TYPESCRIPT

ՁԵՌՆԱՐԿ

Հեղինակ՝ Բաղդասարյան Աշոտ

ԲՈՎԱՆԴԱԿՈՒԹՅՈՒՆ

2. **Փոփոխականներ (Variables)**
    2.1. Ի՞նչ է փոփոխականը
    2.2. Ի՞նչպես ստեղծել փոփոխական
    2.3. Հաճախ օգտագործվող ֆունցիաներ
    2.4. Console Օբյեկտ
    2.5. Հաղորդագրություններ(Comments)
3. **Տվյալների տեսակները (Data Types)**
    3.1. Տեսակի նշանակությունը
    3.2. Հիմնական տվյալների տեսակները
    3.3. Number տեսակ և հաճախ օգտագործվող մեթոդներ
    3.4. String տեսակ և հաճախ օգտագործվող մեթոդներ
4. **Պայմանական օպերատորներ (Conditional operators)**
    4.1. Ինչ է պայմանը
    4.2. Ինչպես ստեղծել պայման և մեկից ավել պայմաններ
    4.3. Եռակի օպերատոր (Ternary Operator ? :)
    4.4. Switch - Case
    4.5. Ինչ է Boolean-ը
    4.6. Ինչպես են ստացվում boolean արժեքներ
    4.7. Օրինակ առօրյա կյանքից
    4.8. Տրամաբանական օպերատորներ
    4.9. Երբ այլ տիպեր վերածվում են boolean-ի
    4.10. Խնդիրների մեջ կիրառելիություն
5. **Ցիկլեր (Loops)**
    5.1. Ի՞նչ է ցիկլը
    5.2. Ի՞նչ է ալգորիթմը
    5.3. Ի՞նչպես Ստեղծել ցիկլ
    5.4. Օրինակ առօրյա կյանքից
    5.5. խնդիրների մեջ կիրառելիություն
6. **Զանգվածներ (Arrays)**
    6.1. Ի՞նչ է զանգվածը
    6.2. Ի՞նչպես ստեղծել զանգված
    6.3. Զանգվածի արժեքներին հասանելիություն
    6.4. Օրինակ առօրյա կյանքից
    6.5. Պարունակության փոփոխում
    6.6. Խնդիրների մեջ կիրառություն
7. **Գործառույթներ (Functions)**
    7.1. Ի՞նչ է գործառույթը
    7.2. Գործառույթի կառուցվածքը
    7.3. Գործառույթներ արգումնետներով
    7.4. Գործառույթներ վերադարձվող արժեքով
    7.5. Օրինակ առօրյա կյանքից
    7.6. խնդիրների մեջ կիրառություն
8. **Զանգվածների մեթոդներ (Array Methods)**
    8.1. Push(), ավելացնել տարր վերջում
    8.2. Pop(), հեռացնել վերջին տարրը
    8.3. Shift() և unshift()

8.4. ForEach(), աշխատել բոլոր տարրերի հետ
8.5. Map(), փոխել զանգվածը նորով
8.6. Filter(), զտել տարրերը
8.7. Find(), գտնել առաջին համապատասխան տարրը

9. Օբյեկտներ (Objects)
9.1. Ի՞նչ է օբյեկտը
9.2. Օբյեկտի ստեղծում
9.3. Տվյալների հասանելիություն
9.4. Օբյեկտի հատկությունների փոփոխում
9.5. Օրինակ առօրյա կյանքից
9.6. Օբյեկտների ներքին օբյեկտներ
9.7. Խնդիրների մեջ կիրառելիություն

10. Template Literals
10.1. Ի՞նչ է Template Literals-ը
10.2. Օրինակ առօրյա կյանքից

11. Falsy և Truthy արժեքներ
11.1. Ի՞նչ է Falsy արժեք
11.2. Ի՞նչ է Truthy արժեք

12. Typeof օպերատոր
12.1. Ի՞նչպես օգտագործել typeof օպերատորը
12.2. Null և undefined տեսակի արժեքներ
12.3. Օրինակ առօրյա կյանքից

13. Math Object
13.1. Math-ի օգտակար մեթոդներ
13.2. Կիրառություն խնդիրների մեջ

14. JSON (JavaScript Object Notation)
14.1. Ինչպիսին է JSON-ը
14.2. JSON մեթոդներ

15. Սխալներ և սխալների կառավարում
15.1. Error-ների հիմնական տեսակներ
15.2. Ի՞նչպես կանխել սխալները(try,catch)
15.3. Ի՞նչպես ստեղծել մեր սեփական Error օբյեկտը

16. ISNaN ֆունկցիա
16.1. Ի՞նչ է NaN-ը
16.2. Ինչ է անում isNaN-ը

17. Continue/Break
17.1. Ի՞նչ է continue հրամանը
17.2. Ի՞նչ է անում continue հրամանը
17.3. Ինչ է break հրամանը
17.4. Ի՞նչ է անում break հրամանը
17.5. Կիրառելիություն խնդիրների մեջ

18. Arrow Functions

19. Generator Functions

20. Scope
20.1. Ինչ է scope-ը
20.2. Արտահայտությունների և հրամանների վերջ

Մաս 1 - Նախաբան

Հակիրճ նկարագրություն

Այս գիրքը նախատեսված է սկսնակներից մինչև փորձառու ծրագրավորողների համար, ովքեր ցանկանում են խորությամբ սովորել TypeScript: Այն ընդգրկում է լեզվի հիմնական հասկացությունները, տիպային համակարգը, օբյեկտ-կողմնորոշված ծրագրավորման (OOP - Object-Oriented Programming) հնարավորությունները, JavaScript-ի հետ համատեղելիությունը, ինչպես նաև գործնական օրինակներ:

Ինչ կսովորեք այս գրքից

- TypeScript-ի հիմունքները
- Տիպեր, ինտերֆեյսներ, կլասներ և ժառանգում
- Տիպային համակարգի ուժեղ կողմերը
- JavaScript-ի և TypeScript-ի տարբերությունները
- Օբյեկտ-կողմնորոշված ծրագրավորում
- Լավագույն պրակտիկաներ և օպտիմալ կոդի կառուցում

Ինչու սովորել TypeScript

Վերջին տարիներին TypeScript-ը դարձել է JavaScript-ի ամենահզոր գործիքներից մեկը: Այն ստեղծվել է Microsoft-ի կողմից՝ որպես JavaScript-ի սուպերսեթ, որը ավելացնում է ստատիկ տիպավորում և այլ հնարավորություններ որոնք չկան JavaScript ում:

Ստատիկ տիպավորումը թույլ է տալիս ծրագրավորողին դեռ կոդ գրելու ժամանակ կանխել սխալներ՝ նշելով, թե տվյալ փոփոխականը կամ ֆունկցիան ինչ տիպի տվյալներ պետք է ունենա:

Սկզբում թվում էր, թե TypeScript-ը պարզապես լրացում է JavaScript-ի համար, բայց այսօր այն լայնորեն օգտագործվում է արտադրության մակարդակով՝ սկսած վեբ հավելվածներից մինչև սերվերային համակարգեր:

TypeScript-ը օգնում է ստեղծել ավելի մեծ ու կայուն ծրագրեր՝ իր տիպային համակարգի շնորհիվ: Այն հիանալի է հատկապես մեծ թիմային նախագծերի համար, որտեղ բոլոր անդամները կարող են վստահ լինել, որ տվյալները փոխանցվում են ճիշտ և կանխատեսելի ձևով:

Ում համար է այս գիրքը նախատեսված

Այս գիրքը նախատեսված է նրանց համար, ովքեր.

- Ցանկանում են սովորել TypeScript՝ հիմքերից մինչև բարդ թեմաներ
- Արդեն օգտագործում են TypeScript և ուզում են բարձրացնել իրենց հմտությունները և ձեռք բերել ավելի շատ փորձ:

Եթե դուք սկսնակ եք, բայց արդեն ծանոթ եք HTML-ին և CSS-ին, ապա այս գիրքը կօգնի ձեզ վստահորեն առաջ շարժվել՝ սկսելով պարզ օրինակներից մինչև իրական նախագծեր:

Ինչ պետք է իմանալ առաջ անցնելիս

TypeScript-ը կառուցված է JavaScript-ի հիման վրա:
Ուստի եթե ունեք JavaScript-ի մասին գիտելիքներ, TypeScript-ը սովորելը շատ ավելի հեշտ կլինի:

Եթե դեռ չգիտեք JavaScript, անհանգստանալու կարիք չկա: Այս գիրքը ներկայացնում է համառոտ համեմատություններ և պարզ օրինակներ, որոնք կօգնեն նույնիսկ սկսնակներին հեշտությամբ ընկալել և օգտագործել TypeScript-ը:

Ի՞նչ գործիքներ ենք օգտագործելու գրքում

- Տեքստային խմբագրիչ, մենք օգտագործելու ենք Visual Studio Code
- Node.js և npm՝ TypeScript-ը տեղադրելու և գործարկելու համար
- Command Prompt՝ հրամաններ աշխատեցնելու համար

Ինչպես օգտվել այս գրքից առավել արդյունավետ

- Սկսեք կարդալ հերթականությամբ, առանց հատվածներ բաց թողնելու
- Փորձեք գրել ձեր սեփական օրինակները ամեն նոր գաղափար սովորելուց հետո
- Եթե ինչ-որ բան անհասկանալի է, կարդացեք այնքան մինչև ամբողջական հասկանաք նյութը և նոր առաջ անցեք:

Մաս 2 - Ծանոթություն

Ի՞նչ է TypeScript-ը

TypeScript-ը ծրագրավորման լեզու է, որը ընդլայնում է JavaScript-ը՝ ավելացնելով ստատիկ տիպավորում:
Սա նշանակում է, որ կարող եք նախապես նշել փոփոխականների, ֆունկցիաների և օբյեկտների տիպերը և ստանալ սխալների մասին ահազանգեր դեռևս գրելու փուլում, մինչև ծրագիրը աշխատեցնելը:

Ստատիկ տիպավորում նշանակում է, որ տվյալների տիպերը որոշվում և ստուգվում են կոդ գրելու պահին, ոչ թե ծրագրի աշխատելու ընթացքում: Սա հակառակն է դինամիկ տիպավորման, ինչպիսին է JavaScript-ը:

TypeScript կոդը պահվում է `.ts` ֆայլերում և կոմպիլացվում (թարգմանվում) է սովորական `.js` ֆայլերի, որպեսզի կարողանա աշխատել ցանկացած բրաուզերում կամ JavaScript միջավայրում:

Օրինակ. (փոքրիկ գործնական օրինակ ավելացնենք)

```
let message: string = "Բարև աշխարհ"; // Ստեղծում ենք փոփոխական message անունով որին վերագրում ենք "Բարև աշխարհ"
console.log(message); // Տպում ենք արդյունքը՝ "Բարև աշխարհ"
```

Այստեղ մենք նշում ենք, որ `message` փոփոխականը պետք է լինի տիպով `string`:
Եթե փորձենք նրան այլ տիպի արժեք նշանակել (օրինակ՝ թիվ), կստանանք սխալ՝ դեռ կոդ գրելու պահին:

Փոփոխականներ TypeScript-ում

TypeScript-ում, ինչպես JavaScript-ում, փոփոխականները օգտագործվում են տվյալներ պահելու համար: Սակայն TypeScript-ը հնարավորություն է տալիս փոփոխականների տիպը բացահայտորեն նշել, ինչը օգնում է կանխել տիպային սխալները կոդ գրելու ընթացքում:
Հետագա բաժիններում մենք ավելի մանրամասն կծանոթանանք փոփոխականների տիպերին և դրանց հայտարարության տարբեր եղանակներին:

Ինչու է TypeScript-ը կարևոր

✅ Օգնում է կանխել շատ սխալներ՝ դեռ կոդ գրելու փուլում

✅ Ուժեղ տիպային համակարգ, որն ապահովում է կոդի հստակություն և կանխատեսելիություն

✅ Ավելի լավ IntelliSense և ավտոմատ լրացում ծրագրավորման միջավայրերում (IDE-ներում)

✅ Աջակցում է օբյեկտ-կողմնորոշված ծրագրավորման (OOP) գաղափարներ՝ կլասներ, ժառանգում, ինտերֆեյսներ *(Ստանդարտ JavaScript-ում կլասներն ու ժառանգումը ավելի թույլ էին, իսկ TypeScript-ը դրանք դարձնում է ավելի հզոր և մոտեցնում դասական OOP լեզուներին՝ ինչպիսիք են Java-ն և C#-ը:)*

✅ Հիանալի է մեծ թիմերում և մասշտաբային նախագծերում օգտագործելու համար *(TypeScript-ը թույլ է տալիս, որ տարբեր ծրագրավորողներ արագ հասկանան միմյանց կոդը և նվազեցնեն թյուրըմբռնումները, ինչը շատ կարևոր է թիմային աշխատանքում)*:

✅ Հեշտությամբ ինտեգրվում է բոլոր հայտնի JavaScript գրադարանների և գործիքների հետ

Տարբերություններ JavaScript-ի և TypeScript-ի միջև

1. Տիպային համակարգ

- **JavaScript**-ում փոփոխականների տիպերը սահմանված չեն: Կարող ես փոփոխականի մեջ ցանկացած տիպի արժեք գրել (թիվ, տեքստ, օբյեկտ և այլն), և սխալը կտեսնես միայն ծրագրի աշխատանքի ընթացքում:

- **TypeScript**-ում կարող ես նախապես հայտարարել փոփոխականի տիպը (օրինակ՝ `string`, `number`, `boolean`): Սա օգնում է վաղ հայտնաբերել սխալները՝ դեռ կոդ գրելու փուլում:

Օրինակն իրականացնենք երկու լեզուներով՝ JavaScript-ով և TypeScript-ով, որպեսզի տարբերությունները լինեն ավելի տեսանելի:

JavaScript-ում՝
Տվյալ դեպքում մեր կոդը կաշխատի առանց սխալի: Երկրորդ տողում փոփոխականին կտրամադրվի նոր արժեք՝ 5, նույնիսկ եթե սկզբում այն տեքստային արժեք էր ստացել: JavaScript-ը թույլ է տալիս փոփոխականների տիպը փոխել կատարողականի ընթացքում՝ առանց զգուշացումների կամ սխալների:

📜 **Նշում**. JavaScript-ը թույլ տիպավորված լեզու է, այսինքն՝ փոփոխականների տիպերը կարող են փոխվել ծրագրի աշխատանքային ընթացքում:

```
let name = "Աննա";
name = 5;
```

