֆայլը կիրառվում է արտադրական փուլում (production mode), երբ կայքն ամբողջությամբ պատրաստ է: Այս դեպքում ֆայլում գրված կոդը գտնվում է մինիմալացված<sup>19</sup> վիճակում, որի դեպքում կոդն անընթեռնելի է:

Ուշագրավ է այն, որ մեր կողմից մշակվող script -ի ֆայլերն անհրաժեշտ է կայքին կցել jQuery -ից հետո, որպեսզի մեր ֆայլում հասանելի լինեն jQuery -ի բոլոր ֆունկցիաները:  
Օրինակ՝

```
<script src="https://ajax.googleapis.com/ajax/libs/jquery/3.3.1/jquery.min.js"></script>
```

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

<sup>18</sup>Հիշողության տարածք է browser -ում, որտեղ տեղակայվում են տարբեր ստատիկ տարրեր՝ նկարներ, ֆայլեր և այլն, որոնք հետագայում կիրառվում են browser -ի կողմից:

<sup>19</sup>JavaScript -ի մինիմալացված ֆայլերում (minified files) կոդն ամբողջությամբ տեղակայվում է հորիզոնական դիրքում, այստեղ չկան ավելորդ բացատներ, սիմվոլներ և այլն: Արդյունքում՝ կոդը դառնում է անընթեռնելի:

### 14.3 jQuery - սինտաքսը

jQuery տեխնոլոգիայում կողի կառուցվածքը շատ պարզ է.  
**\$(selector).action()**

Այստեղ selector (css նշիչ) -ը ցույց է տալիս, թե ո՞ր տեղերի հետ  
է կատարվում գործողությունը, իսկ action -ը այն  
գործողությունը, որը պետք է կատարվի, իսկ \$ նշանը թույլ է  
տալիս դիմել jQuery -ին:

Ըստհանրապես jQuery -ում կոդ գրելու գործընթացը  
մեկնարկվում է document.ready բլոկի ներսում.

**\$document.ready(function(){**

// կոդը գրվում է այստեղ

**})**

Այստեղ ready-ն HTML փաստաթղթի հետ կապված event է: Նրա  
ներսում գրված կոդն աշխատում է միայն այն ժամանակ, եթե  
HTML էջն ամբողջությամբ թողարկվել է և պատրաստ է  
գործարկման: Սա նաև թույլ է տալիս արդեն JS -ի <script> տեղը  
տեղադրել ոչ թե body -ի վերջում, այլ կամայական կետում:

Ուշագրավ է նաև այն, որ \$ նշանի փոխարեն կարելի է կիրառել  
jQuery բառը:

**jQuery(document).ready(function(){**

// կոդը գրվում է այստեղ

**})**

Մենք կարող ենք նաև jQuery -ին թողարկել մեր կողմից  
ընտրված կամայական փոփոխականի միջոցով: Այդ իմաստով  
անհրաժեշտ է կիրառել noConflict ֆունկցիան:

Օրինակ.

**let app = jQuery.noConflict();**

**app(document).ready(function(){**

// կոդը գրվում է այստեղ

**})**

Մեր օրինակում jQuery -ի բոլոր ֆունկցիաները կարող են  
կանչվել արդեն app փոփոխականի օգնությամբ: NoConflict -ի  
կիրառման անհրաժեշտությունը կարող է առաջանալ այն

Ժամանակ, երբ մեր կայքին կցված լինի մեկ այլ գրադարան, որը նույնպես օգտագործում է \$ -ի նշանը՝ որպես իր հիմնական գործիք: Այդ ժամանակ, մենք կարող ենք կիրառել կամ jQuery տարրերակը, կամ մեր կողմից ընտրված որևէ փոփոխական:

#### 14.4 jQuery - Պարզագույն

##### event-ներ

Այժմ ծանոթանանք jQuery -ի հիմնական գործիքներին: Եջի վրա ունենք button (#btn) և div (#main): Div -ին տանք որոշակի գույն, լայնություն և բարձրություն: Մեր նպատակն է button -ի սեղման ժամանակ անհետացնել div -ը:

Դիտարկենք կոդը.

Click

```
$("document").ready(function(){
    $("#btn").click(function(){
        $("#main").hide();
    })
})
```

Ինչպես նկատեցինք, \$("#btn") -ի միջոցով մենք ընտրում ենք այն տեղը, որի նկատմամբ պետք է իրականանա click event -ը: jQuery -ում բոլոր event-ները իրենցից ներկայացնում են ֆունկցիաներ, որոնք որպես պարամետր ստանում են անանուն ֆունկցիա, որն էլ իր հերթին աշխատում է տվյալ event-ի կատարման ժամանակ: Մեր օրինակում, click -ի ժամանակ կատարման ժամանակ: Մեր օրինակում, click -ի ժամանակ կանչվել է \$("#main").hide() ֆունկցիան, որը #main էլեմենտը թարցնում է:

Տեղադրենք մեկ այլ Button (#btn2) -ը, որի սեղման ժամանակ ետ կվերադարձնենք տվյալ div -ը:

Ուշադրություն դարձնենք այն հանգամանքին, որ մեզ մոտ կողն ամբողջությամբ գրվում է `document.ready` -ի բլոկում:

```
$(document).ready(function(){
    $("#btn").click(function(){
        $("#main").hide();
    })
    $("#btn2").click(function(){
        $("#main").show();
    })
})
```

`show` և `hide` մեթոդները `jQuery` -ի ստանդարտ մեթոդներ են, որոնք թույլ են տալիս ավելացնել էֆեկտներ: Երկու մեթոդներն ել կարող են ստանալ պարամետր, որը նախատեսված է էֆեկտի տևողությունը նշելու համար:

Օրինակ՝ `$("#main").show("slow")`; այս դեպքում, `#btn2` -ի սեղման ժամանակ, `div` -ը էկրանին կհայտնվի դանդաղ: Որպեսզի էֆեկտն աշխատի արագ, անհրաժեշտ է նրան փոխանցել `fast` կոչվող պարամետրը:

```
$("#main").show("fast");
```

Նկատենք, որ մենք ունենք հնարավորություն ֆունկցիային փոխանցել նաև միլիվայրելյանների այն քանակը, որն անհրաժեշտ է կիրառել տվյալ էֆեկտի տևողության համար:

Օրինակ՝

```
$("#main").show(3000);
```

Այս դեպքում էֆեկտի տևողությունը կկազմի 3 վայրկյան:

`show` և `hide` էֆեկտների աշխատանքը իհմնված է տվյալ տեղի `width` և `height` հատկությունների փոփոխման վրա:

`jQuery` -ին մեզ տրամադրում է նաև մի շարք այլ էֆեկտներ.

- `fadeIn` - Էֆեկտը opacity<sup>20</sup> հատկության միջոցով էլրանին ցուցադրում է տվյալ տեղը
- `fadeOut` - Էֆեկտը opacity հատկության միջոցով էլրանից հեռացնում է տվյալ տեղը
- `slideDown` - Էֆեկտը կիրառում է տվյալ տեղի height հատկությունը, որպեսզի վերջինս ցուցադրի էջում:
- `slideUp` - Էֆեկտը կիրառում է տվյալ տեղի height հատկությունը, որպեսզի թաքցնի նրան էջից
- `toggle` - Երկակի ֆունկցիա է, հիմնված `show/hide` էնֆեկտների վրա: Եթե տեղը թաքնված է, ապա այն կանչում է `show` ֆունկցիան, հակառակ դեպքում՝ `hide`-ը
- `slideToggle` - Երկակի ֆունկցիա է, հիմնված `slideUp/slideDown` էնֆեկտների վրա: Եթե տեղը թաքնված է, ապա այն կանչում է `slideDown` ֆունկցիան, հակառակ դեպքում՝ `slideUp`-ը
- `fadeToggle` - Երկակի ֆունկցիա է, հիմնված `fadeIn/fadeOut` էնֆեկտների վրա: Եթե էլեմենտը թաքնված է, ապա այն կանչում է `fadeIn` ֆունկցիան, հակառակ դեպքում՝ `fadeOut`-ը:

---

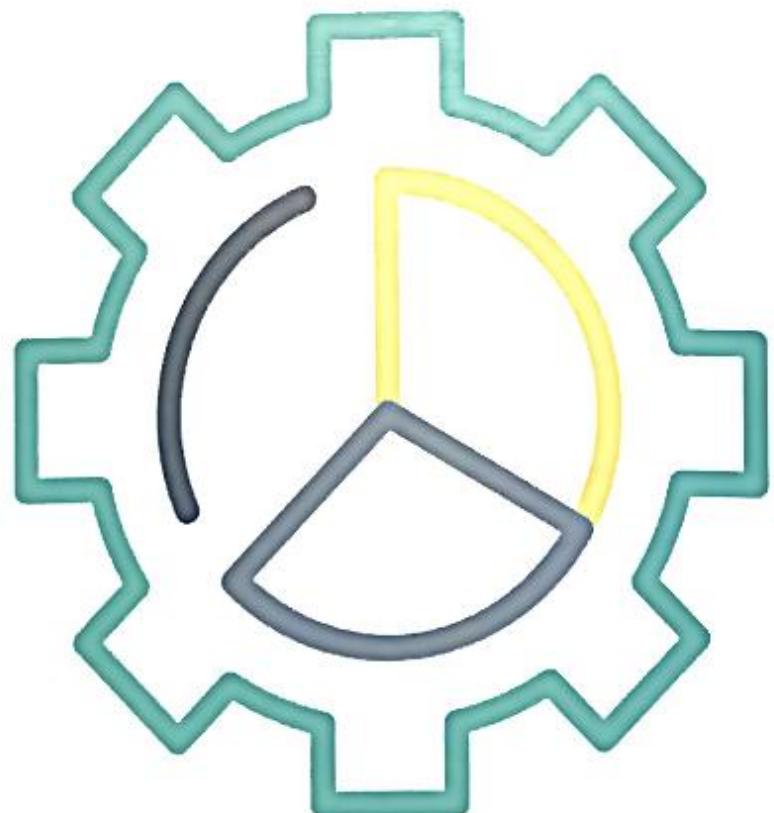
<sup>20</sup> CSS -ի հատկություն է, որի արժեքը [0;1] հատվածի թիվ է: Ցույց է տալիս տվյալ տեղի թափանցելիությունը:

## ՀԱՐՑԵՐ ԵՎ ԱՌԱՋԱԴՐԱՆՔՆԵՐ

1. Ի՞նչ է իրենից ներկայացնում jQuery -ին:
2. Ո՞րն է jquery.js և jquery.min.js ֆայլերի միջև տարբերությունը:
3. Ո՞րն է \$ նշանի դերը jQuery -ում:
4. Բացատրել \$.noConflict ֆունկցիայի նշանակությունը:
5. Ո՞րն է document.ready -ի նշանակությունը:
6. Ի՞նչպիսի էֆեկտների եք ծանոթ jQuery -ում:
7. Ո՞րն է hide/show և toggle էֆեկտների միջև առկա տարբերությունը:
8. Ո՞րն է document.ready -ի նշանակությունը:
9. Ի՞նչ է Library -ն:
10. Ի՞նչ է framework -ը:
11. jQuery -ին գրադարան է, թե framework:
12. Որո՞նք են JS -ի ամենահայտնի framework-ները:
13. Ինչով է տարբերվում Angular -ը jQuery -ից:

# ԴԱՍ 15

## jQuery ԿԱՌԱՎԱՐՈՒՄ



## ԴԱՍ 15: JQUERY ԿԱՌԱՎԱՐՈՒՄ

**15.1 HTML տեգերի կառավարումը**  
Այժմ փորձենք հասկանալ,թե ինչպե՞ս է հնարավոր կառավարել  
HTML տեգերը և դրանց CSS հատկությունները jQuery -ում:  
JavaScript -ի ստանդարտ innerHTML  
հատկությունը      jQuery      -ում  
փոխարակնված է html ֆունկցիայով:  
Դիցուք ունենք div, որի ներսում գրված է  
0 թիվը: Div -ի յուրաքանչյուր սեղման  
ժամանակ ցանկանում ենք ավելացնել  
նրա ներսում գտնվող թիվը մեկով:  
Օրինակ՝

```
$(document).ready(function(){
    $("#main").click(function(){
        let tiv = $(this).html();
        tiv++;
        $(this).html(tiv);
    })
})
```

Ինչպես նկատեցինք, #main – ի սեղման ժամանակ աշխատում  
է ֆունկցիա, որի ներսում **let tiv = \$(this).html()** գործողության  
միջոցով tiv փոփոխականի ներսում պահպանվում է div -ի  
ներսում գրված ընթացիկ արժեքը, այնուհետև կատարվում է  
tiv++ գործողությունը և վերջնական արժեքը կրկին  
պահպանվում է div -ի ներսում:

Այսպիսով՝ **var tiv = \$(this).html();** jQuery -ի հրամանը  
համարժեք է հետևյալ JavaScript -ի հրամանին.  
**var tiv=this.innerHTML;**



Եվ `$(this).html(tiv);` հրամանը համարժեք է  
`this.innerHTML=tiv;` հրամանին:

jQuery -ում որևէ դաշտի html պարունակությունը փոփոխության ենթարկելու և այդ պարունակությունը ստանալու համար կիրառվում է նոյն ֆունկցիան:

Սունձնակի հետաքրքրություն է ներկայացնում `text` ֆունկցիան: Վերջինս համարժեք է html -ին և կիրառվում է նոյն նպատակով, սակայն նրանց միջև առկա է փոքր տարբերություն. html ֆունկցիան կիրառելու դեպքում կարող ենք տվյալ էլեմենտին ավելացնել նաև նոր տեգեր, իսկ `text` -ի դեպքում գոյծ ունենք բացառապես տեքստի հետ:

Օրինակ՝

`$(this).text("<p>" + tiv + "</p>");`

Եթե խոսքը վերաբերվում է input-ների value-ներին, jQuery -ին մեզ տրամադրում է `val`

`<p>1</p>`

ֆունկցիան:

Ենթադրենք ունենք input դաշտ (#inp): Մուտքային դաշտի ներսում գրված արժեքը հնարավոր է վերցնել հետևյալ հրամանի միջոցով.

