Դաս 05: Լամբդա արտահայտություններ

Գծային ֆունկցիաները (Arrow functions) ֆունկցիաների հայտարարման հակիրճ տարբերակն է:

```
// Սովորական ֆունկցիա
function add(a, b) {
    return a + b;
}

// Գծային ֆունկցիա
const add = (a, b) => a + b;
```

Գծային ֆունկցիաները տարբերվում են սովորական ֆունկցիաներից՝

- Բացակայում է function հատուկ անունը
- Եթե գծային ֆունկցիայում միայն մեկ արտահայտություն է, կարելի է բաց թողնել return և {} փակագծերը:
- Գծային ֆունկցիան This-ը վեկալում է այն բլոկից, որտեղ իրեն հայտարարել են, իսկ սովորական ֆունկցիան՝ այն օբյեկտից, որը իրեն կանչել է:

Գծային ֆունկցիաները անանուն ֆունկցիաներ են: Անանուն ֆունկցիա կարելի է ստանալ նաև սովորական ֆունկցիայից, բավարար է ուղղակի չհայտարարել ֆունկցիայի անուն:
Սակայն նման ֆունկցիաներին իրենց հայտարարումից հետո հասանելիություն ստանալը դառնում է անհնարին, եթե իրենց չենք վերագրել որևէ փոփոխականի:

```
const F1 = function () {
    console.log("անանուն ֆունկցիա");
}
```

```
const F1 = () => {
    console.log("գծային ֆունկցիա");
```

```
}
```

Մեկ ֆունկցիան կարելի է հայտարարել մեկ այլ ֆունկցիայի մեջ, ինչպես նաև ֆունկցիան կարող է վերադարձնի մեկ այլ ֆունկցիա:

```
function F1(){
    let x=5;
    return function(){
        let y = 10;
        return function(){
            let z = 7;
            return x + y + z;
        }
    }
}
console.log(F1()()());
```

Սակայն նման դեպքերում որպեսզի կարողանանք տեսնել երրորդ ֆունկցիայի արդյունքը, անհրաժեշտ է առաջին ֆունկցիան կանչել մի քանի () -ի միջոցով, քանի որ առաջին ֆունկցիան վերադարձնում է ֆունկցիա, և որպեսզի այն աշխատի վերադառնալուց հետո, իրեն ևս տալիս ենք () և դա անում ենք այնքան ժամանակ, քանի դեռ ունենք վերադարձվող ֆունկցիա:

Ֆունկցիայում հայտարարված փոփոխականը ունի հասանելիություն իր ներսում հայտարարված ֆունկցիաների բլոկում: Դա աշխատում է JavaScript-ի **փակվածքի (closure)** մեխանիզմի շնորհիվ: Օրինակ եթե մենք հայտարարենք ֆունկցիա F2, իրեն փոխանցենք a փոփոխական, իսկ ներսում հայտարարենք x փոփոխականը, ապա F2-ի մեջ հայտարարված T1 փոփոխականը ֆիքսում է իր շրջակա միջավարի փոփոխականները (a, x) և հետագայում անկախ այն վարից, թե

որտեղ են իրեն կանչում, այն կարող է վերադարձնել տվյալ փոփոխականները:

```
function F2(a){
    let x = 10;
    function T1(){
        let y = 20;
        return x + y + a;
    }
    return T1;
}
console.log(F2(5)());
```

JavaScript-ը նաև հանդիսանում է **բլոկային հասանելիության** լեզու, այսինքն՝ **(lexical scoping)**: Փոփոխականները հասանելի են այն բլոկներում և իրենց դուստր բլոկներում, որոնցում հայտարարվել են: Այսինքն տվյալ բլոկներից դուրս իրենք չունեն հասանելիություն:
Օրինակ եթե ստեղծենք T2 ֆունկցիա և իր ներսում հայտարարենք x փոփոխական, ապա տվյալ փոփոխականը գոյություն կունենա միմիայն տվյալ ֆունկցիայի բլոկի հասանելիության տիրույթում: Այսինքն եթե մենք T2 ֆունկցիան կանչենք որևէ T3 ֆունկցիայի մեջ, և փորձենք T3 ֆունկցիայում տպել T2 ֆունկցիայի X փոփոխականը, ապա կստանանք ReferenceError, քանի որ T3 ֆունկցիայում x-ը հասանելի չէ:

```
function T2(){
    let x = 'ok';
}
function T3(){
    T2();
    console.log(x);
}
T3();
```