TypeScript-ում՝
Տվյալ դեպքում մեր կոդը չի աշխատի: Քանի որ փոփոխականը հայտարարված է որպես տիպ `string`, TypeScript-ը թույլ չի տա հետագայում նրան վերագրել թիվ: Սա կբերի սխալի դեռ կոդ գրելու պահին՝ ապահովելով կոդի ավելի բարձր հուսալիություն:

📜 **Նշում**. TypeScript-ը խիստ տիպավորված լեզու է, որտեղ փոփոխականների, ֆունկցիաների և օբյեկտների տիպերը հստակ սահմանված են և պահպանվում են ամբողջ ծրագրում:

```
let FirstName: string = "Anna";
FirstName = 8;
```

2. Նոր հնարավորություններ

JavaScript ծրագրավորման լեզվում ստանդարտորեն չկան որոշ առաջադեմ հնարավորություններ, ինչպիսիք են ինտերֆեյսները, էնյումերացիաները, ջեներիկները և կատարելագործված մոդուլային կառուցվածքը:

TypeScript-ը լրացնում է այս բացը՝ առաջարկելով. Տիպերի հայտարարում, Ինթերֆեյսներ (Interface), Էնյումերացիաներ (Enum), Ջեներիկներ (Generics), Մոդուլային կառուցվածքի կատարելագործում:

Ավելին, TypeScript-ը նաև ապահովում է ամբողջական օբյեկտ-կողմնորոշված ծրագրավորման (OOP) գործիքակազմ՝ ներառյալ կլասներ, ժառանգում և ինկապսուլյացիա:

3. Կոդի կատարում

JavaScript կոդը կարող է անմիջապես աշխատել բրաուզերում (օրինակ՝ Chrome, Firefox) կամ հատուկ ծրագրերում, ինչպիսին է Node.js-ը: Այսինքն՝ JavaScript ֆայլերը բացելով՝ դրանք միանգամից գործարկվում են:

Իսկ TypeScript կոդը չի կարող անմիջապես աշխատել: Նախ պետք է այն հատուկ ծրագրի միջոցով փոխել (կոմպիլացնել) սովորական JavaScript-ի: Այս փուլում `.ts` (TypeScript ֆայլ) փոխարկվում է `.js` (JavaScript ֆայլ):

Կոմպիլացիա - կոդի փոխակերպում մեկ լեզվից մյուսի՝ մեքենայի կամ համակարգչի կողմից ընկալելի լինելու համար: Այս գործընթացը անհրաժեշտ է, որ TypeScript-ը հասկանալի լինի JavaScript-ի տեսքով:

Node.js-ը JavaScript-ի համար հատուկ ծրագիր է, որը թույլ է տալիս գրել և աշխատեցնել JavaScript կոդը ոչ միայն ինտերնետային էջերում, այլ նաև համակարգչում: Node.js-ը շատ արդյունավետ է, քանի որ աշխատում է արագ և կարող է միաժամանակ շատ առաջադրանքներ կատարել: Այն հիմնականում օգտագործվում է վեբ հավելվածներ ստեղծելու և տվյալներ արագ մշակելու համար:

4. Սխալների հայտնաբերում

JavaScript-ում կոդը աշխատելու ընթացքում կարող են սխալներ առաջանալ, և դրանք հաճախ երևում են միայն ծրագրի աշխատանքին ժամանակ:

Իսկ TypeScript-ը կոդ գրելու պահին արդեն ստուգում է, արդյոք ամեն ինչ ճիշտ է արված: Այսպես՝ շատ սխալներ կարելի է նկատել ու շտկել դեռ կոդ գրելու ընթացքում:

Սխալների ստուգում կոդ գրելու ընթացքում նշանակում է, որ համակարգը օգնում է քեզ ավելի արագ գտնել խնդիրները, նախքան ծրագիրը կաշխատի:

5. Օգտագործման նպատակը

JavaScript-ը հարմար է, երբ ուզում ես արագ ստեղծել փոքր կամ միջին չափի նախագծեր: Այն տալիս է ազատություն, բայց նաև պետք է զգույշ լինել, քանի որ շատ բաներ համակարգը չի ստուգում:

TypeScript-ը ավելի հարմար է, երբ նախագիծը մեծ է, կամ երբ շատ մարդիկ են աշխատում միասին: Այն օգնում է, որ կոդը լինի հստակ, ճիշտ դասավորված և ավելի կանխատեսելի:

6. Սովորելու ընթացքը

Եթե արդեն գիտես JavaScript, ապա TypeScript սովորելն ավելի հեշտ կլինի, քանի որ այն շատ նման է, պարզապես ունի լրացուցիչ հնարավորություններ:

Եթե նոր ես սկսում ծրագրավորումը, խորհուրդ է տրվում նախ լավ հասկանալ JavaScript-ը, հետո անցնել TypeScript-ին: Այսպես կկարողանաս ավելի լավ հասկանալ՝ ինչ առավելություններ է տալիս TypeScript-ը:

TypeScript-ի տեղադրում (Windows և Linux)

TypeScript-ի օգտագործման համար նախ պետք է տեղադրեք Node.js, որը թույլ է տալիս աշխատեցնել JavaScript և TypeScript համակարգչում: Եթե արդեն տեղադրված է Node.js, կարող եք անմիջապես անցնել TypeScript-ի տեղադրման քայլերին:

1. Node.js-ի տեղադրում

Քայլ 1. Բացեք ձեր բրաուզերը և գնացեք Node.js-ի պաշտոնական կայք: Ներբեռնեք վերջին LTS (Long Term Support) տարբերակը:

Քայլ 2. Տեղադրեք ծրագրի ներբեռնված ֆայլը՝ հետևելով էկրանին ցուցադրվող հրահանգներին: Հետևաբար, համակարգում տեղադրվելու են Node.js և npm (Node Package Manager), որոնք անհրաժեշտ են TypeScript-ի տեղադրման համար:

2. TypeScript-ի տեղադրում

Windows-ի համար

Քայլ 1. Windows համակարգում բացեք CMD (Command Prompt) ծրագիրը և մուտքագրեք հետևյալ հրամանը՝

```
npm install -g typescript
```

Այս հրամանը կտեղադրի TypeScript-ը համակարգում:

Քայլ 2. Տեղադրման ավարտից հետո TypeScript-ի կոմպայլերը (tsc հրամանը) հասանելի կլինի համակարգում:

Linux-ի համար

Քայլ 1. Linux-ում բացեք թերմինալ և մուտքագրեք հետևյալ հրամանը՝

```
sudo npm install -g typescript
```

Նախքան տեղադրումը դուք կարող եք անհրաժեշտ լինել ստանալ ձեր համակարգի ադմինիստրատորի իրավունքները՝ մուտքագրելով ձեր գաղտնաբառը:

Քայլ 2. Տեղադրման ավարտից հետո TypeScript-ի կոմպայլերը (tsc հրամանը) հասանելի կլինի համակարգում:

3. TypeScript-ի տեղադրման ստուգում

Քայլ 1. Ստուգելու համար, արդյոք TypeScript-ը ճիշտ տեղադրված է, բացեք CMD (Windows) կամ Թերմինալ (Linux) և մուտքագրեք հետևյալ հրամանը՝

```
tsc --version
```

Եթե TypeScript-ը ճիշտ տեղադրվել է, պետք է տեսնեք նրա տարբերակը, օրինակ՝ Version 4.x.x:

Ստեղծել TypeScript պրոյեկտ

Որպեսզի ստեղծեք TypeScript-ի պրոյեկտ և սկսեք աշխատել ձեր կոդի վրա, պետք է հետևեք որոշ պարզ քայլերի: Այս բաժնում մանրամասն կներկայացնեմ, թե ինչպես պետք է ստեղծեք ֆայլեր, կոնֆիգուրացիա և կառուցեք TypeScript պրոյեկտ՝ օգտագործելով Node.js:

1. Պանակ(Folder) ստեղծելը

Քայլ 1. Նախ պետք է ստեղծեք նոր պանակ(folder), որտեղ կպահեք ձեր TypeScript պրոյեկտը:

Բացեք Terminal-ը կամ Command Prompt-ը (Windows-ի համար) և կատարեք հետևյալ հրամանը՝

```
mkdir my-typescript-project
cd my-typescript-project
```

Այս հրամանների միջոցով դուք կստեղծեք պանակ որին անվանում ենք my-typescript-project, կարող է լինել այլ անվանում, և երկրորդ հրամանի միջոցով մուտք եք գործում ձեր ստեղծած պանակի մեջ:

2. package.json ֆայլի ստեղծում

Ստեղծեք package.json ֆայլը՝ օգտագործելով npm-ի ինտեգրված հրամանը: package.json ֆայլը կարևոր է պրոյեկտի կարգաբերման համար և պարունակում է այնպիսի տվյալներ, ինչպիսիք են՝ պրոյեկտի կոնֆիգուրացիաները և օգտագործվող գրադարանների մասին տեղեկությունները:

Մուտքագրեք հետևյալ հրամանը.

```
npm init -y
```

Այս հրամանը ավտոմատ կերպով կստեղծի package.json ֆայլը ձեր պրոյեկտի մեջ՝ բոլոր անհրաժեշտ մուտքագրումներով:

3. TypeScript-ի կոնֆիգուրացիոն ֆայլի ստեղծում

Քայլ 3. Ստեղծեք tsconfig.json ֆայլը՝ TypeScript-ի կոնֆիգուրացիայի համար: Այս ֆայլը թույլ կտա վերահսկել TypeScript կոմպիլյացիան՝ նշելով, թե ինչքան թարմացում է պետք անել և որտեղ պետք է պահվեն կոմպիլացված JavaScript ֆայլերը:

Ստեղծեք tsconfig.json ֆայլը՝ մուտքագրելով հետևյալ հրամանը՝

```
tsc --init
```

Այս հրամանը կստեղծի tsconfig.json ֆայլը, որտեղ կկարողանաք փոփոխություններ կատարել, եթե անհրաժեշտ է:

4. TypeScript ֆայլի ստեղծում

Քայլ 4. Ստեղծեք ձեր առաջին TypeScript ֆայլը: Օրինակ՝ app.ts անունով ֆայլ, որտեղ կգրեք TypeScript կոդը:

Նոր ֆայլ ստեղծելու համար բացեք տեքստային խմբագրիչ (օրինակ՝ Visual Studio Code) և ստեղծեք app.ts ֆայլը կամ ձեր նախնտրած անվանումով .ts ֆայլ:

Կոդի օրինակ՝

```
let word: string = "Hello, TypeScript!"; // Ստեղծում ենք
փոփոխական string տեսակի որին վերագրում ենք Hello, TypeScript!
console.log(word); // Տպում ենք արդյունքը՝ Hello, TypeScript!
```

Տեքստային խմբագրիչը ծրագիր է, որով կարող ենք ստեղծել և փոփոխել տեքստային ֆայլեր: Այն հիմնականում օգտագործվում է ծրագրավորողների կողմից կոդ գրելու, պահպանելու և խմբագրելու համար: Օրինակներ են Visual Studio Code-ը, Sublime Text-ը և Notepad++-ը:

5. Կոդի կոմպիլացիա

Քայլ 5. Երբ TypeScript կոդը պատրաստ է, անհրաժեշտ է այն կոմպիլացնել JavaScript ֆայլի: Սա անելու համար կատարեք հետևյալ հրամանը թերմինալում՝ `tsc`

Այս հրամանը կկոմպիլացնի ձեր app.ts ֆայլը և ստեղծի app.js ֆայլը: Կոմպիլացիայի ընթացքում օգտագործվում է tsconfig.json ֆայլի կարգաբերման տվյալները:

Երբ .ts ֆայլը կոմպիլացվի .js ֆայլի, դուք .js ֆայլում կնկատեք հենց ֆայլի սկզբում գրված "use strict":

"use strict" որոշում է JavaScript-ի խստացված ռեժիմը (strict mode), որը նպատակ ունի անվտանգ դարձնել կոդը և բացահայտել հնարավոր սխալներ, որոնք սովորաբար թաքնվում են սովորական ռեժիմում:

6. Պրոյեկտի վերլուծություն և Node.js-ի միջոցով կատարում

Քայլ 6. Պարզապես օգտագործեք Node.js համակարգը՝ app.js ֆայլը կատարելու համար: Դա անել՝ օգտագործեք այս հրամանը թերմինալում՝

```
node app.js
```

Եթե ամեն ինչ ճիշտ է արված, պետք է տեսնեք հետևյալ արժեքը.

```
Hello, TypeScript!
```

7. Պրոյեկտի արտաքին ռեսուրսների և գրադարանների կառավարում

Քայլ 7. Եթե ձեր TypeScript պրոյեկտը պահանջում է այլ գործիքներ կամ հավելյալ հնարավորություններ, կարող եք դրանք ավելացնել npm-ի միջոցով: Օրինակ, եթե ցանկանում եք տեղադրել որոշակի գրադարան, օգտագործեք հետևյալ հրամանը.

```
npm install <library-name>
```

Այսպես, անհրաժեշտ ռեսուրսները կհայտնվեն `node_modules` պանակում: Ավելին, այդ ռեսուրսները կտեղադրվեն նաև `package.json` ֆայլում՝ դառնալով ձեր պրոյեկտի մասը:

Գրադարանն այնպիսի հավաքածու է, որը պարունակում է նախապես գրած կոդեր և գործիքներ, որոնք կարող են օգնել ծրագրավորողներին ավելի արագ կատարել որոշակի առաջադրանքներ:

Օրինակ, JavaScript-ի `sort()` ֆունկցիան կազմում է զանգվածի (array) տարրերը որոշակի կարգով: Այն կարող է օգտագործվել ցանկացած ժամանակ, երբ ցանկանում ենք դասավորել թվերը կամ տեքստերը աճման կամ նվազման կարգով:

Մաս 3 - Լեզվի հիմունքները

TypeScript լեզվի պատմություն

TypeScript-ը ստեղծվել է որպես լուծում այն խնդիրների, որոնք առաջացել էին JavaScript-ի ավելի լայն կիրառման պայմաններ ում: Երբ 2010-ականների սկզբին JavaScript-ը սկսեց օգտագործվել ոչ միայն պարզ ինտերակտիվության, այլ նաև բարդ ու խոշոր ծրագրերի ստեղծման համար, երևացին նրա սահմանափակումները՝ հատկապես տիպայնության բացակայությունն ու սխալների ուշ բացահայտումը: Այս պայմաններում Microsoft-ը սկսեց մշակել մի նոր լեզու, որը կհիմնվի JavaScript-ի վրա, բայց կլրացնի դրա թույլ կողմերը՝ ապահովելով ավելի կանխատեսելի և կառավարվող կոդ:

2012 թվականի հոկտեմբերին Microsoft-ը թողարկեց TypeScript 0.8 տարբերակը՝ ներկայացնելով այն որպես բաց կոդով ծրագրավորման լեզու: Այն նախագծվել էր որպես JavaScript-ի վերին շերտ, այսինքն՝ ցանկացած վավեր JavaScript կոդ նաև վավեր TypeScript է, ինչը նշանակում է, որ ծրագրավորողները կարող էին հեշտությամբ անցում կատարել նոր լեզվին՝ չվնասելով գործող նախագծերը:

TypeScript-ը սկզբնական շրջանում ընդունվեց համեմատաբար սահմանափակ շրջանակներում, սակայն նրա աճն արագացավ, երբ 2014-2016 թվականներին Angular ֆրեյմվորթի թիմը հայտարարեց, որ Angular 2-ը ամբողջությամբ գրվելու է TypeScript-ով: Այս քայլը ոչ միայն բարձրացրեց լեզվի հեղինակությունը, այլև խթանեց նրա կիրառումը լայն ծրագրավորման համայնքում: Հետագա տարիներին՝ 2016-ից սկսած, TypeScript-ը ստացավ մի շարք բարելավումներ, ինչպիսիք են մոդուլների աջակցությունը, async/await ֆունկցիոնալությունը և խստացված տիպային ստուգումները, որոնք էլ ավելի հզորացրին լեզուն:

2020-ից սկսած TypeScript-ը արդեն դիտվում էր որպես առաջատար լուծում՝ JavaScript-ի մեծ նախագծերում օգտագործելու համար: Այն լիովին ինտեգրվեց ժամանակակից գործիքակազմի և ֆրեյմվորթների մեջ՝ ներառյալ VS Code խմբագրիչը, React և NestJS գրադարանները: Այսօր TypeScript-ը դարձել է շատ ծրագրավորողների նախընտրելի լեզուն՝ շնորհիվ իր կանխատեսելիության, լավ IDE աջակցման և մեծ նախագծերի կառավարման հարմարությունների:

Այսօր TypeScript լեզուն օգտագործվում է բազմաթիվ հայտնի ծրագրերում, որոնք մարդիկ օգտագործում են ամեն օր: Օրինակ՝ Microsoft-ը հենց այս լեզվով է գրել իր հայտնի կոդ գրելու ծրագիրը՝ Visual Studio Code-ը: Նույնը արել են նաև Google-ը, երբ ստեղծել է Angular անունով գործիքը, որի օգնությամբ պատրաստում են կայքեր: TypeScript են օգտագործում նաև մեծ ընկերություններ, ինչպես օրինակ Slack-ը, GitHub-ը, Airbnb-ն և շատ ուրիշներ:

Փոփոխականներ

Ի՞նչ է փոփոխականը

Ծրագրավորման մեջ փոփոխականը նման է մի փոքրիկ տուփի, որի մեջ մենք պահում ենք ինչ-որ արժեք: Այդ արժեքը կարող է լինել թիվ, տեքստ կամ որևէ այլ բան: Ամեն անգամ, երբ մենք ուզում ենք ծրագրում ինչ-որ բան պահել և հետո օգտագործել՝ դրա համար ստեղծում ենք փոփոխական:

Պատկերացրու մի պահարան՝ բազմաթիվ դարակներով: Յուրաքանչյուր դարակ ունի իր անունը, և դու այնտեղ կարող ես դնել տվյալ: Հետո, երբ քեզ պետք է տվյալը, պարզապես ասում ես դարակի անունը, ու տվյալը ստանում ես: Սա է փոփոխականի հիմնական գաղափարը:

Ի՞նչպես ստեղծել փոփոխական TypeScript ում

TypeScript-ում (ինչպես JavaScript-ում) փոփոխական ստեղծելու համար օգտագործում ենք հետևյալ հրամանները let, const, var

Օրինակ `

`let` – երբ արժեքը կարող է փոխվել.

```
let age : Number = 17; // Ստեղծում ենք փոփոխական 'age' անունով և վերագրում ենք նրան 17 արժեքը, նշելով որ իր արժեքը թիվ է
age = 18; // Փոփոխում ենք 'age' փոփոխականի արժեքը՝ դարձնելով 18
console.log(age); // Տպում ենք արդյունքը՝ 18
```

let-ով ստեղծված փոփոխականը կարող է փոխվել հետո: Այս դեպքում մենք սկզբում տալիս ենք 17, հետո փոխում ենք 18-ով:

`const` – երբ արժեքը հաստատուն է և չի փոխվելու.

```
const year = 2007; // Ստեղծում ենք 'year' անունով փոփոխական, որը հաստատուն է՝ արժեքը 2007
// year = 2008  Սա սխալ է, որովհետև const փոփոխականի արժեքը հնարավոր չէ փոխել
console.log(year); // Տպում ենք արդյունքը՝ 2007
```

Եթե վստահ ենք, որ որևէ արժեք ծրագրի ընթացքում պետք է մնա անփոփոխ, ապա պետք է օգտագործենք const հրամանը:

Սա ոչ միայն ցույց է տալիս, որ արժեքը հաստատուն է, այլ նաև պաշտպանում է տվյալը պատահական կամ չնախատեսված փոփոխությունից, Օրինակ՝ մարդու ծննդյան տարեթիվը կայուն տվյալ է, քանի որ այն միանշանակ է և ծրագրի ընթացքում չի կարող փոխվել:

`var` – նախընտրելի չէ օգտագործել՝ ունի հնացած վարք

```
var name : String = "Աշոտ"; // Ստեղծում ենք 'name' փոփոխականը 'Աշոտ' արժեքով՝ օգտագործելով հնացած 'var'
name= "Արամ"; // Փոփոխում ենք փոփոխականի արժեքը
console.log(name); // Տպում ենք արդյունքը՝ Արամ
```

Չնայած `var`-ը թույլ է տալիս փոփոխել փոփոխականի արժեքը, այն ունի հնացած վարք: Մասնավորապես՝ այն չի պահպանում բլոկի սահմանները, ինչը կարող է հանգեցնել սխալների: Այս հասկացությունը՝ «բլոկային սահմաններ», մենք կքննարկենք առաջիկա գլուխներում:

Հետևաբար, փոփոխականներ ստեղծելու համար այսուհետ կօգտագործենք միայն let և const հրամանները՝ որպես ժամանակակից և անվտանգ տարբերակներ:

alert, prompt, console

Ծրագրեր ստեղծելիս երբեմն ցանկանում ենք օգտատիրոջը ցույց տալ հաղորդագրություն կամ հարցնել ինչ որ բան: TypeScript-ը, ինչպես JavaScript-ը, տրամադրում է երկու հիմնական ֆունկցիա, որոնք մեզ օգնում են այդ գործում, alert(), prompt() :

Այս երկու ֆունկցիաները պատկանում են բրաուզերի հիմնական օբյեկտին՝ window-ին: Այդ պատճառով էլ մենք կարող ենք օգտագործել դրանք առանց որևէ գրադարան ներմուծելու:

> Նշում
> Այս ֆունկցիաները հասանելի են միայն բրաուզերում, որովհետև միայն բրաուզերը ունի «պատուհանի» միջավայր (window), որտեղ հնարավոր է ցուցադրել հաղորդագրություններ:
> Եթե աշխատում եք սերվերային միջավայրում՝ օրինակ Node.js-ում, ապա այդ ֆունկցիաները չեն աշխատի:

`alert()` ֆունկցիան օգտագործվում է՝ օգտատիրոջը պարզ տեղեկություն ցուցադրելու համար. Օրինակ՝ alert("Բարև աշխարհ"):

Եթե ուզում ենք օգտատիրոջից վերցնել արժեք (օրինակ՝ անունը), կարող ենք օգտագործել prompt():

```
let name = prompt("Ի՞նչ է քո անունը:"); // Ստեղծում ենք փոփոխական
```

```
name անունով որին վերագրում ենք prompt ֆունկցիան որը օգտատերից
հարցնում է "Ի՞նչ է քո անունը:"
alert(name); // Այստեղ արդյունքը կախված է օգտատերի պատասխանից
```

Այստեղ առաջին տողում մենք ստեղծում ենք name անունով փոփոխական և նրան վերագրում ենք prompt() ֆունկցիայի վերադարձած արժեքը: Այս ֆունկցիան բացում է մի պատուհան՝ օգտատիրոջից ստանալու համար անունը: Օգտատերը մուտքագրում է իր անունը, և այդ արժեքը պահվում է name փոփոխականի մեջ:

Երկրորդ տողում օգտագործում ենք alert() ֆունկցիան՝ ստացված անունը ցուցադրելու համար: Այսպիսով, օգտատերը տեսնում է իր մուտքագրած անունը՝ հաղորդագրության պատուհանում:

Node.js-ում տպելու համար սովորաբար օգտագործվում է `console.log()` ֆունկցիան, որը տպում է տվյալը կոնսոլում:

Իսկ ի՞նչ է իրականում console.log()-ը:

console սա JavaScript-ի կողմից տրամադրված ներկառուցված օբյեկտ է, որը նախատեսված է տերմինալում կամ բրաուզերի կոնսոլում տարբեր տեղեկություններ ցուցադրելու համար: Այս օբյեկտը պարունակում է մի շարք մեթոդներ, որոնց միջոցով կարող ենք տպել տարբեր տվյալներ, ստանալ սխալներ, զգուշացումներ և այլն: log() սա console օբյեկտի մեթոդներից (ֆունկցիաներից) մեկն է: Այն բառացի նշանակում է՝ "գրանցել", այսինքն՝ ինչ-որ տվյալ ուղարկել և ցույց տալ կոնսոլում: Երբ գրում ենք console.log(...), մենք ասում ենք՝ "դա գրի կոնսոլում":

Այսպիսով, ամբողջ console.log("Բարև աշխարհ") արտահայտությունը նշանակում է. Տպի "Բարև աշխարհ" տերմինալում:

Բացի log() մեթոդից, console օբյեկտը ունի նաև այլ օգտակար մեթոդներ.

- ➢ console.error() – Տպում է սխալ հաղորդագրություն:
- ➢ console.warn() – Տպում է զգուշացում:
- ➢ console.info() – Տպում է ինֆորմացիա:

Հաղորդագրություններ(Comments)

Հաղորդագրությունները կոդում գրվում են ծրագրավորողների համար՝ բացատրելու, թե ինչ է կատարվում և ինչու: Դրանք հատկապես կարևոր են, երբ աշխատում ես թիմում, փոխանցում ես նախագիծը ուրիշին կամ վերադառնում ես կոդին երկար ժամանակ անց: Դրանք օգնում են հասկանալու մտածելակերպը, որոշումների հիմքը և կանխում են սխալ մեկնաբանությունները:

TypeScript-ում comment ստեղծելու երկու ձև կա.

- ➢ // — single-line comment, որը գրում ես մի տողի մեջ:

- /* ... */ — multi-line comment, որը նախատեսված է ավելի երկար բացատրությունների համար:

Տվյալների տեսակները (Data Types)

Ծրագրավորման հիմնարար և ամենակարևոր գաղափարներից մեկն այն է, որ յուրաքանչյուր արժեք ունի իր հստակ սահմանված տեսակը, այսինքն՝ տիպը: Տվյալների տիպերը որոշում են, թե տվյալ արժեքը ինչ բնույթ ունի՝ ամբողջ թիվ է, տեքստ, ցուցակ, թե որևէ այլ բան: Սրանով պայմանավորված՝ համակարգը կարողանում է հասկանալ, թե ինչպես վարվել տվյալ արժեքի հետ, ինչպես այն պահել հիշողության մեջ, ինչպես վերամշակել այն, և ինչպիսի գործողություններ կարելի է կամ չի կարելի կատարել այդ արժեքի վրա:

Ի տարբերություն JavaScript լեզվի, որտեղ տվյալների տիպերի ստուգումը հիմնականում կատարվում է միայն կատարման պահին (runtime), TypeScript-ը տրամադրում է խիստ, կառուցվածքային և կանխատեսելի տիպային համակարգ: Սա նշանակում է, որ TypeScript-ը ծրագրավորողին հնարավորություն է տալիս արդեն ծրագրի գրման փուլում (compile time) ստուգել՝ արդյոք արժեքները համապատասխանո՞ւմ են իրենց սահմանված տիպերին: Եթե ծրագիրը պարունակում է որևէ տիպային անհամապատասխանություն՝ օրինակ, փորձ է արվում տեքստային տիպով արժեք վերագրել փոփոխականի, որը նախատեսված է միայն թվային արժեքների համար, ապա TypeScript-ը դա կբացահայտի և կմատնանշի որպես սխալ:

Տեսակի նշանակությունը

Երբ մտածում ենք փոփոխականի մասին, կարելի է պատկերացնել այն որպես փոքրիկ տուփ՝ համակարգի հիշողության մեջ: Այդ տուփի վրա ամրացված է պիտակ, որն ասում է, թե այդ տուփում ինչ տեսակի արժեք կարող ենք պահել: Եթե տուփի վրա գրված է "թիվ", ապա այնտեղ կարող ենք դնել միայն թվային տվյալներ: Եթե փորձենք այնտեղ տեղադրել տեքստ՝ օրինակ "Բարև" բառը, TypeScript-ը անմիջապես մեզ կզգուշացնի, որ փորձ ենք անում սխալ տվյալ դնել սխալ տուփի մեջ:

Այսպիսի խիստ տիպային հսկողությունը օգնում է մեզ գրել ավելի կանխատեսելի, կայուն և առանց անսպասելի սխալների ծրագիր: Այն նաև մեծ օգուտ է բերում թիմային աշխատանքի ժամանակ՝ երբ մի քանի ծրագրավորող աշխատում են նույն նախագծի վրա և անհրաժեշտ է, որ բոլորն ունենան հստակ պատկերացում՝ յուրաքանչյուր տվյալ ինչ տիպի է:

Հիմնական տվյալների տեսակներ

TypeScript-ում գոյություն ունեն մի շարք հիմնական (primitive) տվյալների տիպեր: Ստորև կանդրադառնանք դրանցից երկուսին՝ `number` և `string` տիպերին, որոնք ամենից հաճախ օգտագործվողներից են:

Number տեսակը

`Number`-ը այն տիպն է, որը ներկայացնում է բոլոր տեսակի թվերը՝ ամբողջ թվեր (integer), տասնորդականներ (floating), բացասական և դրական արժեքներ: TypeScript-ում բոլոր թվերը դիտարկվում են որպես `number`, և առանձին տիպեր, օրինակ՝ `int` կամ `float`, չեն օգտագործվում՝ ինչպես որոշ այլ լեզուներում:

Օրինակ՝

```
let age: number = 17;
let price: number = 2499.99;
let temperature: number = -5;
```

Այս օրինակում մենք սահմանել ենք երեք փոփոխական՝ բոլորն էլ `number` տիպով: Եթե փորձենք դրանցից մեկին վերագրել տեքստային արժեք, TypeScript-ը սխալ կցուցադրի.

```
age = "տասնյոթ"; // Սխալ. սպասվում է թիվ, բայց տրված է տեքստ
```

String տեսակը

String-ը ներկայացնում է տեքստային տվյալներ: Այս տիպով փոփոխականները պահում են նիշերի շարաններ (character sequences), որոնք կարող են պարունակել բառեր, նախադասություններ, տառեր կամ ցանկացած այլ տեքստային բովանդակություն:

Տեքստերը սահմանվում են ', ", կամ ` (backtick) նշանների մեջ: Backtick-ով գրվածները կոչվում են template literals և հնարավորություն են տալիս ինտերպոլացնել փոփոխականներ տեքստում:

Օրինակ՝

```
let name: string = "Անի";
let greeting: string = 'Բարև աշխարհ';
let info: string = `Իմ անունն է ${name}`;
```

Այս օրինակում մենք օգտագործել ենք տարբեր եղանակներով string-ներ ստեղծելու մեթոդներ: `info` փոփոխականը ցույց է տալիս, թե ինչպես կարող ենք ներառել ուրիշ փոփոխականների արժեքները միացման միջոցով (interpolation):

Եթե փորձենք տեքստային փոփոխականին վերագրել թիվ, կրկին TypeScript-ը կզգուշացնի.

```
name = 123; // Սխալ. սպասվում է տեքստային արժեք, բայց տրված է թիվ
```

Այսպիսով, տվյալների տիպերը TypeScript-ի մեջ ոչ միայն կարգավորում են ծրագրի կառուցվածքը, այլ նաև օգնում են խուսափել բարդ սխալներից՝ ծրագրի զարգացումն ավելի հուսալի և վստահելի դարձնելով: Հետևելով տիպային համակարգի կանոններին՝ մենք ստանում ենք ավելի կարդալի, պահպանելի և տեխնիկապես ճշգրիտ կոդ:

Հաջորդ գլուխներում մենք կծանոթանանք նաև մյուս հիմնական տիպերին, կոմպլեքս տիպերին, և TypeScript-ի առաջադեմ տիպային առանձնահատկություններին:

Պայմանական օպերատորներ

Երբ մարդը որոշում է կայացնում, նա մտածում է պայմանների մասին: Եթե հագուստը մաքուր է, հագնում է այն: Եթե եղանակը լավ է, կարող է դուրս գալ զբոսնելու: Եթե ժամը ուշ է, գուցե գնա քնելու: Այսօր՝ համակարգիչը նույնպես որոշումներ է կայացնում հենց այս տրամաբանությամբ:

Ծրագրավորման մեջ այսպիսի պայմաններ գրելու համար մենք օգտագործում ենք պայմանական օպերատորներ:

Ինչ է պայմանը

Պայմանը արտահայտություն է, որը ստուգում է մի բան՝ ճիշտ է, թե ոչ: Եթե ճիշտ է (այսինքն՝ `true`), համակարգիչը կատարում է որոշ գործողություն: Եթե սխալ է (`false`), այն բաց է թողնում այդ հատվածը կամ կատարում այլ բան:

Պատկերացրու, դու ծրագրում ես ռոբոտ, որը պետք է որոշի՝ կարելի՞ է դուրս գալ բակ: Պայմանը կարող է լինել՝ «Եթե արևոտ է, դուրս արի»: Ահա սա է տրամաբանությունը:

`If` – եթե

Ծրագրում «եթե» գրելու համար օգտագործում ենք `if` բառը:

```
if (պայման) {
  // այս կոդը կկատարվի, եթե պայմանը ճիշտ է
}
```

Օրինակ առօրյա կյանքից.

```
let weather = "արևոտ";

if (weather === "արևոտ") {
  alert("Վերցրու ակնոցն ու դուրս արի զբոսնելու:");
}
```

```
// Համեմատում ենք weather փոփոխականը "արևոտ" բառի հետ, եթե պայմանը ճիշտ է ապա ծրագիրը կտպի 'Վերցրու ակնոցն ու դուրս արի զբոսնելու:' եթե պայմանը ճիշտ չլինի ապա ծրագիրը ոչինչ չի անի:
```

Եթե եղանակը իսկապես "արևոտ" է, համակարգը ցույց կտա հաղորդագրությունը: Եթե եղանակը մռայլ է՝ ոչինչ չի պատահի:

`If ... else` – եթե ... ապա այլապես

Երբ ուզում ենք երկու տարբեր իրավիճակներ նկարագրել՝ ճիշտ և սխալ դեպքում, ավելացնում ենք `else`՝ որը նշանակում է «այլապես»:

```
let isHungry = true;

if (isHungry) {
  alert("Ժամանակն է ուտելու:");
} else {
  alert("Դեռ կսպասենք:");
}
```

Այս օրինակում ծրագիրը ստուգում է՝ սովա՞ծ է օգտատերը: Եթե այո՝ առաջարկում է ու

`If ... else if ... else` – մեկից ավելի պայման

Երբ ունենք ոչ միայն երկու, այլ մի քանի հնարավոր իրավիճակ, կարող ենք ստուգել դրանք հերթով՝ օգտագործելով `else if`:

```
let temperature = 30;

if (temperature > 35) {
  alert("Շատ տաք է:");
} else if (temperature >= 20) {
  alert("Լավ եղանակ է:");
} else {
  alert("Սրսում ենք:");
}
```

Սա նման է մեր առօրյայում կայացվող որոշումներին. եթե շոգ է՝ ցրվում ենք, եթե լավ է՝ դուրս ենք գալիս, եթե ցուրտ է՝ տուն ենք մնում:

Եռակի օպերատոր (Template literals ? :)

Երբ մեր ընտրությունը շատ պարզ է՝ երկու արժեքից մեկն ընտրել, կարող ենք գրել դա մեկ տողով:

```
let age = 17;
let message = age >= 18 ? "Դուք չափահաս եք:" : "Դուք դեռ անչափահաս եք:";
```

Այստեղ մենք ասում ենք. եթե տարիքը մեծ է կամ հավասար 18, տեքստը կլինի առաջին տարբերակը: Հակառակ դեպքում՝ երկրորդը:

Սա նույնն է, ինչ գրել `if...else`, ուղղակի ավելի կարճ և օգտագործվում է այն ժամանակ, երբ մենք ուզում ենք որոշում վերադարձնել որպես արժեք:

`Switch` – `Case`

Երբ ունենք մի փոփոխական, որն ունի մի քանի հնարավոր արժեք, և մենք ուզում ենք յուրաքանչյուրի դեպքում տարբեր բան անել, կարելի է օգտագործել `switch` կառուցվածքը: Սա փոխարինում է շատ `if...else if`-երին, երբ մենք միայն համեմատում ենք նույն արժեքը:

```
let day = "ուրբաթ";

switch (day) {
  case "երկուշաբթի":
    alert("Շաբաթվա սկիզբն է:");
    break;
  case "ուրբաթ":
    alert("Աշխատանքային շաբաթն ավարտվում է:");
    break;
  case "շաբաթ":
  case "կիրակի":
    alert("Հանգստյան օրեր:");
    break;
  default:
    alert("Չճանաչված օր:");
}
```

Այս կառուցվածքը նման է դպրոցական օրակարգային ռեժիմի. եթե օրն է երկուշաբթի՝ երգում ենք այս երգը, եթե ուրբաթ՝ պատրաստվում ենք հանգստին, եթե հանգստյան օր է՝ խաղում ենք: Իսկ եթե օրը մեզ անծանոթ է՝ զգուշացնում ենք:

Այս ամենը միասին օգնում է մեզ ծրագրում որոշումներ կայացնել՝ ինչպես մենք ենք անում առօրյայում: Հաջորդում արդեն կտեսնենք՝ ինչպես են այս պայմանները գնահատվում `true` կամ `false` արժեքով, այսինքն՝ կխոսենք Boolean տիպի մասին:

Ինչ է Boolean-ը

Boolean տիպը պահում է ընդամենը երկու արժեքից մեկը՝

- `true` – ճիշտ է
- `false` – սխալ է

TypeScript-ում երբ սահմանում ես փոփոխական, կարող ես հատուկ նշել, որ այն բուլյան է.

```
let isSunny: boolean = true;
let isNight: boolean = false;
```

Բուլյան արժեքները հատկապես կարևոր են պայմանական օպերատորների հետ աշխատելիս, քանի որ հենց այդ արժեքներն են որոշում՝ կաշխատի՞ պայմանը, թե ոչ:

```
if (isSunny) {
  alert("Վերցրու ակնոցդ ու դուրս արի:");
}
```

Եթե `isSunny` փոփոխականը ունի `true` արժեք, ապա հաղորդագրությունը կցուցադրվի: Եթե այն լինի `false`, ծրագիրը այդ հատվածը պարզապես կանցնի:

Ինչպե՞ս են ստացվում Boolean արժեքներ

Բուլյան արժեքները մենք կարող ենք անմիջապես գրել (օրինակ՝ `true`), բայց իրական ծրագրերում դրանք հաճախ ստացվում են համեմատություններից:

```
let age = 20;
let isAdult = age >= 18; // true
```

Այստեղ `age >= 18` արտահայտությունը վերադարձնում է բուլյան արժեք՝ արդյոք դա ճիշտ է, թե ոչ: Եթե `age` մեծ է կամ հավասար 18, ապա ստացվում է `true`, հակառակ դեպքում՝ `false`:

Այսպիսով, ցանկացած նման համեմատություն՝

- `>` (մեծ է)
- `<` (փոքր է)
- `>=` (մեծ կամ հավասար է)
- `<=` (փոքր կամ հավասար է)
- `===` (հավասար է արժեքով և տիպով)
- `!==` (հակառակ՝ հավասար չէ)

վերադարձնում է հենց Boolean արժեք:

Օրինակ առօրյա կյանքից

Պատկերացրու, որ դու ստեղծում ես խելացի տնային համակարգ, որը պետք է որոշի՝ միացնել լույսերը:
Դու գրում ես հետևյալ տրամաբանությունը՝

Եթե լույս չկա և մարդ կա սենյակում, լույսերը պետք է միանան:

Սա կարելի է գրել այսպես.

```
let isDark = true;
let isSomeoneInRoom = true;

let shouldTurnOnLights = isDark && isSomeoneInRoom; // true
```

Երբ ծրագրի մեջ մենք ունենք երկու կամ ավելի պայման, կարող ենք դրանք միավորել՝ օգտագործելով տրամաբանական օպերատորներ: Դրանք սովորաբար կիրառվում են բուլյան արժեքների հետ:

Տրամաբանական օպերատորներ

TypeScript-ում գոյություն ունեն երեք հիմնական տրամաբանական օպերատորներ՝

- && – «և»
  Վերադարձնում է `true` միայն այն դեպքում, երբ երկու կողմերն էլ `true` են:
- || – «կամ»
  Վերադարձնում է `true`, եթե գոնե մեկը `true` է:
- ! – «ոչ»
  Փոխում է արժեքը՝ `true` → `false`, `false` → `true`

Օրինակ՝

```
let isLoggedIn = true;
let isAdmin = false;

let canAccessSettings = isLoggedIn && isAdmin; // false
```

Այս տողը նշանակում է՝ օգտվողը կարող է մուտք գործել կարգավորումներ միայն այն դեպքում, եթե միաժամանակ մուտք է գործել և ադմինիստրատոր է:

Երբ այլ տիպեր վերածվում են Boolean-ի

TypeScript-ը (և JavaScript-ը) թույլ է տալիս որ որոշ արժեքներ ավտոմատ վերածվեն բուլյանի, երբ օգտագործվում են `if`-ի կամ տրամաբանական օպերատորի մեջ:

Ճշմարիտ համարվող արժեքներ (`true`)

- Ցանկացած չդատարկ տեքստ՝ `"hello"`
- Ցանկացած թիվ բացի 0-ից՝ `5`, `-3`
- Ցանկացած ոչ դատարկ array կամ object

Սխալ համարվող արժեքներ (`false`)

- `0`
- `""` (դատարկ տեքստ)
- `null`
- `undefined`
- `NaN`
- `false` (ինքն իրեն)

```
if ("hello") {
  console.log("Այս տողը կաշխատի, որովհետև 'hello' համարվում է
true");
}
```

Այս գլուխում մենք սովորեցինք, որ Boolean արժեքները հիմնարար դեր ունեն ծրագրի ընթացքը կառավարելու մեջ: Դրանք նման են լույսի անջատիչի՝ կա՛մ միացված են, կա՛մ անջատված: Ու հենց դրանց միջոցով է համակարգիչը որոշում՝ որ ուղղությամբ գնալ:

Ցիկլեր

Ծրագրավորման մեջ շատ հաճախ անհրաժեշտ է միևնույն գործողությունը կրկնել մի քանի անգամ: Այդ գործողությունները կարող են նույնական լինել, բայց տվյալները՝ տարբեր: Օրինակ՝ պատկերացրու, որ պետք է ցուցադրել բոլոր ուսանողների անունները կամ հաշվել բոլոր ամիսների միջին ջերմաստիճանը: Եթե մենք փորձենք այդ ամենը անել առանձին տողերով, ծրագիրը կդառնա երկար, միապաղաղ և խոցելի սխալների համար:

Բարեբախտաբար, ծրագրավորման լեզուները տալիս են հատուկ գործիք՝ **ցիկլեր**, որոնք թույլ են տալիս կոդը կրկնել ավտոմատ ձևով՝ առանց մեկ առ մեկ գրելու:

Ի՞նչ է ցիկլը

Ցիկլը ծրագրին տալիս է հրահանգ՝ «Մինչև այս պայմանը ճիշտ է, կրկնի այս գործողությունը»

Սա նման է այն բանին, երբ ասում ես. "Ամեն օր, քանի դեռ չի անցել շաբաթը, ես գնալու եմ մարզասրահ:" Ահա դա հենց ցիկլային մտածողություն է:

TypeScript-ում կան մի քանի տեսակի ցիկլեր: Այս գլխում կծանոթանանք երեք հիմնական տեսակներին՝

- `while`
- `do...while`
- `for`

While ցիկլ

Այս ցիկլն աշխատում է այնքան ժամանակ, քանի դեռ պայմանը ճիշտ է:

Կառուցվածք՝

```
while (պայման) {
  // այս կոդը կկատարվի, քանի դեռ պայմանը true է
}
```

Օրինակ՝

Պատկերացրու, դու ունես 5 բամպեր գնդակ, և ամեն անգամ ուզում ես մեկ հատ գցել տուփի մեջ, մինչև բոլորը չվերջանան:

```
let balls = 5;
while (balls > 0) {
  console.log("Մի գնդակ տուփի մեջ");
  balls = balls - 1;
}
```

Այս ծրագիրը կգրի 5 անգամ հաղորդագրությունը: Յուրաքանչյուր կրկնությունից հետո մեկ գնդակ պակասում է: Երբ `balls` դառնում է 0, ցիկլը կանգ է առնում:

`Do...while` ցիկլ

Այս կառուցվածքը շատ նման է `while`-ին, սակայն տարբերությունն այն է, որ այն գոնե մեկ անգամ միշտ կաշխատի, նույնիսկ եթե պայմանը սխալ է:

```
let attempt = 1;
do {
```

```
  console.log("Փորձ՝", attempt);
  attempt++;
} while (attempt <= 3);
```

Սա կկատարի երեք փորձ, քանի որ պայմանը մինչև 3 է: Բայց եթե even if attempt սկզբում լիներ 5, առաջին փորձը կկատարվեր, և հետո նոր պայմանը կստուգվեր:

For ցիկլ

Այս ցիկլը առավել հարմար է այն դեպքերում, երբ մենք գիտենք, թե քանի անգամ պետք է կրկնել գործողությունը: Այն իր մեջ ներառում է՝

- սկզբնական արժեք
- պայման
- քայլ (ինչքանով աճի կամ պակասի փոփոխականը)

Կառուցվածք՝

```
for (սկիզբ; պայման; քայլ) {
  // կոդ
}
```

Օրինակ՝

Տպել թվերը 1-ից 5:

```
for (let i = 1; i <= 5; i++) {
  console.log("Թիվ", i);
}
```

Այստեղ՝

- `i = 1` նշանակում է՝ սկսում ենք 1-ից
- `i <= 5` պայմանն ասում է՝ ցիկլը աշխատի մինչև i-ն <= 5 է
- `i++` նշանակում է՝ ամեն անգամ i-ին ավելացրու 1

Այս օրինակն ամբողջովին նույն արդյունքն է տալիս, ինչ `while` ցիկլը, բայց գրառման ձևը ավելի կոկիկ է, երբ մենք աշխատում ենք որոշակի քանակությամբ կրկնությունների հետ:

Ցիկլը կյանքում

Ցիկլերը պարզապես մեքենայական կրկնություն չեն: Դրանք թույլ են տալիս ծրագրերին արձագանքել տվյալների հավաքածուներին՝ օրինակ՝ օգտվողների ցանկերին, ապրանքների ցուցակներին կամ խաղի ընթացքի փուլերին:

Օրինակ՝

> «Մինչև չընթերցեմ գրքի բոլոր գլուխները, չեմ հանգստանա» – սա մարդու ուղեղի ցիկլ է:

Ծրագրավորման մեջ դու գրում ես՝

```
let chapter = 1;
let totalChapters = 10;
while (chapter <= totalChapters) {
  console.log(chapter);
  chapter++;
}
```

Այսպիսի մտածողությամբ կարելի է ծրագրին պատվիրել՝
«մինչև պայմանը ճիշտ է՝ աշխատիր», և երբ այլևս պայմանը սխալ է, կանգ առ:

Ցիկլերը ծրագրին տալիս են կրկնողության ուժ: Հաջորդ գլուխներում մենք կտեսնենք, թե ինչպես են դրանք համատեղվում զանգվածների և գործառույթների (functions) հետ:

Զանգվածներ (Arrays)

Երբ աշխատում ենք միայն մեկ արժեքով, ծրագրավորումն հեշտ է: Բայց իրական ծրագրերում մենք հազվադեպ ենք բավարարվում մեկ տարրով: Պատկերացրու, դու ունես ոչ թե մեկ ընկեր, այլ տասը: Կամ՝ պետք է պահես ոչ թե մեկ թիվ, այլ քննական մի ամբողջ արդյունքների շարք:

Այդպիսի դեպքերում մենք չենք ստեղծում տասը տարբեր փոփոխական: Փոխարենը օգտագործում ենք զանգվածներ:

Ի՞նչ է զանգվածը

Զանգվածը (array) տվյալների տեսակ է, որը կարող է պահել բազմաթիվ արժեքներ մեկ փոփոխականի մեջ: Դու կարող ես պատկերացնել զանգվածը որպես մի շարք տարրերից բաղկացած դարակ՝

📦 → `["Ալի", "Նարե", "Արտակ"]`

Այստեղ մենք ունենք երեք անուն, և բոլորը գտնվում են նույն դարակում՝ միայն տարբեր խցերում:

Ինչպես ստեղծել զանգված

TypeScript-ում զանգված ստեղծելու ամենահաճախ օգտագործվող ձևը հետևյալն է՝

```
let names = ["Անի", "Նարե", "Արտակ"];
```

Այստեղ `names` փոփոխականը պահում է երեք տեքստային արժեք:

Այլ օրինակ՝ թվերի զանգված.

```
let scores = [89, 75, 92, 100];
```

Զանգվածի արժեքներին հասանելիություն

Զանգվածի մեջ ամեն արժեք ունի իր **հերթական համարը**, որը կոչվում է **ինդեքս**: Ուշադրություն՝ ինդեքսավորումը սկսվում է 0-ից:

```
let fruits = ["խնձոր", "ուղտի կտավ", "սերկևիլ"];
console.log(fruits[0]); // "խնձոր"
console.log(fruits[1]); // "ուղտի կտավ"
```

Այսինքն՝ `fruits[0]`–ը առաջին տարրն է, ոչ թե զրոյական:

Էլեմենտների քանակը՝ `length`

Զանգվածը ունի մի հատկություն, որը կոչվում է `length`: Այն վերադարձնում է զանգվածում առկա տարրերի քանակը:

```
let colors = ["կարմիր", "կանաչ", "կապույտ"];
console.log(colors.length); // 3
```

Սա շատ օգտակար է, երբ ուզում ենք իմանալ՝ քանի անգամ կրկնել գործողություն: Օրինակ՝ երբ աշխատում ենք ցիկլով:

Զանգված և ցիկլ

Հաճախ մենք ուզում ենք անցնել զանգվածի բոլոր տարրերով և յուրաքանչյուրի հետ ինչ-որ բան անել: Դա կատարվում է սովորաբար `for` ցիկլով:

```
let cities = ["Գորիս", "Երևան", "Վանաձոր"];
for (let i = 0; i < cities.length; i++) {
```

```
  console.log("Քաղաք՝", cities[i]);
}
```

Այս կոդը կտպի բոլոր քաղաքները՝ հերթով: Ցիկլը սկսում է `i = 0`-ից և աշխատում է այնքան ժամանակ, քանի դեռ `i < cities.length` է: Յուրաքանչյուր փուլում մենք հասնում ենք զանգվածի մեկ տարրին:

Օրինակ առօրյա կյանքից

Պատկերացրու, որ դու ստեղծում ես դասարանի ցածր գնահատականների ցուցակ: Դու պահում ես բոլոր գնահատականները զանգվածում՝

```
let marks = [6, 9, 8, 5, 7];
```

Դու կարող ես գրել ծրագիր, որը գտնում է բոլոր այն գնահատականները, որոնք 6-ից ցածր են և զգուշացնում է:

```
for (let i = 0; i < marks.length; i++) {
  if (marks[i] < 6) {
    console.log("Ուշադրություն՝ ցածր գնահատական՝", marks[i]);
  }
}
```

Այսպիսի ծրագրերը արդեն դառնում են գործնական և մի քայլ մոտեցնում իրական համակարգերին:

Պարունակության փոփոխում

Մենք կարող ենք ավելացնել նոր տարր զանգվածի վերջում՝ օգտագործելով `push()` մեթոդը:

```
let guests = ["Աննա", "Մարիամ"];
guests.push("Տիգրան");
```

Այժմ `guests` զանգվածը ունի երեք անուն:

Նմանապես կարող ենք հեռացնել վերջին տարրը՝ `pop()` մեթոդով:

```
guests.pop(); // հեռացնում է "Տիգրան"-ը
```

Զանգվածները մեզ թույլ են տալիս աշխատել մեծաքանակ տվյալների հետ՝ կազմակերպված և հեշտ կառավարելի ձևով: Դրանք ծրագրավորման ամենակարևոր

կառուցվածքներից են, որոնք հատկապես անհրաժեշտ են իրական նախագծերում՝ ցուցակներ, տվյալների բազաներ, խաղեր կառուցելիս:

Գործառույթներ (Functions)

Ծրագրավորման մեջ երբ գործ ունենք մի գործողության հետ, որը պիտի մի քանի անգամ կրկնվի տարբեր մասերում, մենք չենք ուզում այդ գործողությունը նորից ու նորից գրել: Այդպես ոչ միայն ժամանակ է կորչում, այլև ծրագիրը դառնում է խառը, դժվար սպասարկելի:

Իսկ ի՞նչ կանեինք իրական կյանքում: Եթե ամեն անգամ նույն բանն ենք անում՝ օրինակ՝ «երեկոյան ատամները մաքրելը», ապա ստեղծում ենք սովորություն: Ծրագիրը ևս ունի այդպիսի «սովորություններ», և դրանք կոչվում են գործառույթներ:

Ի՞նչ է գործառույթը

Գործառույթը (function) կոդի հատված է, որը ունի անուն և որը կարող ենք կանչել (կատարել) այնքան անգամ, որքան ուզում ենք: Դա մի տեսակ մինի ծրագիր է՝ ծրագրի ներսում:

Պատկերացրու դու ստեղծել ես գործառույթ անունով `sayHello`, որը ողջունում է օգտվողին: Երբ ուզում ես այդ հաղորդագրությունը տեսնել, պարզապես կանչում ես գործառույթը՝ առանց նորից գրել այն:

Գործառույթի կառուցվածք

```
function անուն() {
  // կոդ, որը կկատարվի
}
```

Օրինակ՝

```
function greet() {
  console.log("Բարև, աշխարհ:");
}
greet(); // կանչում ենք գործառույթը
```

Այս կոդը սահմանում է `greet` անունով գործառույթ, որը տպում է հաղորդագրություն: Բայց մինչև այն չկանչենք `greet()`; ոչինչ չի կատարվում:

Գործառույթներ՝ արգումենտներով

Շատ հաճախ գործառույթները պետք է աշխատեն ոչ միայն նախապես գրված արժեքներով, այլ նաև օգտագործողի կողմից տրված տվյալներով: Դրա համար մենք կարող ենք գործառույթին փոխանցել արգումենտներ:

```
function greet(name: string) {
  console.log("Բարև, " + name + ":");
}
greet("Անահիտ"); // Բարև, Անահիտ:
greet("Տիգրան"); // Բարև, Տիգրան:
```

Այստեղ գործառույթը դառնում է ավելի օգտակար: Այն ընդունում է անուն և օգտագործում է այն ներսում՝ տպելու իր՝ անհատական տարբերակը:

Գործառույթներ՝ վերադարձվող արժեքով

Գործառույթները կարող են ոչ միայն ինչ-որ բան անել, այլ նաև վերադարձնել արժեք, որը կարող ենք պահել փոփոխականի մեջ:

```
function add(a: number, b: number): number {
  return a + b;
}
let result = add(5, 3); // 8
```

Այստեղ `add` գործառույթը ընդունում է երկու թիվ, գումարում է, և վերադարձնում արդյունքը: Այդ արդյունքը պահվում է `result` փոփոխականի մեջ:

Նշում՝
`return` բանալի բառը նշանակում է՝ «վերադարձնել»:

Օրինակ առօրյա կյանքից

Պատկերացրու, դու մի ռոբոտ ես կառուցում, որը պետք է հաշվի ջերմաստիճանի միջինը՝ երեք տարբեր օրերի համար: Եթե ամեն անգամ գրում ես այդ հաշվարկը, ծրագրի ծավալը ավելանում է:

Փոխարենը ստեղծում ես գործառույթ՝

```
function average(a: number, b: number, c: number): number {
  return (a + b + c) / 3;
}
```

Եվ հետո կարող ես օգտագործել այն տարբեր պարագաներում՝

```
let avg1 = average(20, 22, 24);
let avg2 = average(15, 18, 21);
```

Այսպես ծրագրում առաջանում է կրկնակի օգտագործելիություն, և կոդը դառնում է մաքուր, գրագետ ու հեշտ հասկանալի:

Հաջորդ գլուխներում կծանոթանանք գործառույթների առավել խոր տարբերակներին՝ ներառյալ arrow functions և callback գաղափարը:

Զանգվածների մեթոդներ

Նախորդ գլխում մենք սովորեցինք, թե ինչ է զանգվածը՝ մի քանի արժեքներ մեկ տեղում պահելու եղանակ: Բայց իրականում զանգվածը միայն տվյալների պահոց չէ: Այն մի ամբողջ «գործիքատուփ» է՝ տվյալները մշակելու համար:

TypeScript-ը (իսկ դրա հիմքում՝ JavaScript-ը) մեզ տալիս է հզոր մեթոդների մի շարք, որոնց միջոցով կարող ենք հեշտությամբ փոխել զանգվածները, զտել, դասավորել կամ վերածել նոր արժեքների:

Push() – ավելացնել տարր վերջում

Երբ ուզում ենք զանգվածի վերջում նոր տարր ավելացնել, օգտագործում ենք `push()` մեթոդը:

```
let students = ["Անի", "Գոռ"];
students.push("Մարիամ");
```

Այժմ `students` զանգվածը պարունակում է երեք անուն:

Առօրյա օրինակ. Դու կազմում ես հյուրերի ցուցակ՝ և յուրաքանչյուր նոր ժամանածին գրանցում ես ցուցակում:

Pop() – հեռացնել վերջին տարրը

Եթե ուզում ենք հեռացնել զանգվածի վերջին տարրը, օգտագործում ենք `pop()` մեթոդը:

```
students.pop();
```

Այս գործողությունը հեռացնում է վերջին անունը՝ "Մարիամ"-ը:

Shift() և unshift()

- `shift()` – հեռացնում է առաջին տարրը
- `unshift()` – ավելացնում է նոր տարր սկզբում

```
let colors = ["կարմիր", "կանաչ"];
colors.unshift("կապույտ"); // ["կապույտ", "կարմիր", "կանաչ"]
colors.shift(); // ["կարմիր", "կանաչ"]
```

Այս մեթոդները շատ օգտակար են, երբ տվյալների կարգը կարևոր է՝ օրինակ՝ երբ պահպանում ենք վերջին գործողությունների հերթականությունը:

ForEach() – աշխատել բոլոր տարրերի հետ

Եթե ուզում ես յուրաքանչյուր տարրի վրա ինչ–որ գործողություն կատարել, օգտագործիր `forEach()` մեթոդը:

```
let fruits = ["խնձոր", "դեղձ", "սերկևիլ"];
fruits.forEach(function(fruit) {
  console.log("Սիրում եմ", fruit);
});
```

Այստեղ մենք չենք փոխում զանգվածը, պարզապես անցնում ենք տարր առ տարր:

Map() – փոխել զանգվածը նորով

map()–ը մի մեթոդ է, որն վերադարձնում է նոր զանգված, որտեղ ամեն տարր փոխված է ըստ գործառույթի:

```
let numbers = [1, 2, 3];
let doubled = numbers.map(function(n) {
  return n * 2;
});
// doubled = [2, 4, 6]
```

Այս մեթոդը շատ օգտակար է, երբ ունես տվյալների ցուցակ և ուզում ես ստեղծել դրանց նոր տարբերակը՝ վերամշակված:

Filter() – գտել տարրերը

Երբ մեզ պետք են միայն որոշ տարրեր զանգվածից՝ պայմանով ընտրված, օգտագործում ենք `filter()` մեթոդը:

```
let scores = [95, 60, 45, 80];
let passed = scores.filter(function(score) {
  return score >= 60;
});
// passed = [95, 60, 80]
```

Սա նման է «զտիչի». միայն այն տարրերը են անցնում, որոնք բավարարում են պայմանին:

`Find()` – գտնել առաջին համապատասխան տարրը

Երբ պետք է գտնել ոչ թե բոլոր համապատասխանները, այլ **միայն առաջինը**, օգտագործում ենք `find()`:

```
let names = ["Անի", "Մարիամ", "Գոռ"];
let found = names.find(function(name) {
  return name.startsWith("Մ");
});
// found = "Մարիամ"
```

Եթե ոչինչ չգտնվի, վերադարձվում է `undefined`:

Օբյեկտներ (Objects)

Մինչ այժմ մենք աշխատում էինք պարզ արժեքների հետ՝ թիվ, տեքստ, զանգված: Բայց երբ ծրագիրը մեծանում է, տվյալները դառնում են ավելի բազմաշերտ: Միայն անունն ու տարիքը քիչ են՝ մեզ նաև պետք է հասցե, մասնագիտություն, սեռ, ընկերներ…

Դրա համար ծրագրավորման լեզուները տալիս են մեզ օբյեկտներ՝ կառուցվածքներ, որոնք թույլ են տալիս միավորված ձևով պահել տարբեր տվյալներ՝ կապված նույն բանի հետ:

Ինչ է օբյեկտը

Օբյեկտը մի կառուցվածք է, որը պահում է տվյալների հավաքածու: Դրանք կազմված են բանալի-արժեք զույգերից:

Պատկերացրու դու ուզում ես ներկայացնել մարդու մասին ամբողջական տեղեկատվություն: Եթե օգտագործես միայն փոփոխականներ՝ կստացվի այսպիսի մի բան.

```
let name = "Անի";
let age = 25;
let city = "Գորիս";
```

Բայց ավելի ճիշտ ու կազմակերպված կլինի այս ամենը պահել մեկ տեղում՝ որպես օբյեկտ:

Օբյեկտի ստեղծում

```
let person = {
  name: "Անի",
  age: 25,
  city: "Գորիս"
};
```

Այսպես մենք ստեղծեցինք մի փոփոխական, որի արժեքը օբյեկտ է: Այն պահում է երեք հատկություն (property)՝

- `name` → "Անի"
- `age` → 25
- `city` → "Գորիս"

Տվյալների հասանելիություն

Օբյեկտի արժեքներին կարող ենք հասնել երկու ձևով՝

Կետային գրառմամբ (dot notation)

```
console.log(person.name); // "Անի"
```

Գրաֆիկ գրառմամբ (bracket notation)

```
console.log(person["city"]); // "Գորիս"
```

Երկու եղանակն էլ հավասարազոր են, բայց երբ բանալին փոփոխական է կամ պարունակում է բացատ, պետք է օգտագործել փակագծեր:

Օբյեկտի հատկությունների փոփոխում

```
person.age = 26;
person.city = "Երևան";
```

Մենք կարող ենք փոխել օբյեկտի արժեքները, ինչպես նաև ավելացնել նոր հատկություններ:

```
person.profession = "ծրագրավորող";
```

Այժմ օբյեկտը ունի նաև `profession` հատկություն:

Օբյեկտ և ֆունկցիա

Օբյեկտը կարող է նաև գործառույթներ (մեթոդներ) պարունակել:

```
let user = {
  name: "Տիգրան",
  greet: function() {
    console.log("Բարև, ես եմ՝ " + this.name);
  }
};
user.greet(); // Բարև, ես եմ՝ Տիգրան
```

Այստեղ `greet`-ը մեթոդ է: Այն օբյեկտի ներսում գործողություն է իրականացնում: `this` բառը նշանակում է՝ "այս օբյեկտը":

Օրինակ առօրյա կյանքից

Պատկերացրու դու կառուցում ես առցանց խանութ: Յուրաքանչյուր ապրանք ունի անվանում, գին, առկա քանակ, նկարագրություն:

Օբյեկտը այստեղ լավագույն տարբերակն է՝

```
let product = {
  name: "Անլար ականջակալ",
  price: 15000,
  inStock: true,
  description: "Որակյալ ձայն և հարմարավետություն"
};
```

Այս տեղեկատվությունը մեկ տեղում ունենալը հեշտացնում է կառավարման գործընթացը: Օբյեկտի միջոցով ծրագիրը կարող է արագ ցուցադրել ապրանքը, հաշվարկել գինը, ստուգել մնացորդը և այլն:

Օբյեկտների ներքին օբյեկտներ

Օբյեկտները կարող են ներառել նաև այլ օբյեկտներ կամ զանգվածներ:

```
let student = {
  name: "Մարիամ",
  marks: {
    math: 18,
    physics: 19
  },
  hobbies: ["գրականություն", "նկարչություն"]
};
```

Այստեղ `marks`–ը օբյեկտ է, իսկ `hobbies`–ը՝ զանգված:

Զուգահեռ՝ զանգված vs օբյեկտ

- Զանգվածը լավ է այն ժամանակ, երբ արժեքները ունեն հերթականություն (օրինակ՝ ցուցակ)
- Օբյեկտը լավ է այն ժամանակ, երբ արժեքները ունեն անուններ ու բովանդակություն

Ի՞նչ կարող ենք անել օբյեկտներով

- Ստեղծել բարդ կառուցվածքներ (օրինակ՝ օգտատիրոջ պրոֆիլ)
- Պահել և խմբավորել տվյալներ ըստ իմաստի
- Ստեղծել համակարգեր՝ օրինակ՝ խաղեր, վեբ կայքեր, տվյալների գրանցման համակարգեր

Template Literals

Ծրագրավորման մեջ հաճախ պետք է մի քանի արժեքներ միավորենք տեքստի մեջ: Օրինակ՝ մենք ուզում ենք օգտվողին ողջունել՝ օգտագործելով նրա անունը:

Դրա համար նախկինում օգտագործում էինք հավելման մեթոդը (string concatenation)՝

```
let name = "Անի";
console.log("Բարև, " + name + ":");
```

Սա աշխատում է, բայց երբ տեքստը երկար է դառնում, կոդը դառնում է անհասկանալի ու դժվար ընթեռնելի:

Ինչ է Template Literal-ը

Template Literal-ը տեքստ գրելու նոր, պարզ ձև է: Այն օգտագործում է **backtick** նշաններ՝ ` (ոչ՝ ' կամ "), և թույլ է տալիս ներսում միանգամից գրել փոփոխականներ՝ ${} սինտաքսով:

Օրինակ՝

```
let name = "Անի";
console.log(`Բարև, ${name}:`);
```

Արդյունքը նույնն է, բայց կոդը շատ ավելի մաքուր ու ընթեռնելի է:

Ավելի բարդ օրինակ

```
let user = "Տիգրան";
let age = 20;
let city = "Գորիս";
let message = `Բարև, իմ անունը ${user} է, ես ${age} տարեկան եմ և
ապրում եմ ${city}-ում:`;
console.log(message);
```

Առանց template literal-ների սա կդառնար.

```
let message = "Բարև, իմ անունը " + user + " է, ես " + age + " տարեկան
եմ և ապրում եմ " + city + "-ում:";
```

Երկրորդ տարբերակն ավելի խճճված է, դժվար է կարդալը, հատկապես երկար նախադասությունների դեպքում:

Օրինակ առօրյա կյանքից

Պատկերացրու՝ քեզ զանգում է ընկերոջդ հայրը և հարցնում, թե որտե՞ղ է իր որդին: Դու պատասխանում ես՝

«Նա հիմա դուրս է եկել Տիգրանի հետ և գնացել է Գորիսի այգի»:

Ծրագրում սա կարելի է կառուցել՝

```
let friend = "Տիգրան";
```

```
let place = "Գորիսի այգի";
console.log(`Նա հիմա դուրս է եկել ${friend}-ի հետ և գնացել է
${place}:`);
```

Ինչու է սա կարևոր

- Կոդը դառնում է ավելի ընթեռնելի
- Հեշտ է աշխատել բազմաթիվ փոփոխականներով
- Շատ ավելի հարմար է օգտագործել շաբլոններ, օրինակ՝ HTML-ի կամ JSON-ի կառուցման ժամանակ

Falsy և Truthy արժեքներ

Ծրագրավորման մեջ երբ օգտագործում ենք պայմաններ (օրինակ՝ `if`), համակարգը որոշում է՝ արժեքը "ճշմարիտ" է, թե "սխալ":

Բայց բոլոր արժեքները չէ, որ ուղղակի `true` կամ `false` են: Որոշ արժեքներ, երբ օգտագործվում են որպես պայմանի մաս, ավտոմատ կերպով համարվում են ճշմարիտ (truthy) կամ սխալ (falsy):

Ինչ է Falsy արժեք

Falsy արժեքը այն է, որն `if` պայմանում համարվում է սխալ:

Բոլոր `falsy` արժեքները՝

- `false`
- `0`
- `""` (դատարկ տող)
- `null`
- `undefined`
- NaN (Not a Number)

Օրինակ՝

```
if (0) {
  console.log("Այսպես չի տպվի");
}
```

Որովհետև 0-ը **falsy** է:

Ինչ է Truthy արժեք

Truthy արժեքն այն է, որն `if` պայմանում համարվում է ճիշտ, նույնիսկ եթե դա ուղիղ `true` չէ:

Օրինակ՝

- Ցանկացած ոչ-դատարկ տող՝ `"Hello"`
- Ցանկացած ոչ զրոյական թիվ՝ `42`
- Ցանկացած օբյեկտ կամ զանգված՝ `[]`, `{}`

```
if ("Գորիս") {
  console.log("Տպվում է, որովհետև սա truthy է");
}
```

Ինչու է սա կարևոր

Երբ գրում ես պայմաններ, հաճախ չես օգտագործում ուղիղ `true` կամ `false`, այլ փոփոխականներ: Եթե չհասկանաս `falsy/truthy`-ի իմաստը, կստանաս անսպասելի պահվածք:

Typeof օպերատոր

Ծրագրում երբեմն անհրաժեշտ է ստուգել արժեքի տիպը: Այս հարցում օգնում է `typeof` օպերատորը:

Այն վերադարձնում է արժեքի տվյալների տեսակը որպես տող:

Օրինակներ

```
typeof 42           // "number"
typeof "Գիրք"       // "string"
typeof true         // "boolean"
typeof undefined    // "undefined"
typeof null         // "object" ❗ (հատուկ դեպք)
typeof [1, 2, 3]    // "object"
typeof {name: "Անի"} // "object"
typeof function(){} // "function"
```

Որտե՞ղ է պետք

Երբ ունես փոփոխական, բայց չգիտես՝ ինչ տիպի արժեք կա ներսում, կարող ես ստուգել՝

```
let x = "Բարև";
if (typeof x === "string") {
  console.log("Այո, սա տեքստ է");
}
```

Ուշադրություն

- Զանգվածները ([ ]) և օբյեկտները ({}) երկուսն էլ վերադարձնում են `"object"`:
- `null`–ը նույնպես վերադարձնում է `"object"`: Սա լեզվի մի փոքր տարօրինակություն է, ոչ սխալ:

Null և undefined

Սրանք հաճախ շփոթող են սկսնակների համար: Դիտենք տարբերությունները:

Undefined

Դա նշանակում է՝ արժեքը դեռ տրված չէ:

```
let a;
console.log(a); // undefined
```

Փոփոխականը գոյություն ունի, բայց դեռ արժեք չունի:

Null

Դա նշանակում է՝ դատարկ արժեք: Այսինքն՝ մենք հստակ ասում ենք, որ այստեղ ոչինչ չկա:

```
let b = null;
console.log(b); // null
```

Օրինակ կյանքից

Պատկերացրու՝ քո ընկերոջ գրասենյակում մի սեղան կա:

- Եթե սեղանին ոչինչ չկա, բայց դու չես պարզել՝ դա պատահակա՞ն է, թե՞ ոչ՝ `undefined`

- Եթե սեղանին դու հատուկ կերպով չես դրել ոչինչ՝ որովհետև դեռ դատարկ պահում ես՝ `null`

Սովորական սխալ

Բազմաթիվ սկսնակներ շփոթում են սա.

```
let a;
console.log(a == null); // true ❗
console.log(a === null); // false ✅
```

`==` համեմատությունը թույլ է և ընդունում `undefined == null` որպես ճշմարիտ:

`===` ավելի խիստ է և կխուսափի սխալներից:

Եզրահանգում

- Եթե արժեքը չես տվել՝ դա `undefined` է
- Եթե տվել ես հատուկ դատարկություն՝ դա `null` է
- Ստուգելու համար տիպերը՝ օգտագործիր `typeof`
- Պայմաններում՝ հասկացի արդյոք արժեքը `truthy` է, թե `falsy`

Math օբյեկտ

Ծրագրերում հաճախ անհրաժեշտ է աշխատել թվերի հետ՝ հաշվել, կլորացնել, ստանալ պատահական թվեր և այլն: JavaScript-ը մեզ տալիս է `Math` անունով ներկառուցված օբյեկտ, որը պարունակում է շատ օգտակար մեթոդներ:

Math-ի օգտակար մեթոդներ

- `Math.floor(x)` - Կլորացնում է ներքև (թվից ցածր)
- `Math.ceil(x)` - Կլորացնում է վերև (թվից բարձր)
- `Math.random()` - Վերադարձնում է պատահական թիվ՝ 0-1
- `Math.min(a, b, ...)` - Վերադարձնում է ամենափոքր արժեքը
- `Math.pow(x, y)` - Հանվում է աստիճանով (x^n)
- `Math.sqrt(x)` - Հանում է քառակուսի արմատը

Օրինակներ

```
Math.round(4.6);      // 5
```

```
Math.floor(4.6);      // 4
Math.ceil(4.1);       // 5
Math.max(10, 20, 5); // 20
Math.random();        // օրինակ՝ 0.37489231
```

Եթե ուզում ես պատահական ամբողջ թիվ 1-ից 10 միջակայքում՝

```
let random = Math.floor(Math.random() * 10) + 1;
```

Որտե՞ղ է պետք

- Խաղերի մեջ՝ պատահական շարժումներ կամ թվեր
- Օգտվողի մուտքերի գնահատում
- Գրաֆիկական և վիճակագրական ծրագրերում

JSON (JavaScript Object Notation)

JSON-ը տեքստային ձևաչափ է, որը թույլ է տալիս փոխանցել տվյալներ օբյեկտների և զանգվածների տեսքով: Այն օգտագործվում է գրեթե բոլոր վեբ ծրագրերում՝ տվյալների պահեստավորման և փոխանցման համար:

Ինչպիսի՞ն է JSON-ը

```
{
  "name": "Անի",
  "age": 25,
  "isStudent": true
}
```

JSON-ը գրեթե նման է JavaScript-ի օբյեկտներին, բայց՝

- Բոլոր բանալիները պետք է լինեն String ով(" "),
- Չի կարող պարունակել ֆունկցիաներ

JSON.stringify()

Անցում է կատարում օբյեկտից դեպի տեքստ (string)

```
let user = { name: "Անի", age: 25 };
let jsonText = JSON.stringify(user);
// '{"name":"Անի","age":25}'
```

JSON.parse()

Անցում է կատարում տեքստից դեպի օբյեկտ

```
let obj = JSON.parse('{"name":"Անի","age":25}');
console.log(obj.name); // "Անի"
```

Ինչու է սա կարևոր

- Տվյալների փոխանակում սերվերի հետ
- Տվյալների պահպանում ֆայլերում կամ տեղային հիշողության մեջ
- Հեշտ փոխակերպում՝ տեքստ ↔ օբյեկտ

Սխալների կառավարում՝ try / catch և Error

Ծրագրում ցանկացած բան կարող է սխալ գնալ՝ օգտագործողը մուտքագրել է սխալ տվյալ, կապը խափանվել է, JSON-ը սխալ ձևաչափով է: Եթե մենք չվարենք սխալները՝ ծրագիրը կփակի աշխատանքը:

Սրա լուծումը՝ try / catch կառուցվածքն է:

Ինչպես է աշխատում

```
try {
  // կոդ, որը կարող է սխալ առաջացնել
  let data = JSON.parse("ասղ"); // սխալ JSON
} catch (error) {
  console.log("Սխալ առաջացավ.", error.message);
}
```

Եթե `try`–ի մեջ սխալ լինի, կոդը չի կանգնում, այլ անցնում է `catch`-ը:

Error օբյեկտ

Մենք կարող ենք նաև ինքներս ստեղծել սխալներ:

```
function divide(a: number, b: number) {
  if (b === 0) {
    throw new Error("Չի կարելի բաժանել զրոյի վրա:");
  }
  return a / b;
}
```

```
try {
  console.log(divide(10, 0));
} catch (e) {
  console.log("Սխալ՝", e.message);
}
```

Ինչու է կարևոր

- Չխափանել ամբողջ ծրագիրը սխալի դեպքում
- Ճշգրտորեն ցույց տալ օգտվողին՝ ինչ է տեղի ունեցել
- Մասնավորապես կարևոր է՝ տվյալների հետ աշխատելիս (JSON, ֆայլեր, սերվերներ)

`IsNaN` ֆունկցիա

Ծրագրում երբեմն փոփոխականը կարող է թվի փոխարեն անվավեր արժեք ստանալ: Օրինակ՝ օգտագործողը մուտք է արել տեքստ, բայց դու ուզում ես այն համարել թիվ: Այստեղ է օգնության գալիս `isNaN()`:

Ինչ է `NaN`

NaN նշանակում է՝ *Not a Number*, այսինքն՝ «թիվ չէ»: Այս արժեքը ստացվում է, երբ փորձում ենք կատարել անհաջող թվային գործողություն:

```
let x = Number("abc"); // NaN
```

Ինչ է անում `isNaN()`

Սա ստուգում է՝ արդյո՞ք արժեքը **NaN** է:

```
let n = Number("abc");
console.log(isNaN(n)); // true
```

Եթե ուզում ես ստուգել՝ արդյոք արժեքը թիվ է, կարող ես գրել՝

```
let value = "123";
if (!isNaN(Number(value))) {
```

```
  console.log("Սա թիվ է:");
} else {
  console.log("Սա թիվ չէ:");
}
```

Ուշադրություն

`isNaN()` երբեմն ստուգում է ոչ այնպես, ինչպես սպասում ես՝ որովհետև այն ներսում թաքուն փոխում է արժեքը թվի: Որպես ավելի հստակ տարբերակ՝ TypeScript-ում երբեմն ավելի լավ է օգտագործել `Number.isNaN()`:

```
Number.isNaN("abc"); // false ❌
Number.isNaN(NaN);   // true ✅
```

Որտե՞ղ է պետք

- Երբ ստանում ես տվյալներ, և չգիտես՝ դրանք ճիշտ թվե՞ր են
- Երբ ուզում ես ծրագրում խուսափել սխալ հաշվարկներից
- Օգտագործողների մուտքերի ստուգման ժամանակ

Continue և break հրամանները

Ծրագրավորման ընթացքում հաճախ է առաջանում անհրաժեշտություն ազդելու ցիկլերի ընթացքի վրա: TypeScript-ում, ինչպես JavaScript-ում, գոյություն ունեն երկու կարևոր հրամաններ, որոնք թույլ են տալիս կառավարել ցիկլի կատարման ընթացքը՝ `continue` և `break`:

Երբեմն մենք ուզում ենք պարզապես բաց թողնել ցիկլի ընթացիկ քայլը ու անցնել հաջորդին, առանց դուրս գալու ամբողջ ցիկլից: Այստեղ է գործի դրվում `continue` հրամանը: Իսկ երբ հարկավոր է ընդհանրապես դադարեցնել ցիկլի աշխատանքը՝ անկախ նրանից, թե քանի քայլ դեռ մնացել է, օգտագործում ենք `break` հրամանը:

Պատկերացնենք իրավիճակ. ունենք թիվերի շարք 1-ից 10-ը, և ուզում ենք միայն տպել այն թվերը, որոնք զույգ են: Մենք կարող ենք գրել հետևյալը՝

```
for (let i = 1; i <= 10; i++) {
  if (i % 2 !== 0) {
    continue;
  }
  console.log(i);
}
```

Այս կոդը ասում է՝ եթե թիվը կենտ է (`i % 2 !== 0`), բաց թող այս քայլը և անցիր հաջորդին: `continue` հրամանը ցիկլին ասում է՝ «Էս քայլին էլ բան չունենք անելու, գնա հաջորդի վրա»:

Այժմ տեսնենք `break` հրամանի օրինակ: Ենթադրենք ուզում ենք գտնել առաջին թիվը 1-ից 100-ի միջակայքում, որը բաժանվում է 17-ի: Հենց որ այն գտնենք, պիտի դուրս գանք ցիկլից՝ առանց շարունակելու ավելորդ քայլեր:

```
console.log("Առաջին թիվը, որը բաժանվում է 17-ի, սա է՝", i);
for (let i = 1; i <= 100; i++) {
  if (i % 17 === 0) {
    break;
  }
}
```

Այստեղ, երբ պայմանը ճիշտ է լինում (`i % 17 === 0`), մենք ոչ միայն տպում ենք այդ թիվը, այլ նաև `break`-ով դադարեցնում ենք ամբողջ ցիկլը: Սա կարևոր է, երբ չենք ուզում ավելորդ հաշվարկներ անել:

Այս հրամանները հատկապես օգտակար են, երբ մենք աշխատում ենք մեծ տվյալների կամ բարդ պայմանների հետ: Դրանք օգնում են մեզ կարճ և արդյունավետ կերպով կառավարել ծրագրի հոսքը՝ խնայելով ռեսուրսներ և բարձրացնելով կոդի ընթեռնելիությունը:

Arrow functions (Սլաքային ֆունկցիաներ)

Երբ սովորական ֆունկցիաները կարճ են ու միանգամայն պարզ, TypeScript-ում (և JavaScript-ում) կարելի է գրել դրանք շատ ավելի կոմպակտ և հարմար ձևով՝ օգտագործելով arrow function գրությունը:

Սլաքային ֆունկցիաների հիմնական առանձնահատկությունը նրա կարճ ձևն է: Դրանք գրելիս օգտագործվում է «=>» սլաքը, որը ցույց է տալիս, թե ինչ է ֆունկցիան վերադարձնելու:

Պարզ օրինակ

Նախ՝ ունենք սովորական ֆունկցիա, որը գումարում է երկու թիվ.

```
function add(a: number, b: number): number {
  return a + b;
}
```

Այդ նույնը կարող ենք գրել սլաքային ֆունկցիայով՝

```
const add = (a: number, b: number): number => {
  return a + b;
};
```

Այստեղ.

- `const add =` նշանակում է՝ հռչակում ենք փոփոխական `add`, որը կպահի ֆունկցիա:
- `(a: number, b: number)` — սա ֆունկցիայի ներմուծվող պարամետրերն են՝ տիպերով:
- `=>` — սլաքը, որը ցույց է տալիս ֆունկցիայի մարմինը:
- `{ return a + b; }` — ֆունկցիայի մարմինը:

Եթե ֆունկցիայի մարմինը շատ պարզ է՝ միայն մեկ արտահայտություն, կարող ենք գրեթե կարճացնել.

```
const add = (a: number, b: number): number => a + b;
```

Առօրյայի օրինակ՝ պատկերացրու, որ դու ուզում ես շտապ հաշվարկել երկու թվի գումարը, բայց չես ուզում գրել երկարատև հայտարարություն. այստեղ կօգնի arrow function-ը՝ կարճ և հստակ:

Generator functions (Գեներատոր ֆունկցիաներ)

Գեներատոր ֆունկցիաները թույլ են տալիս ստեղծել հատուկ ֆունկցիաներ, որոնք աշխատում են կանգ առնելով և հետո շարունակելով՝ հերթական արժեքներ վերադարձնելով: Դրանք շատ օգտակար են են այն դեպքերում, երբ ուզում ենք աշխատել մեծ տվյալների հավաքածուների հետ կամ ստեղծել կրկնվող հաջորդականություններ՝ մի քայլ առաջ տանելով:

Գեներատորները ճանաչվում են `function*` (աստղիկով ֆունկցիա) ձևով և աշխատում են `yield` բանալի խոսքով՝ յուրաքանչյուր կանչով վերադարձնելով հաջորդ արժեքը:

Պարզ օրինակ

Պատկերացրու, որ ուզում ես քայլ առ քայլ թվեր վերադարձնես՝ սկսած 1-ից, ու մինչ 3-ը: Գեներատորով դա կարող է այսպիսին լինել.

```
function* numbers() {
  yield 1;
  yield 2;
  yield 3;
```

```
}
```

Օգտագործման ձևը.

```
const gen = numbers();
console.log(gen.next().value); // 1
console.log(gen.next().value); // 2
console.log(gen.next().value); // 3
console.log(gen.next().done);  // true (կատարել է)
```

Յուրաքանչյուր `next()` կանչը «երկարատև» ֆունկցիան մի քայլ առաջ է տանում և տալիս հաջորդ արժեքը: Երբ արժեքները վերջանում են, `done` դառնում է `true`:

Առօրյայի օրինակ

Կարծի՛ր, որ դու հերթում ես կանգնած մարդիկ, որոնց դու պետք է հերթով կանչես՝ առանց մեկանգամյա զանգված պատրաստելու: Գեներատորը հենց այդպիսին է՝ դու կարող ես կանչել հաջորդին, երբ կուզես, առանց հանելու բոլորին միանգամից:

Scope-ի օրինակներ և դրանց կապը սինթաքսի հետ

Ֆունկցիայի սկոպ և սինթաքս

```
function greet() {
  let message = "Բարեւ";
  console.log(message);
}
greet(); // "Բարեւ"
// console.log(message); // error, message տեսանելի չէ այստեղ
```

Այստեղ `message` փոփոխականը գտնվում է ֆունկցիայի բլոկի `{}` ներսում, ու դուրս ֆունկցիայից չի երևում:

Բլոկի սկոպ և սինթաքս

```
if (true) {
  let a = 5;
  const b = 10;
  console.log(a + b); // 15
}
// console.log(a); // error, a չի տեսանելի դուրս բլոկից
```

Եթե փոխարինենք `let`-ը `var`-ով, ապա

```
if (true) {
  var a = 5;
}
console.log(a); // 5, որովհետև var ունի function scope, ոչ block scope
```

Փոփոխականների հռչակումներ՝ `let`, `const`, `var`

- `let` և `const` ունեն block scope, նշանակում է տեսանելի են միայն {} ներսում:
- `var` ունի function scope, ինչն ավելի հին մոտեցում է և կարող է առաջացնել շփոթություններ:

Արտահայտությունների և հրամանների վերջ

Սա ևս հստակեցման կարիք ունի.

```
let x = 10
let y = 20
console.log(x + y)
```

```
let x = 10;
let y = 20;
console.log(x + y);
```

Երկրորդ տարբերակը համարվում է լավ սովորույթ, որը կխուսափի հնարավոր սխալներից, հատկապես, երբ կոդը մեծանում է կամ միացվում այլ կոդերի հետ:

Առօրյայի պարզ օրինակ

Կարծիր, որ դու ունես մի խցիկ (սենյակ), որտեղ միայն որոշ մարդիկ կարող են մտնել և ինչ-որ բան անել (block scope): Մյուսներն այդ խցիկից դուրս են, ու չգիտեն այնտեղի գաղտնիքները: Իսկ որոշ մարդիկ աշխատում են ողջ տան տարածքում (global scope): Եթե օգտագործես `let` կամ `const`, խցիկի դուռը փակ կլինի և միայն ներսի մարդիկ կկարողանան տեսնել ու փոխել իրերը: Իսկ եթե օգտագործես `var`, դա նման կլինի մեծ դռան, որ բաց է ողջ տան համար (function scope կամ նույնիսկ global, եթե դրսում):

Ֆունկցիայի հայտարարության ձևերը TypeScript-ում

TypeScript-ում, ինչպես JavaScript-ում, ֆունկցիաներ կարելի է հռչակել և ստեղծել մի քանի տարբեր ձևերով: Յուրաքանչյուրը ունի իր առանձնահատկությունները և օգտագործման հատուկ դեպքերը:

1. Սովորական ֆունկցիայի հայտարարություն (Function Declaration)

Սա ամենահայտնի և ավանդական ձևն է, երբ մենք օգտագործում ենք `function` բանալին և անմիջապես անուն ենք տալիս ֆունկցիային:

Սինթաքս՝

```
function ֆունկցիայի_անուն(պարամետրեր): Վերադարձի_տիպ {
  // ֆունկցիայի մարմին
}
```

Օրինակ՝

```
function greet(name: string): string {
  return `Բարեւ, ${name}!`;
}
```

Այս ձևով հռչակված ֆունկցիան կարող է կանչվել ցանկացած տեղից ֆայլում, քանի որ դրանք հիմնապես "hoisted" են (բարձրացվում են ֆայլի վերին մաս):

2. Ֆունկցիայի արտահայտություն (Function Expression)

Այս ձևում ֆունկցիան ստեղծվում է որպես արժեք և պահպանվում փոփոխականի մեջ: Այս դեպքում ֆունկցիայի անունը կարող է լինել կամ բաց թողնվել (անանուն ֆունկցիա):

Սինթաքս՝

```
const ֆունկցիա = function(պարամետրեր): Վերադարձի_տիպ {
  // մարմին
};
```

Օրինակ՝

```
const greet = function(name: string): string {
  return `Բարեւ, ${name}!`;
```

```
};
```

Այս ձևով ֆունկցիան հասանելի կլինի միայն այն հատվածում, որտեղ հայտարարում ենք փոփոխականը: Նման ֆունկցիան չի բարձրացվում (hoisting), ուստի պետք է հռչակել մինչև կանչելը:

Հասարակ տիպեր (Primitive Types)

Հասարակ տիպերն այն տվյալների տեսակներն են, որոնք պահում են պարզ, անփոփոխ արժեքներ՝ օրինակ՝ թվեր, տեքստեր կամ լոգիկ արժեքներ:

Հիմնական պարզ տիպերը TypeScript-ում՝

- **number** — թվային արժեքներ (լրիվ և սահող կետով), օրինակ՝ `42`, `3.14`
- **string** — տեքստային տիպ, օրինակ՝ `"Hello"`, `'Բարեւ'`
  boolean — լոգիկ տիպ, երկու հնարավոր արժեք՝ `true` կամ `false`
- **null** — հատուկ արժեք, որը ցույց է տալիս, որ փոփոխականն ունի բացակայող արժեք
- **undefined** — փոփոխականի ոչ սահմանված արժեքը
- **symbol** — յուրահատուկ և անփոփոխ նշան (առավել հատուկ դեպքերում օգտագործվում է)
- **bigint** — շատ մեծ ամբողջ թվեր, որոնք ավելի մեծ են, քան սովորական `number`-ի հնարավոր արժեքները

Հասարակ տիպերի հատկանիշները

- Նրանք պահվում են անմիջապես հիշողության մեջ, որպես պարզ արժեք
- Կատարում են կոպի արում (copy by value), այսինքն՝ երբ պատճենում ես մեկ փոփոխական մյուսին, ստեղծվում է ամբողջովին անկախ նոր օրինակ

Օրինակ

```
let x: number = 10;
let y = x;  // y կստանա 10-ի պատճենը
y = 20;    // փոփոխելով y-ն, x-ը չի փոխվում
console.log(x); // 10
console.log(y); // 20
```

Այս կոդում x և y ունեն առանձին առանձին արժեքներ՝ անկախ իրարից:

Հղումով տիպեր (Reference Types)

Հղումով տիպերը պահպանվում են հիշողության մեջ իրենց հասցեով (հղումով), ոչ թե անմիջապես արժեքով: Սա նշանակում է, որ երբ նման տիպերի մի փոփոխականին մյուսին փոխանցում ենք, մենք պատճենում ենք հասցեն, ոչ ինքն արժեքը:

Հղումով տիպերի օրինակներ՝

- **Object** — օբյեկտներ՝ որպես կույտեր՝ հատկություններով և մեթոդներով
- **Array** — զանգվածներ
- **Function** — ֆունկցիաներ նույնպես օբյեկտներ են
- **Class instances** — դասի օրինակներ

Հղումով տիպերի հատկանիշները

- Պատճենելիս պահվում է հղումը հիշողության հասցեի վրա, ոչ թե ամբողջ օբյեկտի տվյալները
- Երբ մի փոփոխականի միջոցով փոխում ես օբյեկտը, փոփոխությունը կտեսնեն նաև մյուս բոլոր փոփոխականները, որոնք հղվում են նույն օբյեկտին

Օրինակ

```
let obj1 = { name: "Աշոտ" };
let obj2 = obj1; // obj2 հիմա հղվում է նույն օբյեկտին
obj2.name = "Լիզա";
console.log(obj1.name); // Լիզա
console.log(obj2.name); // Լիզա
```

Այստեղ `obj1` և `obj2` իրականում հղվում են նույն օբյեկտին հիշողության մեջ: Փոփոխելով մեկը՝ փոխվում է նաև մյուսի տեսանելի տվյալը:

Ինչու է կարևոր տարբերությունը հասկանալ

- Եթե ցանկանում ես աշխատել ընդամենը տվյալների պատճենների հետ, պետք է օգտագործել պարզ տիպերը կամ գիտակցված կոպի անել հղումով տիպերի դեպքում:
- Հղումով տիպերով աշխատելիս պետք է զգուշանալ, որ մի քանի փոփոխական կարող են փոխել նույն օբյեկտը՝ առաջացնելով ոչ ակնկալվող փոփոխություններ:
- Այս հասկացությունը կարևոր է նաև ֆունկցիաների պարամետրերի փոխանցման ժամանակ և տվյալների կառավարման ժամանակ:

Array Destructuring (Զանգվածի քանդում)

Array destructuring-ը թույլ է տալիս մաս-մաս վերցնել զանգվածի (array) տարրերը ու անմիջապես հանձնարարվել փոփոխականներին՝ շատ պարզ ու կարճ ձևով:

Սինթաքս

```
const [var1, var2, var3] = array;
```

Օրինակ

```
const numbers = [10, 20, 30];
// Տրամադրենք, ուզում ենք տարրերը պահել առանձին փոփոխականներում
const [a, b, c] = numbers;
console.log(a); // 10
console.log(b); // 20
console.log(c); // 30
```

Առավելություններ

- Կարճ և պարզ կոդ՝ փոխարենը, որ գրել բազմաթիվ տողեր `const a = numbers[0]` և այլն:
- Կարող ես քանդել միայն անհրաժեշտ մասերը

Մասնակի քանդում և մնացորդի պահպանում

```
const [first, , third] = numbers; // Բաց թողնում ենք երկրորդը
console.log(first); // 10
console.log(third); // 30
// Մնացորդը հավաքում ենք rest operator-ով
const [head, ...tail] = numbers;
console.log(head); // 10
console.log(tail); // [20, 30]
```

Object Destructuring (Օբյեկտի քանդում)

Object destructuring-ը թույլ է տալիս օբյեկտի հատկություններն անմիջապես հանձնել փոփոխականների՝ հեշտ ու հստակ:

Սինթաքս

```
const { property1, property2 } = object;
```

Օրինակ

```
const person = {
  name: "Աշոտ",
  age: 18,
  city: "Գորիս"
```

```
};
const { name, city } = person;
console.log(name); // Աշոտ
console.log(city); // Գորիս
```

Կարևոր է՝

- Փոփոխականների անունները պետք է համընկնեն օբյեկտի հատկությունների անունների հետ
- Կարող ես փոխել անունը հետևյալ ձևով՝

```
const { name: personName, age } = person;
console.log(personName); // Աշոտ
console.log(age); // 18
```

Նմուշի քանդում և մնացորդ

```
const { name, ...otherProps } = person;
console.log(name); // Աշոտ
console.log(otherProps); // { age: 18, city: "Գորիս" }
```

Առօրյայի օրինակ՝

Կարծիր, որ դու ունես սանդղակ, որի վրա դրված են գրքեր (զանգված), և ուզում ես առաջին գրքի անունը, կամ ունես անձի տվյալներ (օբյեկտ) և ուզում ես միայն անունն ու քաղաքը առանձին վերցնել՝ հեշտությամբ:

Error-ների հիմնական տեսակներ

1. Syntax Error (Սինթաքսի սխալ)

Սա տեղի է ունենում, երբ կոդը չի հետևում լեզվի գրելու կանոններին: Օրինակ՝ մոռացել ես փակել փակագծերը, կամ գրել ես սխալ բառ:

Օրինակ՝

```
function greet(name: string {
  console.log("Բարեւ, " + name);
}
```

Հիմնական խնդիրը՝ բացակայում է փակագծի փակող մասը): Հետևանքով TypeScript կամ JavaScript չկարողանա կարդալ կոդը և կտանի կոդի վերլուծության սխալ:

2. Reference Error (Հղման սխալ)

Այս սխալը տեղի է ունենում, երբ փորձում ես օգտագործել փոփոխական կամ ֆունկցիա, որը կամ չի հայտարարվել, կամ չի գտնվում տեսանելիության սկոպում:

Օրինակ՝

```
console.log(x); // Error, x-ը չի սահմանված
```

Եթե x փոփոխականը հայտարարված չէ այդ հատվածում, JavaScript-ը կգրի `ReferenceError: x is not defined`:

3. Type Error (Տիպի սխալ)

TypeScript-ում շատ հաճախ հանդիպում են նաև տիպերի սխալներ, երբ փորձում ես գործարկել գործողություններ անհամատեղելի տիպերի վրա:

Օրինակ՝

```
let num: number = 5;
num.toUpperCase(); // Error, number տիպի վրա չկա toUpperCase ֆունկցիա
```

TypeScript կոմպիլյատորը կտարածի տիպի սխալի հաղորդում՝ նախքան ծրագիրը գործարկելը:

4. Range Error (Диапазонային սխալ)

Դուք կստանաք RangeError, երբ օգտագործում եք թվային արժեք, որը դուրս է թույլատրելի սահմաններից: Օրինակ, զանգվածի ինդեքսի սահմաններից դուրս անցնելը:

5. Eval Error

Այս սխալը հատուկ կապված է `eval()` ֆունկցիայի հետ, երբ սխալ է նրա գործարկումը:

6. URI Error

Ծագում է, երբ սխալ օգտագործվում են URL-ների կոդավորման/դեկոդավորման մեթոդներ, օրինակ՝ `decodeURI` կամ `encodeURI` սխալ պարամետրերով:

Ինչպես հայտնաբերել և լուծել սխալները

- **Syntax errors**-ը հայտնաբերվում են կոդը հավաքելիս կամ գրելու պահին (IDE-ներում, օրինակ VS Code)

- **Reference errors**-ը հաճախ հայտնաբերվում են runtime-ի ժամանակ, երբ փորձում ես օգտագործել ոչ սահմանված կամ ոչ հասանելի արժեքներ
- Type errors-ը TypeScript-ը ցույց է տալիս կոմպիլյացիայի պահին, օգնելով խուսափել runtime-ի սխալներից
- Օգտագործիր console.log, debugger, և try...catch բլոկներ՝ runtime սխալները բռնելու և վերլուծելու համար

Spread Operator (. . .)

Spread operator-ը ցույց է տալիս ռեստ (rest) և փոխանցում (spread) նույն նշանը `...`, բայց տարբեր իրավիճակներում օգտագործվում է տարբեր նշանակությամբ:

Այս բաժնում կխոսենք հենց spread գործառույթի մասին, որը թույլ է տալիս «բացել» iterable (օրինակ զանգվածը կամ օբյեկտը)՝ մաս-մաս տարածելով այլ հավաքածուների մեջ:

Spread Array-ների վրա

Եթե ունես մի զանգված և ուզում ես իր տարրերը փոխանցել մեկ ուրիշ զանգվածի, կամ ֆունկցիայի որպես առանձին պարամետրեր՝

```
const arr1 = [1, 2, 3];
const arr2 = [4, 5, 6];
const combined = [...arr1, ...arr2];
console.log(combined); // [1, 2, 3, 4, 5, 6]
```

Այստեղ `...arr1` և `...arr2` բացում են զանգվածները և տեղադրում տարրերը նոր զանգվածի մեջ:

Spread ֆունկցիայի կանչերում

Օրինակ՝

```
function sum(a: number, b: number, c: number) {
  return a + b + c;
}
const numbers = [1, 2, 3];
console.log(sum(...numbers)); // 6
```

Այստեղ զանգվածի տարրերը փոխանցվում են որպես առանձին արգումենտներ:

Spread օբյեկտների վրա

ES2018-ից հնարավոր է օգտագործել spread օբյեկտների հետ՝ ստեղծելու նոր օբյեկտ՝ հավաքելով մեկ կամ մի քանի օբյեկտների հատկություններ:

```
const obj1 = { a: 1, b: 2 };
const obj2 = { b: 3, c: 4 };

const merged = { ...obj1, ...obj2 };
console.log(merged); // { a: 1, b: 3, c: 4 }
```

Ի՞նչ է կատարվում՝ սկզբում գրվում են `obj1`-ի բոլոր հատկությունները, հետո `obj2`-ի, և եթե ունեն նույն անուններով հատկություններ, ապա վերջինը տիրապետում է (այստեղ `b: 3`-ը փոխարինեց `b: 2`):

Կարևոր նկատառումներ

- Spread operator-ը չի փոխարինում deep copy-ը, այսինքն՝ եթե օբյեկտի կամ զանգվածի ներսում ունես nested օբյեկտներ, դրանք հղումով են փոխանցվում:
- Շատ հարմար է օգտագործել immutable data-ի մոտեցումների համար՝ փոփոխության ժամանակ նոր օրինակ ստեղծելու համար:

Օրինակ՝ immutable փոփոխություն զանգվածի մեջ

```
const oldArray = [1, 2, 3];
const newArray = [...oldArray, 4]; // ավելացնում ենք 4-ը նոր զանգվածի մեջ
console.log(oldArray); // [1, 2, 3]
console.log(newArray); // [1, 2, 3, 4]
```