`let x = $("#inp").val();`

Նոյն տրամաբանությամբ, եթե ցանկանանք մուտքային դաշտի ներսում գրել օրինակ "Welcome" բառը, ապա

կկատարենք հետևյալ գործողությունը.

`$("#inp").val("Welcome");`

Այժմ ծանոթանանք `attr` ֆունկցիային, որի օգնությամբ կկարողանանք տարբեր հատկություններ ավելացնել HTML տեգերին:

Օրինակ՝

`$("#inp").attr("id", "my-element-1");`

Հրամանի կատարման արդյունքում #inp տեգի id հատկությունը

կստանա `my-element-1` արժեքը:

Եթե անհրաժեշտ է տեղին փոխանցել մեկից ավել հատկություններ, ապա attr ֆունկցիային կարող ենք փոխանցել օբյեկտ նշանակում է փոփոխական, օրյեկտ: Ըստհանրապես օբյեկտ նշանակում է փոփոխական, որն իր մեջ կարող է պարունակել տարրեր տվյալներ: Օրյեկտների մասին առավել մանրամասն կխոսենք հետագայում: Այժմ, ցույց տանք միայն՝ ինչպե՞ս կարելի է օգտագործել attr ֆունկցիան օբյեկտի հետ:

```
$("#inp").attr({id: 'my-element', max: 4});
```

Տեղից հատկություններ հեռացնելու համար օգտվում ենք removeAttr ֆունկցիայից:

Օրինակ՝  

```
$("#inp").removeAttr("disabled");
```

Այս իրամանի կատարման արդյունքում input տեղից կհեռացվի նրա disabled հատկությունը:

Տեզերին կլասներ ավելացնելու համար, իհարկե, կարող ենք օգտագործել attr ֆունկցիան, սակայն վերջինս փոխում է class հատկության ամբողջ պարունակությունը, հետևաբար, չենք կարող օգտագործել մեկից ավել կլասներ ավելացնելու դեպքում:

Օրինակ՝  

```
$("#inp").attr("class","A");  
$("#inp").attr("class","B");
```

Տեղը կստանա միայն B անունով կլասը:

Այս իրավիճակում կարող ենք օգտագործել addClass ֆունկցիան.

```
$("#inp").addClass("A");  
$("#inp").addClass("B");
```

Այս դեպքում էլեմենտը կունենա և A, և B կլասները:

Ըստհանրապես, հատկությունները լինում են ստանդարտ և արհեստական: Ստանդարտ հատկությունները առկա են HTML -ում ստանդարտ տեզերի համար: Օրինակ՝ id, class, onclick և այլն: Արհեստական հատկությունները, դրանք ծրագրավորողի կողմից ստեղծված հատկություններ են, որոնք ունեն data-\*

տեսքը, որտեղ \* -ի փոխարեն կարող է լինել ցանկացած բառ: Օրինակ՝ data-id, data-animal-type և այլն:

Նման հատկություններ ավելացնելու համար կրկին կարող ենք կիրառել attr ֆունկցիան, սակայն խորհուրդ է տրվուած օգտագործել նաև data ֆունկցիան:

Փորձենք հասկանալ տարբերությունները.

```
$("#main").attr("data-id",4);  
$("#main").data("id",4);
```

attr կիրառման դեպքում, data-id հատկությունը երևալու է browser -ի inspect պատուհանում (տեսնելու համար սեղմել F12 կամ ctrl + shift + c), data -ի դեպքում՝ ոչ: Ֆունկցիաներից որևէ մեկով ավելացված հատկությանը հնարավոր չէ դիմել մյուսով: Այսպիսով՝ եթե հատկությունը ավելացվել է data ֆունկցիայով, սակայն փորձում ենք ետ ստանալ attr -ով՝ կստանանք undefined.

```
$("#main").data("id",4);  
console.log($("#main").attr('data-id')); //undefined
```

Բացի այդ, data ֆունկցիայի օգնությամբ փոխանցվող տվյալի տիպը պահպանվում է, մինչդեռ attr -ն միշտ վերադարձնում է string:

Օրինակ.

```
$("#main").attr("data-id",4)  
let x = $("#main").attr("data-id") + 5  
console.log(x); //45
```

Արդյունքում ստացվել է 45, պատճառն այն է, որ 4 և 5 արժեքները գումարվել են որպես տողային տիպի տվյալներ: Այժմ փորձենք նոյն գործողությունը իրականացնել data ֆունկցիայի օգնությամբ:

```
$("#main").data("id",4);  
let x = $("#main").data("id") + 5  
console.log(x); //9
```

Այս դեպքում արդեն ստացվեց 9, քանի որ data ֆունկցիան աշխատում է ոչ միայն արժեքների, այլև տիպերի հետ: Այժմ կրկին անդրադառնանք կլասների հատկություններին:

Կլասը հեռացնելու համար կիրառենք removeClass ֆունկցիան, իսկ եթե ցանկանում ենք ստուգել՝ արդյո՞ք տվյալ տեղն ունի որևէ կոնկրետ կլաս, թե ոչ, ապա կարող ենք օգտագործել hasClass ֆունկցիան:

Օրինակ՝

```
$("#main").click(function(){
    if($(this).hasClass("active")){
        $(this).removeClass("active");
    }else{
        $(this).addClass("active");
    }
})
```

Մեկնաբանանեք վերևում գրված կոդը: #main տեղը հանդիսանում է մեր div -ը, որի հետ կապված է սեղման event (click), ընդ որում՝ սեղման ժամանակ, եթե div -ն ունի active կլասը, ապա նրանից հեռացվում է այդ կլասը, հակառակ դեպքում՝ ավելացվում:

Այս արդյունքին կարելի էր հասնել՝ պարզապես կիրառելով toggleClass ֆունկցիան: Վերջինս ստուգում է՝ արդյո՞ք տեղն ունի տվյալ կլասը և կախված արդյունքից՝ ավելացնում այն, կամ հեռացնում:

```
$(this).toggleClass("active");
```

## 15.2 CSS և անիմացիա

jQuery -ում տեգերին սթայլեր կարելի է ավելացնել css մեթոդի օգնությամբ:

Օրինակ՝

```
$("#inp").css("color", "red");
```

Մեր օրինակում #inp տեգի համար color -ը կդառնա red:

Նկատենք, որ այստեղ նույնպես մեկից ավելի շատ պարամետրերի դեպքում անհրաժեշտ է կիրառել օբյեկտ. օրինակ՝

```
$("#inp").css({
    color: "red",
    "background-color": "white",
    "font-size": "24px",
    display: "block"
})
```

Նկատենք, որ բոլոր CSS -ի պարամետրերը, որոնք նկարագրվում են մի քանի բառերի օգնությամբ, օրինակ՝ background-color, օբյեկտի ներսում գտնվում են ապաթարցների ներսում:

Այժմ անդրադառնանք անհմացիայի կիրառումներին: jQuery -ում հնարավոր է անհմացիայի ենթարկել HTML տեգերի միայն այն պարամետրերը, որոնք թվային են: Այսպիսով՝ ցանկացած տեգի համար, օրինակ, width -ը կարելի է ենթարկել անհմացիայի, մինչդեռ color -ը՝ ոչ:

jQuery -ում անհմացիաներ ստեղծելու համար կիրառում ենք animate ֆունկցիան, որն ունի երկու հիմնական պարամետրեր՝ անհմացիան նկարագրող օբյեկտ և տևողություն:

Օրինակ՝ ստեղծենք անհմացիա, որը div -ի սեղման ժամանակ նրա լայնությունը և բարձրությունը կդարձնի 400px՝ 2 վայրկյանի ընթացքում:

```
$("#main").click(function(){
    $(this).animate({
        width: "400px",
        height: "400px"
    }, 2000);
})
```

Այստեղ 2000 -ը ցույց է տալիս անհմացիայի տևողությունն արտահայտված միլիվայրեկյաններով:

Ստորև բերված կոդի կատարման արդյունքում, div -ի յուրաքանչյուր սեղման ժամանակ նրա չափսերը կմեծանան 50px -ով:

```

$( "#main" ).click(function(){
    $(this).animate({
        width: "+=50px",
        height: "+=50px"
    }, 2000);
})

```

jQuery -ում անհմացիաները աշխատում են հերթի սկզբունքով: Այսպիսով՝ եթե մենք ցանկանում ենք, որպեսզի `div` -ի լայնությունը նախ մեծանա `100px` -ով 2 վայրկյանում և ապա 3 վայրկյանի ընթացքում մեծացնի իր բարձրությունը `200px` -ով, ապա կոդը կունենա հետևյալ տեսքը.

```

$(this).animate({width: "+=100px"}, 2000);
$(this).animate({height: "+=200px"}, 3000);

```

### 15.3 Callback ֆունկցիաներ

Շարունակենք նախորդ օրինակի դիտարկումը: Այսպիսով՝ ունենք `div`, որի սեղման ժամանակ անհրաժեշտ է ստեղծել անհմացիա, որը `div` -ի բարձրությունը կդարձնի `300px`: Անհրաժեշտ է նաև անհմացիայի ավարտից հետո `alert` -ի օգնությամբ ցույց տալ *Animation has finished* տեքստը:

Դիտարկենք կոդը.

```

$( "#main" ).click(function(){
    $(this).animate({height: "300px"}, 5000);
    alert("Animation has finished");
})

```

Աշխատեցնելով գրված կոդը՝ կնկատենք, որ սկզբում տեղի է ունենում `alert` -ը, ապա՝ ավարտվում անհմացիան: Նշանակում է, որ այս իրավիճակում իրամանների կատարման

հերթականությունը չի պահպանվել և գրված կոդն աշխատել է ասինքրոն<sup>21</sup>:

Պատճառն այն է, որ animate ֆունկցիան, որը պետք է ստեղծի անհմացիան, ունի կատարման  $t = 5000$  միլիվայրկյան տևողություն: Նման իրավիճակներում JavaScript -ի մյուս հրամանները չեն սպասում, որպեսզի նախնական գործողությունը ավարտվի և շարունակում են կատարվել: Հենց այսպիսի իրավիճակներում է անհրաժեշտ կիրառել callback ֆունկցիաները:

Callback -ը ֆունկցիայի տեսակ է, որը կատարվում է ասինքրոն հրամանի կատարման ավարտին:

jQuery -ի մի շարք ֆունկցիաներ, ինչպիսիք են animate, fadeIn, show, hide և այլն, կարող են ստանալ ևս մեկ պարամետր, որն իրենից ներկայացնում է callback ֆունկիա և այն կկանչվի միայն ասինքրոն հրամանի կատարման ավարտին:

Դիտարկենք կոդը.

```
$("#main").click(function(){
```

```
    $(this).animate({height: "300px"}, 5000, function(){
        alert("Animation has finished")
    });
})
```

Այս դեպքում , div -ի սեղման ժամանակ, 5 վայրկյանի ընթացքում նրա բարձրությունը կդառնա 300px և միայն այդ ժամանակ կաշխատի alert հրամանը՝ էկրանին ցույց տալով անհրաժեշտ տեքստը:

---

<sup>21</sup> Կոդն անվանում են ասինքրոն, եթե հրամաններից որևէ մեկը կատարվում է ավելի վաղ, քան նրան նախորդող հրամաններից որևէ մեկը:

## ՀԱՐՏԵՐ ԵՎ ԱՌԱՋԱԴՐԱՆՔՆԵՐ

1. Որո՞նք են այն մեթոդները, որոնց միջոցով փոխարինվում է JavaScript -ի innerHTML և value հատկությունները jQuery -ում:
2. Ի՞նչ տարբերություն կա html և text ֆունկցիաների միջև:
3. Բացատրել attr և removeAttr ֆունկցիաների նշանակությունը:
4. Բացատրել addClass, removeClass, hasClass և toggleClass ֆունկցիաների նշանակությունը:
5. Ինչպես կարելի է փոխել որևէ էլեմենտի CSS հատկությունները jQuery -ում
6. Ո՞րն է animate ֆունկցիայի նշանակությունը:
7. Կա՞ն այնպիսի հատկություններ, որոնց հնարավոր չէ անհմացիայի ենթարկել jQuery -ի animate ֆունկցիայով:
8. Ի՞նչ է նշանակում ասինքրոն կոդ արտահայտությունը:

ԴԱՍ 16

# CALCULATOR – Ի ՍՏԵՂՑՈՒՄ



## ԴԱՍ 16: CALCULATOR -Ի ՄՇԱԿՈՒՄ

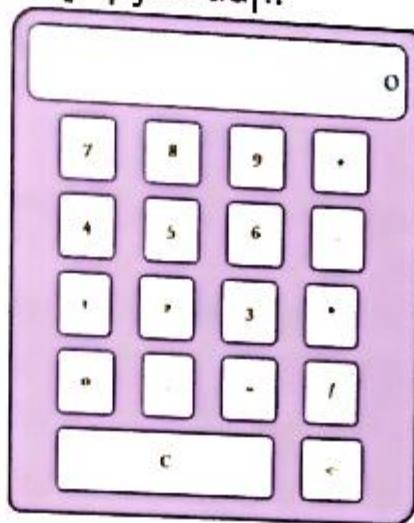
### 16.1 Արտաքին տեսքի մշակումը HTML, CSS -ով:

Ենթադրենք ցանկանում ենք ստեղծել հաշվիչ (calculator), որը կկատարի մի շարք պարզագույն գործողություններ:

Calculator -ի տեսքը, որը ակնկալում ենք ստանալ ցոյց է տրված նկարում:

Տեսքը ստանալու համար անհրաժեշտ է օգտագործել HTML կոդի ստորև բերված կառուցվածքը.

```
<div id="calculator">
<div id="result">0</div>
  <div class="row">
    <button class="num">7</button>
    <button class="num">8</button>
    <button class="num">9</button>
    <button class="action">+</button>
  </div>
  <div class="row">
    <button class="num">4</button>
    <button class="num">5</button>
    <button class="num">6</button>
    <button class="action">-</button>
  </div>
  <div class="row">
    <button class="num">1</button>
    <button class="num">2</button>
    <button class="num">3</button>
    <button class="action">*</button>
  </div>
  <div class="row">
    <button class="num">0</button>
    <button>.;</button>
```



```

        <button id="equal">=</button>
        <button class="action">/</button>
    </div>
    <div class="row">
        <button class="clear action">C</button>
        <button class="back action"><-</button>
    </div>
</div>

```

Անհրաժեշտ է նաև կիրառել հետևյալ CSS կոդը.

```

/*
    font-family: gabiola;
}

.clear{
    width: 345px;
}

#calculator{
    width: 600px;
    padding: 20px;
    background: #b28cdf;
    border: 4px solid black;
    border-radius: 40px 20px 40px 20px;
}

#result {
    border: 3px solid black;
    font-size: 4em;
    border-radius: 20px;
    background: #c2effa;
    text-align: right;
    padding: 0px 10px;
    min-height: 120px;
}

.row{
    margin: 10px;
}

```

```

button{
    padding: 16px 35px;
    background:white;
    font-size: 2.2em;
    margin-left: 40px;
    margin-top: 10px;
    font-weight: bolder;
    border: 3px solid black;
    border-radius: 10px;
}

```

Այսպիսով՝ ստացանք Calculator –ի արտաքին տեսքը: Ինչպես նկատեցինք, #result դիվն իրենից ներկայացնում է այն դաշտը, որտեղ պետք է ցուցադրվեն կատարվող գործողությունները: Բոլոր այն կոճակները (button), որոնք թվային են, ունեն .num կլասը, իսկ գործողությունները ներկայացված են action կլասով:

## 16.2 Ֆունկցիոնալ մասի մշակումը

Որպեսզի Calculator –ը դառնա ֆունկցիոնալ, ընդհանուր առմամբ, անհրաժեշտ է կիրառել 3 փոփոխական՝ number1, number2 և action: Օրինակ՝  $5+4$  արտահայտության հաշվման դեպքում, number1 փոփոխականի ներսում կպահենք 5 –ը, number2 –ի ներսում՝ 4 –ը, իսկ action –ի ներսում՝ գումարման նշանը:

Արդյունքում, նախնական կոդը, որը կունենանք մեր script.js ֆայլում կլինի հետևյալը.

```

$(document).ready(function(){
    let number1, number2, action;
})

```

Եթե սեղմվում է արվում calculator –ի թվային կոճակներից որևէ մեկի վրա: Անհրաժեշտ է #result – ի մեջ ցույց տալ, թե ո՞ր թիվն է սեղմվել: Օրինակ՝ եթե սեղմվել է 5 թվանշանը, ապա #result div –ի ներսում անհրաժեշտ կլինի ցույց տալ 5, սակայն եթե հաջորդիվ, սեղմվի 4 –ի վրա, այս դեպքում արդեն ցույց կտանք ոչ թե 4, այլ՝ 54:

Ասկածը դիտարկենք .num կլասն ունեցող կոճակների սեղման ժամանակ աշխատող ֆունկցիայում.

```
$(".num").click(function(){
    //num -ի մեջ պահենք սեղմված button -ի ներսում գրված թիվը
    let num = $(this).html();
    //current -ի մեջ պահենք #result - ներսում գրված ընթացիկ թիվը
    let current = $("#result").html();
    //current -ին ավելացնենք սեղմված button -ի թիվը
    current += num;
    //արդյունքը կրկին տեղադրենք #result- մեջ
    $("#result").html(current);
})
```

Այս դեպքում, փորձենք Calculator -ի օգնությամբ գրել 456 թիվը.

0456

Ինչպես նկատեցինք, թվի դիմացից գրված է 0: Պատճառն այն է, որ կողի ներսում մենք current+=num գործողությունն ենք իրականացնում: Հաշվի առնելով այն, որ #current -ի ներսում գրված էր 0, ապա բնական է, որ այդ թվանշանը չի բացակայի հաջորդ գործողությունների ժամանակ:

Հետևաբար, կարող ենք ավելացնել պայմանական արտահայտություն, որը կսուզի՝ արդյո՞ք տեքստը սկսվում է 0 -ով, և այդ դեպքում կհեռացնի տեքստից 0 -ը՝ տարրը՝ օգտագործելով substring ֆունկցիան:

```
$(".num").click(function(){
    let num = $(this).html();
    let current = $("#result").html();
    current += num;
    if(current.startsWith("0")){
        current = current.substring(1);
    }
    $("#result").html(current);
})
```

Ինչպես գիտենք, number1 փոփոխականը ցոյց է տալիս արտահայտության առաջին տարրը<sup>22</sup>: Իսկ ե՞րբ է հասկանալի դառնում, որ առաջին թիվը մուտքագրված է և ժամանակն է մուտքագրել երկրորդը: Իհարկե, այն ժամանակ, երբ օգտատերը սեղմում է որևէ գործողության վրա:

Այս դեպքում անհրաժեշտ է կատարել 3 գործողություն.

Այս դեպքում անհրաժեշտ է կատարել #result -ում

1. number1 փոփոխականի ներսում պահել #result -ում գրված ընթացիկ արժեքը
2. action -ի մեջ պահել այն գործողությունը, որի վրա սեղմվել է (օրինակ՝ գումարում, հանում և այլն)
3. Դատարկել #result -ը, որպեսզի օգտատերը մուտքագրի հաջորդ թիվը

```
$(".action").click(function(){
    //number1 -ի մեջ պահենք ընթացիկ թիվը
    number1 = Number($("#result").html())
    // action -ի մեջ պահենք այն գործողությունը, որը պետք է
    //կատարվի
    // այսինքն՝ սեղմված button -ի ներսում գրված սիմվոլը
    action = $(this).html();
    // Դատարկենք #result -ը որպեսզի մուտքագրվի հաջորդ
    // թիվը

    $("#result").html("");
})
```

Այժմ արդեն հայտնի է, number1, action փոփոխականների արժեքները: Անդրադառնանքը number2 -ին: Բնական է, որ number2 -ի արժեքը պարզ է դառնում այն ժամանակ, երբ օգտատերը սեղմում է = նշանին: Հենց այստեղ էլ տեղի է ունենում արտահայտության հաշվարկման գործողությունը՝ կախված action -ի արժեքից.

---

<sup>22</sup> Օրինակ 5+4 արտահայտության մեջ number1-ը կինի 5-ը:

```

$( "#equal" ).click(function(){
    number2 = Number($("#result").html());
    let res;
    if(action == "+"){
        res = number1 + number2;
    }else if(action == "-"){
        res = number1 - number2;
    }else if(action == "*"){
        res = number1 * number2;
    }else if(action == "/"){
        res = number1 / number2;
    }
    $("#result").html(res);
})

```

Այսպիսով՝ պարզագույն գործողություններ կատարող Calculator -ն արդեն պատրաստ է:

### ՀԱՐՑԵՐ ԵՎ ԱՌԱՋԱԴՐԱՆՔՆԵՐ

1. Calculator -ին ավելացնել իրական թվերի հետ գործողություններ կատարելու հնարավորություն:
2. Թույլ չտալ, որպեսզի . (կետ) սիմվոլը մեկից ավել անգամ կիրառվի որևէ թվում:
3. C կոճակի սեղման ժամանակ #result դաշտում գրել 0 արժեքը:
4. ← կոճակի յուրաքանչյուր սեղման ժամանակ ջնջել #result -ում առկա արժեքի վերջին սիմվոլը:
5. Բացասական թվերի հետ գործողություններ կատարելու համար ավելացնել նոր կոճակ՝ minus -ը: Սեղման ժամանակ, եթե #result -ում գրված է  $x$  թիվ, ապա այն պետք է դառնա  $-x$ , այսինքն՝ բացասական: Հաջորդ սեղմանը՝ կրկին  $x$  և այսպես շարունակ:
6. Ավելացնել BIN կոճակը, որի սեղման ժամանակ մուտքագրված թիվը կներկայացվի երկուական համակարգում:

**ԴԱՍ 17**

**ԴԻՆԱՄԻԿ  
ԵՐԱԳՐԱՎՈՐՈՒՄ**



## ԴԱՍ 17: ԴԻՆԱՄԻԿ ԾՐԱԳՐԱՎՈՐՈՒՄ

### 17.1 Ժամանակի կարգավորման ֆունկցիաներ

Ծրագրավորման գործընթացում երբեմն անհրաժեշտ է լինուած կազմակերպել գործողությունների որոշակի բազմություն, որոնք կամ սպասում են որոշակի ժամանակ և հետո կատարվուած, կամ էլ կատարվուած են ու պարբերականությամբ:

Նման գործողություններ կազմակերպելու համար JavaScript -ը մեզ տրամադրուած է setTimeout և setInterval ֆունկցիաները:

Դիտարկենք setTimeout ֆունկցիայի պարզ կառուցվածքը.