Հայտարարում (Hosting) և Կատարում (Execution)

```
A1();           // Կանչում ենք A1() ֆունկցիան առաջին անգամ:
console.log(x);       // Փորձում ենք տպել x փոփոխականը:
function A1() {
    alert('Hello');// Սկզբնական հայտարարությունը՝ A1 ֆունկցիա:
}
var x = 10;           // x փոփոխականի var-ով հայտարարությունը:
let x = 10;           // x փոփոխականի let-ով հայտարարությունը:
function A1() {
    alert('Barev');  // A1 ֆունկցիայի նոր հայտարարություն:
}
A1();             // Կանչում ենք A1() ֆունկցիան երկրորդ անգամ:
console.log(x);       // Փորձում ենք տպել x փոփոխականը:
```

1. Հայտարարությունների փուլ
   - Բոլոր **ֆունկցիաները** և **var-ով հայտարարած փոփոխականները** բարձրացվում են վերև (hoisted):
   - Ֆունկցիաները ամբողջությամբ "բարձրացվում են", այսինքն՝ նրանց ամբողջ մարմինը հասանելի է ցանկացած վայրում:
   - **var** փոփոխականները նույնպես բարձրացվում են, սակայն ի տարբերություն ֆունկցիաների, բարձրանում են միայն նրանց **հայտարարությունները**, չներառելով իրենց արժեքավորումները:
   - **let փոփոխականները չեն հոստվում**, այլ մնում են **"Temporal Dead Zone"**-ում **(ժամանակավոր մահացած գոտի)**, մինչև նրանց փաստացի հայտարարվելը:
2. Կատարման փուլ (Execution):
   - Կոդը կատարվում է գծային՝ **վերևից ներքև**:

Հոսթինգից հետո կոդը ստանում է հետևյալ արտաքին տեսքը՝

```
function A1() {
    alert('Barev'); // Վերջին հայտարարված A1 ֆունկցիան "հաղթում
է":
}
var x;
```

```
// x փոփոխականի var-ով հայտարարումը բարձրացվել է վերև:
let x;                  // let x-ը մնում է Temporal Dead Zone-ում:

A1();
// Կատարվում է A1() ֆունկցիան, որը ցույց է տալիս "Barev":
console.log(x);
// Արդյունք՝ undefined (քանի որ x-ի var-ն է բարձրացվել, բայց դեռ արժեքավորված չէ):
x = 10;                 // x-ին var-ով տրվում է արժեք՝ 10:
x = 10;                 // let x-ի պատճառով կոդը կսեղծի սխալ:
console.log(x);
// Արդյունք՝ կտեսնեք 10 եթե let-ի սխալը վերացնեք:
```

Ֆունկցիան կարելի է նաև դարձնել մեկանգամյա օգտագործման՝ (immediately invoked function, IIFE)

```
let A3 = function(x, y){
// մեկ անգամյա օգտագործման ֆունկցիա
    return x * y;
}(7, 8)      // Վերադարձնում ենք սկզբնական արժեքներ 7 և 8
console.log(typeof(A3));
```

Մենք հայտարարել ենք ֆունկցիա, որը վերագրել ենք **let** փոփոխականին: Ֆունկցիան ակնկալում է ստանալ երկու փոփոխական՝ x, y և վերադարձնում է x * y:

{} փակագծերից հետո կանչել ենք այդ ֆունկցիան, փոխանցելով նրան փոփոխականների արժեքները՝ (7, 8): Դրանից հետո let A3 փոփոխականը ստանում է անանուն ֆունկցիայի արժեքը՝ 56: Այս գործողությունից հետո անանուն ֆունկցիան դառնում է անհասանելի հետագա կանչերի համար:

Ֆունկցիան կարող է վերադարձնի նաև տերնար օպերատորի արտահայտություն:

```
let A3 = function(x, y){
    // if(x > y){
    //     return x*y;
    // } else {
```

```
    //      x ** y;
    // }
    return (x > y) ? x * y : x ** y;     // ternar operator
}
```

Նույն արտահայտությունը կարող ենք գրել նաև օգտագործելով գծային ֆունկցիաներ՝

```
let A3 = (x,y) => x>y?x*y:x**y;
console.log(typeof(A3));
```

Գծային ֆունկցիան նույնպես կարելի է գրել IIFE եղանակով՝

```
let A4 = ((x,y) => x>y ? x*y : x**y) (2,3);
console.log(typeof(A4));
```

[Tutor code visualizer]