**setTimeout(F,t);**

Այս դեպքուած, F ֆունկցիան կկանչվի մեկ անգամ, իրամասի կատարուածից հետո, եթե կանցնի ու միլիվայրկյան:

```
$(<document>).ready(function(){
    setTimeout(function(){
        alert("hello");
    }, 2000);
})
```

Այս դեպքուած, էջի բացվելուց անմիջապես 2 վայրկյան հետո էկրանին կհայտնվի hello տեքստը՝ alert պատուհանով:

setInterval -ը նախատեսված է որևէ ֆունկցիա պարբերաբար կանչելու համար:

```
setInterval(function(){
    console.log("ok")
}, 2000);
```

Բերված օրինակուած, էջի բացվելուց հետո, յուրաքանչյուր 2 վայրկյանը մեկ, կկատարվի console.log իրամասը:

Իսկ ի՞նչ տեղի կունենա, եթե մենք setInterval -ը ստեղծենք button -ի սեղման ժամանակ և պարբերաբար սեղմենք այն:

```
Դիտարկենք կոդը.  
 $("button").click(function(){  
   setInterval(function(){  
     console.log("ok")  
   }, 2000);
```

» Երբ առաջին անգամ սեղմենք button -ը, մենք կունենանք interval, որն աշխատում է յուրաքանչյուր 2 վայրկյանը մեկ։ Հաջորդ սեղմանն ինտերվալը կրկնապատկվում է, քանի որ նախկին ստեղծված ինտերվալը չի հեռացվում, իետևաբար գործողությունը տեղի կունենա արդեն յուրաքանչյուր վայրկյանը մեկ։ Հաջորդ անգամ՝ կես վայրկյանը մեկ և այսպես շարունակ։ Նման խնդիրներից խուսափելու համար JavaScript -ը մեզ տրամադրում է clearTimeout և clearInterval ֆունկցիաները, որոնք թույլ են տալիս հիշողությունն ազատել գործող ֆունկցիաներից։

Այս դեպքում.

```
var x = null;  
$("button").click(function(){  
    clearInterval(x);  
    x = setInterval(function(){  
        console.log("ok")  
    }, 2000);
```

» Բերված օրինակում ինտերվալի տվյալները պահպանվում են չգլոբալ փոփոխականի ներսում, իսկ սեղման ժամանակ նախ ջնջվում է չ ինտերվալը, եթե այն առկա է, և ապա ստեղծվում է նորը, ինչը չի ունենա ազդեցություն ինտերվալի արագագործության և աշխատանքի վրա:

## 17.2 Դինամիկ ծրագրավորում

17.2 Դրամակիզական գույքը կազմում է 1 000 մլն դրամիկ Նախորդ բաժիններում արդեն խոսել ենք դինամիկ ծրագրավորման մեթոդի մասին։ Այս մեթոդը մեզ

Ինարավորություն էր տալիս JavaScript -ի օգնությամբ էջին ավելացնել տարրեր տեղեր:

JavaScript -ում, ընդհանուր առմամբ, դինամիկ ծրագրավորման գործընթացը տեղի է ունենում createElement, appendChild ստանդարտ ֆունկցիաների օգնությամբ, սակայն jQuery -ն մեզ է տրամադրում մի շարք նոր գործիքներ, որոնք թույլ են տալիս զգալիորեն հեշտացնել դինամիկ ծրագրավորման գործընթացը: Այդ գործիքներն են՝ append, prepend, after, before, appendTo, prependTo և այլն:

Ենթադրենք անհրաժեշտ է body -ին ավելացնել h1 տեղ, որի մեջ գրված է Hello բառը, իսկ նրա id -ին title է.

```
$("body").append("<h1 id='title'>Hello</h1>");
```

Նոյն արդյունքին կարելի էր հասնել, եթե օգտագործեինք prepend ֆունկցիան: Տարբերությունն այն է, որ append -ի դեպքում դինամիկ ավելացվող տարրը ավելանում է տվյալ տեղի (մեր օրինակում՝ body) վերջում, իսկ prepend -ի դեպքում ավելանում է սկզբում:

Այժմ ծանոթանանք after և before ֆունկցիաների հետ:

Նախորդ իրամանի կատարման արդյունքում՝ body -ի մեջ ավելացավ h1 տեղ, որն ուներ id = title հատկությունը:

Եթե ցանկանում ենք ավելացնել որևէ տեղ անմիջապես h1 -ից հետո, ապա կարող ենք օգտագործել after ֆունկցիան.

```
$("#title").after("<p>jQuery is cool </p>");
```

Նոյն տրամաբանությամբ կարող ենք ավելացնել նոր դինամիկ տարր մինչ #title -ը: Այդ նպատակով կկիրառենք before ֆունկցիան.

```
$("#title").before("<p>jQuery is Awesome!! </p>");
```

JavaScript -ում դինամիկ ծրագրավորման գործընթացը տեղի էր ունենում փուլերով նկարագրման միջոցով, այսինքն, ստեղծվում էր տարրը, ապա նրան ավելացվում որոշակի հատկություններ, այնուհետև այն ավելացվում էր անհրաժեշտ տեղում:

Նոյն ալգորիթմը, անշուշտ, կարելի է կիրառել նաև jQuery -ում: Դիտարկենք օրինակը.

```

let elm = $("<h1></h1>");
elm.html("I love jQuery");
elm.attr("id", "title");
elm.css("color", "red");
$("body").append(elm);

```

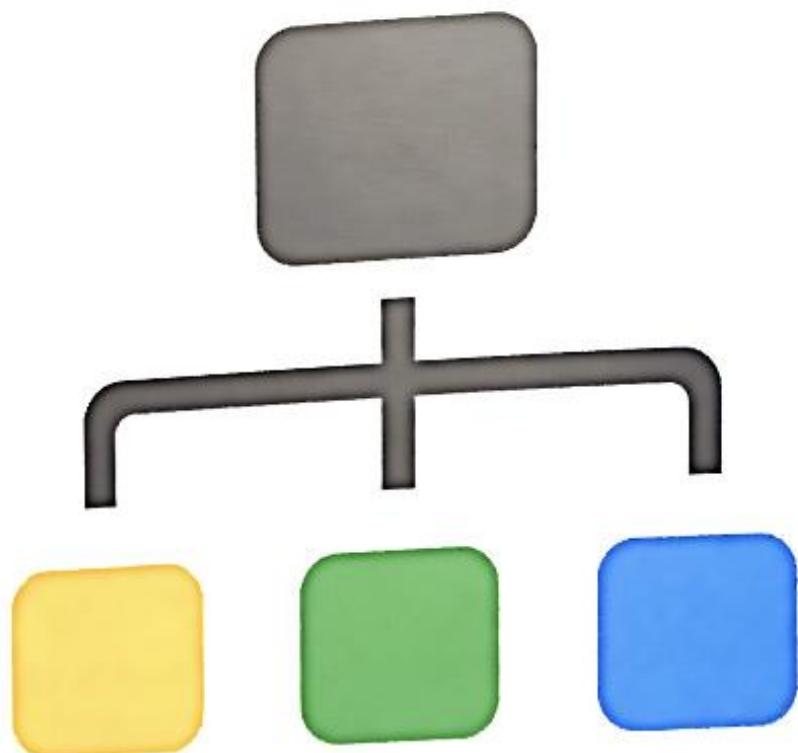
Այստեղ կարող ենք կիրառել նաև **appendTo** ֆունկցիան, որը համարժեք է **append** -ին, սակայն **appendTo** -ի դեպքում ֆունկցիան որպես պարամետր ստանում է այլ տարրը, որտեղ պետք է ավելացվի դինամիկ ստեղծված տեգը:  
 Այսպիսով՝ վերջին տողում կունենանք.  
 elm.appendTo("body");

## ՀԱՐՑԵՐ ԵՎ ԱՌԱՋԱԴՐԱՆՔՆԵՐ

- Ի՞նչ նպատակով են կիրառվում **setTimeout**, **setInterval**, **clearTimeout** և **clearInterval** ֆունկցիաները:
- Բացատրել **append**, **prepend**, **after** և **before** ֆունկցիաների նշանակությունը:
- Ո՞րն է **append/appendTo** և **prepend/prependTo** ֆունկցիաների տարբերությունը:
- Ի՞նչ հիմնական եղանակներ կան, որոնց միջոցով հնարավոր է ստեղծել դինամիկ տեգեր:
- Input դաշտում մուտքագրվում է րոպե և վայրկյան ցույց տվող արտահայտություն: Օրինակ՝ 14:53: **button** -ի սեղման ժամանակ անհրաժեշտ է դինամիկ կերպով էջին ավելացնել **h1**, որում գրված է 14:53 տեքստը: Յուրաքանչյուր վայրկյան **h1** -ում գրված արտահայտությունը պետք է փոխվի: Օրինակ՝ 14:52, 14:51 և այսպես շարունակ: Եթե վայրկյան ցույց տվող թիվը հավասարվում է 0 -ի, անհրաժեշտ է մեկով պակասեցնել րոպե -ի արժեքը: Ինտերվալի աշխատանքը պետք է դադարեցնել 0:0 պահին:

ԴԱՍ 18

# JQUERY ԵՎ DOM



### 18.1 DOM -ի ծառային կառուցվածքի ղեկավարման

#### ֆունկցիաներ

Այս դասի ընթացքում կանդրադառնանք մի շարք ֆունկցիաների, որոնք jQuery -ին տրամադրում է մեզ HTML տարրերի ծառային կառուցվածքում գործողություններ իրականացնելու համար:

**find**: ֆունկցիան թույլ է տալիս փնտրել ժառանգ տարրեր՝ ըստ տրված CSS նշիչի (selector): Ենթադրենք անհրաժեշտ է `#main` div -ի ներսում գտնվող բոլոր նկարներին ավելացնել `.active` կլասը: Այս դեպքում հրամանը կունենա հետևյալ տեսքը.

```
$("#main").find("img").addClass("active");
```

Նկատենք, որ `find` ֆունկցիան որոնում է ոչ միայն ժառանգ տարրերում, այլ նաև ժառանգների ժառանգներում, այսինքն՝ ամբողջ խորությամբ: Եթե ցանկանում ենք, որ գործընթացը վերաբերվի միայն անմիջական ժառանգներին, ապա անհրաժեշտ է կիրառել **children** ֆունկցիան:

**first**: տեղերի բազմության մեջ վերադարձնում է առաջին տարրը: Օրինակ՝ `#main` div -ում գտնվող նկարներից առաջինին `active` կլասը կարելի է ավելացնել հետևյալ հրամանի միջոցով.

```
$("#main > img").first().addClass("active");
```

**last**: տեղերի բազմության մեջ վերադարձնում է վերջին տարրը: Օրինակ՝ `#main` div -ում գտնվող նկարներից վերջինին `border` կարելի է ավելացնել հետևյալ հրամանի միջոցով.

```
$("#main > img").last().css("border", "5px solid");
```

**eq(n)**: տեղերի բազմության մեջ վերադարձնում է `n+1` -րդ տարրը: Օրինակ՝ `#main` div -ում գտնվող 4-րդ նկարին `border` կարելի է ավելացնել հետևյալ հրամանի միջոցով.

`$("#main > img").eq(3).css("border", "5px solid");`  
Նկատենք, որ ինչպես զանգվածներում, այստեղ նույնպես  
համարակալումը սկսվում է 0 -ից:

**filter:** թույլ է տալիս առանձնացնել ենթաբազմություն որոնված  
տեգերում: Օրինակ՝ ենթադրենք անհրաժեշտ է .active կլասն  
ունեցող նկարների բարձրությունը դարձնել 100%: Այս դեպքում  
անհրաժեշտ է կատարել հետևյալ հրամանը.

`$("img").filter(".active").css("height", "100%");`

**has:** Թույլ է տալիս կատարել որոնում՝ ըստ ժառանգ տեգերի:  
Ենթադրենք անհրաժեշտ է էջից թաքցնել այն div-երը, որոնցում  
առկա է h1 տեգ. Դիտարկենք կոդը.

`$("div").has("h1").hide();`

**closest:** Թույլ է տալիս գտնել տվյալ տարրին ամենամոտ ծնող  
տարրը՝ ըստ տրված CSS selector -ի: Ենթադրենք ունենք իրար  
մեջ ներդրված երեք div-եր, որոնցից վերջինի ներսում առկա է  
նաև span:

Դիտարկենք հրամանը.

`$("span").closest("div").css({  
 "border": "1px solid red",  
 "padding": "20px"  
 })`

Մեր օրինակում border և padding կստանա այն div-ը, որի մեջ  
գտնվում է span -ը, այսինքն՝ նրա ծնող տարրը, քանի որ այդ  
div -ը հանդիսանում է նրան ամենամոտը:

**parent:** Թույլ է տալիս գտնել տվյալ տարրը ամենամոտ ծնող  
տեգը՝ առանց որևէ CSS selector -ի: Ենթադրենք ունենք իրար  
մեջ ներդրված երեք div-եր, որոնցից վերջինի ներսում առկա է  
նաև span:

Դիտարկենք հրամանը.

```
$( "span" ).parent().css({  
    "border": "1px solid red",  
    "padding": "20px"  
})
```

Մեր օրինակում border և padding կստանա այն div -ը, որի մեջ գտնվում է span -ը, քանի որ այդ div -ը հանդիսանում է նրան ամենամոտը: Նկատենք, որ closest և parent ֆունկցիաները համարժեք են իրար, սակայն մի տարբերությամբ. closest ֆունկցիան կարող է ստանալ պարամետր, որը selector է, իսկ parent -ը՝ ոչ:

parents: Վերադարձնում է տվյալ տեղի բոլոր ծնող էլեմենտները՝ ըստ տրված selector -ի: Ենթադրենք ցանկանում ենք գտնել span -ի բոլոր այն ծնող էլեմենտները, որոնք div-եր են:

Այս դեպքում կարիք կիහնի կիրառել հետևյալ կոդը.

```
$( "span" ).parents("div")
```

is: Թույլ է տալիս համեմատել որոնված տեղը մեկ այլ selector -ի հետ: Ենթադրենք ցանկանում ենք ստուգել՝ արդյո՞ք span -ի ծնող տեղը div է: Այս դեպքում պայմանական օպերատորը կունենա հետևյալ տեսքը.

```
if( $( "span" ).parent().is("div") ){  
    // զործողություններ  
}
```

not: Թույլ է տալիս ֆիլտրել որոնված տեղերը բացառման կարգով: Ենթադրենք անկիրաժեշտ է գտնել բոլոր այն div—երը, որոնք չունեն .active կլասը և նրանց ֆոնի գույնը դարձնել կարմիր:

Դիտարկենք կոդը.

```
$( "div" ).not(".active").css("background","red");
```

**parentsUntil:** Թույլ է տալիս առանձնացնել տվյալ տեղի բոլոր ծնող տարրերը մինչ այն սահմանը. որը կնշվի ֆունկցիայի պարամետրով.

Ենթադրենք body -ի ներսում ունենք իրար մեջ ներդրված 3 div, որոնցից վերջինի մեջ առկա է span:  
Դիտարկենք կոդը.

`$(“span”).parentsUntil(“body”).hide();`

Այս դեպքում, կանհետանան span -ի բոլոր ծնող էլեմենտները (div-երը), բացառությամբ body -ի, քանի որ այն հանդիսանում է սահմանը:

**next:** Թույլ է տալիս դիմել տվյալ տեղից անմիջապես հետո գտնվող տեղին:

Օրինակ՝

`<h1>some Text</h1>`

`<span>some text 2</span>`

Դիտարկենք կոդը.

`$(“h1”).next().hide();`

Այս իրամանի կատարման արդյունքում էջից կանհետանան span -ը, քանի որ այն գտնվում է h1 տեղից անմիջապես հետո:

**prev:** Թույլ է տալիս դիմել մինչ տվյալ տեղը գտնվող տեղին, այսինքն՝ prev ֆունկցիան հանդիսանում է next -ի հակառակը: Նկատենք, որ գործում են նաև prevUntil և nextUntil ֆունկցիաները, որոնք նման են իրենց աշխատանքի տրամաբանությամբ parentsUntil -ին:

## 18.2 Էլեմենտների հեռացումը էջից

Այս բաժնում մեր ուշադրությունը իրավիրենք երեք տարրեր ֆունկցիաների՝ remove -ի, empty -ի և detach -ի վրա:

**empty:** Թույլ է տալիս դատարկել տվյալ տեղի ամբողջ պարունակությունը:

Դիտարկենք կոդը.

`$("#main").empty()`

Այս հրամանի կատարման արդյունքում կդատարկվի #main տեղի ամբողջ պարունակությունը:

remove: Էջից հեռացնում է տվյալ տեղը և վերադարձնում նրա

արժեքը:

Օրինակ՝

`$("#main").remove()`

Հրամանի կատարման արդյունքում, այս անգամ, ոչ թե կդատարկվի #main տեղը, այլ այն կջնջվի էջից:

detach: Էջից հեռացնում է տվյալ տեղը և վերադարձնում նրա

արժեքը:

Օրինակ՝

`$("#main").detach()`

Հրամանի կատարման արդյունքում, կրկին, ոչ թե կդատարկվի #main տեղը, այլ այն կհեռացվի էջից:

Թվային նկատում ենք, remove և detach ֆունկցիաները հնչյած նկատում ենք, քորոշողությունը: Իհարկե, նրանց միջև կատարում են նոյն գործողությունը: Մեր ուշադրությունը առկա է որոշակի տարբերություն: Մեր ուշադրությունը հրավիրենք այն հանգամանքի վրա, որ թե' remove, թե' detach ֆունկցիաները վերադարձնում են ջնջված տեղը:

Այլ կերպ ասած, դա նշանակում է, որ ջնջված տեղը հեշտությամբ կարելի է պահել փոփոխականի ներսում:

Օրինակ՝

`var elm = $("#main").remove()`

Այժմ, եթե #main -ը պահված է փոփոխականի ներսում, մենք կարող ենք օգտագործել դինամիկ ծրագրավորման մեթոդից՝ մեզ ծանոթ append ֆունկիան, որպեսզի կրկին ետ վերադարձնենք ջնջված տարրը:

```

Այսպիսով՝
$("#main").click(function(){
    alert("ok");

})

var elm = $("#main").remove()
$("body").append(elm);

```

Նոյնը կարելի է անել detach ֆունկցիայի միջոցով.

```

$("#main").click(function(){
    alert("ok");

})

let elm = $("#main").detach()
$("body").append(elm);

```

Այս ֆունկցիաների միջև տարրերությունն արտահայտվում է այն ժամանակ, երբ ջնջվող տեզի նկատմամբ առկա է եղել որևէ event: Մեր օրինակում button -ի սեղման ժամանակ alert -ի օգնությամբ ելքագրվում է “ok” տեքստը:

Այս դեպքում, երբ էջին կրկին ավելացնենք այդ button -ը, ապա.

- Եթե button -ը ջնջվել էր remove ֆունկցիայով, ապա նրա հետ կապված event-ները կդադարեն գործել նրան վերականգնելուց հետո
- Եթե button -ը ջնջվել էր detach ֆունկցիայով, ապա նրան վերականգնելուց հետո նրա հետ կապված event-ները կրկին կշարունակեն գործել.

### 18.3 Էլեմենտների կառավարում

Դիտարկենք DOM -ի հետ աշխատանքի և մի քանի ֆունկցիաներ, որոնք ունեն լայն կիրառում jQuery -ում:

**each:** Կիրառվում է html տեգերի, ինչպես նաև, ստանդարտ զանգվածների նկատմամբ ցիկլ կիրառելու համար: Ցիկլի ներսում հերթական տարրին կարելի է դիմել \$(this) -ի օգնությամբ:

```
$("#main > div").each(function(){
```

```
})
```

Նկատենք, որ each ֆունկցիային փոխանցված անանուն ֆունկցիան, կարող է ստանալ երկու պարամետրեր, որոնցից առաջինը ցույց է տալիս հերթական համարը, իսկ երկրորդը զանգվածի տարրն է:

```
$("#main > div").each(function(index,item){
```

```
})
```

**clone:** Թույլ է տալիս կլոնավորել ընտրված տարրերը՝ ներառելով ժառանգ տեգերը և բոլոր հատկությունները:

Օրինակ՝

```
$("#main").clone().appendTo("body")
```

Մեր օրինակում ստեղծվում է #main div -ի կրկնօրինակը և այն անմիջապես ավելացվում է body -ին: Նկատենք, որ clone ֆունկցիան կարող է ստանալ բույսան պարամետր, այսինքն՝ true/false: Եթե պարամետրը true է, ապա էլեմենտի հետ կապված event-ները գործում են նաև կրկնօրինակի համար, հակառակ դեպքում՝ ոչ:

**wrap:** Հնարավորություն է տալիս դինամիկ մեթոդով նոր տեղ արտագծել ընտրված տարրին: Ենթադրենք անհրաժեշտ է

#main div -ը ընդգրկել <div class='parent'></div> տարրի  
ներսում: Այս դեպքում.

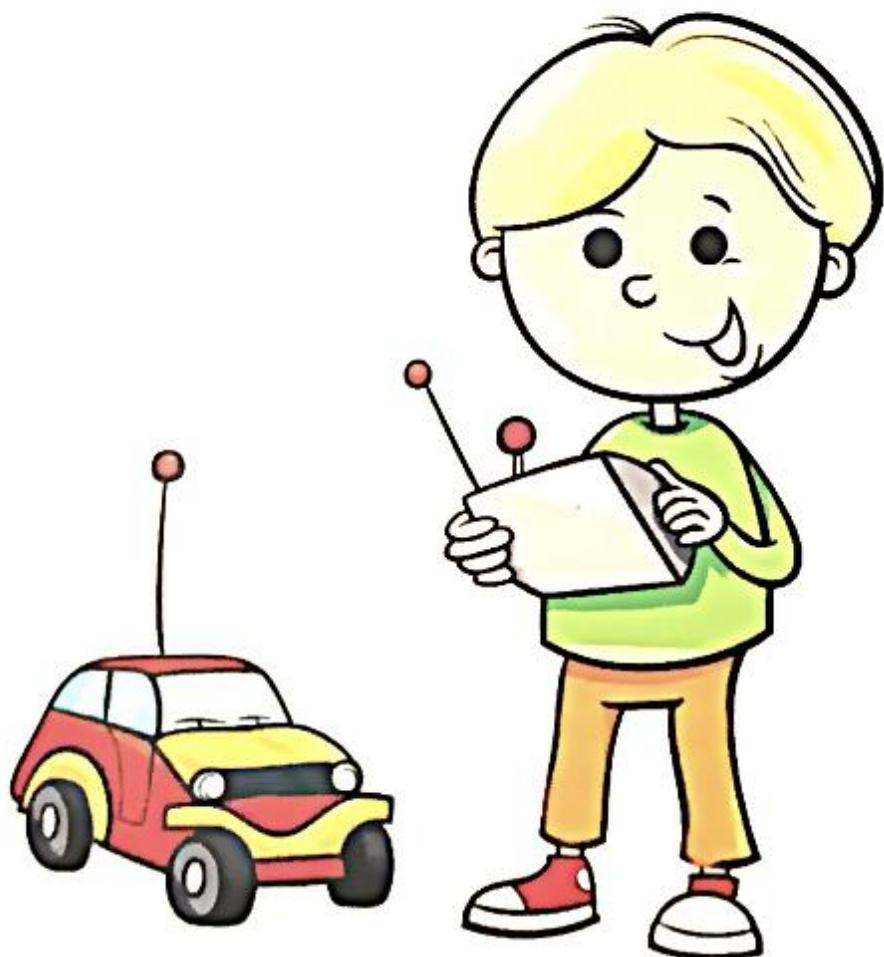
```
$( "#main" ).wrap( "<div class='parent'></div>" )
```

## ՀԱՐՑԵՐ ԵՎ ԱՌԱՋԱԴՐԱՆՔՆԵՐ

1. Ո՞րն է parent, parents և parentsUntil ֆունկցիաների տարրերությունը:
2. Բացատրել next/prev ֆունկցիաների դերը:
3. Ինչպես կարելի է ցիկլի օգնությամբ դիտարկել HTML տարրերը:
4. Բացատրել clone ֆունկցիայի դերը:
5. Ո՞րն է remove, empty և detach ֆունկցիաների միջև տարրերությունը:
6. Ի՞նչ նպատակով կարող ենք կիրառել first և last ֆունկցիաները:
7. Ի՞նչ նպատակով է կիրառվում wrap ֆունկցիան:

ԴԱՍ 19

EVENT  
ԵՐԱԳՐԱՎՈՐՈՒՄ



## ԴԱՍ 19: EVENT ԾՐԱԳՐԱՎՈՐՈՒՄ

### 19.1 Պարզագույն event-ներ

Դաս 3 - ում քննարկել ենք event-ների հետ կապված որոշ գաղափարներ: Այս բաժնում փորձենք հասկանալ՝ ինչպես է տեղի ունենում jQuery -ի աշխատանքը event-ների հետ: Ինչպես գիտենք, event-ները հնարավորություն են տալիս ապահովել ինտերակտիվ ծրագրավորում: Եթե օգտագործողը սեղմում է որոշակի դաշտում, փոխում է select - ի option-ը, կամ պարզապես մուտքագրում է որոշակի ինֆորմացիա, այդ ժամանակ աշխատում են event-ները: Ընդհանուր առմամբ event-ները կարելի է բաժանել 2 խմբի.

1. «մենիկի հետ կապված» event-ներ
2. «ստեղնաշարային» event-ներ:

Դիտարկենք body -ին սեղման event ավելացնելու օրինակը.

```
$("body").click(function(){  
    alert("Hello");  
})
```

Այժմ, body -ի ցանկացած կետում սեղման դեպքում կբացվի alert պատուիան: Իհարկե, պատուիանը կբացվի միայն այն դեպքում, եթե body -ի ներսում կան տարբեր էլեմենտներ, կամ body -ին ունի որոշակի բարձրություն, օրինակ՝ 100vh, ինչը նշանակում է էկրանի տեսանելի հատվածի ամբողջ բարձրությունը:

Այժմ ենթադրենք, ցանկանում ենք դինամիկ մեթոդով body -ին ավելացնել շրջանագիծ հենց այն կետում, որտեղ սեղմվել է: Այս իմաստով, անհրաժեշտ է իմանալ, որ event-ների հետ կապված անանուն ֆունկցիաները կարող են ստանալ պարամետր, որը ինֆորմացիա է պարունակում տվյալ event -ի մասին: Ընդունված է, այդ փոփոխականի համար, որպես անուն նշանակել են:

```
$(“body”).click(function(e){
```

```
)
```

Ե -Ն ԻՐԵՆԻԾ ՆԵՐԿԱՅԱԳՆՈՒՄ Է ՕՐԵԿՈ, ԱՅՍԻՆՔՆ՝ ՎԻՌԻԴԱԿԱՆ, ՈՐԻ ՄԵԶ Առկա ԵՆ ՄԻ ՉԱՐՔ ՏՎՅԱԼՆԵՐ ՄԵՂԻ ՈՒՆԵցող ԵՎԵՆՏ -Ի ՄԱՍԻՆ: Այդ հատկություններից ԵՆ clientX, clientY, screenX, screenY, pageX, pageY Դաշտերը, որոնք համապատասխանաբար ինֆորմացիա ԵՆ պարունակում ԷՉԻ ԱՅՆ X,Y ԿՈՈՐԴԻՆԱՏՆԵՐԻ ՄԱՍԻՆ, ՈՐՄԵՂ ՄԵՂԻ Է ՈՒՆԵցԵԼ ՄԵՂՄՈԱՐ: ՓՈՐՁԵՆՔ ՀԱՍԿԱՆԱԼ ԴՐԱՆՑ ՆՉԱՆԱԿՈՒԹՅՈՒՆՆԵՐԸ.

- **clientX, clientY** - այս դեպքում X,Y կոորդինատները հարաբերական են դիրքավորված viewport -ի նկատմամբ: Ընդհանրապես, viewport ասելով հասկանում ենք էկրանի տեսանելի հատվածը: Հետևաբար, եթե scroll -ի միջոցով հայտնվում ենք էջի մեկ այլ հատվածում, միևնույն է դրանից չի փոխվում clientX, clientY կոորդինատները:
- **pageX, pageY** - այս դեպքում դիրքավորումը հարաբերական է էջի պարունակության նկատմամբ: Օրինակ body -ի սեղման դեպքում, եթե էջն ունի scroll, ապա pageY կոորդինատը կարող է վերադարձնել մեծ արժեք, մինչդեռ clientY -ը, այնուամենայնիվ, շարունակում է մնալ հարաբերական viewport<sup>23</sup> -ի նկատմամբ:
- **screenX, screenY** - այս տարրերակը շատ նման է clientX/clientY դեպքին, այսինքն՝ կոորդինատները կրկին դիրքավորված viewport -ի նկատմամբ, սակայն տարրերությունն այն է, որ կոորդինատները փոխվում են browser -ում zoom -ի (խոշորացում) կիրառելու ժամանակ:

Հաշվի առնելով, որ մեր դեպքում body -ին դատարկ է՝ կարող ենք օգտվել clientX և clientY հատկություններից:

---

<sup>23</sup> Վեր էջի այն հատվածը, որը տվյալ պահին տեսանելի է:

Այսպիսով՝ սեղման ժամանակ ստեղծենք դինամիկ տեղ, որն ունի absolute դիրքավորում, որոշակի պատահական գույն, իսկ նրա top/left կոորդինատները կապենք e.clientX և e.clientY դաշտերին:

Ստեղծված տեղի ներսում նաև գրենք նրա ընթացիկ կոորդինատները:

```
$("body").click(function(e){  
    let elm = $("<div></div>");
```

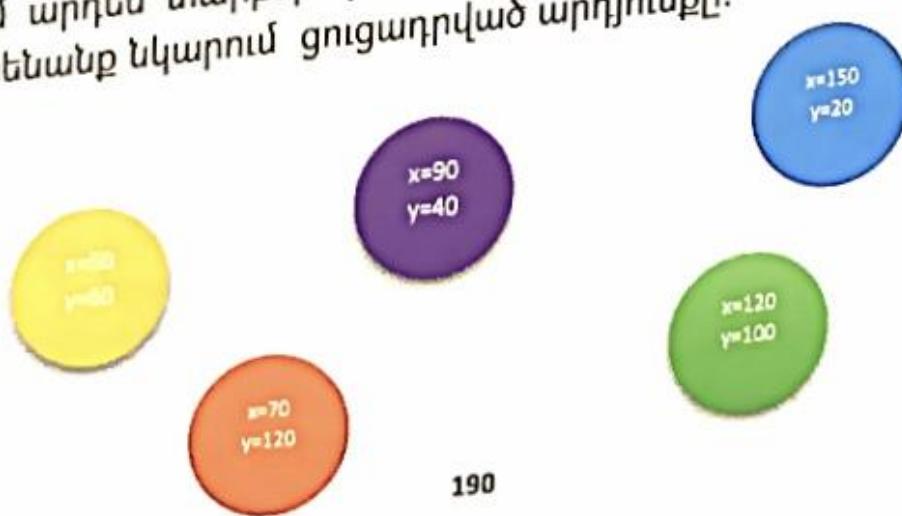
```
    elm.css({  
        position: "absolute",  
        top: e.clientY + "px",  
        left: e.clientX + "px",  
        width: "100px",  
        height: "100px",  
        "border-radius": "50%"  
    })
```

```
    let r = parseInt(Math.random() * 256)  
    let g = parseInt(Math.random() * 256)  
    let b = parseInt(Math.random() * 256)
```

```
    let randomColor = `rgb(${r},${g},${b})`;  
    elm.css("background", randomColor);
```

```
    elm.html("x=" + e.clientX + "<br>y=" + e.clientY);  
    elm.appendTo("body");
```

Այժմ արդեն տարբեր կետերում սեղման ժամանակ էլերանին կունենանք նկարում ցուցադրված արդյունքը:



Բացի click -ից, կան մի շարք event-ներ, որոնք ունեն լայն կիրառում:

- **change** - ակտիվանում է այն ժամանակ, եթե տվյալ տեղի պարունակությունը փոփոխության է ենթարկվում: Օրինակ, եթե select -ի option-ներից ընտրվում է որևէ մեկը:
- **dblclick** - ակտիվանում է double click -ի (կրկնակի սեղմում) ժամանակ
- **keydown** - ակտիվանում է, եթե սեղմվում է ստեղներից որևէ մեկը
- **mouseup** - ակտիվանում է, եթե մկնիկի կոճակը սեղմված վիճակից բաց է թողնվում
- **mousedown** - ակտիվանում է, եթե մկնիկի կոճակը սեղմվում է
- **mousemove** - ակտիվանում է, եթե մկնիկը սկսում է շարժվել:

Ենթադրենք ունենք #inp անունով դաշտ և ցանկանում ենք, որ օգտագործողը նրա մեջ կարողանա մուտքագրել միայն թվեր: Բնական է, որ այս իրավիճակում #inp -ի նկատմամբ պետք է կիրառել keydown event -ը:

Օգտագործելով jQuery -ի \$.isNumeric ֆունկցիան՝ կարող ենք հեշտությամբ համոզվել՝ արդյո՞ք մուտքագրված տվյալը թվային է, թե ոչ:

Դիտարկենք կոդը.

```
$("#inp").keydown(function(e){  
    if($.isNumeric(e.key) == false){  
        e.preventDefault();  
    }  
})
```

Այսպիսով՝ ֆունկցիայի ներսում ստուգված է՝ արդյո՞ք մուտքագրված տվյալը (e.key) թվային չէ: Բնական է, որ նման իրավիճակում նպատակ ունենք թվի՝ input -ի ներսում

հայտնվելը դադարեցնել: Ինչպես գիտենք, ստեղծաշարից որևէ ստեղն սեղմելիս, input -ի ներսում հայտնվում է համապատասխան սիմվոլը: Այդ հատկությունը չի ստեղծվում է ծրագրավորողի կողմից, այն արդեն ներդրված է browser - event -ի ստանդարտ հատկությունը, որը տեղի է ունենում preventDefault() ֆունկցիայի կիրառմամբ: Նոյն մեթոդով կարելի է լուծել այլ խնդիրներ: Օրինակ՝ browser -ում աջ սեղման ժամանակ բացվում է ենթատեքստային մենյու, այս գործնթացը նույնպես կարելի է կանխել preventDefault() ֆունկցիայով, այս դեպքում, սակայն, event -ը կլինի contextmenu -ն:

## 19.2 Դինամիկ event-ներ

Այժմ էջի վրա տեղադրենք #btn կոճակը, որի սեղման ժամանակ ստեղծեք մեկ այլ button, որն իր հերթին կունենա .item կլասը:

```
<button id='btn'>Create Button</button>
$("#btn").click(function(){
    var btn = $("<button></button>");
    btn.addClass("item");
    btn.html("click here");
    $("body").append(btn);
})
```

Յուրաքանչյուր անգամ #btn -ի սեղման ժամանակ էկրանին կհայտնվի նոր տեղ, որը կունենա item կլասը:

Այժմ փորձենք click event ավելացնել նաև item -ին, այսինքն՝ դինամիկ ստեղծվող կոճակին:

```
$(".item").click(function(){
    alert("it's ok");
})
```

Սեղմելով դինամիկ ստեղծված կոճակների վրա՝ կնկատենք, որ alert պատուիանը չի բացվում: Պատճառն այն է, որ click ֆունկցիան և առհասարակ դրան համարժեք մյուս event-ները (dblclick, mousedown, keydown...) կարող են աշխատել միայն այն

տեգերի հետ, որոնք ստատիկ կերպով առկա են HTML -ում:  
Այսինքն՝ դրանց աշխատանքը չի վերաբերվում դինամիկ  
մեթոդով ստեղծված տեգերին: Այս իմաստով, jQuery -ին մեզ  
տրամադրում է ոռ ֆունկցիան, որը թույլ է տալիս ընդունել  
ավելացնել ինչպես դինամիկ, այնպես էլ ստատիկ տեգերին:

```
$("document").on("click", ".item", function(){
```

// կոդը

)

Այսպիսով՝ ոռ ֆունկցիայի առաջին պարամետրը event - անունն  
է, երկրորդն այն տեգը, որի նկատմամբ կիրառվում է event -ը,  
իսկ երրորդը՝ անանուն ֆունկցիան, որը պետք է կանչվի:

Օռ ֆունկցիան ունի ևս մի քանի կիրառություն: Նրա օգնությամբ  
կարող ենք նոյն տեգին, կամ տեգերի խմբին, տարբեր event-ների ժամանակ կապել նոյն ֆունկցիան:

```
$("document").on("click mousemove mouseenter", ".item",
```

```
function(){
```

// կոդը

)

Կարող ենք նաև տարբեր event-ներին տարբեր ֆունկցիաներ  
կապել նոյն տեգի կամ տեգերի խմբի նկատմամբ.

```
$("#main").on({
```

```
    mousemove: function(){
```

```
},
```

```
    click: function(){
```

```
},
```

```
    mouseleave: function(){
```

```
}
```

)

### 19.3 Event-ների հեռացում

Ինչպես արդեն գիտենք, JS -ի դասընթացից, ցանկացած HTML  
տեգի կարելի է ոչ միայն ավելացնել, այլև նրանից հեռացնել  
event-ներ: JavaScript -ում այս նպատակով կիրառում ենք  
addEventListener և removeEventListener ֆունկցիաները: jQuery -

ում արդեն օգտվում ենք unbind և off ֆունկցիաներից: Ըստ որում, առավել հաճախ հանդիպում է off -ը, իսկ unbind -ը jQuery -ի հետագա տարրերակներում այլևս չի կիրառվելու, ինչպես delegate, live և մի շարք այլ ֆունկցիաներ:

Դիտարկենք օրինակը.

```
$document.on("click", ".item", function(){
    alert("ok");
    $document.off("click", ".item");
```

)  
Մեր օրինակում, .item կլասն ունեցող տեգերին կապվել է click event, որի ժամանակ alert է արվում օկ տեքստը և անմիջապես դրանից հետո, .item -ից հեռացվում է այդ event -ը: Այս դեպքում, button -ի սեղման event -ը կաշխատի ընդամենը մեկ անգամ, քանի որ event -ն անմիջապես հեռացվում է:

Եթե event -ը կապվել է ոչ թե անանուն ֆունկցիայի այլ կոնկրետ անուն ունեցող ֆունկցիայի օգնությամբ, ապա off ֆունկցիային հնարավոր է փոխանցել նաև երրորդ պարամետր, որն արդեն կիյնի ֆունկցիայի անունը: Այս դեպքում կիեռացվի ոչ թե ամբողջական event -ը, այլ event -ի ժամանակ գործող կոնկրետ ֆունկցիան:

Միանգամյա գործարկման event-ներ հնարավոր է ստեղծել նաև one ֆունկցիայով.

```
$document.one("click", ".item", function(){
    alert("ok");
```

)  
Այս դեպքում ֆունկցիան կաշխատի ընդամենը մեկ անգամ և event -ն անմիջապես կիեռացվի: Այսպիսով՝ off գործողությունը տեղի կունենա ավտոմատ:

## ՀԱՐՑԵՐ ԵՎ ԱՌԱՋԱԴՐԱՆՔՆԵՐ

1. Ի՞նչ է իրենից ներկայացնում event -ը, ինչպիսի՞ առաջիկ հետ եք ծանոթ:
2. Բացատրել ո՛ո, ուղ և off ֆունկցիաների դերը:
3. Ո՞րն է preventDefault ֆունկցիայի նշանակությունը:
4. Մկնիկի հետ կապված ի՞նչ առաջիկ հետ եք ծանոթ:
5. Ո՞րն է mousedown -ի և click -ի տարրերությունը:
6. Ո՞րն է սեղման ժամանակ clientX և pageX հատկությունների միջև տարրերությունը:

# ԴԱՍ 20

## ԽԱՂԵՐԻ ԵՐԱԳՐԱՎՈՐՈՒՄ



## ԴԱՍ 20: 2D ԽԱՂԵՐԻ ԾՐԱԳՐԱՎՈՐՄԱՆ ՕՐԻՆԱԿ

### 20.1 Խաղի նկարագրությունը

Փորձենք կիրառել jQuery -ի հնարավորությունները պարզագույն համակարգչային խաղի մշակելիս:

Խաղում գործող օբյեկտները երկու մեքենաներն են, որոնցից առաջինը պատկանում է խաղացողին և կարող է շարժվել միայն աջ և ձախ, իսկ երկրորդը իջնում է վերևից ներքև՝ ընթացքը սկսելով պատահական կետից: Խաղացողը պետք է փորձի խոսափել մեքենաների միջև հնարավոր բախումից:



Բախման դեպքում խաղն ավարտվում է: Բերված նկարում կարող ենք տեսնել խաղի արտաքին տեսքը (html, css):

### 20.2 Խաղի արտաքին տեսքի կառուցումը

Դիտարկենք կիրառված HTML կոդը.

**<body>**

```
<div id="road">
    <div id="my-car"></div>
    <div id="car-2"></div>
    <section></section>
</div>
```

**</body>**

#road div - ը ներկայացնում է ճանապարհի սև հատվածը: #my-car -ը՝ խաղացողի ավտոմեքենան: #car-2 -ն այն մեքենան է, որը պետք է շարժվի վերևից ներքև:

Ճանապարհի բաժանարար սպիտակ գծերն իրենցից ներկայացնում են section տեղեր, որոնք div -ին կավելացվեն դինամիկ մեթոդով:

Դիտարկենք կիրառված CSS կոդը.

```
*{ margin: 0; padding: 0; }
body{ height: 100vh; background:gray; }
#road{ position: relative; height: 100vh; width: 40%; margin: auto; background: #000; overflow: hidden; }
#my-car{ position: absolute; bottom: 20px; right: 30px; width: 64px; z-index: 999; height: 131px; background-image: url(..../img/c1.png); background-size: cover; }
#car-2{ position: absolute; top: 20px; left: 30px; width: 64px; height: 131px; z-index: 999; background-image: url(..../img/c2.png); background-size: cover; }
```

```
}

#road > section{
    position: absolute;
    left: 0;
    right: 0;
    width: 8px;
    height: 40px;
    background: white;
    margin: auto;
```

```
}
```

### 20.3 Խաղի ֆունկցիոնալ մասը JavaScript -ում

Այժմ ուսումնասիրենք JS -ում գրված կոդը.

```
$(document).ready(function(){
    drawLines();
    bindEvents();
    setInterval(function(){
        animateLines();
        moveCar();
        check();
    },700)
})
```

Էջի բացմանը զուգընթաց իսկզբանե կանչվում են drawLines և bindEvents ֆունկցիաները, ապա սկսում է աշխատել 700 միլիվայրկյան պարբերությամբ ինտերվալը, որի ներսում կանչվում են animateLines(), moveCar() և check() ֆունկցիաները:

drawLines ֆունկցիան նախատեսված է դինամիկ մեթոդով էջին բաժանարար սպիտակ գծերի ավելացման համար:

bindEvents ֆունկցիայի ներսում մենք կգրենք խաղի հետ

կապված բոլոր event-ները:

animateLines ֆունկցիան նախատեսված է ճանապարհի բաժանարար գծերի շարժումն ապահովելու համար:

check ֆունկցիան պարբերաբար պետք է ստուգի՝ արդյո՞ք մերենաները բախվում են, թե ոչ:

Իսկ moveCar ֆունկցիան նախատեսված է վերևից ներքև իջնող մերենայի շարժման անհմացիան ապահովելու համար:

Ինչպես նկատեցինք, կոդի նախնական նկարագրության մեջ պահպանվել է ֆունկցիոնալ ծրագրավորման սկզբունքները, այսինքն՝ ստեղծել առանձին ֆունկցիաներ և կիրառել դրանց կանչերը:

Այժմ հերթականությամբ անդրադառնանք այդ ֆունկցիաներից յուրաքանչյուրին:

Դիտարկենք drawLines ֆունկցիան:

```
function drawLines(){
    for(let i = 0; i < 12; i++){
        let sect = $("<section></section>");
        sect.css("top", 80 * i + "px");
        sect.appendTo("#road");
    }
}
```

Ինչպես նկատեցինք, ֆունկցիան ցիկլի օգնությամբ էջին ավելացնում է 12 տարրեր section էլեմենտներ, որոնք ել հանդիսանում են ճանապարհի բաժանարար գծերը:

Յուրաքանչյուր գծի CSS -ում top հատկությունը սահմանվել է որպես 80 \* i: Պատճառն այն է, որ section տեղերն ունեն position:absolute դիրքավորումը, ինչի արդյունքում, դինամիկ ավելացնելու դեպքում նրանք կունենան նույն կոորդինատը: Գրված իրամանի կատարման արդյունքում յուրաքանչյուր երկու հարկան գծերի միջև հեռավորությունը կազմում է 80px:

Հաջորդ ֆունկցիան նախատեսված է գծերի շարժումն ապահովելու համար:

Այժմ դիտարկենք animateLines ֆունկցիան.

```
function animateLines(){
    $("section").each(function(){
        let t = parseInt($(this).css("top"))
        t += 35;
        $(this).css("top", t+"px");
    })
    prepend.NewLine();
}
```

ֆունկցիայի ներսում, ցիկլի օգնությամբ, դիտարկված են բոլոր section տեգերը: Յուրաքանչյուր հերթական section -ի համար, որ կոփոխականի մեջ պահպել է նրա տօք կոորդինատը: Նկատենք, որ եթե տեգի տօք կոորդինատը հավասար է օրինակ 470px -ի, ապա parseInt կիրառելու դեպքում արտահայտության տողային մասը՝ px -ը հեռացվում է տեքստից: Ինչպես նկատեցինք կոորդինատին ավելացվում է 35 արժեքը և ստացված նոր արժեքը կրկին փոխանցվում է տօք -ին: Ամփոփելով ֆունկցիայի աշխատանքի առաջին հատվածը՝ կարող ենք ասել, որ յուրաքանչյուր section -ի կոորդինատը փոխվում է 35 -ով: Բնական է, որ նման իրավիճակում վերևում գտնվող գիծը կիշնի ներքև և ճանապարհի վերին հատվածը կմնա դատարկ: Այդ է պատճառը, որ ֆունկցիայի վերջում կանչվել է մեկ այլ ֆունկցիա՝ prependNewLine: Վերջինս նախատեսված է #road -ի սկզբում դինամիկ ծրագրավորման մեթոդով նոր գիծ ավելացնելու համար:

Դիտարկենք ֆունկցիան.

```
function prependNewLine(){  
    let sect = $("<section></section>");  
  
    let t = $("section").first().css("top")  
  
    let first = parseInt(t) - 80;  
  
    sect.css("top", first + "px");  
  
    sect.prependTo("#road");  
  
    removeLastLine();  
}
```

Ինչպես կարելի է հասկանալ կողից, դինամիկ մեթոդով ստեղծվում է section տեգ, նրա համար, որպես կոորդինատ, սահմանվում է section -ներից առաջինի կոորդինատից 80-ով պակաս արժեք:

`$("#section").first()` ֆունկցիան վերադարձնում է առաջին section էլեմենտը, որը գտնվում է էջում:  
Ուշագրավ է այստեղ հատկապես `removeLastLine()` ֆունկցիայի կանչը:

Ինչպես գիտենք, մեզ մոտ ֆունկցիան կանչվել է ինտերվալի ներսում, հետևաբար, բնական է, որ արդյունքում էջին պարբերաբար ավելանում են section-ներ: `removeLastLine` ֆունկցիան ջնջում է section-ներից վերջինը, եթե նրա կոորդինատը գերազանցել է 700px -ը:

Այստեղ 700px -ը հարաբերական է, կախված է մոնիթորի չափությունից: Եթե այն ցանկանում ենք դարձնել բացարձակ, ապա կարելի է px -ների փոխարեն կիրառել %-ները, որպես չափման միավորներ:

Դիտարկենք `removeLastLine` ֆունկցիան:

```
function removeLastLine(){
    var t = parseInt($("#section").last().css("top"))
    if(t > $("#road").height())
        $("#section").last().remove();
    var t2 = parseInt($("#section").first().css("top"))
    if(t2 < -80)
        $("#section").first().remove();
}
```

Ֆունկցիան աշխատում է հետևյալ տրամաբանությամբ. Եթե կոորդինատների ներսում պահպանվում է section տեղերից վերջինի տոր կոորդինատը: Եթե այդ կոորդինատը գերազանցում է ճանապարհի բարձրությանը, ապա տվյալ էլեմենտը հեռացվում է էջից jQuery -ի `remove` ֆունկցիայի

օգնությամբ: Որպեսզի տեգերի կուտակում չունենանք բացասական դիրքում մենք ստուգում ենք նաև -80 դիրքից վեր գտնվող տարրերի առկայությունը: t2 փոփոխականի ներսում պահում ենք section-ներից առաջինի կոորդինատը: Եթե այն փոքր է -80 -ից, նշանակում է, որ ճանապարհի սկզբնագծից վեր առկա են տեգեր, այդ դեպքում ջնջում ենք նաև first էլեմենտը:

#### 20.4 Մեքենաների կառավարումը

Խաղի ընթացքում մեքենաների կառավարման գործընթացը կարելի է բաժանել երկու փուլի:

1. Վերևից ներքև իջնող մեքենայի շարժման ապահովումը
2. Խաղացողի մեքենայի շարժման ապահովումը

Առաջին կետի ապահովման համար ստեղծված ֆունկցիան կոչվում էր moveCar: Ֆունկցիան գտնվում էր ինտերվալի ներսում, իետևաբար այն պարբերաբար կանչվում է: Դիտարկենք գրված կոդը.

```
var carTop = 0;
function moveCar(){
    carTop += 10;
    $("#car-2").css("top", carTop + "%");
    if(carTop > 90){
        carTop = -20;
        let carLeft = parseInt(Math.random() * 80);
        $("#car-2").css("left", carLeft + "%");
    }
}
```

Այսպիսով՝ ունենք carTop գլոբալ փոփոխականը, որին նախապես տրված է 0 արժեք: Փոփոխականը ներկայացնում է մեքենայի նախնական կոորդինատը: Ֆունկցիայի յուրաքանչյուր կանչի դեպքում այդ կոորդինատն ավելանում է 10 -ով՝ փոփոխանցվելով #car-2 div -ին %-ների օգնությամբ: Եթե այդ կոորդինատը գերազանցում է 90% -ը, ապա բնական է, որ մեքենան արդեն դուրս է եկել խաղադաշտի սահմաններից, իետևաբար անհրաժեշտ է նրան ետ վերադարձնել: Այդ նպատակով էլ carTop փոփոխականի արժեքը դառնում է -20:

Քանի որ անհրաժեշտ է, որպեսզի մեքենան յուրաքանչյուր անգամ իջնի պատահական կետից, ուստի հայտարարվել է նաև carLeft փոփոխականը, որի վերագրվել է պատահական արժեք 0-ից 80%<sup>24</sup> սահմաններում: Վերջնական արդյունքում, մեքենան պարբերաբար իջնում է վերևից ներքև, յուրաքանչյուր անգամ պատահական դիրքից:

Եթե խոսքը վերաբերվում է արդեն խաղացողի մեքենայի շարժմանը, ապա այստեղ անհրաժեշտ է event կապել ստեղնաշարին, որպեսզի աջ կամ ձախ սլաքները սեղմելու դեպքում մեքենան շարժվի համապատասխան ուղղություններով: Ինչպես արդեն նշվել է, event-ների հետ կապված բոլոր գործողությունները պետք է գրվեն bindEvents ֆունկցիայում:

Դիտարկենք ֆունկցիան.

```
function bindEvents(){
```

```
    $("body").keydown(function(e){  
        if(e.key == "ArrowRight"){  
            goRight();  
        }else if(e.key == "ArrowLeft"){  
            goLeft();  
        }  
    })
```

```
}
```

Ինչպես նկատեցինք, body -ի նկատմամբ keydown event -ի ցանկացած աշխատանքի դեպքում իսկզբանե ստուգվում է արդյո՞ք սեղմվել է ArrowRight կամ ArrowLeft ստեղներից որևէ մեկը: Այդ իրավիճակներից յուրաքանչյուրում կանչվում է համապատասխանաբար goLeft և goRight ֆունկցիաները: goLeft ֆունկցիան նախատեսված է մեքենան դեպի ձախ տեղաշարժելու համար, իսկ goRight -ը՝ աջ:

Դիտարկենք երկու ֆունկցիաները.

<sup>24</sup> Այստեղ 80% կիրառելու պատճառն այն է, որ մեքենան գրավում է մոտ 20% տարածք ճանապարհի վրա:

```

var rightCoord = 0;
function goRight(){
    rightCoord -= 10;
    if(rightCoord < 5){
        rightCoord = 5;
    }
    $("#my-car").css("right", rightCoord + "%");
}

function goLeft(){
    rightCoord += 10;
    if(rightCoord > 80){
        rightCoord = 80;
    }
    $("#my-car").css("right", rightCoord + "%");
}

```

Մեքենայի աջ և ձախ ուղղություններով շարժման համար կիրառվել է նրա նոյն right պարամետրը: Մի դեպքում այն աճում է, մյուսում՝ նվազում:  
 Որպես նախնական արժեք rightCoord -ի համար ընտրվել է 0 -ն: goRight -ի ժամանակ փոփոխականի արժեքը պակասել է 10 -ով, իսկ goLeft -ի դեպքում՝ ավելացել: Ինչպես կարելի է նկատել, կոդում նաև ստուգվել են պայմաններ: Պատճառն այն է, որ միայն աջ, կամ, միայն ձախ ուղղությամբ շարժվելու դեպքում մեքենան դուրս կգա ճանապարհի սահմաններից: Հետևաբար, աջ շարժման դեպքում, եթե մեքենայի աջ կողորդինատը փոքր է 5 -ից<sup>25</sup>, ապա այն կրկին դառնում է 5: Մեքենան երբեք չի կարող գերազանցել աջ



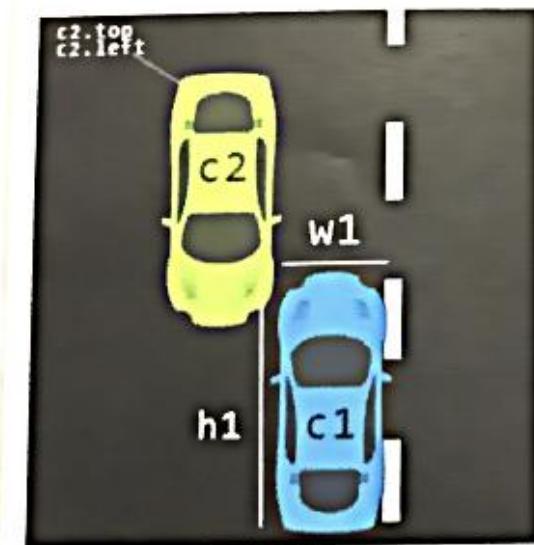
<sup>25</sup> Այստեղ 5 -ը ներկայացնում է մեքենայի և ճանապարհի ձախ եզրագծի միջև առկա նվազագույն հեռավորությունը:

ուղղությունը 5 -ից պակաս արժեքով: Զախ շարժման դեպքում, ստուգվել է աջ ուղղության արժեքը 80% սահմանով: Այսինքն, եթե մեքենայի right պարամետրը գերազանցում է 80% -ը, ապա այն կրկին դառնում է 80:

## 20.5 Մեքենաների բախումը

Ըստհանրապես ֆիզիկայից հայտնի է, որ երկու մարմիններ հանդիպում են միմյանց, եթե նրանց x,y կոորդինատները նույն են: Մեր օրինակում, սակայն, եթե մեքենաները ունեն w1 լայնություն և h1 բարձրություն, խնդրի կառուցվածքը փոխվում է:

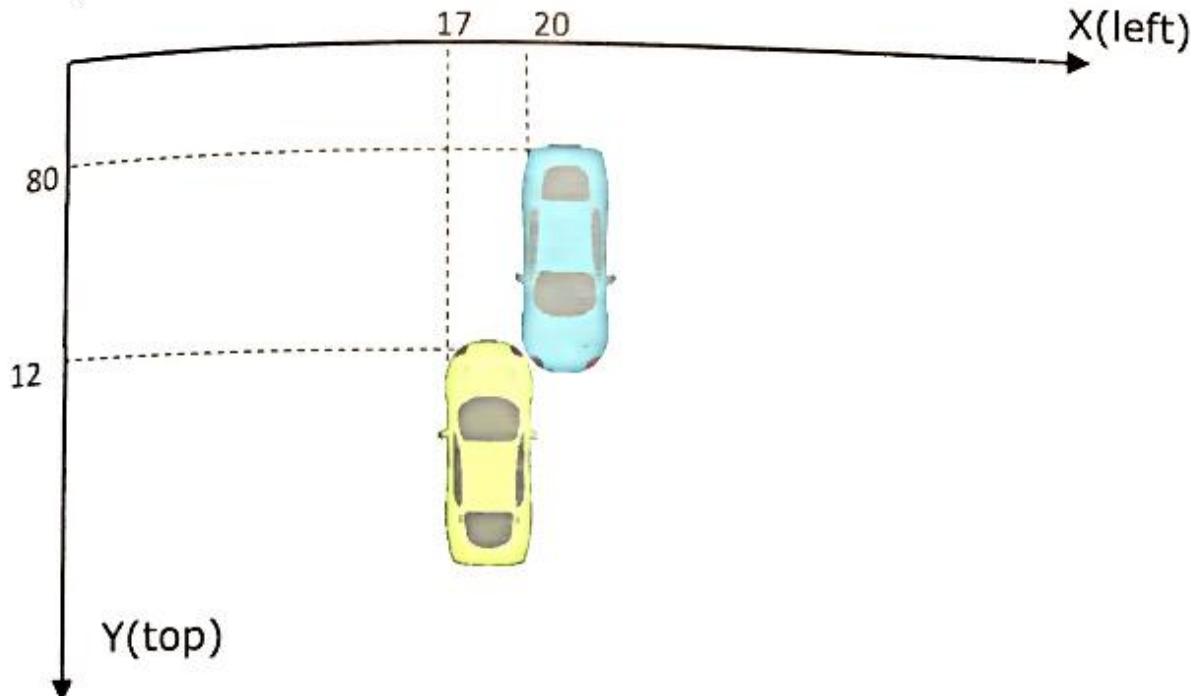
jQuery -ին HTML էլեմենտների կոորդինատները որոշելու համար մեզ տրամադրում է offset հատուկ ֆունկցիան, որը վերադարձնում է տվյալ էլեմենտի top և left կոորդինատները: Ձեզ մոտ, գուցե, հարց առաջանա, թե ինչու չի վերադարձնում նաև bottom և right կոորդինատները: Պատճառն այն է, որ հարթության վրա ցանկացած մարմնի դիրք կարելի է որոշել ընդամենը երկու կոորդինատով: Մեր օրինակում left -ը ցույց կտա տեղի դիրքը հորիզոնական ուղղությամբ, իսկ top -ը՝ ուղղահայաց:



Ինչպես նկատեցինք պատկերված նկարում, top և left ասելով նկատի ունենք, մեքենայի վերին ձախ գագաթի

Կոորդինատները:  $w1$  -ով նշանակված է մեքենայի լայնությունը, իսկ  $h1$  -ով՝ բարձրությունը: Բնական է, որ մեքենաների բախումը որոշելու համար նպատակահարմար է դիտարկել նրանց կոորդինատների տարբերությունները մոդովով:

Դիտարկենք պատկերը



Ինչպես նկատեցինք, ստանդարտ կոորդինատային հարթության վրա բերված է մեքենաների որոշակի դիրք: Նկատենք, որ ընտրված կոորդինատները պատահական են, ընտրվել են բախման պրոցեսն առավել հասկանալի ներկայացնելու համար:

Այսպիսով՝ կանաչ մեքենայի (խաղացողի մեքենա)  $x1$  կոորդինատը կլինի 170 -ը,  $y1$  -ը 120 -ը: Վերևից ներքև իջնող մեքենայի կոորդինատները կլինեն՝  $x2 = 200$ ,  $y2 = 80$ :

Նկատենք, որ  $x$  կոորդինատը ցույց է տալիս jQuery -ում offset<sup>26</sup> -ի left պարամետրը, իսկ  $y$  -ը՝ top:

Ընդունենք  $w1 = 60$ ,  $h1 = 90$ , որտեղ  $w1$  -ը մեքենաների լայնությունն է, իսկ  $h1$  -ը՝ բարձրությունը: Դիտարկենք  $|x1 - x2|$  տարբերության բացարձակ արժեքը, որը կստացվի  $| -30 |$ ,

<sup>26</sup> ֆունկցիա է, որը վերադասնում է տվյալ HTML տեգի ընթացիկ կոորդինատները՝ top և left:

այսինքն՝ 30: Համեմատելով  $30 - w$  մեքենաների  $w$  լայնության հետ՝ կտևսնեք, որ  $w > 30$ , նշանակում է, հորիզոնական հարթության մեջ առկա է բախման համար անհրաժեշտ պայմանը:

Դիտարկենք  $|y_1 - y_2|$  տարրերությունը, որի արժեքը կլինի 40: Համեմատելով  $40 - w$  մեքենաների բարձրության հետ՝ կստանանք  $h > 40$ : Նշանակում է, որ առկա է ուղղահայաց բախման անհրաժեշտ պայմանը: Եթե երկու մեքենաների համար առկա են ուղղահայաց և հորիզոնական բախման պայմանները<sup>27</sup>, ապա մեքենաների բախումն անխուսափելի է: Ասկան անհրաժեշտ է ստուգել և իրականացնել check մոդուլը:

Դիտարկենք կոդը.

```
function check(){
    let c1 = $("#my-car").offset();
    let c2 = $("#car-2").offset();
    let w1 = $("#my-car").width();
    let h1 = $("#my-car").height();
    let d1 = Math.abs(c2.left - c1.left);
    let d2 = Math.abs(c2.top - c1.top)

    if(d1 < w1 && d2 < h1){
        //կոդը պետք է գրվի այստեղ
    }
}
```

}  
If -ի ներսում կգրվի այն կոդը, որը պետք է աշխատի բախման դեպքում:

<sup>27</sup> Որպեսզի մեքենաները բախվեն, անհրաժեշտ է, որ նրանց միջև եղած հորիզոնական հեռավորությունը լինի ավելի փոքր, քան մեքենայի լայնությունը, իսկ ուղղահայաց հեռավորությունը լինի ավելի փոքր, քան մեքենայի բարձրությունը: