

O‘ZBEKISTON RESPUBLIKASI AXBOROT
TEKNOLOGIYALARI VA KOMMUNIKASIYALARINI
RIVOJLANTIRISH VAZIRLIGI

MUHAMMAD AL-XORAZMIY NOMIDAGI
TOSHKENT AXBOROT TEKNOLOGIYALARI
UNIVERSITETI
FARG‘ONA FILIALI

X.Sh.MUSAYEV

KOTLIN DASTURLASH TILI

(O‘quv qo‘llanma)

Farg‘ona – 2021

UDK 004.43
BBK 22.1

Ushbu o'quv qo'llanma hozirgi kundagi zamonaviy dasturlash tillaridan biri bo'lgan Kotlin dasturlash tiliga bag'ishlangan. Kotlin dasturlash tili eng mashhur dasturlash tillaridan biri hisoblanadi. Kotlin dasturlash tili ommalashganligiga asosiy sabablardan biri bu mobil ilovalar yaratishdagi asosiy tillardan biri bo'lganidadir. Kotlin dasturlash tili platforma tanlamaydigan tillar sarasiga kiradi. Kotlin dasturlash tilida yozilgan buyruqlar ketma-ketligi Windows operatsion tizimida, Linux operatsion tizimida yoki MacOS kabi operatsion tizimlarda ishlashi mumkin.

Kotlin dasturlash tili yuqori darajali interaktiv va obyektga yo'naltirilgan dasturlash tillaridan biri hisoblanadi. Ushbu qo'llanma Kotlin dasturlash tilini boshlovchilari uchun keng qo'llaniladigan sodda ma'lumotlardan tashkil topgan.

Tuzuvchi: – X.Sh.Musayev, Muhammad al–Xorazmiy nomidagi
TATU Farg'ona filiali Dasturiy injiniring kafedrası
o'qituvchisi

Taqrizchilar: – T.X.Tojiyev, Farg'ona davlat universiteti Axborot
texnologiyalari kafedrası mudiri, fizika-matematika fanlari
nomzodi, dotsent

– B.A.Mirzakarimov, Muhammad al–Xorazmiy nomidagi
TATU Farg'ona filiali Dasturiy injiniring kafedrası
dotsenti, fizika-matematika fanlari nomzodi

*Ushbu o'quv qo'llanma Muhammad al–Xorazmiy
nomidagi TATU Farg'ona filiali Kengashining
2021 yil 30 sentabr kungi 2 – sonli yig'ilishida
muhokama qilingan va chop etishga tavsiya
etilgan.*

1. KIRISH

1.1. Kotlin nima?

Kotlin¹ – dasturlash tili obyektga yo‘naltirilgan dasturlash tili hisoblanib, Java Virtual Machinening yuqori qismida ishlaydigan va JetBrains tomonidan ishlab chiqarilgan dasturlash tili hisoblanadi. Shuningdek, bu dasturlash tili JavaScript va Low Level Virtual Machine (LLVM – Quyi darajadagi virtual mashina) infratuzilmasi orqali bir qator platformalarda bajariladigan kodlardan tuzilgan.

Kotlin dasturlash tili Java dasturlash tili kabi yozilgan dasturni bayt kodga kompilyatsiya qiladi. Ya’ni, Java dasturlash tilida bo‘lgani kabi bu dasturlash tili ham Java Virtual Machine o‘rnatilgan joyda ishlay oladi. **Kotlin** dasturlash tilida yozilgan dasturlarni Windows, Linux, Mac OS, iOS va Android platformalarida ishlatish mumkin.

Tilning birinchi versiyasi 2016-yil 15-fevralda ommaga taqdim qilingan. Lekin tilni rivojlantirish 2010-yildan amalga oshirilgan. Hozirgi vaqtda tilning 1.5.31 versiyasi (2021-yil 20-sentabr ishlab chiqarilgan) ishlatilmoqda.

Til sintaksistida Pascal, TypeScript, Haxe, PL/SQL, F#, Go, Scala, C++, Java, C#, Rust va D dasturlash tillari elementlaridan foydalanilgan. O‘zgaruvchilar va parametrlar e‘lon qilishda o‘zgaruvchining nomidan keyin ma‘lumot turi ko‘rsatilishi kerak. Ko‘p dasturlash tillarining kompilyatorlari buyruq tugaganini bilish uchun nuqtali verguldan foydalanadi. **Kotlin** dasturlash tilida buyruqlarni alohida qatorlarga yozishda nuqtali vergul (;) ishlatilishi shart emas. Agar ikki yoki undan ortiq buyruqlar bir qatorda yozilsa, unda nuqtali vergul ishlatilishi kerak bo‘ladi. Bu dasturlash tili obyektga yo‘naltirilgan yondashuvdan tashqari funksiyalardan foydalanilgan holda oddiy ketma–ketlikdagi jarayonlar yordamida yaratilgan kodlarni kompilyatsiya qilish uchun ham mo‘ljallangan. C/C++ va D dasturlash tillari kabi bu tilda ham oddiy va sodda dasturlar tuzish uchun **main** funksiyasidan foydalaniladi. **Kotlin** dasturlash tili shuningdek Perl va Shell dagi kabi satrlarning interpolyatsiya²sini ham tushuna oladi. **Kotlin** dasturlash tilida **type inference** (avtomatik tipni aniqlash) ni ham ishlatish mumkin.

Kotlin dasturlash tili ishlatiladigan eng mashhur sohalardan biri, birinchi navbatda Android platformasi hisoblanadi. Shu qadar mashhurki

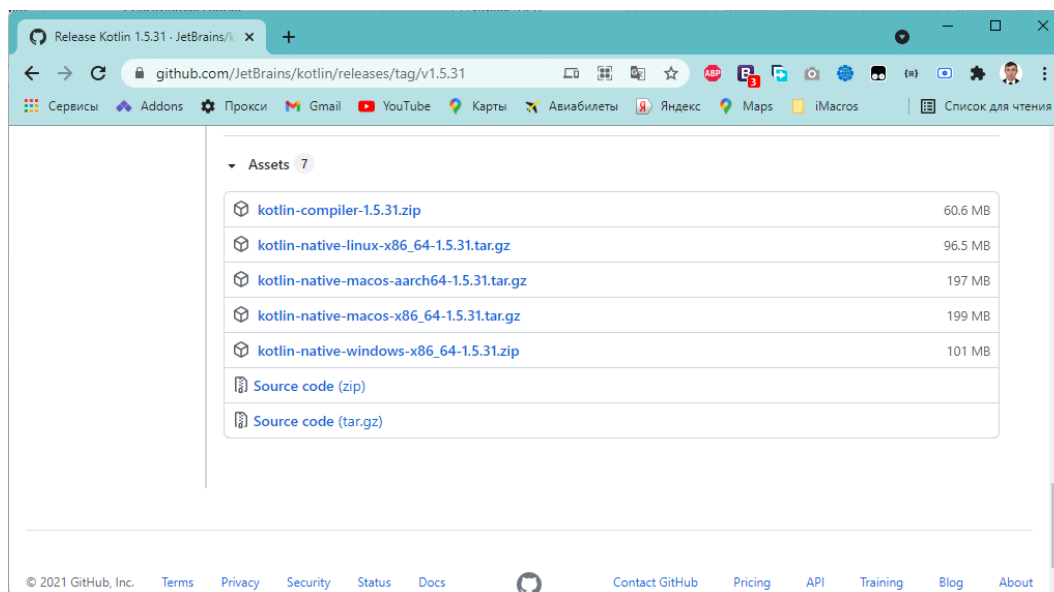
¹ Bu tilga Finlyandiya ko‘rfazidagi Kotlin orolining nomi berilgan

² Interpolyatsiya bu – satrga kiritilgan o‘zgaruvchini uning qiymati bilan almashtirish

Google I/O 2017 konferensiyasida **Kotlin** dasturlash tilini **Java** va **C++** dasturlash tillari bilan bir qatorda Android uchun rasmiy tillardan biri deb e'lon qildi. Android Studio dasturining 3-versiyasidan boshlab ushbu til bilan ishlay oladigan kompilyatsiya fayllari muhitiga odatiy ravishda kiritildi. **Kotlin** dasturlash tilining rasmiy veb sayti <https://kotlinlang.org/> hisoblanib, unda dasturchiga kerakli bo'lgan ma'lumotlarni olish mumkin.

1.2. Kotlin kompilyatorini o'rnatish va sozlash

Kotlin dasturlash tilining oxirgi versiyasini <https://github.com/JetBrains/kotlin/releases/tag/v1.5.31> havola orqali ko'chirib olish mumkin. Bu havolani brauzerga kiritilganida foydalanuvchi ishlatayotgan operatsion tizimga mos ravishda kompilyatorlar ro'yxati gavdalanadi.

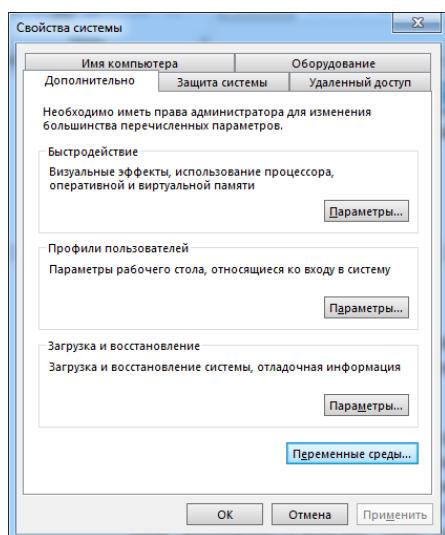


1.1–rasm: Kotlin dasturlash tilining kompilyatori ko'chirib olish.

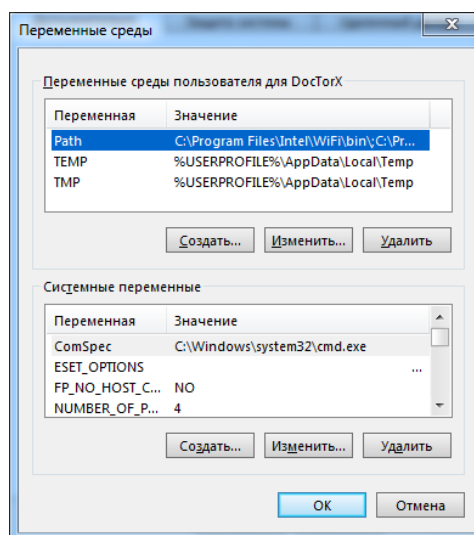
Ko'rinib turgan ro'yxatdagi fayllardan Windows operatsion tizim uchun `kotlin-compiler-1.5.31.zip` faylini ko'chirib olinadi. Ko'chirib olingan faylni arxivdan chiqariladi. Masalan, **Program Files** katalogini ichiga **Kotlin** nomdagi yangi katalog yaratilib, uning ichiga arxiv fayli ichidagi katalog va fayllar ko'chiriladi.

Kotlin dasturlash tilining versiyasini ko'rish uchun **Командная строка** (yoki **Windows PowerShell**) oynasiga `kotlinc -version` buyrug'ini yozishga to'g'ri keladi. Bu buyruq ishlamasligi mumkin, sababi operatsion tizim `kotlinc` buyrug'ini tanimaydi. Bu buyruqni tanishi uchun operatsion tizimdagi Мой компьютер => Свойства siga kiriladi va chap tomondagi «Дополнительные параметры системы» bo'limi tanlanadi. Shunda 1.2–rasmdagi oyna hosil bo'ladi.

Bu oynadan **Переменные среды** tugmasi bosiladi (1.3–rasm), hosil bo‘lgan oynaning **Системные переменные** bo‘limidan **PATH** o‘zgaruvchisini (переменная) qidirib, uni belgilanadi va **Изменить** tugmasi bosiladi.

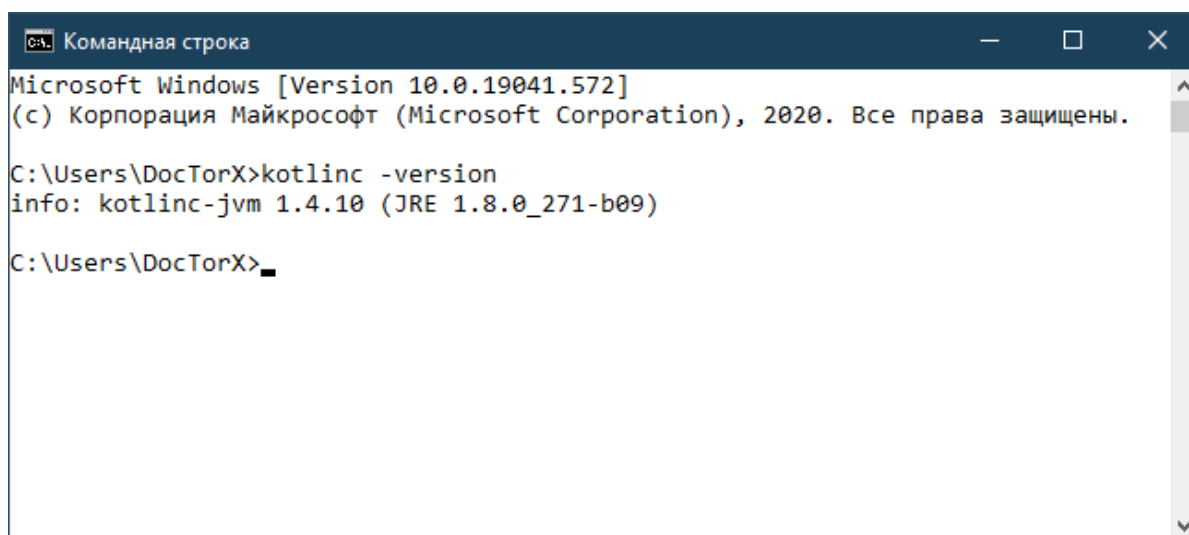


1.2–rasm: Свойства системы oynasi



1.3-rasm: Переменные среды oynasi

Shundan so‘ng **Значение переменной** dagi qiymatlarning oxiriga o‘tiladi va **C:\Program Files\Kotlin\bin** manzili kiritiladi. Hosil bo‘lgan barcha oynalardagi **OK** tugmalarini bosib oynalardan chiqiladi va o‘zgarishlar kompyuterga saqlanadi.



1.4–rasm: Kotlin dasturlash tilining versiyasini aniqlash

Yuqoridagi ishlarni bajarilganidan so‘ng **Командная строка** (yoki **Windows PowerShell**) oynasiga **kotlinc -version** buyrug‘ini yozib dasturlash tilining versiyasi aniqlanadi. Versiyani aniqlash dasturlash tili kompyuterda ishlayotganini bildiradi. **kotlinc** — bu **Kotlin** dasturlash

tilining kompilyatoridir. **Kotlin** dasturlash tilida yozilgan kodni bayt kodga aylantirib beruvchi dastur **kotlinc** bo‘lib, boshqa utilitlar kabi **bin** katalogida joylashgan bo‘ladi. Agar hamma ketma–ketlik to‘g‘ri bajarilgan bo‘lsa, 1.4–rasmda keltirilgan yozuv chiqadi, aks holda agar “**kotlinc** не является внутренней или внешней командной, исполняемой программой или пакетным файлом” kabi xatolik chiqsa, «Переменные среды» ni tekshirib chiqish lozim bo‘ladi.

1.3. Kotlin tilida Hello World dasturini yaratish

Kotlin dasturlash tilida birinchi dastur yozish uchun, oddiy **Блокнот**, **Notepad++** yoki **Sublime Text** dasturidan foydalanish mumkin. **Kotlin** dasturlash tilida yozilgan buyruqlar ketma–ketligini ***.kt** kengaytmasi bilan saqlanadi. **Sublime Text** dasturi yordamida **Kotlin** dasturlash tilida birinchi dastur quyidagicha yoziladi.

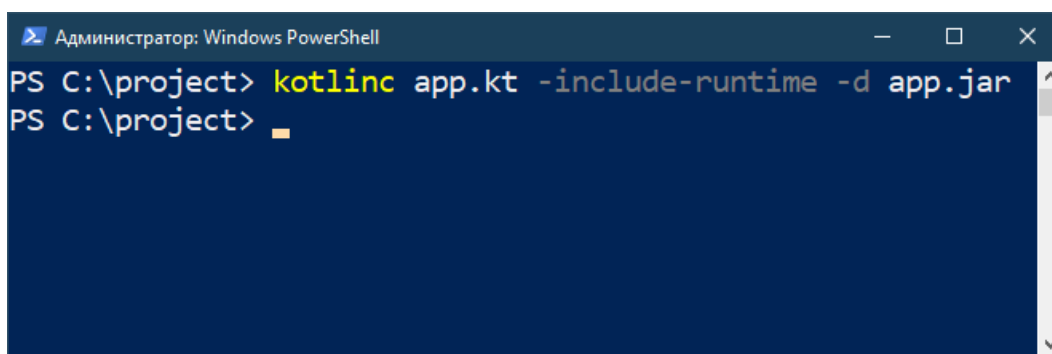


1.5–rasm: Sublime Text dasturi yozilgan dastur kodi

1.5–rasmda **fun** xizmatchi so‘zi yordamida **main** asosiy funksiyasi e‘lon qilinmoqda. Asosiy funksiyaning tana qismi **Java** dasturlash tilidagi kabi figurali qavslar ichida yoziladi. **println** funksiyasi yordamida **Hello World** so‘zi dasturning natijasi sifatida ekranda paydo bo‘ladi.

Bu yozilgan buyruqlar ketma–ketligi bitta fayl sifatida, ya’ni **app.kt** nomi bilan saqlab, **Командная строка** (yoki **Windows PowerShell**) oynasida kompilyatsiya qilinadi. Kompilyatsiya qilish uchun quyidagi buyruq beriladi.

```
kotlinc app.kt -include-runtime -d app.jar
```



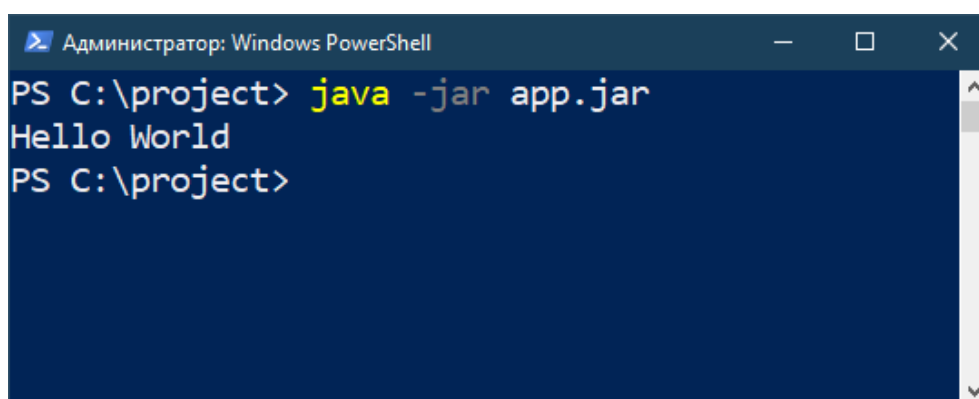
```
Администратор: Windows PowerShell
PS C:\project> kotlinc app.kt -include-runtime -d app.jar
PS C:\project>
```

1.6–rasm: Kotlinida yozilgan kodni kompilyatsiya qilish

Yuqoridagi yozilgan buyruqlar ketma–ketligi yordamida **Kotlin** dasturlash tilida yozilgan buyruqlar ketma–ketligi kompilyatsiya qilinib, ***.jar** fayl ko‘rinishga keltiriladi. Yaratilgan faylni ishga tushirish uchun Java dasturlash tilining buyruqlaridan foydalaniladi. Bu buyruq quyidagicha:

```
java -jar app.jar
```

Buyruq ishga tushirilganda ekranda quyidagicha ma’lumotlar hosil bo‘ladi (1.7–rasm).



```
Администратор: Windows PowerShell
PS C:\project> java -jar app.jar
Hello World
PS C:\project>
```

1.7–rasm: Yaratilgan jar faylni ishlatish

1.4. Kotlin dasturlash tilining muhitlari

Kotlin dasturlash tilining muhitlari bir nechta bo‘lib, ularning asosiylari quyidagilar hisoblanadi:

1. IntelliJ IDEA
2. TryKotlin
3. Android Studio
4. Vim
5. Sublime Text
6. Visual Studio Code

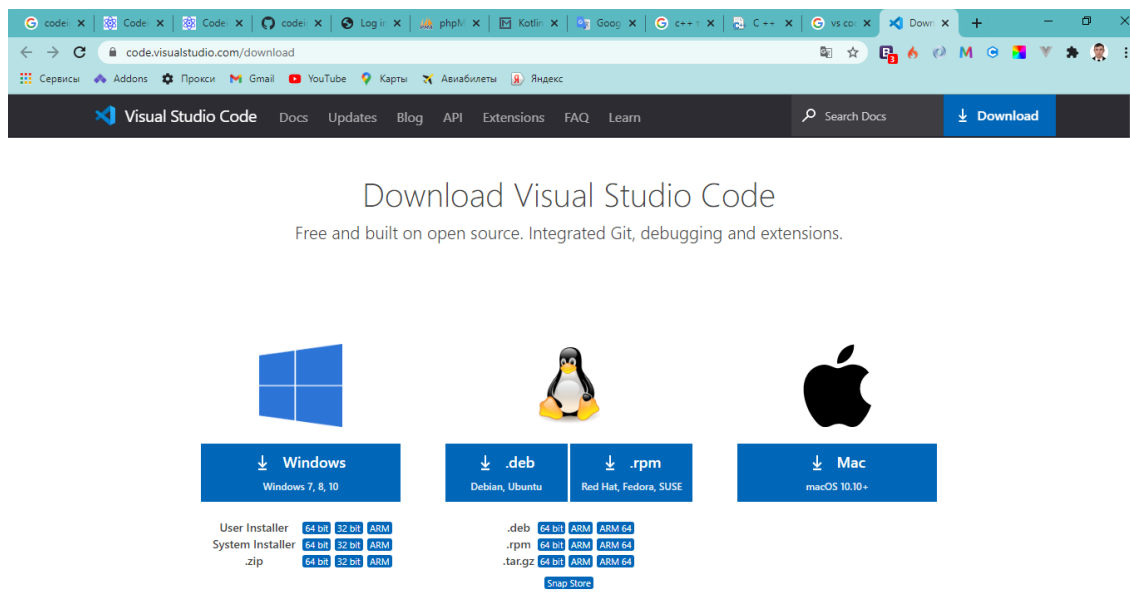
Yuqorida keltirilgan muhitlardan eng mashhurlari bu IntelliJ IDEA, Android Studio va Visual Studio Code lar hisoblanadi. Quyida IntelliJ

IDEA va Visual Studio Code muhitlarda **Kotlin** dasturlash tilini ishlatish va undan natija olish ko'rsatib o'tilgan.

Visual Studio Code muhitini o'rnatish va sozlash

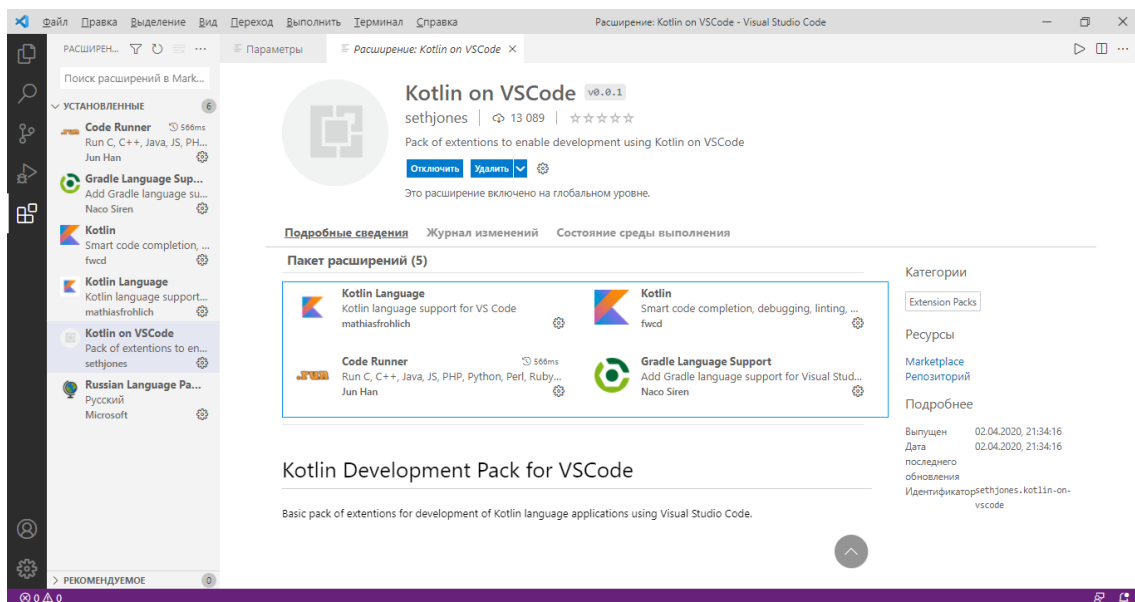
Visual Studio Code – **Kotlin** dasturlash tilida ishlash imkoniyatini beruvchi muhitlardan biri hisoblanadi. Ushbu muhitning boshqa IDElardan ustun tomonlari mavjud. Masalan, Veb ilovalar yaratish uchun ishlatiladigan qo'shimcha xizmatlar o'rnatish mumkin. Hozirgi kunda **Visual Studio** o'zining soddalashtirilgan **Visual Studio Code** dasturini foydalanuvchilarga taqdim etgan. Bu dastur yordamida foydalanuvchi o'ziga tegishli bo'lgan xizmatlarni butun jahon tarmog'idan yuklab olib, bu dastur orqali ishlatishi mumkin, ya'ni **VS Code** dasturi **IDE** vazifasini bajarib beradi.

VS Code dasturini <https://code.visualstudio.com/download> manzili orqali yuklab olinadi. Yuklangan faylni ishga tushirib, dasturni kompyuterga o'rnatiladi.



1.8–rasm: VS Code dasturi uchun xizmatlar

Visual Studio Code ishga tushgach, **Kotlin** dasturlash tilining xizmatini o'rnatiladi. Buning uchun dasturning chap tomonida joylashgan **Extensions** bandi tanlanadi. Xizmatni qidirish yordamida topib o'rnatiladi. Xizmatning nomi **Kotlin on VSCode** deb nomlangan. Bu xizmat **Visual Studio Code** dasturiga o'rnatilganidan so'ng **Kotlin** dasturlash tilining kodlari yoziladi. Yangi fayl yaratish uchun **File** (Файл) → **New** (“Новая файл”) bandlari tanlanadi.



1.9–rasm: VS Code dasturiga **Kotlin on VSCode** xizmatini o‘rnatish

Yangi yaratilgan faylni Hello.kt nomi bilan saqlanadi. Saqlangan fayl bo‘sh bo‘lgani uchun uning ichiga quyidagi **Kotlin** dasturlash tilida yozilgan kod yoziladi:

```
fun main() {
    println("Hello Kotlin from Visual Studio Code")
}
```

Visual Studio Code dasturida yozilgan kodni ishlashini tekshirish uchun Ctrl+Alt+N tugmalarini birgalikda bosiladi. Dasturning ostki qismida dasturning natijasi hosil bo‘ladi.

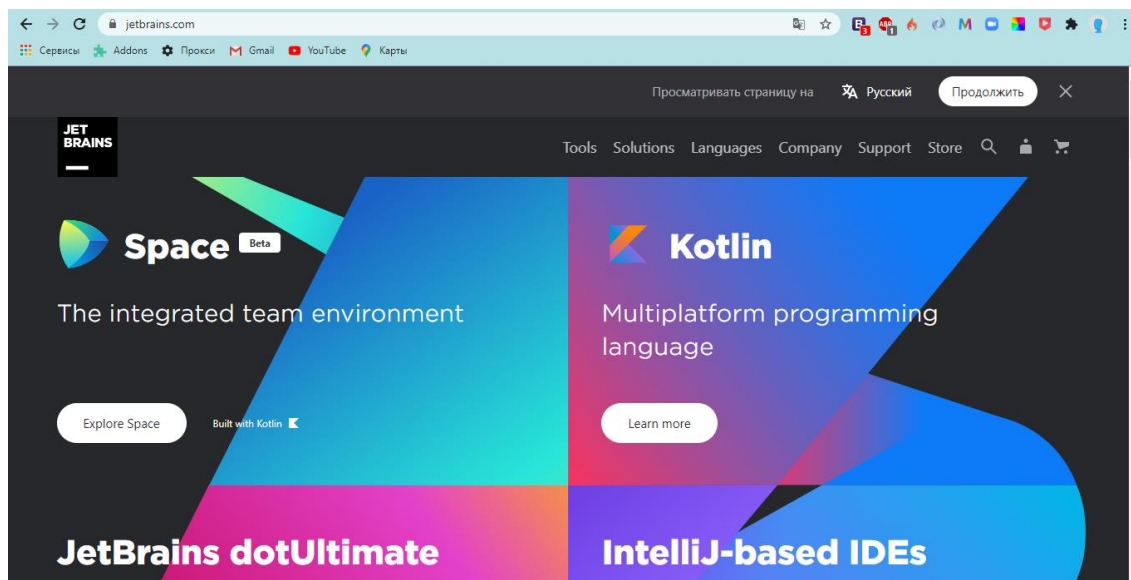
```
PROBLEMS  ВЫХОДНЫЕ ДАННЫЕ  TERMINAL  КОНСОЛЬ ОТЛАДКИ  Code
[Running] cd "c:\Users\DocTorX\" && kotlinc hello.kt
-include-runtime -d hello.jar && java -jar hello.jar
Hello Kotlin from Visual Studio Code

[Done] exited with code=0 in 6.771 seconds
```

1.10–rasm: Visual Studio Code dasturida olingan natija

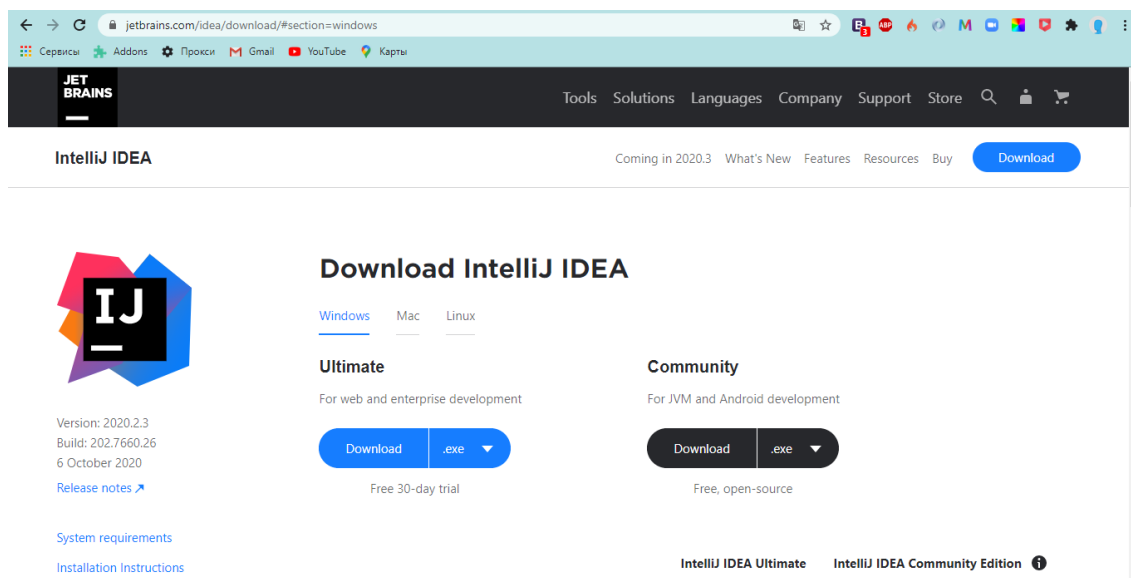
IntelliJ IDEA muhitini o‘rnatish va sozlash

Kotlin dasturlash tili **JetBrains** firmasi tomonidan yaratilganligi sababli bu dasturlash tilining asosiy muhiti shu firmaning mahsuloti bo‘lgan **IntelliJ IDEA** hisoblanadi. Bu muhitni o‘rnatish uchun JetBrains firmasining rasmiy veb sayti <https://www.jetbrains.com> manzilidan muhitning asosiy o‘rnatiluvchi faylini ko‘chirib olinadi. Veb saytning ko‘rinishi 1.11–rasmda keltirilgan.



1.11–rasm: JetBrains firmasining rasmiy veb sayti

1.11–rasmda ko‘rsatilgan sayt oynasidan Tools menyusidagi IntelliJ IDEA bandini tanlab hosil bo‘lgan sahifadagi Download tugmasi chertiladi. Shunda 1.12–rasmda keltirilgan sahifa hosil bo‘ladi.



1.12–rasm: IntelliJ IDEA dasturini yuklab olish oynasi

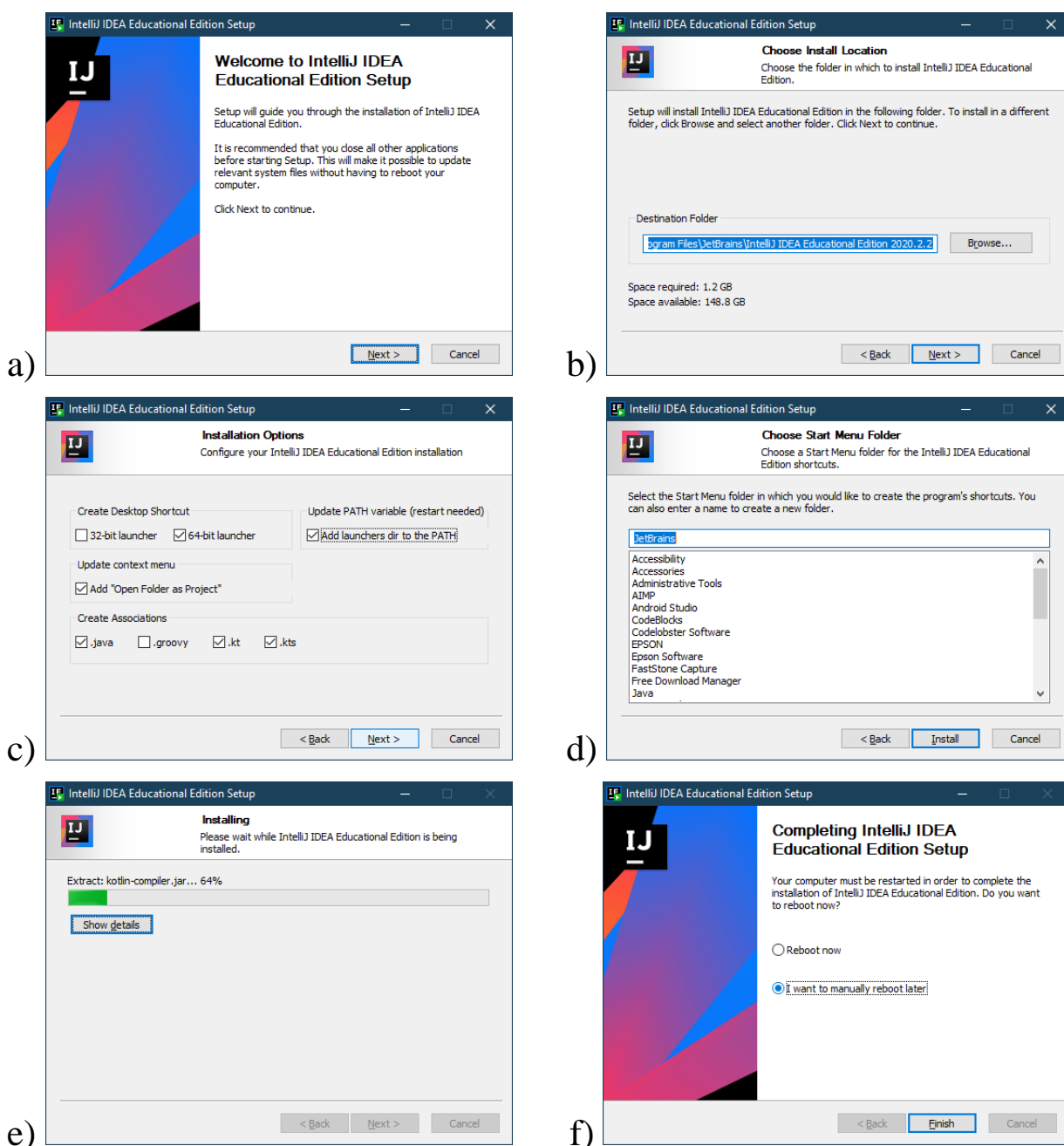
IntelliJ IDEA dasturi turli xil platformalar uchun ishlaydigan yagona platformaga ega bo‘lgan dastur hisoblanib, foydalanuvchining platformasiga mos keladigan dasturni yuklab olishi mumkin. IntelliJ IDEA dasturi Windows, Mac va Linux operatsion tizimlari bir ko‘rinishga ega bo‘lgan dastur interfeysi taqdim etadi. Bundan tashqari JetBrains firmasi bu dasturni ikki xil variantda foydalanuvchilarga taqdim etgan.

Birinchi ko‘rinishi **IntelliJ IDEA Ultimate** bo‘lib, bu ko‘rinish pulli ya’ni foydalanuvchilar tomonidan sotib olishga mo‘ljallangan bo‘ladi. Bu ko‘rinishini olgandan so‘ng foydalanuvchiga 30 kunga bepul ishlatishga

ruhsat beradi va undan keyin yiliga va foydalanuvchilar soniga qarab o'zining narxini taqdim etadi.

Ikkinchi ko'rinishi **IntelliJ IDEA Community** bo'lib, bu ko'rinish bepul ko'rinishda bo'ladi. Ultimate dan farqi bu ko'rinishda ayrim cheklovlar mavjud. Bu ko'rinishni asosan yangi o'rganuvchilar ishlatadi yoki o'qitish uchun ishlatilsa ham bo'ladi.

IntelliJ IDEA dasturi bir vaqtning o'zida bir nechta dasturlash tillari bilan ishlash xususiyatiga ega. Bu dastur yordamida Java, Kotlin, Groovy va Scala dasturlash tillari bilan ishlash mumkin.



1.13–rasm: **IntelliJ IDEA** dasturini o'rnatish jarayoni

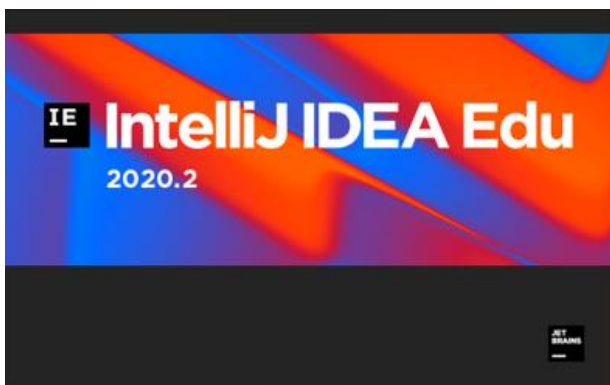
IntelliJ IDEA dasturini o'rnatish oynalari 1.13–rasmda ko'rsatilgan.

- 1.13a – rasmda IntelliJ IDEA muhitini oʻrnatishga tayyorgarlik oynasi hisoblanib, bu oynada IntelliJ IDEA muhitini qay tarzdagisi oʻrnatilayotgani va oʻrnatgandan soʻng kompyuterni qayta yuklash haqida axborot bermoqda. Oʻrnatilayotgan muhitning toʻliq nomi **IntelliJ IDEA Educational Edition**. Bu oʻqitishga asoslangan muhit xisoblanadi.
- 1.13b – rasmda IntelliJ IDEA muhitini oʻrnatilayotgan katalog nomi koʻrsatilgan. Xohlasa foydalanuvchi bu ketma–ketlikni oʻzgartirishi mumkin. Bu oynada oʻrnatilayotgan diskdagi boʻsh joy hajmi va oʻrnatilayotgan dasturning hajmi koʻrsatilgan.
- 1.13c–rasmdagi oyna oʻrnatishni tanlash oynasi hisoblanib, IntelliJ IDEA muhitini kerakli boʻlgan qoʻshimcha xizmatlaridan foydalanish uchun foydalanuvchiga turli koʻrinishdagi variantlarni taqdim etadi.
- 1.13d–rasmdagi oynada «Пуск» menyusida qaysi nomdagi katalog yoki papkada joylashishini koʻrsatish uchun foydalanuvchidan kerakli boʻlgan maʼlumot soʻraydi. Agar foydalanuvchi koʻrsatilgan nomga rozi boʻlsa, Next tugmasini bosib keyingi oynaga oʻtishni bildiradi.
- 1.13e–rasmdagi oynada IntelliJ IDEA oʻrnatilish jarayoni ketayotganini bildiruvchi progress koʻrsatilgan. Agar foydalanuvchi qanday maʼlumotlar yoki fayllarni oʻrnatayotgani bilan qiziqsa Show details tugmasini bosib koʻrishi mumkin.
- 1.13e–rasmda IntelliJ IDEA muhiti oʻrnatilganligini bildiruvchi oyna keltirib oʻtilgan. Bu oynada kompyuterni qayta yuklashni ikki xil varianti keltirilgan. Birinchi koʻrinishi bu **Reboot now** boʻlib, **Finish** tugmasi bosilishi bilan kompyuter qayta yuklash jarayoniga oʻtishni bildiradi. Ikkinchi koʻrinish **I want to manually reboot later** boʻlib, **Finish** tugmasi bosilishidan soʻng oʻrnatilish jarayoni yakunlanadi. Foydalanuvchi xohlagan vaqtda kompyuterga qayta yuklash buyrugʻini berishini bildiradi.

IntelliJ IDEA dasturini ishga tushirish uchun quyidagi ketma–ketlik amalga oshiriladi.

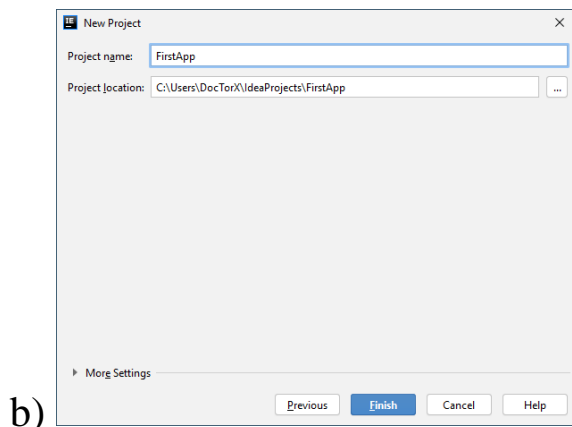
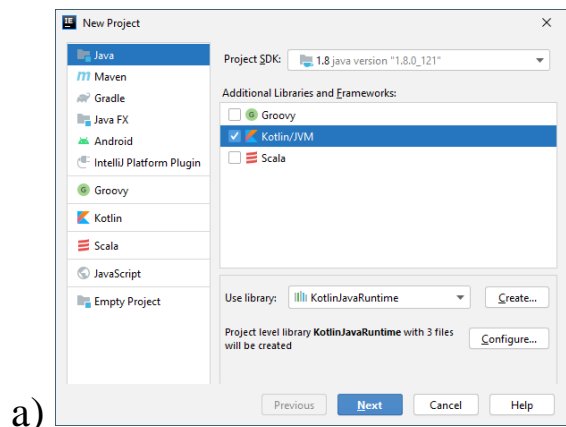
ПУСК => Все программы => JetBrains => IntelliJ IDEA Educational Edition

Dastur ishga tushganda 1.14–rasmdagi kirish oynasi hosil boʻladi.



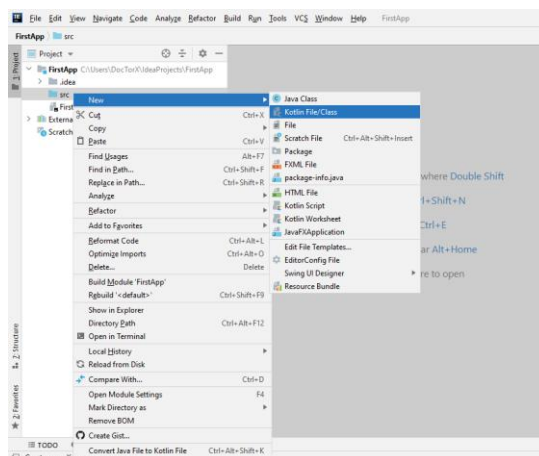
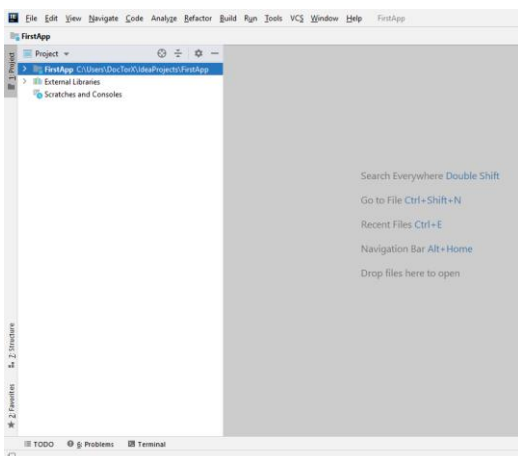
1.14–rasm: IntelliJ IDEA Edu dasturning kirish oynasi va yangi loyiha yoki mavjud loyihani ochish oynasi

IntelliJ IDEA muhitida yangi loyiha yaratish Welcome to IntelliJ IDEA oynasidagi New Project tugmasi orqali amalga oshiriladi. Bu tugma chertilganda 1.15a–rasmda keltirilgan oyna hosil bo‘ladi. Bu oynadan Additional Libraries and Frameworks bo‘limidagi Kotlin/JVM bandini tanlab Next tugmasi chertiladi. Shundan so‘ng 1.15b–rasmda keltirilgan oyna hosil bo‘ladi. Bu oynadagi Project name bandiga yaratilayotgan ilovaning nomini kiritish lozim. Ilovaning nomi kiritilganidan so‘ng Finish tugmasi chertiladi.



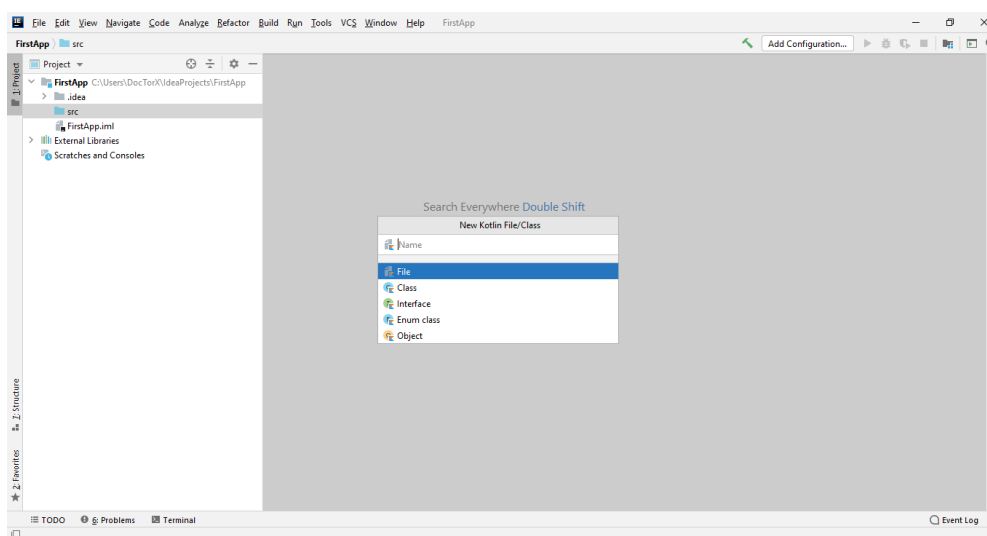
1.15–rasm: IntelliJ IDEA muhitida yangi loyiha yaratish oynalari

Yaratilgan yangi loyihada hech qanday fayl yoki klass mavjud bo‘lmaydi (1.16a–rasm). Bo‘sh loyihaga kerakli bo‘lgan faylni loyihaning ichida joylashgan **src** papkasining ustida sichqoncha ko‘rsatgichini joylashtirib, sichqonchaning o‘ng tugmasini chertib, hosil bo‘lgan menyudan New bo‘limi tanlanadi. Keyingi hosil bo‘lgan ostki menyudan Kotlin File/Class bandi tanlanadi. Shunda 1.17–rasmdagi oyna hosil bo‘ladi.



1.16–rasm: Yangi loyiha uchun kerakli fayllar yaratish

Hosil bo‘lgan oynada Kotlin dasturlash tilida yaratilishi kerak bo‘lgan File, Class, Interface, Enum class va Object bo‘limlari mavjud. Bu bo‘limlardan kerakli bo‘lganini tanlab loyihani to‘ldiriladi. Bu holatda File bandi tanlanib yangi fayl yaratiladi.

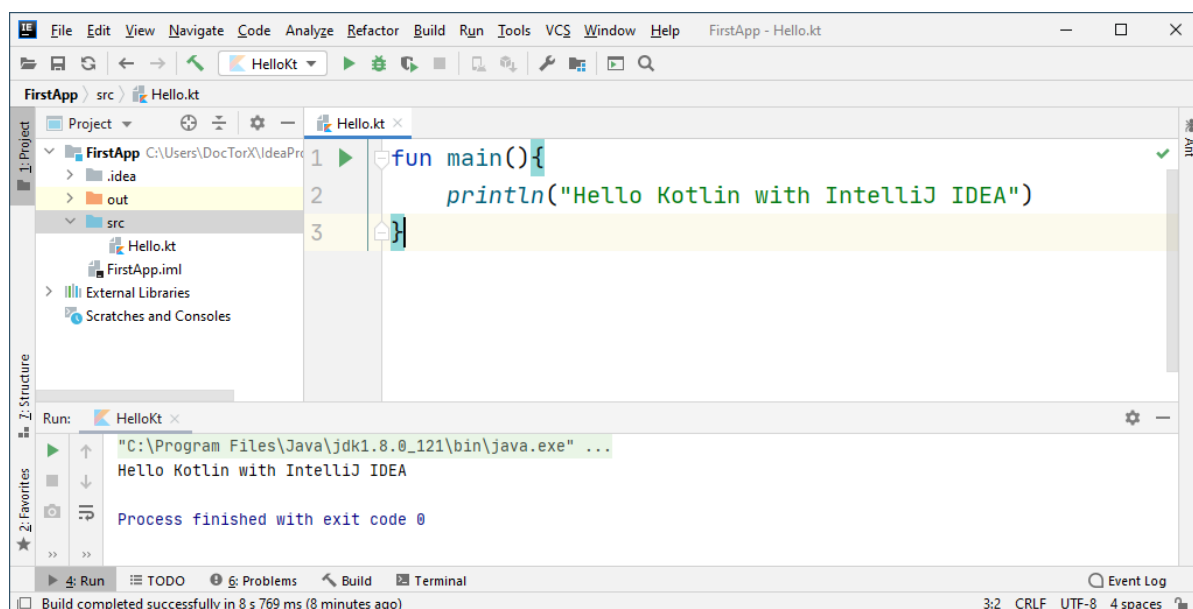


1.17–rasm: Yangi loyihaga fayl yoki boshqa a’zo yaratish oynasi

Yaratilgan faylga quyidagi yozilgan **Kotlin** dasturlash tilidagi buyruqlar ketma–ketligi yozililadi.

```
fun main() {
    println("Hello Kotlin with IntelliJ IDEA")
}
```

Yuqorida keltirilgan ketma–ketlikni IntelliJ IDEA muhitiga kiritilganidan so‘ng Ctrl+Shift+F10 tugmalari birgalikda bosilib natija olinadi (1.18–rasm).



1.18–rasm: Loyiha kodlari va uning natijasi

IntelliJ IDEA muhitining natija oynasi shu muhitning ostki qismida joylashgan bo‘lib, foydalanuvchilarga oynadan chiqib ketmasdan shu oynaning o‘zida natijalarni ko‘rish va tahlil qilishni osonlashtiradi.

Nazorat savollari:

1. Kotlin dasturlash tilining nomi qayerdan kelib chiqqan?
2. Kotlin dasturlash tilining ishlatiladigan sohalarini sanab bering.
3. Kotlin dasturlash tilining birinchi versiyasi qachon ishlab chiqilgan?
4. Kotlin dasturlash tili kim tomonidan ishlab chiqilgan?
5. Kotlin dasturlash tili sintaksisida qaysi dasturlash tillarining uslublaridan foydalanilgan?
6. Kotlin dasturlash tilining asosiy muhitlari sanab bering.
7. Kotlin dasturlash tilini ishlatish uchun VS Code dasturiga qanday o‘zgartirish kiritish kerak?

2. Kotlin dasturlash tilining asoslari

2.1. O'zgaruvchilar

Kotlin dasturlash tilida ma'lumotlarni saqlash uchun boshqa dasturlash tillaridagi kabi o'zgaruvchilardan foydalanadi. O'zgaruvchi ma'lum bir qiymatni saqlash uchun xotiraning nomlangan qismi hisoblanadi. Har bir o'zgaruvchi ma'lum bir nom, ma'lumot turi va qiymatdan iborat bo'ladi. O'zgaruvchining nomi identifikator hisoblanib, **identifikator** bu – lotin harfi yoki ostki chiziq (_) belgisi bilan boshlanuvchi, lotin harfi, raqamlar va ostki chiziq belgisining ixtiyoriy ketma-ketligiga aytiladi. **Kotlin** dasturlash tilida o'zgaruvchini aniqlash uchun **val** yoki **var** kalit so'zlaridan foydalanish mumkin. Bu ikki kalit so'zlarning farqi shundaki, **val** bilan e'lon qilingan o'zgaruvchi, o'zgarmas o'zgaruvchi, **var** bilan e'lon qilingan o'zgaruvchi, o'zgaruvchan o'zgaruvchi deb yuritiladi.

O'zgarmas o'zgaruvchi bu – dastur davomida o'zining qiymatini o'zgartirmaydigan va o'zgarishiga yo'l qo'ymaydigan o'zgaruvchidir.

O'zgaruvchan o'zgaruvchi bu – dastur davomida o'zining qiymatini o'zgartirib turadigan o'zgaruvchidir.

Quyida o'zgarmas o'zgaruvchiga misol keltirilgan.

```
fun main(args: Array<String>) {  
    val age = 23  
    println(age)  
}
```

Agar **val** ko'rinishda e'lon qilingan o'zgaruvchiga boshqa qiymat beriladigan bo'lsa, xatolik yuzaga keladi (2.1–rasm).

```
fun main(args: Array<String>) {  
    val age = 23  
    println(age)  
    age = 56  
    println(age)  
}
```

2.1–rasm: O'zgarmas o'zgaruvchi qayta qiymat berish

2.1–rasmda ko'rinib turibdiki qayta qiymat berishda integrallashgan muhitning o'zi xatolik haqida qizil chiziq yordamida xabar bermoqda. **val** bilan e'lon qilingan o'zgaruvchi bir marta qiymat qabul qiladi va boshqa

qiymat qabul qilinishiga yo‘l qo‘ymaydi. O‘zgarmas o‘zgaruvchini oldin e‘lon qilib, dasturning ixtiyoriy joyida bir marta qiymat berish ham mumkin.

O‘zgaruvchan o‘zgaruvchi yuqorida aytilganidek **var** kalit so‘zi bilan e‘lon qilinadi. Quyida **var** kalit so‘zi yordamida e‘lon qilingan o‘zgaruvchiga misol keltirilgan.

```
fun main(args: Array<String>) {
    var age = 23
    println(age)
    age = 56
    println(age)
}
```

Yuqoridagi kodlar ketma–ketligida **age** o‘zgaruvchisi birinchi bo‘lib, 23 qiymatni o‘ziga o‘zlashtirib oladi, bu o‘zgaruvchi o‘ziga biriktirgan qiymatni ekranga chiqarilgandan so‘ng boshqa qiymat ya‘ni, 56 ni o‘zlashtiradi va uning ham qiymatini ekranga chiqarganidan so‘ng dastur o‘z ishini yakunlaydi.

2.2. Ma‘lumotlar turlari

Kotlin dasturlash tilida har bir o‘zgaruvchining o‘ziga xos turi mavjud bo‘lib, ushbu turdagi o‘zgaruvchilar ustida turli amallar bajarish imkonini beradi. **Kotlin** dasturlash tilida Java dasturlash tili va boshqa tillardagi kabi ichki sodda turlar mavjud emas, barcha turlar ma‘lum bir klassni ifodalaydi. **Kotlin** dasturlash tilida sonli tur, mantiqiy tur, belgili tur, satri tur va ixtiyoriy turlar mavjud.

Sonli turlar

Tur nomi	Qiymatlar oralig‘i	Xotiradan oladigan joy
Byte	–128 ... 127	1 bayt
Short	–32 768 ... 32 767	2 bayt
Int	–2 147 483 648 ... 2 147 483 647	4 bayt
Long	–9 223 372 036 854 775 808 ... 9 223 372 036 854 775 807	8 bayt
Float	$-3.4 \cdot 10^{38} \dots 3.4 \cdot 10^{38}$	4 bayt
Double	$\pm 5.0 \cdot 10^{-324} \dots \pm 1.7 \cdot 10^{308}$	8 bayt

Literal o'zgaruvchilar va qiymatlar

Literal qiymatlar asosan **Kotlin** dasturlash tilida o'zgarmas o'zgaruvchilarga qiymat sifatida beriladi (ba'zan ularni doimiy qiymatlar deb ham yuritiladi). Literal qiymatlar mantiqiy, sonli, haqiqiy, belgili va satrli qiymat bo'lishi mumkin.

Butun sonlarni ifodalaydigan har qanday o'zgaruvchilar **Int** ma'lumotlar turi sifatida e'lon qilinadi. Quyida butun sonni qabul qiluvchi o'zgarmas o'zgaruvchi ko'rsatilgan:

```
val age: Int = 45
```

Katta butun sonlarni ifodalash uchun o'zgaruvchilar **Long** ma'lumotlar turi sifatida e'lon qilinadi va unga qiymat sifatida berilayotgan sonning oxiriga "L" qo'shimchasi qo'yiladi.

```
val age: Long = 125L
```

Xuddi shu tarzda suzuvchi vergulli sonlarni (haqiqiy sonlar) ham o'zgaruvchilarga qiymat sifatida berish mumkin. O'zguruvchiga qiymat berish uchun Double ma'lumotlar turidan foydalanish mumkin.

```
val weight: Double = 68.71
```

Suzuvchi vergulli sonlarni ishlatishda yana bir ma'lumotlar turidan foydalanish mumkin. Bu ma'lumotlar turining nomi Float bo'lib, bu turda e'lon qilingan o'zgaruvchining qiymatiga Long turida ishlatilganidek qo'shimcha qo'shiladi. Bu qo'shimcha F hisoblanib, quyida misol keltirilgan.

```
val weight: Float = 158.17F
```

Kotlin dasturlash tilida o'nlik sanoq sistemasidagi sonlardan tashqari ikkilik va o'n oltilik sanoq sistemalaridagi sonlarni o'zgaruvchilar qiymat sifatida o'zlashtirish mumkin.

Sonlarni o'n oltilik sanoq sistemasida ifodalash uchun son **0x** bilan boshlanishi, 0 dan 9 gacha arab raqamlari va A dan F gacha bo'lgan lotin harflari ishlatilishi kerak bo'ladi.

Sonlarning ikkilik sanoq sistemasida yozish uchun **0b** old qo'shimcha bilan boshlab, 0 va 1 sonlaridan foydalanish kerak.

Yuqorida keltirilgan ikki sanoq sistemasidagi sonlar asosan butun sonlarga ishlatiladi. Masalan,

```
val age: Int = 0x1A    // 26
val a: Int = 0b0101    // 5
val b: Int = 0b1011    // 11
```

Mantiqiy tur

Barcha dasturlash tillarida bo‘lgani kabi **Kotlin** dasturlash tilida ham mantiqiy ifodalarning qiymatlarini va mantiqiy qiymatlarni saqlash uchun ishlatiladigan tur mavjud. Bu turning nomi **Boolean** deb nomlanadi. Bu tur o‘zida ikkita qiymat saqlash xususiyatiga ega. Bu ikki qiymat **true** (rost) va **false** (yolg‘on).

```
val a: Boolean = true
val b: Boolean = false
```

Belgili kattaliklar

Belgili kattaliklarni ifodalash uchun **Char** ma’lumotlar turidan foydalaniladi. Belgi deganda apostrof (‘) ichida yozilgan bitta **Unicode** tizimidagi ma’lumotga aytiladi. **Unicode** tizimida 65536 ta belgi mavjud.

```
val a: Char = 'A'
val b: Char = 'B'
val c: Char = 'T'
```

Shuningdek **Kotlin** dasturlash tilida **C++** dasturlash tilidagi kabi boshqaruvchi belgilar mavjud. Bu belgilar quyidagi belgilar hisoblanadi.

\t	Bo‘sh joylar majmuasi (tabulatsiya)
\n	Yangi qatorga tushish (Enter belgisi)
\r	Yangi qatorga tushish (Return belgisi)
\'	Apostrof belgisi
\"	Qo‘shirnoq belgisi
\\	Teskari chiziq (Back Slash)

Satrlı kattaliklar

Satrlı kattaliklar **String** ma’lumot turi bilan ifodalanib, satrlar asosan qo‘shirnoq ichiga olingan yoki 3 tali qo‘shirnoqni ichiga olingan shaklda ko‘rsatiladi.

```
fun main(args: Array<String>) {
    val name: String = "Xurshidbek"
    println(name)
}
```

Satrlı kattaliklar boshqaruv belgilarini o‘z ichiga oladi. Bunga misol sifatida “\n” belgisini o‘z ichiga olgan biron–bir satrni ko‘rsatish mumkin. Bu belgi satrda qatnashgan bo‘lsada ekranga chiqarish vaqtida ko‘rinmaydi. Uning o‘rniga belgi qatnashgan joydan boshlab, satrning

qolgan qismi ikkinchi qatorga chiqariladi, ya'ni bu belgi Enter vazifasini bajarib beradi.

```
val text: String = "Salom TATU Talabalari.\nSizlarni kirib kelayotgan Navro'z ayyomi bilan tabriklayman."
```

Agar satr umumiy olganda bir nechta qatordan tashkil topgan bo'lsa, u holda bu satrni 3 lik qo'shtirnoqdan foydalaniladi.

```
val text: String = ""Salom TATU Talabalari. Sizlarni kirib kelayotgan Navro'z ayyomi bilan tabriklayman. ""
```

Andoza yordamida satrlar yaratish

Andoza yordamida satrlar yaratish uchun turli ko'rinishdagi qiymatlardan foydalanish mumkin. Xususan bu usulda o'zgaruvchan o'zgaruvchidan foydalaniladi. Andoza yordamida satrlar yaratish uchun dollar (\$) belgisi ishlatiladi, misol:

```
fun main(args: Array<String>) {  
    val firstName = "Xurshidbek"  
    val lastName = "Musayev"  
    val fullString = "Salom $firstName $lastName"  
    println(fullString)  
}
```

Bunday holda, satrning ichki qismidagi \$firstName, \$lastName o'zgaruvchining o'rniga uning qiymatlari joylashtiriladi va yaratilayotgan satrning tipini ko'rsatish shart emas.

Any turi

Any turi **Kotlin** dasturlash tilidagi boshqa barcha turlarning asosi hisoblanadi. Ya'ni, **Int** yoki **Double** ma'lumotlar turi **Any** turidan kelib chiqqan. Shuning uchun ushbu turdan foydalanish vaqtida o'zgaruvchi ixtiyoriy turdagi qiymatlarni qabul qilish xususiyatiga ega.

```
var name: Any = "Xurshidbek Musayev"  
name = 1985
```

2.3. Arifmetik amallar

Boshqa dasturlash tillaridagi kabi **Kotlin** dasturlash tilida ham, arifmetik amallar mavjud. Bu arifmetik amallarga qo'shish, ayirish, ko'paytirish va shunga o'xshash bir nechta amallarni ko'rsatish mumkin.

+ (qo'shish). Bu ikki sonning yig'indisini qaytaradi.

```
val x = 5  
val y = 6  
val z = x + y  
println(z)    // z = 11
```

– (ayrish). Bu ikki sonning orasidagi farqni qaytaradi.

```
val x = 5
val y = 6
val z = x - y
println(z)    // z = -1
```

* (ko‘paytirish). Bu ikki sonning ko‘paytmasini qaytaradi.

```
val x = 5
val y = 6
val z = x * y
println(z)    // z = 30
```

/ (bo‘lish). Bu ikki sonning bo‘linmasini qaytaradi.

```
val x = 60
val y = 10
val z = x / y
println(z)    // z = 6
```

% (qoldikli bo‘lish). Bu ikki sonni bo‘lgandan hosil bo‘lgan qoldiq qismi.

```
val x = 65
val y = 10
val z = x % y
println(z)    // z = 5
```

++ (inkrement). O‘zgaruvchining qiymatini bittaga orttirish. Inkrement amali ikki xil ko‘rinishga ega bo‘lib, prefiks va postfiks ko‘rinishidir. Quyida prefiks ko‘rinishdagi inkrement keltirilgan.

```
var x = 5
val y = ++x
println(x)    // x = 6
println(y)    // y = 6
```

Inkrementning postfiks ko‘rinishi quyida berilgan.

```
var x = 5
val y = x++
println(x)    // x = 6
println(y)    // y = 5
```

-- (dekrement). O‘zgaruvchining qiymatini bittaga kamaytirish. Dekrement amali inkrement amaliga o‘xshash ikki xil ko‘rinishga ega bo‘lib, prefiks va postfiksdur. Quyida prefiks ko‘rinishdagi dekrement keltirilgan.

```
var x = 5
val y = --x
println(x)    // x = 4
println(y)    // y = 4
```

Dekrementning postfiks ko‘rinishi quyida berilgan.

```

var x = 5
val y = x--
println(x)      // x = 4
println(y)      // y = 5

```

Yuqorida keltirilgan amallardan tashqari **Kotlin** dasturlash tilida qiymat berishli amallar ham mavjud. Bu amallarni quyidagilardan iborat.

+= (qiymat berishli qo'shish). Bu amal amalning chap qismidagi operantga, o'ng tomonida turgan operant qo'shib, chiqqan natija chap tomondagi operantga qiymat sifatida qaytaradi. Ya'ni, chap tomondagi operantning qiymatini o'ng tomondagi operantga ortirish.

-= (qiymat berishli ayirish). Bu amal amalning chap qismidagi operantga, o'ng tomonida turgan operant ayirilib, chiqqan natija chap tomondagi operantga qiymat sifatida qaytaradi. Ya'ni, chap tomondagi operantning qiymatini o'ng tomondagi operantga kamaytirish.

***=** (qiymat berishli qo'shish). Bu amal amalning chap qismidagi operantga, o'ng tomonida turgan operant ko'paytirilib, chiqqan natija chap tomondagi operantga qiymat sifatida qaytaradi. Ya'ni, chap tomondagi operantning qiymatini o'ng tomondagi operantga marta ortirish.

/= (qiymat berishli bo'lish). Bu amal amalning chap qismidagi operantga, o'ng tomonida turgan operant bo'linib, chiqqan natija chap tomondagi operantga qiymat sifatida qaytaradi. Ya'ni, chap tomondagi operantning qiymatini o'ng tomondagi operantga marta kamaytirish.

%= (qiymat berishli qoldikli bo'lish). Bu amal amalning chap qismidagi operantga, o'ng tomonida turgan operant bo'linib, qolgan qoldiq chap tomondagi operantga qiymat sifatida qaytaradi. Ya'ni, chap tomondagi operantning qiymatini o'ng tomondagi operantga marta kamaytirish orqali qolgan qoldiqni topish.

Bitlar ustida amallar bajaruvchi operatorlar

Dasturlash tillarida bitli amallar nomidagi operatorlar mavjud. **Kotlin** dasturlash tili ham bu operatorlardan holi emas. **Kotlin** dasturlash tilidagi bitli amallar faqat **Int** va **Long** ma'lumot turlarida e'lon qilingan o'zgaruvchilar yoki qiymatlar ustida amallar bajaradi. **Kotlin** dasturlash tilida bu operatorlar quyidagicha ko'rinishga ega.

shl (shift left (signed shift left) – chapga surish). Bu operator butun sonlar ustida bitlarni ishorali chapga surish vazifasini bajaradi.

```

val z = 3 shl 2           // z = 11 << 2 = 1100
println(z)                 // z = 12
val d = 0b11 shl 2
println(d)                 // d = 12

```

Yuqorida keltirilgan misolda 3 soni oldin ikkilik ko‘rinishga keltirilib chap tomonga ikki xona suriladi, ya’ni bu sonning o‘ng tomoniga ikkita 0 raqamini yozish bilan teng bo‘ladi. 3 sonining ikkilik ko‘rinishi 11 ga teng. 11 ning o‘ng qismiga ikkita 0 raqamini joylashtiradigan bo‘lsak, 1100 ko‘rinishdagi son hosil bo‘ladi. Bu sonning ikkilikdagi ko‘rinishi bo‘lib, bu sonning 10 lik sanoq sistemasidagi ko‘rinishi 12 soniga teng bo‘ladi.

shr (shift right (signed shift right) – o‘ngga surish). Bu operator butun sonlar ustida bitlarni ishorali o‘ngga surish vazifasini bajaradi.

```
val z = 12 shr 2          // z = 1100 >> 2 = 1100
println(z)               // z = 3
val d = 0b1100 shr 2
println(d)               // d = 3
```

Yuqorida keltirilgan misolda 12 soni oldin ikkilik ko‘rinishga keltirilib o‘ng tomonga ikki xona suriladi, ya’ni bu sonning chap tomoniga ikkita oxirgi raqamini o‘chirish bilan teng bo‘ladi. 12 sonining ikkilik ko‘rinishi 1100 ga teng. 1100 ning chap qismidagi ikkita 0 raqamini o‘chiradigan bo‘lsak, 11 ko‘rinishdagi son hosil bo‘ladi. Bu sonning ikkilikdagi ko‘rinishi bo‘lib, bu sonning 10 lik sanoq sistemasidagi ko‘rinishi 3 soniga teng bo‘ladi.

ushr (unsigned shift right – o‘ngga surish). Bu operator butun sonlar ustida bitlarni o‘ngga surish vazifasini bajaradi.

```
val z = 12 ushr 2        // z = 1100 >> 2 = 1100
println(z)               // z = 3
```

and (bitlar ustida ko‘paytirish yoki kon’yunksiya). Bu operator ikki butun sonning mos bitlari ustida mantiqiy ko‘paytirish amalini bajaradi. Ya’ni sonlarning har biri ikkilik ko‘rinishga keltirilib, mos bitlari mantiqiy ravishda ko‘paytiriladi. Mantiqiy ko‘paytirish quyida berilgan jadval asosida amalga oshiriladi.

A	B	A and B
1	1	1
1	0	0
0	1	0
0	0	0

Bu jadvaldan ko‘rinadiki, agar mos bitlar 1 ga teng bo‘lsa, ularning mantiqiy ko‘paytmasi ham 1 ga, qolgan hollarda esa 0 ga teng bo‘ladi.

```

val x = 5           // 101
val y = 6           // 110
val z = x and y      // z = 101 & 110 = 100
println(z)          // z = 4

```

```

val d = 0b101 and 0b110
println(d)          // d = 4

```

or (bitlar ustida qo'shish yoki diz'yunksiya). Bu operator ikki butun sonning mos bitlari ustida mantiqiy qo'shish amalini bajaradi. Ya'ni sonlarning har biri ikkilik ko'rinishga keltirilib, mos bitlari mantiqiy ravishda qo'shadi. Mantiqiy qo'shish quyida berilgan jadval asosida amalga oshiriladi.

A	B	A or B
1	1	1
1	0	1
0	1	1
0	0	0

Bu jadvaldan ko'rinadiki, agar mos bitlar 0 ga teng bo'lsa, ularning mantiqiy yig'indisi ham 0 ga, qolgan hollarda esa 1 ga teng bo'ladi.

```

val x = 5           // 101
val y = 6           // 110
val z = x or y       // z = 101 | 110 = 111
println(z)          // z = 7
val d = 0b101 or 0b110
println(d)          // d = 7

```

xor (bitlar ustida istisnoli qo'shish). Bu operator ikki butun sonning mos bitlari ustida mantiqiy istisnoli qo'shish amalini bajaradi. Ya'ni sonlarning har biri ikkilik ko'rinishga keltirilib, mos bitlari mantiqiy ravishda istisnoli qo'shadi. Mantiqiy istisnoli qo'shish quyida berilgan jadval asosida amalga oshiriladi.

A	B	A xor B
1	1	0
1	0	1
0	1	1
0	0	0

Bu jadvaldan ko‘rinadiki, agar mos bitlar 0 yoki 1 ga teng bo‘lsa, ularning mantiqiy istisnoli yig‘indisi 0 ga, qolgan hollarda esa 1 ga teng bo‘ladi.

```
val x = 5           // 101
val y = 6           // 110
val z = x xor y     // z = 101 ^ 110 = 011
println(z)         // z = 3
val d = 0b101 xor 0b110
println(d)         // d = 3
```

inv (inversiya). Bu operator emas metod hisoblanib, yuqorida ta’kidlanganidek butun sonlarning Int va Long turlarida mavjud bo‘lib, ko‘rsatilgan o‘zgaruvchining inversiyasini ko‘rsatib beradi. Inversiya bu – sonning bitlarini teskarisiga aylantirish degan ma’noni bildiradi.

```
val b = 11          // 1011
val c = b.inv()
println(c)          // -12
```

2.4. Solishtirish belgilari va shartli ifodalar

Ba’zi bir shartlarni o‘z ichiga olgan ifodalar shartli ifodalar hisoblanib, bu ifodalar turli ko‘rinishdagi solishtirish belgilaridan tashkil topadi. Bu ifodalarning natijaviy qiymati **Boolean** turiga mansub bo‘lib, bular asosan **true** (rost) va **false** (yolg‘on) qiymatlar qaytaradi.

Solishtirish belgilari

> (...dan katta) – agar birinchi operand ikkinchi operanddan katta bo‘lsa rost (**True**), aks holda yolg‘on (**False**) qiymat qaytaradi.

```
val a = 11
val b = 12
val c:Boolean = a > b
println(c)     // false - a kichik b dan
val d = 35 > 12
println(d)     // true - 35 katta 12 dan
```

< (...dan kichik) – agar birinchi operand ikkinchi operanddan kichik bo‘lsa rost (**True**), aks holda yolg‘on (**False**) qiymat qaytaradi.

```
val a = 11
val b = 12
val c = a < b   // true
val d = 35 < 12 // false
```

>= (...dan katta yoki teng) – agar birinchi operand ikkinchi operanddan kichik bo‘lmasa ya’ni, katta yoki teng bo‘lsa rost (**True**), aks holda yolg‘on (**False**) qiymat qaytaradi.

```
val a = 11
val b = 12
```

```
val c = a >= b    // false
val d = 11 >= a    // true
```

<= (...dan kichik yoki teng) – agar birinchi operand ikkinchi operanddan katta bo‘lmasa ya’ni, kichik yoki teng bo‘lsa rost (**True**), aks holda yolg‘on (**False**) qiymat qaytaradi.

```
val a = 11
val b = 12
val c = a <= b    // true
val d = 11 <= a    // false
```

== (teng) – agar ikkita operand bir–biriga teng bo‘lsa rost (**True**), aks holda yolg‘on (**False**) qiymat qaytaradi.

```
val a = 11
val b = 12
val c = a == b    // false
val d = b == 12    // true
```

!= (teng emas) – agar ikki operand bir–biriga teng bo‘lmasa rost (**True**), aks holda yolg‘on (**False**) qiymat qaytaradi.

```
val a = 11
val b = 12
val c = a != b    // true
val d = b != 12    // false
```

Mantiqiy operatorlar

Mantiqiy ifodalar bilan ishlash davrida solishtirish belgilaridan tashqari mantiqiy operatorlar ham ishlatiladi. Mantiqiy operatorlar mantiqiy qiymatlarning bir qanchasini birlashtirish imkonini beradi.

and (mantiqiy ko‘paytirish operatori) – ikki ifodaning qiymati rost bo‘lganda rost, qolgan hollarda yolg‘on qiymat qaytaruvchi mantiqiy operator hisoblanadi.

```
val age = 25
val weight = 67
val result = (age > 21) and (weight == 67)
println(result)    // true
```

Bu holatda **and** operatori ikki mantiqiy ifodaning natijasini mantiqiy ko‘paytiradi, bu ikki mantiqiy ifoda **age > 21** va **weight == 58**. Agar ikki mantiqiy ifoda rost bo‘lsa, umumiy natija rost (**true**) qiymat qaytaradi, agar mantiqiy ifodalardan biri yolg‘on bo‘lsa umumiy natija ham yolg‘on (**false**) bo‘ladi. Masalan:

```
val age = 25
val weight = 67
val isMarried = false
val result = (age > 21) and (weight == 67) and isMarried
println(result)
```

or (mantiqiy qo'shish operatori) – mantiqiy qo'shish operatori ikki operandning qiymati yolg'on bo'lsa, yolg'on, qolgan hollarda rost qiymat qaytaruvchi mantiqiy operator.

```
val age = 22
val isMarried = false
val result = (age > 21) or isMarried
println(result)
```

! (mantiqiy inkor operatori) – mantiqiy inkor operatori operandning qiymati rost bo'lsa, yolg'on, yolg'on bo'lsa, rost qiymat qaytaruvchi mantiqiy operatoridir.

```
var age = 22
var weight = 58
var isMarried = false
println(!(age > 21))           // false
println(isMarried)            // true
```

Mantiqiy inkor operatorining ikkinchi ko'rinishi bu har bir mantiqiy qiymatni o'zida saqlovchi obyektning **not()** metodi mavjud.

```
val a = true
val b = a.not()           // false
val c = b.not()           // true
```

xor (mantiqiy istisnoli qo'shish operatori) – mantiqiy istisnoli qo'shish operatori ikki operandning qiymati rost yoki yolg'on bo'lsa, yolg'on, qolgan hollarda rost qiymat qaytaruvchi mantiqiy operator.

```
val a = true
val b = false
val c = a xor b           // true
val d = a xor (90 > 10)   // false
```

in (to'plam yoki ketma-ketlikning ichida) – agar ko'rsatilgan operand ketma-ketlik yoki to'plamning ichida mavjud bo'lsa, rost qiymat, aks holda yolg'on qiymat qaytaradi.

```
val a = 5
val b = a in 1..6         // true
```

Yuqorida ko'rsatilgan misolda 1..6 bu ketma-ketlik hisoblanadi. a o'zgaruvchisining qiymati ketma-ketlikning ichida mavjudligi uchun rost qiymat qaytaradi.

2.5. Tarmoqlanuvchi jarayonlar

Tarmoqlanuvchi jarayonni shart operatorisiz tasavvur qilib bo'lmaydi. Shart operatori bu qandaydir bir shartga asosan ikki holatdan biri bajaruvchi tarmoqlanuvchi jarayon hisoblanadi. Tarmoqlanuvchi

jarayonlarni operatori if shart operatori deb yuritiladi. Bu operatorining ko‘rinishi quyidagicha:

```
if(mantiqiy ifoda yoki shart){
    buyruqlar ketma-ketligi
}
else{
    buyruqlar ketma-ketligi
}
```

Shart operatorining ikki xil ko‘rinishi mavjud bo‘lib, bu ko‘rinishlar to‘la shartli va chala shartli operatorlar deb yuritiladi. Chala shartli operatorida shart rost bo‘lsa, qandaydir buyruqlar ketma-ketligi bajarilib shart operatoridan keyingi qator ishlatiladi. Agar shart yolg‘on bo‘lsa, **if** operatoridan keyin kelgan qator ishga tushadi.

```
val a = 10
if(a == 10){
    println("a ning qiymati 10 ga teng.")
}
```

Yuqorida aytilganidek a o‘zgaruvchining qiymati 10 ga teng bo‘lsa, figurali qavslar ichida kelgan ma’lumot ekranga chiqariladi, agar o‘zgaruvchining qiymati 10 dan farqli bo‘lsa, figurali qavs ichida kelgan ma’lumotlar ekranga chiqarilmaydi.

Agar shart operatorida **else** operatori ishlatilsa, bunday ko‘rinishdagi operator to‘la shartli operator deb yuritiladi. To‘la shartli operatorga quyida keltirilgan dastur qismi misol bo‘ladi:

```
val a = 10
if(a == 10){
    println("a ning qiymati 10 ga teng.")
} else {
    println("a ning qiymati 10 ga teng emas.")
}
```

Dasturda ikki va undan ortiq shartlarni tekshirishga to‘g‘ri kelib qolganda **Kotlin** dasturlash tilida shart operatori bir necha marta ishlatiladi.

```
val a = 10
if(a == 10) {
    println("a ning qiymati 10 ga teng.")
}
else if(a == 9){
    println("a ning qiymati 9 ga teng.")
}
else if(a == 8){
    println("a ning qiymati 8 ga teng.")
}
```

```
else{
    println("a ning qiymati aniqlanmagan.")
}
```

Kotlin dasturlash tilida **if** operatori ma'lum bir funksiyaga o'xshab qiymat qayratish xususiyatiga ega.

```
val a = 10
```

```
val b = 20
```

```
val c = if (a > b) a else b
```

```
println(c)    // 20
```

Bu yerda **if** operatori Java yoki C++ dasturlash tilidagi ternar amali (?:) sifatida qo'llanilmoqda. Bundan tashqari **if** operatori bir vaqtning o'zida ekranga qandaydir ma'lumot chiqarib, ham qiymat qaytarib xususiyatiga ega. Bunday xolatda qaytariladigan qiymat har bir blokning oxirida ko'rsatilishi kerak.

```
val a = 10
```

```
val b = 20
```

```
val c = if (a > b) {
    println("a = $a")
    a
} else {
    println("b = $b")
    b
}
```

Variant tanlash operatori (when operatori)

Ayrim algoritmlarning hisoblash jarayonlari ko'p tarmoqliligi bilan ajralib turadi. Umuman olganda, tarmoqli jarayonlarni hisoblash uchun shartli operatoridan foydalanish yetarlidir. Lekin, tarmoqlar soni ko'p bo'lsa, shartli operatoridan foylanish algoritmnining ko'rinishini qo'pollashtirib yuboradi. Bu hollarda shartli operatorning umumlashmasi bo'lgan variant tanlash operatoridan foydalanish maqsadga muvofiqdir. Variant tanlash operatori quyidiga qoidalarga amal qiladi:

- **when** operatorining variantlari faqat butun sonlar, satrlar va to'plamlar bo'lishi mumkin;

- variantlar iboralardan tashkil topishi mumkin. Har bir variant yo'naltirgich (→) bilan tugaydi;

- variantlarning qiymati ifodaning qiymati bilan bir xil bo'lishi kerak;

- ifodaning qiymati topilganidan so'ng, bitta buyruq yoki buyruqlar to'plamini bajarilishidan iborat;

- bitta buyruq yoki buyruqlar to'plami tugagandan so'ng, **when** operatorlar blokidan keyingi qatorga o'tkazadi;

- agar bir nechta variantlar uchun bir xil operatorlar ketma–ketligi ishlatiladigan bo‘lsa, variantlar vergul bilan ajratilib yozilishi mumkin;
- agar variantlarni to‘plam sifatida beriladigan bo‘lsa, **in** kalit so‘zidan foydalanish kerak;
- variant tanlash operatorida berilgan variantlardan tashqari nazarda tutilmagan variantlar uchun **else** varianti mavjud.

```

when(<operator selektori>)
{
    <qiymat> -> <operator yoki operatorlar guruhi>
    <qiymat1>, <qiymat2> -> <operator>
    <qiymat> -> {
        <operatorlar guruhi>
    }
    in <to‘plam> -> {
        <operatorlar guruhi>
    }
    !in <to‘plam> -> {
        <operatorlar guruhi>
    }
    else -> <operator yoki operatorlar guruhi>
}

```

Yuqorida keltirilgan konstruksiyaga misol sifatida quyidagilarni ko‘rish mumkin.

```

val a = 10
when(a){
    10 -> println("a = 10")
    20 -> println("a = 20")
    else -> println("aniqlanmagan qiymat")
}

```

Keltirilgan misolda har bir variant uchun bittada operatori bajarilishi ko‘rsatib o‘tilgan. Variantlarni birlashtirish yoki operatorlar blokidan foydalanish quyidagi misol keltirilib o‘tilgan.

```

var a = 10
when(a){
    10, 15 -> {
        println("a = 10 yoki a = 15")
        a *= 2
    }
    20 -> {
        println("a = 20")
        a *= 5
    }
    else -> { println("aniqlanmagan qiymat") }
}
println(a)

```

Agar variantda ko'rsatilgan qiymatlar to'plam shaklda bo'lsa, variant tanlash operatori quyidagi ko'rinishda ifodalaniladi.

```
val a = 10
```

```
when(a) {  
    in 10..19 -> println("a ning qiymati 10 dan 19 gacha")  
    in 20..29 -> println("a ning qiymati 20 dan 29 gacha ")  
    !in 30..40 -> println("a ning qiymati 30 dan 40 gacha  
emas")  
    else -> println("aniqlanmagan qiymat")  
}
```

Yuqoridagi dasturda **in** buyrug'i ko'rsatilgan qiymatni berilgan to'plamning ichidagiligini ifodalasa, **in** buyrug'ini inkor etish (!) esa ko'rsatilgan qiymatning berilgan to'plamga tegishli emasligini ifodalaydi.

Variant tanlash operatori yordamida qiymat qaytarish

Kotlin dasturlash tilida variant tanlash operatori huddi tarmoqlanuvchi operatori singari qiymat qaytarish xususiyatiga ega. Buning uchun variant tanlash operatori yozishdan oldin qandaydir o'zgaruvchiga undan qaytadigan qiymatni olish kifoya etadi. Bunda operatorlar blokining o'rniga qiymatlar yoki qiymatini olish uchun o'zgaruvchi ishlatish mumkin.

```
val sum = 1000  
val rate = when(sum) {  
    in 100..999 -> 10  
    in 1000..9999 -> 15  
    else -> 20  
}
```

```
println(rate)          // 15
```

Yuqorida keltirilgan dastur qismida **sum** o'zgaruvchisining qiymati berilgan ikki to'plamning biriga tegishli bo'lsa, yo'naltiruvchi belgilardan keyingi qiymati, agar berilgan ikki to'plamga tegishli bo'lmasa, **else** operatoridagi yo'naltirgichdan keyingi qiymati olishi mumkin.

2.6. Takrorlanuvchi jarayonlar

Barcha dasturlash tillarida mavjud bo'lgani kabi **Kotlin** dasturlash tilida ham takrorlanuvchi jarayonlarni dasturlash mumkin. Buning uchun **Kotlin** dasturlash tili 3 turdagi takrorlanish operatorlarini taqdim etgan. Bu operator **for (parametrli)**, **while (sharti oldin berilgan yoki shartdan keyin bajariladigan)** va **do..while (sharti keyin berilgan yoki shartdan oldin bajariladigan)** takrorlanish operatorlari hisoblanadi.

for – parametrli takrorlanish operatori

Parametrli takrorlanish operatori ko'rsatilgan to'plamdagi qiymatlarga birma–bir murojaat qiluvchi takrorlanish operatori hisoblanadi. Bu operatorning ko'rinishi quyida berilgan:

```
for(o'zgaruvchi in ketma-ketlik){  
    operatorlar guruhi  
}
```

Misol uchun, 1 dan 9 gacha bo'lgan sonlarning kvadratlarini chiqaruvchi dastur quyidagi ko'rinishda bo'ladi.

```
for(n in 1..9){  
    print("${n * n} \t")  
}
```

Yuqoridagi dasturning qismi 1 dan 9 gacha bo'lgan sonlar ketma–ketligidagi har bir songa murojaat qilgan holda, uning kvadratlarini hosil qiladi va konsol oynasida quyidagicha natija olish mumkin.

```
1      4      9      16     25     36     49     64     81
```

Takrorlanish jarayonlarini ichma–ich ko'rinishda tashkil qilish ham mumkin. Misol uchun matematika kursidagi Pifagor karra jadvalini tuzish talab qilinsin. Bu jadvalni chiqarish uchun ikki takrorlanuvchi operatorni ichma–ich yozishdan foydalaniladi.

```
for(i in 1..9){  
    for(j in 1..9)  
        print("${i * j} \t")  
    println()  
}
```

while – sharti oldin berilgan takrorlanuvchi jarayon

Sharti oldin berilgan takrorlanuvchi operator dastur tuzishda ko'p ishlatiladigan takrorlanuvchi operatori hisoblanadi. Bu operator bir yoki bir nechta operatorlar guruhini, qo'yilgan shart yolg'on (false) bo'lgunga qadar bajaradi. Agar shart rost (true) bo'lsa, takrorlanuvchi jarayon o'z ishini bajaraveradi. Haqiqiy dasturlarda takrorlanuvchi jarayon ichida kamida 2 ta operator bo'ladi, bu operatorlardan biri biror buyruqni ifodalasa, ikkinchisi shartni o'zgartirish uchun ishlatiladigan operator bo'lishi mumkin. Shartni o'zgartirish uchun asosan inkrement yoki dekrement operatoridan foydalanish mumkin. Operatorlar ketma–ketligi 1 tadan ko'p bo'lsa, barcha operatorlar ketma–ketligi operatorlar blokiga olinadi. Sintaksisi quyidagicha:

```
while (<shart>) {  
    <operatorlar_bloki>  
}
```


Sharti oldin berilgan takrorlanish operatori yordamida 1 dan 9 gacha bo'lgan sonlarning kvadratini topish quyidagi dastur qismida berilgan.

```
var i = 9
while (i > 0) {
    println(i*i)
    i--;
}
```

Bu dastur 1 dan 9 gacha bo'lgan sonlar ketma-ketligi uchun emas, balki 1 dan 9 gacha bo'lgan sonlarga teskari tartibda murojaat qilib, ularning kvadratlarini ekranga chiqaradi. Ushbu dasturdagi **while** operatorining shartiga e'tibor berilsa, u yerda o'zgaruvchi 0 dan katta bo'lgan holatlar ko'rib chiqilmoqda, o'zgaruvchining qiymat 0 dan katta bo'lgan vaqtda takrorlanuvchi jarayon takrorlanishda davom etib, o'zgaruvchi 0 ga teng holatda takrorlanish tugatiladi.

do..while sharti keyin berilgan takrorlanish operatori

Yuqorida **while** takrorlanish operatorida, agar shart yolg'on bo'lsa, takrorlanish tanasi umuman ishlamasligini ko'rib chiqildi. Agar shart yolg'on bo'lsa ham, takrorlanish tanasidagi operatorlar bir marotaba bajarilishi kerak bo'lsa, **do..while** takrorlanish operatoridan foydalaniladi. Bu operatorida bajarilishidan oldin operatorlar bajariladi, so'ng takrorlanish uchun qo'yilgan shart tekshiriladi. Shu sababli takrorlanish tanasi kamida bir marotaba bajariladi. Shuning uchun ham bu takrorlanish operatorini **shartdan oldin bajariladigan operator** deb nomlash mumkin. Sintaksisi quyidagicha:

```
do {
    <operatorlar_bloki>
} while (<shart>)
```

Sharti keyin berilgan takrorlanish operatoriga quyidagi misolni keltirish mumkin.

```
var i = -1
do{
    println(i*i)
    i--;
}
while (i > 0)
```

Bu dastur bajarilish natijasida kamida takrorlanish operatori ishlashi hisobiga ekranga natija chiqariladi. Ammo shart tekshirilgan jarayonda o'zgaruvchining qiymati shartga mos bo'lmaganligi sababli takrorlanish operatori ishini tugatadi. Natijada esa ekranda 1 soni hosil bo'ladi.

continue va break operatori

Kotlin dasturlash tilida **break** operatori uzulishlarni boshqarish uchun ishlatiladi. Uzilish yuzaga keltirish uchun bu operatori takrorlanish operatori ichida ishlatish kerak bo'ladi. Agar takrorlanish operatori tanasida **break** operatori ishlatilsa, dastur o'z ishini takrorlanish operatoridan keyingi operatorlar ketma–ketligiga o'tkazadi.

Boshqa dasturlash tillari singari **Kotlin** dasturlash tilida ham **continue** operatori asosan sikl operatorlarining tana qismida ishlatiladi:

- **for** siklida asosan iteratorning yangi qiymatiga o'tishini ta'minlaydi;

- **while** operatori yoki **do–while** operatorida boshqaruv shartlar joylashgan qatorga o'tadi.

```
for(n in 1..8){  
    if(n == 5) continue;  
    print("${n * n} \t")  
}
```

Bu dasturning natijasi quyidagicha bo'ladi:

```
1    4    9    16   36   49   64
```

E'tiborlisi shundaki, 1 dan 8 gacha bo'lgan ketma–ketligidagi sonlarning kvadratini hisoblashda o'zgaruvchining qiymati 5 ga teng bo'lgan holatda uning kvadrati chiqarilmasdan, keyingi qiymatga o'tish uchun **continue** operatoridan foydalanilgan. Ya'ni ketma–ketlikning qaysidir qiymatini e'tiborga olmaslik kerak bo'lgan vaqtda bu operatoridan foydalanish maqsadga muvofiq bo'ladi.

Takrorlanish jarayonida uzilish tashkil etish quyidagi dasturda berilgan:

```
for(n in 1..10){  
    if(n == 5) break;  
    println(n * n)  
}
```

Bu dasturda 1 dan 10 gacha sonlarning kvadratini chiqarish nazarda tutilgan. Lekin takrorlanish operatorining ichida shart operatori yordamida uzilish tashkil etilgan. O'zgaruvchining qiymati 5 ga teng bo'lganda uzilish amalga oshirilib, ekranda 1 dan 4 gacha bo'lgan sonlarning kvadrlari chiqariladi. Chunki uzilish yuqorida aytib o'tilganidek buyruqlar ketma–ketligini takrorlanish operatori oxiriga yetmasdan tashqariga uzatib beradi.

2.7. Ketma–ketliklar

Kotlin dasturlash tilida ketma–ketliklarni tashkil etish uchun ketma–ket ikki nuqtadan (..) foydalaniladi. Bu operatoridan tashqari **downto**, **step** va **until** operatorlaridan ham foydalanish mumkin. Bu operatorlar haqida quyida berilgan ko‘rsatmalar orqali tushinish mumkin.

(..) – a..b, bu buyruq a dan b gacha bo‘lgan sonlardan tashkil topgan ketma–ketlikni qaytaradi. Bunda $a < b$ dan va b qiymat ham ketma–ketlikning tarkibiga kiritiladi;

downto – a downto b, bu buyruq b dan a gacha bo‘lgan sonlardan tashkil topgan ketma–ketlikni teskari tartibda qaytaradi. Bunda $a > b$ dan va b qiymat ham ketma–ketlikning tarkibiga kiritiladi;

until – a until b, bu buyruq a dan b gacha bo‘lgan sonlardan tashkil topgan ketma–ketlikni qaytaradi. Bunda $a < b$ dan va b qiymati ketma–ketlik tarkibiga kiritilmaydi;

step – bu buyruq yuqorida keltirilgan buyruqlarga qo‘shimcha buyruq hisoblanib, yaratilayotgan ketma–ketlikni hosil qilish uchun elementlar farqini ifodalash yoki qadamni ko‘rsatish uchun ishlatiladi. Bu buyruqda 0 va 0 dan kichik son ya’ni, manfiy sonlar ishlatish mumkin emas.

Keltirilgan buyruqlarni ishlash jarayoni quyida berilgan:

```
var range1 = 1..5           // [1, 2, 3, 4, 5]
var range2 = "a".."d"       // [a, b, c, d]
var range3 = 1..5           // 1 2 3 4 5
var range4 = 5 downto 1     // 5 4 3 2 1
var range5 = 1..10 step 2   // 1 3 5 7 9
var range6 = 10 downto 1 step 3 // 10 7 4 1
var range7 = 1 until 9      // 1 2 3 4 5 6 7 8
var range8 = 1 until 9 step 2 // 1 3 5 7
```

2.8. Massivlar

Massiv - bu bir xil tipli, chekli qiymatlarning tartiblangan to‘plamidir. Massivlarga misol sifatida matematika kursidan ma’lum bo‘lgan vektorlar, matritsalar va tenzorlarni ko‘rsatish mumkin. Dasturda ishlatiluvchi barcha massivlarga o‘ziga xos ism berish kerak. Massivning har bir hadiga murojaat esa, uning nomi va o‘rta qavs ichiga olib yozilgan tartib hadi orqali amalga oshiriladi:

```
<massiv_nomi> [<indeks>]
```

Kotlin dasturlash tilida massivlar **Array** turi yordamida e’lon qilinadi. Bunda **Array** turidan so‘ng burchakli qavslar orasida massiv

elementlarining turi ko'rsatilib o'tiladi. Masalan, quyida elementlari butun sonlardan iborat massiv e'lon qilish tartibi ko'rsatilib o'tilgan:

```
val numbers: Array<Int>
```

Agar massivga boshlang'ich qiymatlar beriladigan bo'linsa, unda **arrayOf** funksiyasidan foydalanish tavsiya etiladi. Masalan,

```
val numbers: Array<Int> = arrayOf(1, 2, 3, 4, 5)
```

Bu yerda 5 ta elementdan va elementlari 1 dan 5 gacha bo'lgan sonlardan iborat. **Kotlin** dasturlash tili **Java** dasturlash tiliga o'xshaganligi sababli massiv elementlarining indeksi 0 dan boshlanadi. Massiv elementlarini ustida turli amallarni bajarish mumkin. Bularga massiv elementlarini boshqa bir o'zgaruvchiga olish yoki massiv elementiga boshqa bir qiymat berishlar kiradi. Masalan,

```
val numbers: Array<Int> = arrayOf(1, 2, 3, 4, 5)
```

```
val n = numbers[1]
```

```
numbers[2] = 7
```

Kotlin dasturlash tilida massiv elementlariga bitta qiymatni berish ham mumkin. Masalan,

```
val numbers = Array(3, {5})
```

Bu yerda 3 ta elementdan iborat bo'lgan massiv e'lon qilinmoqda va barcha elementlariga 5 qiymati berilmoqda. Massiv e'lon qilish jarayonida **Array** sinfining konstruktoriga murojaat amalga oshirilgan. Ushbu konstruktor 2 ta parametrga ega bo'lib, birinchi parametr massiv elementlari sonini, ikkinchi parametr figurali qavslar orqali barcha elementlarga qiymat o'rnatish uchun ishlatilmoqda.

Massiv yaratishni soddalashtirish uchun **Kotlin** dasturlash tili ma'lum bir turlarga tegishli bo'lgan qo'shimcha turlardan foydalanadi. Bu turlarga **BooleanArray**, **ByteArray**, **ShortArray**, **IntArray**, **LongArray**, **CharArray**, **FloatArray** va **DoubleArray** turlari kiradi. Quyida ushbu turlardan **IntArray** va **DoubleArray** lardan foydalanish ko'rsatib o'tilgan. **IntArray** yordamida e'lon qilingan massiv **Int** turidagi elementlardan tashkil topgan massivni, **DoubleArray** yordamida esa **Double** turidagi elementlardan tashkil topgan massivni ifodalaydi.

```
val numbers: IntArray = intArrayOf(1, 2, 3, 4, 5)
```

```
val doubles: DoubleArray = doubleArrayOf(2.4, 4.5, 1.2)
```

Bu turdagi massivlarni e'lon qilishda va massiv elementlariga boshlang'ich qiymat berishda turga mos funksiyalardan foydalanish tavsiya etiladi. Bu funksiyalar qo'shimcha massiv turlarining nomlari bilan bir xil ko'rinishda bo'lib, birinchi harfi quyi registrda bo'lsa, oxiriga **Of**

qo‘shimchasi qo‘shilgan. Massiv yaratish uchun ishlatiladigan qo‘shimcha turlarning konstruktorlaridan ham foydalanish mumkin.

```
val numbers = IntArray(3, {5})
val doubles = DoubleArray(3, {1.5})
```

Ketma–ketliklardagi kabi **in** operatori yordamida berilgan qiymat massiv elementlarining ichida mavjud yoki yo‘qligini topish uchun xizmat qiladi.

```
val numbers: Array<Int> = arrayOf(1, 2, 3, 4, 5)
println(4 in numbers)      // true
println(2 !in numbers)     // false
```

Ikki o‘lchovli massivlar

Kotlin dasturlash tili bir o‘lchovli massivlardan tashqari ikki va undan ko‘p o‘lchovli massivlar bilan ham ishlay olish xususiyatiga ega. Bu dasturlash tilida ikki o‘lchovli massivni har bir elementi massivdan iborat bir o‘lchovli massiv sifatida qarash mumkin. Ikki o‘lchovli massivni inson hayotidagi jadvalga qiyoslash mumkin.

Kotlin dasturlash tilida ikki o‘lchovli massivni quyidagicha e‘lon qilish mumkin:

```
val table: Array<Array<Int>> = Array(3, { Array(5, {0}) })
```

Yuqorida keltirilgan massiv e‘lon qilish jarayonida birinchi navbatda 3 ta elementdan iborat bo‘lgan massiv e‘lon qilinadi. E‘lon qilingan massiv elementlari uchun 5 ta elementdan iborat bo‘lgan boshqa bir massiv e‘lon qilinib har bir elementi 0 ga tenglanadi. Sodaroq qilib aytganda 3 ta satr, 5 ta ustundan iborat bo‘lgan va elementlari 0 ga teng massiv e‘lon qilinadi.

Massiv e‘lon qilishda va uning boshlang‘ich qiymatlarini berishda quyidagicha yo‘l tutish ham mumkin:

```
val table = Array(3, { Array(3, {0}) })
table[0] = arrayOf(1, 2, 3)
table[1] = arrayOf(4, 5, 6)
table[2] = arrayOf(7, 8, 9)
```

Bu dastur qismida 3x3 bo‘lgan massiv e‘lon qilinmoqda va uning elementlari 0 bilan to‘ldirilmoqda. Shundan so‘ng massivning har bir satri uchun alohida–alohida qiymatlar o‘rnatilmoqda.

Massiv elementlariga murojaat qilish

Kotlin dasturlash tilida massiv elementlariga murojaat qilish takrorlanuvchi jarayon operatorlaridan bo‘lgan **for** operatori yordamida amalga oshiriladi. Bu operatoridan foydalanish quyidagi misolda keltirilgan:

```

val phones: Array<String> = arrayOf("Galaxy S8", "iPhone X",
"Motorola C350")
for(phone in phones){
    println(phone)
}

```

Yuqorida keltirilgan dastur kodida telefon versiyalaridan tashkil topgan massiv berilgan. Dasturdagi **for** operatori massivning har bir elementiga murojaat qilib, uning qiymatlarini ekranga yoki konsolga chiqarish uchun xizmat qilmoqda. Bu operator yangi **phone** nomdagi o‘zgaruvchiga **phones** nomdagi massiv elementlarini har biriga murojaat qilish vazifasini yuklaydi va u yordamida ko‘rsatilgan ko‘rsatmani bajarishni ta’minlaydi.

Agar massiv ikki o‘lchovli bo‘lsa, u holda **for** operatoridan ikki marta foydalanish tavsiya etiladi. Ikki o‘lchovli massivlar elementlariga murojaat quyidagi misolda ko‘rsatib o‘tilgan:

```

fun main(args: Array<String>) {
    val table: Array<Array<Int>> = Array(3, {Array(3, {0})})
    table[0] = arrayOf(1, 2, 3)
    table[1] = arrayOf(4, 5, 6)
    table[2] = arrayOf(7, 8, 9)
    for(row in table){
        for(cell in row)
            print("$cell \t")
        println()
    }
}

```

Nazorat savollari:

1. O‘zgaruvchi va o‘zgarmas deganda nimani tushunasiz?
2. O‘zgarmas o‘zgaruvchi bilan o‘zgaruvchan o‘zgaruvchini farqi nimada?
3. Ma’lumotlar turini sanab bering.
4. Arifmetik amallar va ularning turlarini ayting.
5. Solishtirish belgilarini sanab bering.
6. Tarmoqlanuvchi jarayonlarni boshqaruvchi operatorlarni tushuntirib bering.
7. Takrorlanuvchi jarayon hosil qiluvchi operatorlarni tushuntiring.
8. Ketma–ketlik deganda nimani tushunasiz?
9. Ketma–ketliklarga misol keltiring.
10. Kotlin dasturlash tilida ketma–ketliklar qanday ifodalanadi?
11. Massivlar va ularning turlari tushuntiring.
12. Kotlin dasturlash tilida massivlar qanday e’lon qilinadi?

3. Funksional dasturlash

3.1. Funksiyalar va ularning parametrlari

Funksiya – bu qism dastur bo‘lib, ma’lum bir vazifalarni bajarishga asoslangan buyruqlar ketma–ketligidan tashkil topadi. **Kotlin** dasturlash tilida funksiya **fun** xizmatchi so‘zi bilan e‘lon qilinib, bu xizmatchi so‘zdan so‘ng e‘lon qilinayotgan funksiyaning nomi yoziladi. Funksiyaning nomidan keyin oddiy qavslar qo‘yilib, uning ichiga parametrlar ro‘yxati vergullar bilan ajratilgan holda shakllantiriladi. Agar parametrlar ro‘yxati mavjud bo‘lmasa, qavslar ochib yopib qo‘yiladi. Ba’zida funksiyalar qiymat ham qaytarib turishi lozim bo‘lib, **Kotlin** dasturlash tilida qiymat qaytaruvchi funksiyalarni e‘lon qilinganda parametrlardan so‘ng qaytayotgan qiymatning turi ko‘rsatilishi kerak bo‘ladi.

```
fun funksiya_nomi(parametrlar): qaytariluvchi_qiymat{  
    ko'rsatmalar ketma-ketligi  
}
```

Parametrsiz funksiyalar

Barcha dasturlash tillarida bo‘lgani kabi **Kotlin** dasturlash tilida ham parametrsiz funksiyalar bilan ishlash, ularni e‘lon qilish mumkin. Quyidagi dastur parametrsiz funksiyaga misol bo‘la oladi:

```
fun main(args: Array<String>) {  
    hello() // hello nomli funksiyaga murojaat  
    hello() // hello nomli funksiyaga murojaat  
    hello() // hello nomli funksiyaga murojaat  
}  
// hello funksiyasini aniqlash  
fun hello(){  
    println("Assalomu alaykum o'rganuvchi")  
}
```

Yuqoridagi misolda **hello** nomli parametrsiz funksiya ketma–ket 3 marta chaqirilmoqda. Bunda **hello** nomi bilan aniqlangan funksiya dasturning asosiy funksiyasi bo‘lgan **main** nomli funksiyadan ya’ni, asosiy funksiyadan keyin aniqlangan. E‘lon qilinayotgan funksiya asosiy funksiyadan keyin aniqlanayotgan bo‘lsa ham, yuqori darajadagi funksiya bo‘lganligi sababli bu funksiya chaqiriladi va ekranga **hello** funksiyasida keltirilgan buyruqlar ketma–ketligi bajariladi.

Kotlin dasturlash tili lokal funksiyalardan tashqari yuqori darajali (top-level) funksiyalar bilan ham ishlay oladi. Yuqori darajadagi funksiyalarni e‘lon qilishda klass, obyekt yoki interfeysdan tashqarida e‘lon qilingan, to‘g‘ridan – to‘g‘ri fayl ichida aniqlangan funksiya nazarda

tutiladi. Yuqori darajadagi funksiyalar biror–bir tuzilmaning ichida joylashmaganligi sababli, ular klass va funksiyalar ierarxiyasidan yuqori qismda joylashadi.

Yuqoridagi dastur qismida keltirilgan funksiyalar yuqori darajali funksiyalar deb nomlanadi.

Paramertli funksiyalar

Kotlin dasturlash tilida funksiya chaqirilayotgan joyidan unga turli ko‘rinishdagi qiymatlarni berish mumkin. Bunda parametrlil funksiyalardan foydalanish tavsiya etiladi. Parametrlil funksiyalar e‘lon qilishda qabul qilinadigan qiymatning turi ko‘rsatib o‘tilishi shart. Parametrlil funksiyaning nomidan keyingi qavslar ichida vergul bilan ajratilgan holda, parametrlil nomi va ikki nuqta qo‘yilib, parametrlil turi ko‘rsatiladi. Quyida ko‘rsatilgan sonning faktorialini topuvchi parametrlil funksiya e‘lon qilingan:

```
fun main(args: Array<String>) {  
    factorial(4)  
    factorial(5)  
    factorial(6)  
}  
  
fun factorial(n: Int) {  
    var result = 1;  
    for(d in 1..n) {  
        result *= d  
    }  
    println("$n! = $result ga teng")  
}
```

Yuqorida keltirilgan funksiya bitta parametrdan iborat bo‘lgan butun qiymatli son qabul qiladi. Bu dastur bajarilganidan so‘ng konsolga quyidagi qiymatlar chiqadi:

```
4! = 24 ga teng  
5! = 120 ga teng  
6! = 720 ga teng
```

Yana bir boshqa funksiyanil misol sifatida keltirish mumkin. Keltiriladigan funksiyaning parametri yuqoridagi funksiyadagi kabi bitta emas, balki ikkita bo‘ladi. Bu funksiya foydalanuvchi va uning yoshi haqida axborot beradi.

```
fun main(args: Array<String>) {  
    displayUser("Toxir", 23)  
    displayUser("Aziza", 19)  
    displayUser("Kamola", 25)  
}
```



```
fun displayUser(name: String, age: Int) {
    println("Nomi: $name   Yoshi: $age")
}
```

Yuqoridagi dasturda keltirilgan **displayUser** nomli funksiya ikki parametr qabul qiladi. Birinchi parametr foydalanuvchining ismini, ikkinchi parametr esa uning yoshini tasvirlaydi. Dasturning asosiy funksiyasidan unga 3 marta murojaat etilayotganda, foydalanuvchining ismi va yoshi funksiyaga parametr sifatida jo‘natiladi. Jo‘natilgan qiymatlarni hisobga olgan holda, dastur konsolga quyidagi natijani chiqaradi:

```
Nomi: Toxir   Yoshi: 23
Nomi: Aziza   Yoshi: 19
Nomi: Kamola  Yoshi: 25
```

Boshlang‘ich qiymatli parametrlar

Yuqorida keltirilgan **factorial** va **displayUser** funksiyalari uchun funksiyaning parametrlariga mos bo‘lgan qiymatlar uzatishga to‘g‘ri keladi. Agar parametrlari mavjud bo‘lgan funksiyaga parametrlari ko‘rsatilmagan holda murojaat qilinsa, u holda dastur xatolik haqida habar beradi. Bunday xatoliklarni oldini olish uchun, **Kotlin** dasturlash tili boshlang‘ich qiymatli parametrga ega bo‘lgan funksiyalar bilan ishlashni tavsiya etadi. Bunday ko‘rinishdagi funksiya e‘lon qilinayotgan vaqtda parametrning turiga mos boshlang‘ich qiymatlar beriladi. Quyidagi dasturda yuqorida keltirilgan **displayUser** funksiyasini boshlang‘ich parametrlari ko‘rinishi keltirib o‘tilgan:

```
fun displayUser(name:String, age:Int=18,
position:String="ishsiz"){
    println("Nomi: $name   Yoshi: $age   Lavozimi: $position")
}
fun main(args: Array<String>) {
    displayUser("Toxir", 23, "Boshqaruvchi")
    displayUser("Aziza", 21)
    displayUser("Kamola")
}
```

Yuqorida keltirilgan dasturda yaratilgan funksiyaning birinchi parametri boshlang‘ich qiymatga ega bo‘lmagan parametr hisoblanib, qolgan ikki parametri boshlang‘ich qiymatga ega bo‘lgan parametrdir. Agar funksiya foydalanilayotgan joyda boshlang‘ich qiymatga ega bo‘lgan parametr uchun qiymat o‘rnatilmasa, u holda funksiya to‘g‘ridan – to‘g‘ri boshlang‘ich qiymat bo‘yicha funksiyada ko‘rsatilgan buyruqlar ketma–ketligi bajaradi. Ammo yuqorida ko‘rsatilgan funksiyada boshlang‘ich qiymatga ega bo‘lmagan parametr mavjudligi sababli,

funksiyani chaqirish joyida kamida bitta parametr bilan chaqirish tavsiya etiladi. Dastur quyidagi natijani konsolga chiqaradi:

```
Nomi: Toxir   Yoshi: 23   Lavozimi: Boshqaruvchi
Nomi: Aziza   Yoshi: 21   Lavozimi: ishsiz
Nomi: Kamola  Yoshi: 18   Lavozimi: ishsiz
```

Nomlangan argumentlar

Parametrli funksiyalarda qiymatlar quyidagi ko‘rinishda uzatiladi: birinchi qiymat – birinchi parametr, ikkinchi qiymat – ikkinchi parametr va hokazo. Lekin nomlangan argumentlardan foydalanib, qiymatlarni ixtiyoriy tartibda uzatish mumkin.

Yuqoridagi dasturning asosiy funksiyasiga quyidagicha o‘zgaritirish kiritish bilan nomlangan argumentli ko‘rinishga keltirish mumkin.

```
fun main(args: Array<String>) {
    displayUser(name="Toxir",position="Boshqaruvchi",age=28)
    displayUser(age=21, name="Aziza")
    displayUser("Kamola", position="Kichik dasturchi")
}
```

Nomlangan argumentlarda funksiyaning parametri nomi ko‘rsatilib unga qiymat beriladi. Bundan tashqari boshlang‘ich qiymatga ega bo‘lgan parametrlarga nomlangan argument sifatida qiymat uzatish shart emas. Agar funksiyada boshlang‘ich qiymatga ega bo‘lmagan parametr mavjud bo‘lsa va boshqa parametrlarga nomlangan argument sifatida murojaat qilinsa, boshlang‘ich qiymatga ega bo‘lmagan parametr ham nomlangan argument sifatida murojaat qilinadi.

3.2. O‘zgaruvchan sonli parametrlar. vararg kaliti

Kotlin dasturlash tilida funksiya bir xil turdagi parametrlarning o‘zgaruvchan sonini qabul qiladi. Bunday parametrlardan foydalanish uchun bu dasturlash tili **vararg** xizmatchi so‘zidan foydalanadi. Masalan, bir funksiyaga bir nechta satrlardan tashkil topgan qiymatlarni uzatish kerak, lekin satrlar soni ma’lum emas.

```
fun printStrings(vararg strings: String) {
    for(str in strings)
        println(str)
}
fun main(args: Array<String>) {
    printStrings("Toxir", "Bobur", "Samad")
    printStrings("Kotlin", "JavaScript", "Java", "C#", "C++")
}
```

Yuqoridagi dasturda **printStrings** nomli funksiya noma’lum sondagi qiymatlarni qabul qiluvchi funksiya shaklida e’lon qilingan. Noma’lum

sondagi qiymatlarni takrorlanish operatori yordamida har bir qiymatga murojaat qilib, uning ustida ko'rsatilgan buyruqlarni bajarish mumkin. Ko'rsatilgan dasturda har bir qiymatga murojaat qilinib, tanlangan qiymatlar konsol oynasiga chiqarish jarayoni ko'rsatilgan. Sonli qiymatlar uchun ham shu shaklda funksiyalar e'lon qilinib, ularning ustida turli amallar bajarish mumkin. Bunga quyidagi dastur misol bo'ladi:

```
fun sum(vararg numbers: Int) {  
    var result=0  
    for(n in numbers)  
        result += n  
    println("Sonlar yig'indisi: $result")  
}  
fun main(args: Array<String>) {  
    sum(1, 2, 3, 4, 5)  
    sum(1, 2, 3, 4, 5, 6, 7, 8, 9)  
}
```

Bu dasturda funksiyaning argumentlari sifatida keltirilgan sonlar yig'indisi ekranga chiqariladi. **vararg** xizmatchi so'zi funksiyaning ikkinchi yoki ixtiyoriy argumentiga ham berilishi mumkin. Agar ushbu xizmatchi so'z ikkinchi argumentga berilsa, birinchi argument uchun berilgan qiymat birinchi parametrga, qolgan qiymatlar esa **vararg** xizmatchi so'zi bilan kelgan parametrga uzatiladi.

```
fun printUserGroup(count:Int, vararg users: String) {  
    println("Soni: $count")  
    for(user in users)  
        println(user)  
}  
  
fun main(args: Array<String>) {  
    printUserGroup(3, "Toxir", "Bobur", "Aziza")  
}
```

Agar **vararg** xizmatchi so'zi yordamida e'lon qilingan parametrdan keyin oddiy argumentlar mavjud bo'lsa, u holda nomlangan parametr yordamida qiymatlar uzatiladi.

```
fun printUserGroup(group: String, vararg users: String,  
count:Int) {  
    println("Guruh: $group")  
    println("Soni: $count")  
    for(user in users)  
        println(user)  
}  
fun main(args: Array<String>) {  
    printUserGroup("650-18", "Toxir", "Bobur", "Aziza", count=3)  
}
```

Yuqoridagi dasturda **printUserGroup** funksiyasi keltirilgan bo‘lib, bu funksiyaning birinchi parametri guruhni, oxirgi parametri esa guruh a‘zolari sonini ifodalaydi. Qolgan qiymatlar guruh a‘zolarining nomlarini haqida ma’lumot beradi. Funksiyaga murojaat qilingan joyga qaralsa, oxirgi parametr nomlangan parametr sifatida yozilgan. Bu dasturning natijasi quyidagidan iborat:

```
Guruh: 650-18
Soni: 3
Toxir
Bobur
Aziza
```

Operator * (Yoyish operatori)

* operatori – yoyish operatori deb ham yuritiladi. Bu yerda bu operatorini ko‘paytirish operatori bilan almashtirib yubormaslik kerak. Bu operator asosan funksiyada **vararg** xizmatchi so‘zi yordamida e‘lon qilingan parametrga massivni uzatishda ishlatiladi. Bunda massiv o‘zgaruvchisining oldida qo‘yiladi. Quyida bu operatoridan foydalanish ko‘rsatib o‘tilgan:

```
fun printUserGroup(group:String, vararg users:String,
count:Int) {
    println("Soni: $count")
    for(user in users)
        println(user)
}
fun main(args: Array<String>) {
    val users = arrayOf("Toxir", "Bobur", "Aziza")
    printUserGroup("650-18", *users, count=3)
}
```

Funksiyaga murojaat qilingan joyga e‘tibor berish lozim. Agar yoyish operatoridan foydalanilmasa, unda xatolik yuz beradi. Chunki funksiyaga qator jo‘natilishi lozim. Ammo massiv jo‘natilayotganligi sababli massivni qator sifatida qaralishi uchun yoyish operatoridan foydalaniladi.

3.3. Qiymat qaytaruvchi funksiyalar. return operatori

Boshqa dasturlash tillari kabi **Kotlin** dasturlash tilidagi funksiya ham qiymat qaytarish xususiyatiga ega. Funksiya qiymat qaytaruvchi bo‘lishi uchun funksiyaning tana qismi boshlanishidan oldin uning qaytaruvchi qiymatining turi qo‘rsatilib o‘tiladi. Funksiya qiymat qaytarishi uchun **return** xizmatchi so‘zidan foydalaniladi. Bu xizmatchi so‘zdan keyin qaytariluvchi qiymat yoki o‘zgaruvchi yoziladi. Qiymat qaytaruvchi funksiyaga quyidagi faktorial topuvchi dastur kodi misol bo‘ladi:

```

fun factorial(n: Int):Int{ // Int turida qiymat qaytaradi
    var result = 1;
    for(d in 1..n){
        result *= d
    }
    return result           // qaytariluvchi qiymat
}
fun main(args: Array<String>) {
    val a = factorial(4)
    val b = factorial(5)
    val c = factorial(6)
    println("a=$a b=$b c=$c")
}

```

Dasturda ko‘rinib turibdiki, funksiyani e‘lon qilinayotgan vaqtda qaytariluvchi qiymatning turi ko‘rsatib o‘tilgan. Bu funksiya **Int** turida qiymat qaytaradi:

```

fun factorial(n: Int):Int

```

Qiymatni qaytarish uchun **return** xizmatchi so‘zidan foydalanilgan:

```

return result

```

Dasturning asosiy qismida o‘zgaruvchiga funksiyadan qaytgan qiymat o‘zlashtirilib olinmoqda:

```

val a = factorial(4)

```

Unit turi

Kotlin dasturlash tilida agar funksiya qiymat qaytarmasa, uning turini **Unit** turi bilan ham e‘lon qilish mumkin. Bu tur **C** dasturlash tili oilasidagi **void** turi bilan o‘xshash tur hisoblanadi. Bu turga quyida keltirilgan funksiya misol bo‘ladi:

```

fun hello(){
    println("Salom")
}

```

Yuqorida keltirilgan funksiya **Unit** turisiz keltirilgan. Agar dasturchi tomonidan tur ko‘rsatiladigan bo‘lsa, funksiyaning ko‘rinishi quyidagi ko‘rinishga keladi:

```

fun hello() : Unit{
    println("Salom")
}

```

Unit turidagi funksiyalarni ham qandaydir o‘zgaruvchiga o‘zlashtirish mumkin. Bunda o‘zgaruvchining qiymati **Unit** ko‘rinishdagi obyekt hisoblanadi. **Unit** turidagi yoki turi ko‘rsatilmagan funksiyalarda ham **return** buyrug‘i ishlatilishi mumkin. Bunda funksiya **return** xizmatchi so‘ziga kelganida bu xizmatchi so‘zdan keyin kelgan ma‘lumotlarni bajarmasdan, funksiya ishini yakunlaydi. Quyida qiymat

qaytarmaydigan funksiyada **return** xizmatchi soʻzini ishlatish boʻyicha dastur kodi keltirilgan:

```
fun checkAge(age: Int) {  
    if(age < 0 || age > 150){  
        println("Noto'g'ri yosh")  
        return  
    }  
    println("Yosh to'g'ri keltirilgan")  
}  
fun main() {  
    checkAge(-10)  
    checkAge(10)  
}
```

Yuqorida keltirilgan dasturda **checkAge** nomli parametrlı funksiya argument sifatida koʻrsatilgan, qiymatini berilgan shartlar asosida tekshirib, shartlarga mos ravishda qiymat chiqaradi. Agar shart bajarilsa, ketma–ketlik shart operatorining ichida joylashgan **return** operatorini bajaradi va funksiya shu yerda faoliyatini tugatadi. Agar shart bajarilmasa, u holda shart operatoridan keyingi operatorlar ketma–ketligi bajarilib funksiya oxiriga qadar bajariladi.

3.4. Bir qatorli va mahalliy funksiyalar

Bir qatorli funksiyalar

Kotlin dasturlash tilida bir qatorli funksiyalar bilan ishlash tushunchasi mavjud. Bir qatorli funksiya oʻzi nima? Bu savolga shunday javob berish mumkin. Funksiya bitta ifodani bajarib qiymat qaytarishi kerak boʻlsa, bunday funksiya bir qatorli funksiya deb yuritiladi. Bunday funksiyalarni oddiy funksiya tarzda quyidagicha yozish mumkin:

```
fun square(x:Int):Int{  
    val d:Int = x * x  
    return d  
}
```

Yuqorida keltirilgan funksiyani bir qarashda bir qatorli funksiya deb boʻlmaydi. Chunki bir qatorli funksiya aynan bir qatorda joylashishi kerak. Funksiya bir qatorda joylashishi uchun **Kotlin** dasturlash tili quyidagi formatni taqdim etgan:

```
fun funksiya_nomi (funksiya_parametrlari) = funksiya_tanasi
```

Yuqorida keltirilgan funksiyani bir qatorli funksiya formatida quyidagicha yozish mumkin.

```
fun square(x: Int) = x * x  
yoki
```

```
fun square(x: Int): Int = x * x
```

Bu ikki bir qatorli funksiya bir xil vazifa bajaradi. Farqi shundaki ikkinchi e'lon qilingan ko'rinishda funksiyadan qaytayotgan qiymatning turi ko'rsatilgan, ya'ni funksiya turi.

Bir qatorli funksiyalarda tarmoqlanuvchi jarayonlarni ham ishlatish mumkin. Bunda tarmoqlanuvchi jarayonning qiymat qaytaruvchi ko'rinishidan foydalaniladi. Bunga misol sifatida ikki sondan kattasini topish dasturini ko'rish mumkin.

```
fun max(x: Int, y: Int) = if (x > y) x else y
fun main() {
    println(max(5, 6))
}
```

Mahalliy funksiyalar

Kotlin dasturlash tilida mahalliy funksiyalar mavjud. Mahalliy funksiyalar bu – yaratilgan funksiyalar ichidagi shu funksiyaga tegishli funksiyadir. Bunday funksiya o'zi e'lon qilingan funksiyadan tashqarida bo'lgan funksiyalarga murojaat qila olmaydi. Mahalliy funksiyaga quyida keltirilgan yoshni taqqoslash funksiyasi misol bo'ladi. Yosh chegarasi musbat va 120 sonidan katta bo'lmagan son sifatida qaralgan.

```
fun compareAge(age1: Int, age2: Int) {
    fun ageIsValid(age: Int): Boolean {
        return (age > 0 && age <= 120);
    }
    if (!ageIsValid(age1) || !ageIsValid(age2)) {
        println("Yosh noto'g'ri")
        return
    }
    when {
        age1 == age2 -> println("Ikki yosh teng")
        age1 > age2 -> println("Birinchi yosh katta")
        age1 < age2 -> println("Ikkinchi yosh katta")
    }
}
fun main() {
    compareAge(20, 25)
    compareAge(-3, 20)
    compareAge(30, 121)
}
```

Yuqoridagi dasturda mahalliy funksiya **ageIsValid** nomi bilan keltirilgan. Bu mahalliy funksiyani bir qatorli funksiya tarzda yozish ham mumkin. Bunda mahalliy funksiya va u tegishli bo'lgan asosiy funksiya quyidagicha ko'rinishga keladi:

```

fun compareAge(age1: Int, age2: Int) {
    fun ageIsValid(age: Int) = (age > 0 && age <= 120);
    if (!ageIsValid(age1) || !ageIsValid(age2)) {
        println("Yosh noto'g'ri")
        return
    }
    when {
        age1 == age2 -> println("Ikki yosh teng")
        age1 > age2 -> println("Birinchi yosh katta")
        age1 < age2 -> println("Ikkinchi yosh katta")
    }
}

```

Mahalliy funksiyadagi *age* o'zgaruvchisini ketma-ketlik shakllida yozish yanada tushunarli bo'ladi. Bunda **ageIsValid** nomli mahalliy funksiya quyidagi ko'rinishga keladi:

```

fun ageIsValid(age: Int) = (age in 1..120)

```

3.5. Funksiyalarni qayta yuklash

Barcha dasturlash tillaridagi kabi **Kotlin** dasturlash tilida ham funksiyalarni qayta yuklash mumkin. Funksiyalarni qayta yuklash deganda nima tushuniladi? Funksiyalarni qayta yuklash deganda bir nomli funksiyalarni bir necha marta e'lon qilish, ya'ni nomi bir xil bo'lgan, parametrlari turi, parametrlar soni va qaytariluvchi qiymat turi bilan farqlanadigan funksiyalarga aytiladi. Masalan, ikki sonni qo'shish funksiyasini ko'riladigan bo'lsa, bunda son butun yoki haqiqiy turda bo'lishi mumkin.

```

fun sum(a: Int, b: Int): Int{
    return a + b
}
fun sum(a: Double, b: Double): Double{
    return a + b
}
fun sum(a: Int, b: Int, c: Int): Int{
    return a + b + c
}
fun sum(a: Int, b: Double): Double{
    return a + b
}
fun sum(a: Double, b: Int): Double{
    return a + b
}

```

Yuqorida bir nechta **sum** nomidagi funksiyalar berilgan. Bu yerdagi funksiyalarning parametrlari yuqorida aytilganidek turi yoki soni bilan farq qiladi. Bu funksiyalardan foydalanish quyida ko'rsatilgan:


```

fun main() {
    val a = sum(1, 2)
    val b = sum(1.5, 2.5)
    val c = sum(1, 2, 3)
    val d = sum(2, 1.5)
    val e = sum(1.5, 2)
}

```

Funksiyalarni parametrlari soni va ularning turlari ustma–ust tushib, funksiyaning qiymati turi turlicha bo‘lsa, unda bu funksiyalardan biri xato hisoblanadi. Chunki bu funksiyalarga murojaat qilinganda asosan parametrlari soni va ularning turlari bo‘yicha qaysi funksiyaning bajarish kompilyator tomonidan tanlab olinadi. Bunga misol sifatida quyidagi dastur kodini ko‘rish mumkin.

```

fun sum(a: Double, b: Int) : Double{
    return a + b
}
fun sum(a: Double, b: Int) : String{
    return "$a + $b"
}

```

Yuqorida aytilganidek, bu ikki funksiyaning parametrlari soni va ularning turlari ustma–ust tushadi. Ammo funksiya qaytayotgan qiymat har xil. Bu esa dasturchining xatosi hisoblanadi.

Funksiya turini aniqlash

Kotlin dasturlash tilida barcha o‘zgaruvchilar obyekt hisoblaniladi, shu jumladan funksiyalar ham obyektidir. Funksiyalar ham boshqa obyektlar singari o‘ziga xos turga ega. Funksiyaning turi quyidagicha aniqlanadi.

```

(parametrlar_turi) -> qaytariluvchi_tur

```

Qiymat qaytarmaydigan va parametrlarga ega bo‘lmagan funksiya quyidagicha aniqlanadi:

```

fun hello(){
    println("Salom Kotlin")
}

```

Bu funksiyaning turi quyidagicha bo‘ladi:

```

() -> Unit

```

Agar funksiyaning parametrlari mavjud bo‘lsa, u holda funksiyaning turi parametrlarining turlari qavs ichida yozilgan holatda aniqlanadi. Masalan,

```

fun sum(a: Int, b: Int): Int{
    return a + b
}

```

Bu funksiyaning ikki parametri mavjud bo'lib, ikkala parametr ham butun sonli turda. Bundan tashqari funksiya butun sonli turda qiymat qaytaradi. Bu funksiyaning turi quyidagicha aniqlanadi:

```
(Int, Int) -> Int
```

Funksiyaning turini aniqlash nima uchun kerak? Funksiyaning turi dasturda turli nomdagi parametrlari soni va qaytariluvchi turi bir xil bo'lgan bir nechta funksiyalarga bir nom bilan murojaat qilish huquqini beradi. Bundan tashqari murakkab nomdagi funksiyalarni nomini o'zgartirish huquqini ham beradi.

Funksiya nomini o'zgartirish

Kotlin dasturlash tilida o'zgaruvchi e'lon qilinayotganda uning turini funksiya turi sifatida ko'rsatish mumkin. Bunda ko'rsatilgan turdagi funksiyalarni o'zgaruvchiga qiymat sifatida uzatish mumkin bo'ladi. O'zgaruvchini funksiya tiri bilan quyidagicha e'lon qilinadi:

```
val message: () -> Unit
```

Bu yerda **message** nomli o'zgaruvchi parametrغا ega bo'lmagan qiymat qaytarmaydigan funksiya sifatida e'lon qilinmoqda. Funksiya turi yordamida e'lon qilingan o'zgaruvchiga funksiyaning ko'rsatish uchun ketma-ket ikkita ikki nuqtadan (::) foydalaniladi. Bu buyruq funksiya nomining oldidan qo'yilib, funksiyaning parametrlari va parametrlarni o'z ichiga oladigan qavslardan foydalanishni cheklaydi. Shundan so'ng funksiya turi yordamida e'lon qilingan o'zgaruvchiga oddiy funksiya sifatida murojaat qilish mumkin. Bunga misol quyidagi dasturda keltirilgan:

```
fun main() {  
    val message: () -> Unit  
    message = ::hello  
    message()  
}  
fun hello() {  
    println("Salom Kotlin")  
}
```

Bu dasturda yuqorida e'lon qilingan **message** nomli o'zgaruvchiga murojaat qilinganda bu o'zgaruvchi to'g'ridan-to'g'ri **hello** nomli funksiyaga murojaat qiladi.

Bu usul yordamida parametrlarga ega bo'lgan funksiyalarga ham murojaat qilish mumkin.

```
fun main() {  
    val operation: (Int, Int) -> Int = ::sum  
    val result = operation(3, 5)
```

```

println(result) // 8
}
fun sum(a: Int, b: Int): Int{
    return a + b
}

```

Bu usuldan foydalanib, klaviatura yordamida kiritilgan sonlarni berilgan amal orqali hisoblash dasturi quyidagicha yoziladi. Berilgan amallar sifatida qo'shish (+), ayirish (-), ko'paytirish (*), bo'lish (/) va qoldikli bo'lish (%) ko'rish mumkin.

Bu berilgan topshiriqni bajarishda funksiya turini qabul qiladigan har bir amal uchun funksiya yaratiladi. Funksiya nomlari quyidagicha bo'lsin: qo'shish (add), ayirish (sub), ko'paytirish (mul), bo'lish (div) va qoldikli bo'lish (mod).

```

import java.util.*
fun add(x: Int, y: Int): Float {
    return (x + y).toFloat()
}
fun sub(x: Int, y: Int): Float {
    return (x - y).toFloat()
}
fun mul(x: Int, y: Int): Float {
    return (x * y).toFloat()
}
fun div(x: Int, y: Int): Float {
    return (1.0 * x / y).toFloat()
}
fun mod(x: Int, y: Int): Float {
    return (x % y).toFloat()
}

fun main() {
    val input = Scanner(System.`in`)
    val a: Int = input.nextInt()
    val ch: String = input.next()
    val b: Int = input.nextInt()
    var action: (Int, Int) -> Float = ::add
    when (ch) {
        "+" -> action = ::add
        "-" -> action = ::sub
        "*" -> action = ::mul
        "/" -> action = ::div
        "%" -> action = ::mod
    }
    val res: Float = action(a, b)
    print("$a $ch $b = $res")
}

```

Yuqorida keltirilgan dasturning natijasi quyida berilgan:

$15 + 5$	$15 - 23$	$15 * 12$
$15 + 5 = 20.0$	$15 - 23 = -8.0$	$15 * 12 = 180.0$

Bu natijalardan ko‘rinib turibdiki, tuzilgan dastur to‘g‘ri ishlamoqda.

3.6. Yuqori darajadagi funksiyalar

Yuqori darajali funksiyalar (high order function) – bu funksiyani parametr sifatida qabul qiladigan, funksiyani qaytaradigan yoki yuqorida ko‘rsatilgan ikkalasini ham bajaradigan funksiyalardir.

Funksiyalarni parametr sifatida qabul qilish

Funksiya parametri boshqa bir funksiyani qabul qilishi uchun parametrni funksiya turi sifatida e‘lon qilinishi yoki ko‘rsatilishi kerak:

```
fun main() {  
    displayMessage(::morning)  
    displayMessage(::evening)  
}  
fun displayMessage(mes: () -> Unit) {  
    mes()  
}  
fun morning() {  
    println("Xayrli tong")  
}  
fun evening() {  
    println("Xayrli kech")  
}
```

Yuqorida keltirilgan dasturda **displayMessage** nomli funksiyaning **mes** parametri qiymat qaytarmaydigan va parametrlarga ega bo‘lmagan funksiya qabul qilishini bildiradi. Bu funksiya chaqirilganda parametrda qaysi funksiyaning nomi mavjud bo‘lsa, funksiya parametrda kelgan funksiyani ishlatib beradi. Funksiya birinchi bo‘lib, **morning** funksiyasini ishlatib bermoqda. Parametr sifatida kelgan funksiya bajarilganida ekranga “Xayrli tong” ko‘rinishidagi axborot uzatiladi.

Parametrli funksiyalarni ham funksiyaning parametri sifatida ishlatish mumkin. Buning uchun funksiyaning turini ko‘rsatayotgan vaqtda parametrlari mavjud ko‘rinishga keltirish kifoya. Parametrli funksiyani funksiya parametri sifatida foydalanish quyidagi dasturda ko‘rsatib o‘tilgan.

```
fun main() {  
    action(5, 3, ::sum)           // 8  
    action(5, 3, ::multiply)     // 15
```

```

        action(5, 3, ::subtract)    // 2
    }
    fun action(n1: Int, n2: Int, op: (Int, Int) -> Int) {
        val result = op(n1, n2)
        println(result)
    }
    fun sum(a: Int, b: Int): Int{
        return a + b
    }
    fun subtract(a: Int, b: Int): Int{
        return a - b
    }
    fun multiply(a: Int, b: Int): Int{
        return a * b
    }
}

```

Bu yerda **action** nomli funksiya uchta parametr qabul qiladi. Birinchi va ikkinchi parametrlar butun sonli turda, uchinchi parametr esa funksiya turida bo‘lib, **(Int, Int) -> Int** ko‘rinishga ega. Funksiya turidan ko‘rinib turibdiki, bu parametr parametrlari butun son dan iborat bo‘lgan, butun sonli qiymat qaytaradigan funksiyaning nomi yoziladi.

action nomli funksiyaning tana qismida kerakli bo‘lgan funksiya chaqirilib, hosil bo‘lgan natija konsol oynasiga chiqariladi.

Funksiyalarni qiymat sifatida qaytarish

Kamdan – kam hollarda funksiyalar funksiya qaytarishi mumkin. Funksiya qaytarishi uchun yaratilayotgan funksiyaning turi funksiya turida e‘lon qilinadi. Funksiya qaytaradigan funksiya tanasida **return** xizmatchi so‘zidan so‘ng funksiyaning ifodalovchi funksiyaning nomi yoziladi. Masalan,

```

fun main() {
    val action1 = selectAction(1)
    println(action1(8,5)) // 13
    val action2 = selectAction(2)
    println(action2(8,5)) // 3
}
fun selectAction(key: Int): (Int, Int) -> Int{
    when(key) {
        1 -> return ::sum
        2 -> return ::subtract
        3 -> return ::multiply
        else -> return ::empty
    }
}
fun empty (a: Int, b: Int): Int{
    return 0
}

```

```

fun sum(a: Int, b: Int): Int{
    return a + b
}
fun subtract(a: Int, b: Int): Int{
    return a - b
}
fun multiply(a: Int, b: Int): Int{
    return a * b
}

```

bu yerda – **selectAction** funksiyasi butun sonli turni ifodalovchi bitta parametrga ega. Funksiyadan qaytish turi sifatida (**Int**, **Int**) -> **Int** funksiya turi ko'rsatilgan. Ya'ni, **selectAction** funksiyasi ikkita butun sonli parametrga ega bo'lgan va butun sonli qiymat qaytaradigan funksiyaning qaytaradi.

selectAction funksiyasining tana qismida **key** parametrining qiymatiga qarab, ma'lum bir funksiya qiymat sifatida qaytariladi. **selectAction** funksiyasining natijasi birinchi navbatda **action1** o'zgaruvchisiga uzatiladi. **selectAction** funksiyasi funksiya qaytarganligi sababli, **action1** o'zgaruvchisi o'zida funksiya saqlaydi. Keyin **action1** o'zgaruvchisi orqali **selectAction** funksiyasidan qaytgan funksiyaning chaqirish mumkin. Qaytgan funksiya 2 ta butun sonli qiymatdan iborat parametrga ega bo'lganligi sababli **action1** o'zgaruvchisi funksiya ko'rinishda chaqirilib, unga parametrlar ko'rsatiladi.

3.7. Anonim funksiyalar va ulardan foydalanish

Anonim funksiyalar odatdagi funksiyalarga o'xshaydi, faqat bu funksiyalarning nomi mavjud bo'lmaydi. Anonim funksiya bitta ifodaga ega bo'lishi mumkin. Masalan,

```

fun(x: Int, y: Int): Int = x + y

```

Yoki operatorlar bloki yordamida tana qismini ko'rsatish ham mumkin.

```

fun(x: Int, y: Int): Int{
    return x + y
}

```

Anonim funksiyalar o'zgaruvchining qiymatini sifatida ham berilishi mumkin.

```

fun main() {
    val message = fun() = println("Hello")
    message()
}

```

Bu yerda **fun**() = println("Hello") anonim funksiyasi message o'zgaruvchisiga uzatilmoqda. Ushbu anonim funksiya hech qanday

parametriga ega emas va shunchaki konsolga "Hello" qatorini chiqaradi. Shunday qilib message o'zgaruvchisi () -> **Unit** turini aks ettiradi. Shundan so'ng bu funksiyani oddiy o'zgaruvchining nomi bilan chaqirish mumkin **message()**.

Parametriga ega bo'lgan anonim funksiyalar ham mavjud. Bunga quyidagi dasturda misol keltirilgan:

```
fun main() {  
    val sum = fun(x: Int, y: Int): Int = x + y  
    val result = sum(5, 4)  
    println(result)    // 9  
}
```

Bu dasturda **sum** nomli o'zgaruvchiga anonim funksiya qiymat sifatida berilgan. Bu anonim funksiya ikkita butun sonli turdagi parametrlarni qabul qiladi va anonim funksiya ko'rsatilgan qiymatlarning yig'indisini qaytaradi.

Agar o'zgaruvchiga murojaat qilinsa, u holda bu o'zgaruvchi anonim funksiyaning vazifasini bajarib, natija qaytaradi. Natija qaytarishi uchun unga to'g'ri murojaat qilinsa, bo'ldi. O'zgaruvchiga to'g'ri murojaat qilish huddi funksiyaga murojaat qilish kabidir, ya'ni **val result = sum(5, 4)**

Anonim funksiya – funksiya parametri sifatida

Anonim funksiyalarni – funksiya parametri sifatida ishlatish ham mumkin. Bu holda anonim funksiyaning turi funksiyaning parametridagi funksiya turiga mos bo'lishi kerak.

```
fun main() {  
    doOperation(9, 5, fun(x: Int, y: Int): Int = x + y)  
    doOperation(9, 5, fun(x: Int, y: Int): Int = x - y)  
    val op = fun(x: Int, y: Int): Int = x * y  
    doOperation(9, 5, op)  
}  
fun doOperation(x: Int, y: Int, op: (Int, Int) -> Int) {  
    val result = op(x, y)  
    println(result)  
}
```

Anonim funksiyani – funksiyadan qaytarish

Kotlin dasturlash tilida anonim funksiyani – funksiyaning qiymati sifatida qaytarish mumkin. Quyidagi dasturni bunga misol bo'ladi:

```
fun main() {  
    val action1 = selectAction(1)  
    val result1 = action1(4, 5)  
    println(result1)  
  
    val action2 = selectAction(3)
```

```

    val result2 = action2(4, 5)
    println(result2)

    val action3 = selectAction(9)
    val result3 = action3(4, 5)
    println(result3)
}
fun selectAction(key: Int): (Int, Int) -> Int{
    when(key){
        1 -> return fun(x: Int, y: Int): Int = x + y
        2 -> return fun(x: Int, y: Int): Int = x - y
        3 -> return fun(x: Int, y: Int): Int = x * y
        else -> return fun(x: Int, y: Int): Int = 0
    }
}

```

Anonim funksiyalar **selectAction** funksiyasining ichida ko'rsatilgan. Bu funksiyaning **key** nomli parametrining qiymatiga qarab to'rtta anonim funksiyadan birini qiymat sifatida qaytarish mumkin.

3.8. Lambda amallari

Lambda amali – bu biror–bir operatorlardan tashkil topgan buyruqlar ketma–ketligi, ya'ni dastur kodi. Aslida lambda amali funksiyalar uchun yo'llanma hisoblanadi. Bu holda lambda amallar oddiy va anonim funksiyalar singari o'zgaruvchilar va funksiya parametriga qiymat sifatida berilishi mumkin. Lambda amallari figurali qavslar bilan ifodalanadi. Masalan,

```
{ println("hello") }
```

Bu holda, lambda ifodasi konsolga "hello" qatorini chiqaradi. Lambda amali oddiy o'zgaruvchida saqlanishi mumkin, so'ngra bu o'zgaruvchining nomi orqali oddiy funksiya sifatida chaqiriladi.

```

fun main() {
    val hello = { println("Hello Kotlin") }
    hello()
    hello()
}

```

Bu yerda, lambda amali *hello* o'zgaruvchisiga saqlanadi va ushbu o'zgaruvchi orqali ikki marta chaqiriladi. Lambda amali funksiya uchun yo'llanma bo'lganligi sababli *hello* o'zgaruvchisiga huddi funksiya murojaat qilinganday murojaat qilinadi. Bu lambda amalining turi ya'ni, funksiya turi () -> **Unit** ko'rinishda bo'ladi.

```
val hello: ()->Unit = { println("Hello Kotlin") }
```

Lambda amallarini o'zgaruvchilarga qiymat sifatida bermagan holda ishlatish uchun **run** buyrug'idan foydalaniladi.


```
fun main() {
    run { println("Hello Kotlin") }
}
```

Shu bilan birga lambda amallarini oddiy qavslar yordamida funksiya sifatida ishlatish mumkin:

```
fun main() {
    { println("Hello Kotlin") }()
}
```

Shuni yodda tutish kerakki, agar lambda amallari ketma–ket yozilib, ishga tushirish uchun oddiy qavslardan foydalanilayotgan bo‘lsa, **Kotlin** dasturlash tili keyingi ifodalarni yangi ko‘rsatma sifatida qabul qilishi uchun, har bir lambda amalidan keyin nuqtali vergul ishlatish lozim:

```
fun main() {
    { println("Hello Kotlin") }();
    { println("Kotlin in Uzbek") }()
}
```

Parametrlarni uzatish

Lambda amallari funksiyalar singari parametrlarni qabul qilishi mumkin. Parametrlar uzatish uchun \rightarrow yo‘nalish buyrug‘i ishlatiladi. Parametrlar yo‘nalish buyrug‘ining chap tomonida va lambda amalining tanasi ya’ni, tana qismi yo‘nalish buyrug‘ining o‘ng tomonida ko‘rsatiladi.

```
fun main() {
    val printer = {message: String -> println(message)}
    printer("Hello")
    printer("Good Bye")
}
```

Bu yerda lambda ifodasi **String** turidagi bitta parametрни qabul qiladi, uning qiymatini konsolga chiqaradi. Lambda amali chaqirilganida uning parametri qavslar ichida beriladi:

```
fun main() {
    { message: String -> println(message) }("Welcome to
Kotlin")
}
```

Agar parametrlar bir nechta bo‘lsa, yo‘nalish buyrug‘idan oldin ular vergul bilan ajratilgan holda yoziladi.

```
fun main() {
    val sum = {x:Int, y:Int -> println(x + y)}
    sum(2, 3) // 5
    sum(4, 5) // 9
}
```

Agar lambda amali bir nechta buyruqlardan iborat bo'lsa, yo'nalish buyrug'idan so'ng har bir buyruq alohida satrlarda ko'rsatilishi kerak bo'ladi:

```
val sum = { x:Int, y:Int ->
    val result = x + y
    println("$x + $y = $result")
}
```

Lambda amallarida qiymat qaytarish

Yo'nalish buyrug'idan keyingi ifoda lambda amalining natijasini aniqlaydi. Ushbu natijani ixtiyoriy o'zgaruvchiga qiymat sifatida berish mumkin. Agar lambda amali hech qanday natija qaytarmasa, lambda amali funksiyalardagi kabi **Unit** turida qiymat qaytaradi:

```
val hello = { println("Hello") }
val h = hello()
val printer = { message: String -> println(message) }
val p = printer("Welcome")
```

Ikkala holatda ham **println** funksiyasi ishlatilmoqda, funksiya aslida hech qanday ma'lumot qaytarmaydi deyish xato bo'ladi, chunki bu holatda **Unit** turiga tegishli bo'lgan obyekt qaytariladi.

Lambda amallari ham ma'lumot qaytaradi. Ma'lumot qaytarishga quyidagi dastur misol bo'ladi:

```
fun main() {
    val sum = {x:Int, y:Int -> x + y}
    val a = sum(2, 3)
    val b = sum(4, 5)
    println("a=$a b=$b")
}
```

Bu yerda yo'nalish buyrug'idan o'ng tomonda $x + y$ ifoda turganligi sababli, bu ifodaning qiymati hisoblanib, lambda amalini chaqirgan o'zgaruvchiga qiymat uzatiladi. Lambda amalining turi **(Int, Int) -> Int** funksiya turiga mansub. Agar lambda ifodasi ko'p buyruqli bo'lsa, u bir nechta satrlardan iborat bo'lib, satrlar oxirida qiymat qaytarishi mumkin.

```
val sum = {x:Int, y:Int ->
    val result = x + y
    println("$x + $y = $result")
    result
}
```

Lambda amalining oxirgi satrida sonni ifodalaydi, u satrda joylashgan o'zgaruvchining qiymati x va y sonlarining yig'indisidan iborat bo'ladi. Bu yig'indi lambda amalining natijasi sifatida qaytariladi.

Lambda amali funksiyaning parametrlari sifatida

Lambda amali funksiyaning parametri sifatida kelishi mumkin, agar funksiyaning parametrda kelgan funksiya turi lambda amalining funksiya turi bilan ustma–ust tushshishi quyidagi dastur ko‘rsatilgan:

```
fun main() {  
    val sum = {x:Int, y:Int -> x + y }  
    doOperation(3, 4, sum)  
    doOperation(3, 4, {a: Int, b: Int -> a * b})  
}  
  
fun doOperation(x: Int, y: Int, op: (Int, Int) -> Int) {  
    val result = op(x, y)  
    println(result)  
}
```

Lambda amalining funksiya turi aniq ko‘rsatilgan parametr yoki o‘zgaruvchiga yuborilayotganda, lambda amalidagi parametrlar turlarini ko‘rsatish shart emas:

```
fun main() {  
    val sum: (Int, Int) -> Int = {x, y -> x + y }  
    doOperation(3, 4, {a, b -> a * b})  
}  
  
fun doOperation(x: Int, y: Int, op: (Int, Int) ->Int){  
    val result = op(x, y)  
    println(result)  
}
```

Bu yerda, **sum** o‘zgaruvchisi funksiya turi bilan aniqlanganligi sababli, undan keyin kelgan lambda amalidagi birinchi va ikkinchi parametrlar avtomatik ravishda funksiya turida ko‘rsatilgan turga o‘tadi va qaytuvchi qiymat ham funksiya turidagi qaytuvchi qiymatning turiga moslashadi.

Lambda amali yordamida **doOperation** funksiyaga murojaat qilingan vaqtda ham funksiya parametri sifatida kelgan funksiyani ifodalovchi funksiya turiga amal uzatilganda avtomatik ravishda parametr turi va qaytarish turi moslashadi.

Agar funksiya parametrlari orasida funksiya turi oxirgisi bo‘lib, unga lambda amali yordamida murojaat qilinayotgan bo‘lsa, unda lambda amali funksiya parametrlari yozilganidan so‘ng qavslardan tashqarida yozilishi mumkin.

Yuqorida keltirilgan **doOperation** funksiyasida funksiya turi parametrlar so‘ngisidir. Bunga murojaat qilishning ikki turi mavjud.

Birinchi turidan foydalanish yuqoridagi dasturda keltirib o'tildi. Ya'ni, lambda amali funksiya parametri sifatida keltirilgan.

```
doOperation(3, 4, {a, b -> a * b})
```

Ikkinchi ko'rinishida lambda amali funksiya tashqariga chiqarish bilan ifodalanadi.

```
doOperation(3, 4) {a, b -> a * b}
```

Lambda amalini funksiyaning qiymati sifatida qaytarish

Kotlin dasturlash tilida funksiya – funksiya turiga mos lambda amalini qaytarishi mumkin. Bunda funksiya yuqorida ko'rib o'tilganidek funksiya turida aniqlanishi lozim.

```
fun main() {  
    val action1 = selectAction(1)  
    val result1 = action1(4, 5)  
    println(result1)  
    val action2 = selectAction(3)  
    val result2 = action2(4, 5)  
    println(result2)  
    val action3 = selectAction(9)  
    val result3 = action3(4, 5)  
    println(result3)  
}  
  
fun selectAction(key: Int): (Int, Int) -> Int {  
    when(key) {  
        1 -> return {x, y -> x + y }  
        2 -> return {x, y -> x - y }  
        3 -> return {x, y -> x * y }  
        else -> return {x, y -> 0 }  
    }  
}
```

Bu yerda **selectAction** nomli funksiya – funksiya turiga mos lambda amalini qaytarmoqda. Agar lambda amalida keltirilgan parametrlar lambda amalining tanasida ishlatilmasa, ya'ni foydalanmaydigan parametr bo'lsa, u holda parametrlar o'rniga ostki chiziqdan foydalanish mumkin. Bunday holat **selectAction** funksiya tanasidagi **when** buyrug'ining **else** bo'limida ishlatilmoqda. Bu ko'rinishni **Kotlin** dasturlash tili quyidagicha yozishni taklif etadi.

```
else -> return {_, _ -> 0 }
```

Nazorat savollari:

1. Funksiya deganda nimani tushunasiz?
2. Funksiyalar necha turli bo'ladi?
3. Funksiya parametrlarini nima ajratiladi?
4. Bir qatorli funksiya deganda nimani tushunasiz?

5. Mahalliy funksiya nima?
6. Kotlin dasturlash tilida funksiyalarni qayta yuklash mumkinmi?
7. Yuqori darajali funksiya nima?
8. Anonim funksiya nima?
9. Lambda amali qanday vazifa bajaradi?

4. Obyektga yo'naltirilgan dasturlash

4.1. Klasslar va obyektlar

Kotlin dasturlash tili obyektga yo'naltirilgan dasturlash tillari sarasiga kiradi. Ya'ni, bu tilda yaratilgan dasturlar bir–biri bilan obyektlar yordamida bog'lanadi. **Kotlin** dasturlash tili boshqa obyektga yo'naltirilgan dasturlash tillari kabi obyekt yaratish uchun klasslardan foydalanadi. Obyekt klassning hosilasi hisoblanadi. Masalan, mashina obyektini ko'riladigan bo'lsak, u quyidagilardan iborat. Korpus, g'ildiraklar, rul va boshqalar. Bu sanab o'tilgan ma'lumotlar obyektning xususiyatlari hisoblanadi.

Bu sanab o'tgan ma'lumotlarni bitta umumiy ma'lumotga birlashtirilganda klass tushunchasi kelib chiqadi. Mashinalar bir–biriga o'xshamaydigan ko'rinishlarga ega. Qaysidir mashinaning korpusi boshqasidan farq qilsa, boshqasining g'ildiraklari farq qiladi. Shuning uchun obyekt yaratilayotgan vaqtda uning xususiyatlariga turlicha ma'lumotlarni kiritish mumkin.

Klass qanday yaratiladi? Shu savolga quyidagicha javob berish mumkin.

Kotlin dasturlash tilida klass yaratish uchun **class** xizmatchi so'zi foydalaniladi. Bu xizmatchi so'zdan so'ng yaratilayotgan klassning nomi keltiriladi va shundan keyin figurali qavslar orasida klassning xususiyatlari yoki maydonlari keltiriladi. **Kotlin** dasturlash tilida klassning tana qismi mavjud bo'lmasa, figurali qavslarni yozish shart emas. Masalan, quyida insonni ifodalovchi **Person** nomdagi klass keltirilgan:

```
class Person
// yoki quyidagi ko'rinishda
class Person { }
```

Klass boshqa dasturlash tillaridagi kabi **Kotlin** dasturlash tilida ham foydalanuvchi tomonidan yaratilgan ma'lumotlar turi hisoblanadi. Shuning uchun, yaratilgan klassni ma'lumotlar turi sifatida ishlatish va u yordamida o'zgaruvchilar e'lon qilish mumkin.

```
fun main() {
    val tohir: Person
    val bobur: Person
    val aziza: Person
}
class Person
```

Yuqorida keltirilgan dasturning asosiy funksiyasida uchta o'zgaruvchi foydalanuvchi tomonidan yaratilgan ma'lumotlar turi ya'ni

Person klassi yordamida yaratilmoqda. Shuni ta’kidlash lozimki **Kotlin** dasturlash tilining asosiy funksiyasi C# va Java dasturlash tillaridagi kabi klasslarda emas, balki barcha yaratilgan klasslardan tashqarida yoziladi.

Klassning obyektini yaratish uchun ushbu klassning konstruktorini chaqirish kerak bo’ladi. Konstruktor klass nomi bilan yuritiladigan funksiya hisoblanadi. Agar foydalanuvchi tomonidan klassning konstruktori aniqlanmasa, komplyatorning o’zi bo’sh konstruktor yaratadi va undan foylanish mumkin:

```
fun main() {  
    val tohir: Person = Person()  
    val bobur: Person = Person()  
    val aziza: Person = Person()  
}  
class Person
```

Yuqorida keltirilgan dasturdagi o’zgaruvchilar bu holdan so’ng o’zida obyektlarni saqlaydi.

4.2. Xususiyatlar va metodlar

Har bir klass o’ziga kerakli ma’lumotlarni va holatlarni – xususiyatlarda saqlaydi. Xususiyatlar klass darajasida **val** yoki **var** kalit so’zlari bilan belgilanadigan o’zgaruvchilarni ifodalaydi. Agar xususiyat **val** yordamida aniqlansa, unda bu xususiyatga bir marta qiymat tayinlanadi, ya’ni bu xususiyat o’zgarmas bo’ladi. Agar xususiyat **var** kalit so’zi yordamida aniqlansa, u holda bu xususiyatning qiymatlarini o’zgartirish imkoni mavjud.

Xususiyatlar boshlang’ich qiymatga ega bo’lishi mumkin. Quyida bunga misol keltirilgan:

```
class Person{  
    var name: String = "Noma'lum"  
    var age: Int = 18  
}
```

Bu yerda, insonni ifodalovchi **Person** nomidagi klass yaratilgan bo’lib, uning ikkita xususiyati mavjud. Bu xususiyatlardan bir *name* – insonning nomini, ikkichisi *age* – insonning yoshini ifodalaydi. Bu ikki xususiyatlarga boshlang’ich qiymatlar berilgan. Bu ikki xususiyatlar **var** kalit so’zi bilan aniqlanganligi sababli ularning qiymatini dasturning ixtiyoriy joyida o’zgartirish mumkin:

```
fun main() {  
    val bobur: Person = Person()    // obyekt yaratish  
    println(bobur.name)             // Noma'lum  
    println(bobur.age)              // 18
```

```

    bobur.name = "Bobur"
    bobur.age = 25
    println(bobur.name)           // Bobur
    println(bobur.age)           // 25
}
class Person{
    var name: String = "Noma'lum"
    var age: Int = 18
}

```

Xususiyatlarga murojaat qilish uchun obyektning ifodalovchi o'zgaruvchining nomi, obyektning ichidagi ma'nosini berish uchun nuqta (.) va nuqtadan so'ng xususiyatning nomi ko'rsatiladi. Masalan, o'zgaruvchiga obyekt xususiyatining qiymatini olish quyidagicha ifodalanadi:

```
val personName : String = bobur.name
```

Agar obyektning xususiyatiga qiymat berish kerak bo'lsa, quyidagicha ma'lumotni uzatish mumkin:

```
bobur.name = "Bobur"
```

Klass metodlari

Klass ichida o'zgaruvchi e'lon qilib ishlatish kabi klassning ichida funksiyalar e'lon qilish ham mumkin. Klassda e'lon qilingan o'zgaruvchilar maydon yoki xususiyat deb nomlanganidek, klassga tegishli bo'lgan funksiyalarni metodlar deb yuritiladi. Metodlarning vazifasi klassdan olingan obyektlarning xatti-harakatini belgilab beradi. Umumiy olganda klassning xususiyatlari va metodlari klass a'zolari deb nomlanadi. Masalan, klass metodi bilan ishlash quyidagi dasturda keltirilgan:

```

class Person{
    var name: String = "Undefined"
    var age: Int = 18
    fun sayHello(){
        println("Salom, mening ismim $name")
    }
    fun go(location: String){
        println("$name $location \bga bormoqda")
    }
    fun toString() : String{
        return "Ismi: $name Yoshi: $age"
    }
}

```

Klassning metodlari oddiy funksiyalar kabi aniqlanadi. Hususan, yuqorida keltirilgan **Person** klassida 3 ta metod mavjud bo'lib, bu metodlardan birinchisi **sayHello**. Bu metod konsolga salomlashish va obyektning ismini chiqarishga xizmat qiladi. Ikkinchi metod **go** metodi

bo‘lib, obyektning harakatini ya’ni qaysi manzilga borishini ko‘rsatib beradi. Joylashuv manzilini metodga *location* parametri orqali uzatiladi. Uchinchi metod **personToString** bo‘lib, obyekt haqidagi ma’lumotlarni satr ko‘rinishiga keltirib, qiymat sifatida uzatish uchun xizmat qiladi.

Klassning metodlarida klass xususiyatlaridan foydalanish to‘g‘ridan-to‘g‘ri amalga oshiriladi. Yuqorida keltirilgan **Person** klassida aniqlangan ikki xususiyat mavjud. Bular *name* va *age* xususiyatlaridir. Yuqorida klassning o‘zini yaratish ko‘rsatib o‘tilgan. Bu klassdan foydalanish quyidagi to‘liq dasturda keltirilgan:

```
fun main() {
    val tohir = Person()
    tohir.name = "Tohir"
    tohir.age = 37
    tohir.sayHello()
    tohir.go("do'kon")
    println(tohir.personToString())
}

class Person{
    var name: String = "Undefined"
    var age: Int = 18
    fun sayHello(){
        println("Salom, mening ismim $name")
    }
    fun go(location: String){
        println("$name $location \bga bormoqda")
    }
    fun personToString() : String{
        return "Ismi: $name      Yoshi: $age"
    }
}
```

Bu dasturni natijasi konsolda quyidagicha bo‘ladi:

```
Salom, mening ismim Tohir
Tohir do'konga bormoqda
Ismi: Tohir      Yoshi: 37
```

4.3. Konstruktorlar

Kotlin dasturlash tilida obyekt yaratish uchun klassning konstruktori chaqiriladi. Odatda kompilyator tomonidan konstruktor yaratilgan bo‘lib, shu konstruktordan foydalaniladi. Bu konstruktor hech qanday parametrlarga ega bo‘lmagan konstruktor hisoblanadi. Lekin dasturchi tomonidan konstruktor yaratish mumkin. Konstruktorni aniqlash uchun **constructor** xizmatchi so‘zi ishlatiladi.

Kotlin dasturlash tili klasslarida bitta asosiy konstruktor (primary constructor) va bir yoki bir nechta ikkilamchi konstruktorlar (secondary constructor) mavjud bo‘lishi mumkin.

Asosiy konstruktor

Asosiy konstruktor klass sarlavhasining bir qismi bo‘lib, klass nomidan keyin darhol aniqlanadi:

```
class Person constructor(_name: String) { }
```

Konstruktorlar oddiy funksiyalar kabi parametrlarga ega bo‘lishi mumkin. Yuqoridagi klass konstruktori **String** ma’lumotlar turidagi *_name* parametriga ega. Klass konstruktori orqali tashqi tomondan ma’lumotlar uzatish va undan obyektни yaratish uchun foydalanish mumkin. Konstruktor klass metodidan farq qilib, hatti–xarakatni emas, balki boshlang‘ich ma’lumotlarni tashqi tomondan uzatish uchun ishlatiladi.

Kotlin dasturlash tilida asosiy konstruktor yaratishda **constructor** kalit so‘zini ishlatmasdan ham konstruktor yaratish mumkin. Unda yuqorida keltirilgan klass quyidagi ko‘rinishga keladi:

```
class Person(_name: String) { }
```

Bu ko‘rinishda klass va uning asosiy konstruktorini bir vaqtning o‘zida yaratish amalga oshirilmoqda. Asosiy konstruktor orqali jo‘natilgan ma’lumotlar, agar klassning xususiyatlari mavjud bo‘lsa, kerakli xususiyatlarga qiymatlarni o‘rnatish uchun **init** nomli metod yozish kerak. Bu metod asosiy konstruktorning tanasida yozilishi kerak bo‘lgan buyruqlar ketma–ketligidan iborat bo‘ladi. Masalan,

```
class Person(_name: String) {  
    val name: String  
    init{  
        name = _name  
    }  
}
```

Bu yerda **Person** klassi va uning asosiy konstruktori yaratilib, klassning **name** xususiyatiga boshlang‘ich qiymat o‘rnatish uchun **init** metodi yordamida ish bajarilmoqda. Asosiy konstruktorda *_name* nomli parametr mavjud bo‘lib, bu parametr orqali **name** xususiyatining boshlang‘ich qiymati olib kelinmoqda. Asosiy konstruktorning tana qismi esa yuqorida aytilganidek **init** metodining tana qismida aniqlangan.

Kotlin dasturlash tilidagi klassga tegishli bo‘lgan **init** metodning maqsadi asosiy konstruktor yordamida boshlang‘ich qiymat o‘rnatilishi

kerak bo'lgan xususiyatlarga ularning qiymatlarni o'rnatish ketma-ketligini aniqlashdan iborat.

Asosiy konstruktordan foydalanib, obyekt yaratish jarayoni quyidagi dasturda berilgan:

```
fun main() {  
    val tohir = Person("Tohir")  
    val bobur = Person("Bobur")  
    val aziza = Person("Aziza")  
    println(tohir.name)  
    println(bobur.name)  
    println(aziza.name)  
}  
class Person(_name: String) {  
    val name: String  
    init{  
        name = _name  
    }  
}
```

Shuni hisobga olish kerakki, asosiy konstruktor aniqlangan bo'lsa, kompilyator tomonidan yaratiladigan standart konstruktordan foydalanib bo'lmaydi. Asosiy konstruktor yaratilgan bo'lsa, shu konstruktordan foydalanilgan holda obyektlar yaratishga to'g'ri keladi. **Kotlin** dasturlash tilida asosiy konstruktor tomonidan jo'natilgan ma'lumotlarni klassning o'zida to'g'ridan-to'g'ri xususiyatlarga o'rnatish mumkin:

```
class Person(_name: String) {  
    val name: String = _name  
}
```

Asosiy konstruktordagi xususiyatlar

Asosiy konstruktor klassga tegishli bo'lgan xususiyatlarini aniqlash uchun ham ishlatilishi mumkin. Asosiy konstruktorlar yordamida xususiyatlar yaratish quyidagi dasturda keltirilgan:

```
fun main() {  
    val bobur: Person = Person("Bobur", 23)  
    println("Name: ${bobur.name} Age: ${bobur.age}")  
}  
class Person(val name: String, var age: Int) {  
}
```

Asosiy konstruktor yordamida xususiyatlar yaratishda har bir xususiyat parametr sifatida aniqlanib, ularning ta'rifi **val** (agar xususiyatning qiymati o'zgartirish rejalashtirilmagan bo'lsa) yoki **var** (agar xususiyatning qiymati o'zgartirish rejalashtirilgan bo'lsa) kalit so'zlari ifodalanadi. Asosiy konstruktorda aniqlangan xususiyatlarni

klassning tanasida aniqlashning hojati yo‘q, chunki asosiy konstruktor bu xususiyatlarni o‘zi aniqlab qo‘yadi. Asosiy konstruktor chaqirilganda bu xususiyatlarga qiymatlar avtomatik ravishda uzatiladi.

Ikkilamchi konstruktorlar

Kotlin dasturlash tilida klasslar uchun ikkilamchi konstruktor tushunchasi mavjud bo‘lib, bu konstruktorlar asosiy konstruktorni qayta yuklash vazifasini bajaradi. Ikkilamchi konstruktorlar klassning tana qismida aniqlanadi. Agar klass uchun asosiy konstruktor aniqlangan bo‘lsa, ikkilamchi konstruktor asosiy konstruktorni **this** kalit so‘zi yordamida chaqirish bilan aniqlanadi:

```
class Person(_name: String) {  
    val name: String = _name  
    var age: Int = 0  
    constructor(_name: String, _age: Int) : this(_name) {  
        age = _age  
    }  
}
```

Bu yerda **Person** klassining asosiy konstruktori **name** xususiyatiga qiymat o‘rnatishni belgilaydi. Ikkilamchi konstruktor esa bu klassning **name** va **age** xususiyatlarini boshlang‘ich qiymatlarini o‘rnatishga xizmat qiladi. Ikkilamchi konstruktor ikki parametr qabul qiladi. Bu parametrlar **_name** va **_age**. Ikkilamchi konstruktorda asosiy konstruktorga murojaat qilinmoqda. Murojaatda klassning **name** xususiyatiga qiymat uzatilish ko‘rsatib o‘tilgan. Ya’ni asosiy konstruktorga ikkilamchi konstruktorning **_name** parametri joylashtirilgan. Ikkilamchi konstruktorning tana qismida klassning **age** xususiyatiga qiymat uzatilmogda. Klassdagi ikkilamchi konstruktor quyidagi ko‘rinishga ega:

```
constructor(_name: String, _age: Int) : this(_name) {  
    age = _age  
}
```

Kotlin dasturlash tilida ikkilamchi konstruktor chaqirilganida birinchi o‘rinda asosiy konstruktor bajariladi, shundan so‘ng ikkilamchi konstruktorning tana qismi bajariladi. Yuqoridagi ikkilamchi konstruktor bajarilganida birinchi o‘rinda **name** xususiyatiga qiymat o‘rnatiladi keyin esa **age** xususiyatiga qiymat beriladi. Ikkilamchi konstruktordan to‘liq foydalanish quyidagi dasturda berilgan:

```
fun main() {  
    val tohir: Person = Person("Tohir")  
    val bobur: Person = Person("Bobur", 45)  
    println("Ismi: ${tohir.name}   Yoshi: ${tohir.age}")  
    println("Ismi: ${bobur.name}   Yoshi: ${bobur.age}")  
}
```

```

}

class Person(_name: String) {
    val name: String = _name
    var age: Int = 0
    constructor(_name: String, _age: Int) : this(_name) {
        age = _age
    }
}

```

Dasturning asosiy funksiyasi bo‘lgan **main** funksiyasida **Person** klassi yordamida ikkita obyekt yaratilgan. Bu obyektlarning birinchisi *tohir* nomli o‘zgaruvchida saqlanmoqda va obyekt yaratishda klassning asosiy konstruktoridan foydalanilgan. Asosiy konstruktordan ko‘rinib turibdiki, u bitta parametr oladi. Ikkinchi obyekt *bobur* nomli o‘zgaruvchida saqlangan. Bu o‘zgaruvchiga obyekt yaratishda klassning ikkilamchi konstruktoridan foydalanilgan. Ikkilamchi konstruktor ikkita parametrغا ega. Dasturni ishlatilsa, konsolda quyidagi natija hosil bo‘ladi:

```

Ismi: Tohir   Yoshi: 0
Ismi: Bobur   Yoshi: 45

```

Kotlin dasturlash tilida ikkilamchi konstruktorlar bir nechta bo‘lishi mumkin. Yuqorida keltirilgan **Person** klassiga ish joyi uchun *company* nomli xususiyat qo‘shiladi. Bu xususiyatdan tashqari yana bitta ikkilamchi konstruktor qo‘shiladi. Qo‘shilayotgan ikkilamchi konstruktor quyidagi ko‘rinishga ega.

```

constructor(_name: String, _age: Int, _company: String) :
this(_name, _age) {
    company = _company
}

```

Bu ikkilamchi konstruktor va **company** nomli xususiyat qo‘shilganidan so‘ng dasturning ko‘rinishi quyidagi ko‘rinishga keladi.

```

fun main() {
    val tohir: Person = Person("Tohir")
    val bobur: Person = Person("Bobur", 45)
    val sobir: Person = Person("Sobir", 32, "TATU")
    println("Ismi: ${tohir.name}   Yoshi: ${tohir.age}   Ish
joyi: ${tohir.company}")
    println("Ismi: ${bobur.name}   Yoshi: ${bobur.age}   Ish
joyi: ${bobur.company}")
    println("Ismi: ${sobir.name}   Yoshi: ${sobir.age}   Ish
joyi: ${sobir.company}")
}

class Person(_name: String) {
    val name: String = _name
    var age: Int = 0
}

```

```

var company: String = "Noma'lum"
constructor(_name: String, _age: Int) : this(_name) {
    age = _age
}
constructor(_name: String, _age: Int, _company: String) :
this(_name, _age) {
    company = _company
}
}

```

Bu dastur natijasi quyidagi ko‘rinishda bo‘ladi:

```

Ismi: Tohir   Yoshi: 0   Ish joyi: Noma'lum
Ismi: Bobur   Yoshi: 45  Ish joyi: Noma'lum
Ismi: Sobir   Yoshi: 32  Ish joyi: TATU

```

4.4. Ko‘rinish huquqlari

Dasturda ishlatiladigan barcha turlar shuningdek, komponentalar (klasslar, obyektlar, interfeyslar, konstruktorlar, funksiyalar, xususiyatlar) ko‘rinish huquqi bilan tasniflanadi. Ko‘rinish huquqi ma’lum turlar va ularning tarkibiy qismlariga ishlatilib, bu huquqlar klass a’zolarini qayerda foydalanish yoki foydalanmasligini aniqlaydi. **Kotlin** dasturlash tilida quyidagi ko‘rinish huquqlari mavjud:

- **private** – klasslar, obyektlar, interfeyslar va klassdan tashqari aniqlangan funksiyalar ushbu ko‘rinish huquqi bilan aniqlangan bo‘lsa, bu kattaliklar ular berilgan faylda ko‘rinadi. Ushbu ko‘rinish huquqi bilan klass a’zolari aniqlansa, aniqlangan a’zolar faqat klass ichida ko‘rinadi, ya’ni faqat klass ichida foydalanish mumkin;

- **protected** – ushbu ko‘rinish huquqi bilan aniqlangan klass a’zolari aniqlangan klassda va shu klass yordamida olingan hosila klasslarda ko‘rinadi, ya’ni foydalanish mumkin;

- **internal** – ushbu ko‘rinish huquqi bilan aniqlangan klasslar, obyektlar, interfeyslar, funksiyalar, xususiyatlar, konstruktorlar va ular aniqlangan modulning istalgan qismida ko‘rinadi, ya’ni foydalanish mumkin. Modul bir katalogda joylashgan **Kotlin** dasturlash tilida yozilgan fayllar to‘plamini ifodalaydi;

- **public** – ushbu ko‘rinish huquqi bilan aniqlangan klasslar, funksiyalar, xususiyatlar, obyektlar, interfeyslar dasturning istalgan qismida ko‘rinadi, ya’ni foydalanish mumkin. Bundan tashqari, agar bu ko‘rinish huquqi funksiyalar yoki klasslar boshqa paketda aniqlangan bo‘lsa, ularni **import** qilib foydalanish mumkin.

Ommaviy (public) ko‘rinish huquqi

Ko‘rinish huquqi xususiyatlar va metodlarning oldidagi **var/val/fun** kalit so‘zlaridan oldin joylashtiriladi. Agar ko‘rinish huquqi aniq ko‘rsatilmagan bo‘lsa, unda bu a‘zolar **public** ko‘rinishida bo‘ladi. Quyidagi klass ko‘rinish huquqlari ko‘rsatilmagan holda tuzilgan:

```
class Person() {  
    var name = "Noma'lum"  
    var age = 18  
    fun printPerson() {  
        println("Ismi: $name   Yoshi: $age")  
    }  
}
```

Yuqorida keltirilgan klass quyida keltirilgan klass bilan teng kuchli hisoblanadi:

```
class Person() {  
    public var name = "Noma'lum"  
    public var age = 18  
    public fun printPerson() {  
        println("Ismi: $name   Yoshi: $age")  
    }  
}
```

Agar xususiyatlar asosiy konstruktor orqali e‘lon qilingan bo‘lsa va ko‘rinish huquqlari ko‘rsatilmagan bo‘lsa, u holda bu xususiyatlar ham **public** huquqida bo‘ladi. Quyidagi ikki klass bunda misol bo‘la oladi:

```
class Person(val name: String, val age: Int) {  
    public fun printPerson() {  
        println("Ismi: $name   Yoshi: $age")  
    }  
}
```

Asosiy konstruktorda keltirilgan ikki xususiyatning ko‘rinish huquqi ko‘rsatilmagan bo‘lsada, ular **public** ko‘rinish huquqiga ega, ya‘ni:

```
class Person(public val name: String, public val age: Int) {  
    public fun printPerson() {  
        println("Ismi: $name   Yoshi: $age")  
    }  
}
```

Hususiyl (private) ko‘rinish huquqi

Agar klass a‘zolariga **private** ko‘rinish huquqi qo‘llanilsa, bu a‘zolariga tashqi tomondan, ya‘ni klassdan tashqarida murojaat qilish mumkin bo‘lmaydi. Bu ko‘rinish huquqi klass a‘zolarini o‘zining ichida ishlatish va ularning ustida turli amallar bajarish uchun ruhsat beradi.

```

class Person(private val name:String, _age: Int){
    private val age = _age
    fun printPerson(){
        printName()
        printAge()
    }
    private fun printName(){
        println("Ismi: $name")
    }
    private fun printAge(){
        println("Yoshi: $age")
    }
}
fun main() {
    val tohir = Person("Tohir", 37)
    tohir.printPerson()
    //println(tohir.name) //Xatolik!-name xususiyati -
private
    //tohir.printAge() //Xatolik!-printAge metodi - private
}

```

Yuqorida keltirilgan dasturdagi klassda **Person** klassi ikkita xususiyatni aniqlaydi: *name* (shaxsning ismi) va *age* (shaxsning yoshi). Aniqroq qilib aytganda bitta xususiyat konstruktor yordamida, ikkinchi xususiyat klass maydoni sifatida aniqlangan. Bu ikki xususiyat **private** ko‘rinish huquqi bilan aniqlanganligi uchun ularga faqat klass ichida murojaat qilish mumkin. Ularga klassdan tashqari murojaat qilish mumkin emas.

Yuqoridagi klassda xususiyatlardan tashqari uchta metod ham mavjud. Bu metodlar *printPerson*, *printAge* va *printName*. Bu metodlardan *printAge* va *printName* ikkisi **private** ko‘rinish huquqi bilan belgilangan. Bitta *printPerson* metodiga ko‘rinish huquqi belgilanmagan va bu metod xususiy ko‘rinish huquqidagi ikki metodga murojaat qilishi ko‘rsatib o‘tilgan.

Xususiy ko‘rinish huquqi bilan belgilangan metodlarga faqat klass ichida murojaat qilinganligi sababli, ularni klassdan tashqarida chaqirish xatolikni yuzaga keltiradi.

```

println(tohir.name)    //Xatolik!-name xususiyati - private
tohir.printAge()       //Xatolik!-printAge metodi - private

```

Agar dasturda yuqorida keltirilgandek **private** ko‘rinish huquqiga ega bo‘lgan metod yoki xususiyatga murojaat qilinsa, dastur xatoligi yuzaga keladi.

Konstruktor ham ko‘rinish huquqlariga ega bo‘ladi. Agar asosiy konstruktorga ko‘rinish huquqi qo‘llanilsa, unda bu konstruktorni aniqlash

uchun **constructor** kalit soʻzidan foydalanish lozim boʻladi. Asosiy va ikkilamchi konstruktorlarni aniqlashdagi koʻrinish huquqi **constructor** kalit soʻzidan oldin ishlatiladi.

```
class Person private constructor(val name:String) {  
    protected constructor(val name:String, var age:Int) {}  
}
```

4.5. Qiymat oʻrnatuvchi va qiymat oluvchi metodlar

Qiymat oʻrnatuvchi va qiymat oluvchi metodlar bu klassning qaysidir xususiy maydoni ustida amallar bajarishdan iborat boʻladi. Har bir xususiyat aslida oʻzining maydoniga ega boʻladi. Bu maydon koʻp holatlarda **private** koʻrinish huquqi bilan belgilanadi. Xususiy koʻrinish huquqi bilan belgilangan maydonlarga maʼlumotlar kiritish va undagi maʼlumotlarni oʻqish uchun qiymat oʻrnatuvchi va qiymat oluvchi metodlar yozishga toʻgʻri keladi. **Kotlin** dasturlash tilida bunday metodlarni yozish uchun xususiyatlardan foydalangan holda amalga oshiriladi. Bu ikki metod dasturlashda getter (**getter** – qiymat oluvchi) va setter (**setter** – qiymat oʻrnatuvchi) metodlar deb yuritiladi. Bu metodlarni yozish qoidasi quyida keltirilgan:

```
var xususiyat_nomi[: xususiyat_tipi] [=boshlangʻich_qiymat]  
    [getter]  
    [setter]
```

Kotlin dasturlash tilida **getter** va **setter** metodlari klassning aʼzolaridan tashqari yuqori darajadagi oʻzgaruvchilarga yaʼni, global yoki asosiy funksiyadan tashqarida eʼlon qilingan oʻzgurchan oʻzgaruvchilarga ham qoʻllanilishi mumkin. Quyida global oʻzgaruvchiga **setter** metodi qoʻllanilgan:

```
var age:Int = 1  
    set(value) {  
        if((value > 0) and (value <= 150))  
            field = value  
    }  
fun main() {  
    print(age)  
    age = 45  
    print(age)  
    age = -35  
    print(age)  
}
```

Global oʻzgaruvchiga **setter** metodi u aniqlangandan soʻng darhol yoziladi. Yuqorida keltirilgan dasturda **age** nomli global oʻzgaruvchi berilgan. Unda soʻng bu oʻzgaruvchiga **setter** metodi qoʻshimcha

yozilgan. **Setter** metodi bitta parametrga ega bo'lgan anonim funksiyadan iborat. Agar dasturning ixtiyoriy qismida bu global o'zgaruvchining qiymatini o'zgartirish uchun chaqirilsa, u holda **setter** metodi orqali uning qiymati o'zgartirilishi mumkin. Sababi **setter** metodi global o'zgaruvchi qabul qilishi mumkin bo'lgan qiymatlarni tekshiradi. Agar tekshirishdagi shart bajarilsa, global o'zgaruvchining qiymati o'zgartiriladi, aks holda uning qiymati o'zgarishsiz qoldiriladi. Masalan, global o'zgaruvchiga shartni qanoatlantiruvchi 45 soni qiymat sifatida uzatilganda uning qiymati o'zgaradi. **Setter** metodiga e'tibor berilsa, unda *field* nomdagi o'zgaruvchi mavjud bo'lib, bu o'zgaruvchi **setter** metodiga tegishli bo'lgan o'zgaruvchini bildiradi. Bu *field* o'zgaruvchisi kompilyator tomonidan avtomatik ravishda e'lon qilinadi. Yuqorida keltirilgan dasturning natijasi quyidagilardan iborat:

```
18
45
45
```

Yuqoridagi dasturda faqat **setter** metodi berilgan. Shu dasturga **getter** metodini qo'shib kengaytirish mumkin. Dasturni kengaytirishda qaysi metod ish bajarganini bilish maqsadida **setter** va **getter** metodlarga bitta qo'shimcha chiqarish operatorini qo'shib qo'yilsa va **getter** metodini **setter** metodidan so'ng yozilsa, maqsad muvofiq bo'ladi.

```
var age: Int = 18
    set(value) {
        println("Qiymat joylashtitish: ")
        if((value > 0) and (value < 150))
            field = value
    }
    get() {
        print("Qiymatni olish: ")
        return field
    }
fun main() {
    println(age)
    age = 45
    println(age)
    age = -35
    println(age)
}
```

Bu dasturni ishga tushirilganda konsol oynasida quyidagi natija hosil bo'ladi:

```
Qiymatni olish: 18
Qiymat joylashtitish:
Qiymatni olish: 45
```

Qiymat joylashtirish:
Qiymatni olish: 45

Klasslarda getter va setter metodlaridan foydalanish

Getter va **setter** metodlarini global o'zgaruvchilarda ishlatish mumkin bo'lsada, bu metodlar odatda klasslar xususiyatlarining qiymatlari ustida vositachilik qiladi. Yuqorida keltirilgan global o'zgaruvchi klassning xususiyati sifatida kelganda, klassning ko'rinishi quyidagicha bo'ladi:

```
class Person(val name: String) {  
    var age: Int = 0  
    set(value) {  
        if((value > 0) and (value <= 150)) field = value  
    }  
}
```

Getter metodini xususiyatlari boshqa bo'lgan klassda ko'riladigan bo'lsa, bu klassning ko'rinishi quyidagicha bo'ladi:

```
class Person(var firstname: String, var lastname: String) {  
    val fullname: String  
    get() = "$firstname $lastname"  
}
```

Bu yerda *fullname* xususiyati klassning ikki maydonidagi, ya'ni *firstname* (ismi) va *lastname* (familiyasi) qiymatlarni birlashtirib beradi. Shu bilan birga klass shaxsni anglatgani uchun shu shaxsning to'liq ismini qaytaradi. Klassning *fullname* xususiyatiga e'tibor qaratilsa, bu xususiyat **val** kalit so'zi, ya'ni o'zgarmas o'zgaruvchi sifatida aniqlangan. Bu bilan ushbu xususiyatni ma'lumot qabul qilishi cheklangan. Ammo klassdagi *firstname* va *lastname* xususiyatlarining qiymatlari o'zgarsa, *fullname* xususiyatining qiymati o'zgaradi.

Qiymatlarni saqlash uchun maydonlardan foydalanish

Klasslarda xususiy maydonlar mavjud. Bu maydonlarga qiymatlarni uzatish va undagi qiymatni olish uchun yangi bitta maydon yoki xususiyat yaratilishi kerak. Bunda yangi yaratilgan xususiyat uchun **getter** va **setter** metodlarini qo'llagan holda, xususiy maydon bilan ishlashni tashkil qilsa bo'ladi. Shunda xususiyatning qiymati shu xususiyatga tegishli bo'lgan maydonda saqlanishini ta'minlash mumkin. Yuqoridagi **Person** nomli klassga tegishli o'zgartirishlar kiritgan holda to'liq klass qanday ko'rinishda bo'lishini quyidagi dasturda ko'rish mumkin:

```
class Person(private var _firstname:String, private var  
_lastname:String) {  
    private var _age:Int = 1  
    private var _company:String = ""
```

```

    constructor(_firstname:String, _lastname:String,
_age:Int) : this(_firstname, _lastname) { this._age = _age }
    constructor(_firstname:String, _lastname:String,
_age:Int, _company:String) : this(_firstname, _lastname,
_age) { this._company = _company }
    var firstname:String
        set(value) {
            _firstname = value
        }
        get() = _firstname
    var lastname:String
        set(value) {
            _lastname = value
        }
        get() = _lastname
    var age:Int
        set(value) {
            _age = value
        }
        get() = _age
    var company:String
        set(value) {
            _company = value
        }
        get() = _company
    fun personString():String = "Ismi: $_firstname\nSharifi:
$_lastname\nYoshi: $_age\nIsh joyi: $_company";
}

```

Bu klassda to'rtta xususiy maydon mavjud bo'lib, ular *_firstname*, *_lastname*, *_age* va *_company* lardir. Bu maydonlar mos ravishda shaxsning ismi, familiyasi, yoshi va ish joyini bildiradi. Bu maydonlarga mos ravishda xususiyatlar ham mavjud. Bu xususiyatlarning har biriga **getter** va **setter** metodlari yozilgan. Bu xususiyatlar *firstname*, *lastname*, *age* va *company* lardir. Keltirilgan maydonlar va xususiyatlardan tashqari bitta alohida metod ham yozilgan. Bu metodning nomi *personString* bo'lib, shaxs haqida to'liq ma'lumot beradi. Klassning **setter** metodida maydon uchun hech qanday cheklovlar qo'yilgan emas. Agar cheklov qo'yilishi kerak bo'lsa, birinchi navbatda qaysi maydonga qanday ko'rinishda cheklov qo'yish kerakligini tasavvur qilib olish va uni **setter** metodida buyruqlar yordamida qo'llash tavsiya etiladi.

Bu klassdan foydalanilgan holda dasturning asosiy funksiyasi quyidagicha bo'ladi:

```

fun main() {
    val bobur = Person("Bobur", "Qosimov")
    bobur.age = 28
}

```

```

    bobur.company = "Soliq boshqarmasi"
    println(bobur.personString())
    println("=====")
    val aziza = Person("Aziza", "Komilova", 32)
    aziza.company="TATU"
    println(aziza.personString())
}

```

Bu dastur konsolda quyidagicha natija beradi:

```

Ismi: Bobur
Sharifi: Qosimov
Yoshi: 28
Ish joyi: Soliq boshqarmasi
=====
Ismi: Aziza
Sharifi: Komilova
Yoshi: 32
Ish joyi: TATU

```

4.6. Paketlar va ularni qo‘shib olish

Kotlin dasturlash tilida klasslar va funksiyalarni mantiqiy birlashmasi bo‘lgan paketlardan foydalanish tavsiya etiladi. Bu paketlarda bitta muammoni hal etish uchun qaratilgan klasslar yoki funksiyalar to‘plamidan iborat bo‘ladi. Bunday paketlar hosil qilishda kompyuterning kataloglari va fayl nomlaridan keng foydalaniladi. Paketlarni **package** kalit so‘zi bilan aniqlash mumkin. Masalan, *email* nomli paket hosil qilish quyida berilgan:

```
package email
```

Bu buyruq faylning birinchi qatorida joylashgan bo‘ladi va faylni ichidagi barcha tarkib ushbu paketga tegishli hisoblanadi. Masalan, loyihaga yangi *email.kt* nomdagi fayl qo‘shiladi. Yaratilgan faylga quyida berilgan klass yoziladi:

```

package email
class Message(val text: String)
fun send(message: Message, address: String) {
    println("'${message.text}' ushbu habar $address manzilga
jo'natildi")
}

```

Yaratilgan paket *email* deb nomlanadi. Paket **Message** nomli klassni o‘z ichiga olgan, unda bitta *text* nomli xususiyat mavjud. Nisbatan aytganda, bu klass elektron pochta xabarini ifodalaydi va *text* xususiyatida matn joylashadi. Bu paketda shuningdek shartli ravishda elektron pochtaga xabar jo‘natuvchi *send* nomli funksiya ham aniqlangan. Ushbu paketda aniqlangan kattaliklarni boshqa faylda ishlatish kerak bo‘ldi.

```
fun main() {
    var mes = email.Message("Salom Kotlin")
    email.send(mes, "bobur@gmail.com")
}
```

Yuqoridagi dasturda paket ko'rsatilmagan bo'lsa ham, aslida *email* paketiga tengishli bo'lgan klass va funksiyadan foydalanilmoqda. Klass va funksiyaning oldida paketning ismi yozilgan. Agar paketdagi har bir klass yoki funksiyaning oldiga har safar paket nomi yoziladigan bo'lsa, dasturchining vaqtini ko'p ketishiga va dasturning hajmini ortishiga olib keladi. Buni oldini olish uchun, paketni ulash kerak bo'ladi. Paketni ulash uchun **import** kalit so'zidan foydalaniladi. Paket ulab ishlab chiqilgan dastur quyida keltirilgan:

```
import email.*
fun main() {
    var mes = Message("Salom Kotlin")
    send(mes, "bobur@gmail.com")
}
```

Bu dasturda paketdagi barcha klasslar va funksiyalarni yaratilayotgan dasturga ulash ko'rsatib o'tilgan. Paketning barcha kattaliklarini chaqirish **import email.*** buyrug'idan iborat. Aynan kerak bo'lgan klasslar va funksiyalarni chaqirish quyida keltirib o'tilgan:

```
import email.Message
import email.send
```

Chaqirilgan kattalikni belgilab olish

Paketdan kerakli kattaliklarni chaqirib olinganda uning nomi uzun yoki tushunarsiz bo'lishi mumkin. Bunday holatlar dastur tuzmoqchi bo'lgan foydalanuvchiga noqulayliklar keltirib chiqaradi. Bu noqulaylikni bartaraf etish uchun **as** kalit so'zidan foydalanib, chaqirilgan kattalikni foydalanuvchi o'ziga mos nom bilan belgilab olishi mumkin. Masalan, yuqorida keltirilgan dasturdagi ikki kattalikni boshqa nom bilan belgilash quyidagicha amalga oshiriladi:

```
import email.Message as EmailMessage
import email.send as sendEmail
fun main() {
    var mes = EmailMessage("Salom Kotlin")
    sendEmail(mes, "bobur@gmail.com")
}
```

Nazorat savollari:

1. Klass tushunchasini tushuntirib bering.
2. Xususiyat deganda nimani tushunasiz?
3. Klass maydoni deganda nimani tushunasiz?

4. Konstruktor va uning turlarini aytib bering.
5. Obyektlar qanday aniqlanadi?
6. Ko‘rinish huquqlari sanab bering.
7. Metod deganda nimani tushunasiz?
8. Getter va setter metodlarning vazifasini tushuntirib bering.
9. Paket deganda nimani tushunasiz?
10. Paketlardan foydalanish uchun qaysi kalit so‘zdan foydalaniladi?
11. Klass yaratishda qanday kalit so‘z ishlatiladi?
12. Qaysi ko‘rinish huquqini yozmaslik mumkin?

5. Interfeyslar va vorislilik

5.1. Vorislilik

Vorislilik bu obyektga yo'naltirilgan dasturlash tilining asosiy tamoyillaridan bir bo'lib, mavjud klasslarning funksional imkoniyatlaridan foydalanilgan holda, imkoniyatlari kengaytirib, yangi klass hosil qilish imkoniyatini yaratadi. Vorislilikning asosiy ikki komponentasi mavjud. Bu komponentalar yaratilayotgan klassning vazifasini belgilaydigan asosiy klass (*parent-class*, *super-class*) va asosiy klassning vazifalarini meros qilib oladigan va uni kengaytirish yoki o'zgaritirish mumkin bo'lgan voris klass (*child-class*, *sub-class*) deb nomlash mumkin.

Klassning funksional imkoniyatlaridan meros olish uchun, bu klass **open** kalit so'zi yordamida aniqlangan bo'lishi kerak. Agar bu kalit so'z yordamida klass aniqlangan bo'lmasa, bu klassdan voris olib bo'lmaydi. Vorislilik yordamida klass yaratilayotganda klass qaysi klassdan meros olayotgan bo'lsa, yaratilayotgan klassning nomidan so'ng ikki nuqta (:) qo'yilib, asosiy klassning nomi ko'rsatiladi. Masalan:

```
open class Person{
    var name: String = "Noma'lum"
    fun printName(){
        println(name)
    }
}
class Employee: Person()
```

Bu yerda, **Person** klassi *name* xususiyati va *printName* metodiga ega bo'lgan shaxsni ifodalaydi. Korxona yoki firmaning xodimini ifodalash uchun shartli ravishda **Employee** nomli klass keltirilgan. Xodimlar shaxs bo'lganligi uchun, xodimlarni ifodalovchi klassning umumiy vazifalari shaxsning vazifalari bilan bir-biriga tushadi. Shuning uchun **Employee** klassdagi *name* hususiyatini qayta aniqlash o'rniga, **Person** klassining barcha funksiyalaridan meros olish ma'qul bo'ladi. Ya'ni, bu holda **Person** klassi asosiy klass yoki *super-class*, **Employee** klassi esa voris klass yoki *sub-class* deb yuritiladi.

Shuni yodda tutish kerakki, klasslardan vorislilik olinayotgan vaqtda asosiy klassning konstruktorini chaqirish kerak. Agar konstruktori mavjud bo'lmasa, u holda kompilyator aniqlagan standart konstruktorini yozish kerak.

Yuqorida keltirilgan **Person** klassining asosiy konstruktori mavjud emas, shuning uchun **Person** klassining standart konstruktorini **Employee** klassini aniqlashda chaqirish lozim.

Vorislik yordamida klass yaratish vaqtida asosiy klass konstruktoridan foydalanishning ikki yo‘li mavjud. Birinchi usul – bu yuqorida ko‘rsatilgan usul bo‘lib, unda yaratilayotgan klass nomidan keyin asosiy klassning konstruktorini ko‘rsatish, ya’ni:

```
class Employee: Person()
```

Asosiy klassning konstruktorini chaqirishning ikkinchi usuli, bu hosil qilinadigan klassning ikkilamchi konstruktorini aniqlash va unda **super** kalit so‘zi yordamida asosiy klassning konstruktorini chaqirishdan iborat. Masalan,

```
open class Person{
    var name: String = "Noma'lum"
    fun printName(){
        println(name)
    }
}
class Employee: Person{
    constructor():super(){}
}
```

Bu yerda **Employee** klassining ikkinchi darajali konstruktoridan **constructor** kalit so‘zida foydalanilgan holda, asosiy klassning konstruktoridan voris klass yaratilmoqda: **constructor():super()**. Ya’ni, bu yerda **super** kalit so‘zi asosiy klassning konstruktoriga murojaatni anglatadi.

Qaysi usuldan foydalanishdan qat’iy nazar, **Employee** klassining obyektini yaratish va **Person** klassidan meros sifatida o‘tgan funksiya va xususiyatlardan foydalanish mumkin:

```
fun main() {
    val bobur: Employee = Employee()
    bobur.name = "Bobur"
    bobur.printName()
}
open class Person{
    var name: String = "Noma'lum"
    fun printName(){
        println(name)
    }
}
class Employee: Person()
```

Asosiy konstruktor yordamida vorislik

Agar asosiy klassning konstruktori aniq belgilansa (asosiy yoki ikkilamchi konstruktor), voris klass asosiy klassning konstruktorini

chaqirishi lozim. Birinchi usul – konstruktorni voris klass nomidan keyin ikki nuqta qo‘yib chaqirish:

```
open class Person(val name: String) {  
    fun printName() {  
        println(name)  
    }  
}  
  
class Employee(empName: String) : Person(empName)
```

Bu yerda **Person** klassi konstruktor orqali *name* xususiyatini o‘rnatadi. Shuning uchun, **Employee** klassi **String** qiymatini oluvchi va uni **Person** klassi konstruktoriga uzatuvchi konstruktorni aniqlaydi.

Agar voris klass aniq boshlang‘ich konstruktor yoki asosiy konstruktorga ega bo‘lmasa, ikkilamchi konstruktor chaqirilganda asosiy klass konstruktori **super** kalit so‘z orqali chaqirilishi kerak:

```
open class Person(val name: String) {  
    fun printName() {  
        println(name)  
    }  
}  
  
class Employee: Person {  
    constructor(empName: String) : super(empName) {}  
}
```

Person klassining konstruktori bitta parametr qabul qilganligi va bu konstruktorga murojaat qilish uchun **super** kalit so‘ziga bitta parametr berish yordamida murojaat qilinadi. Klasslardan foydalanish quyidagi dasturda keltirilgan:

```
fun main() {  
    val bobur = Employee("Bobur")  
    bobur.printName()  
}  
  
open class Person(val name: String) {  
    fun printName() {  
        println(name)  
    }  
}  
  
class Employee(empName: String) : Person(empName)
```

Yuqorida keltirilgan asosiy klassda asosiy konstruktor aniqlangan, lekin shunday holatlar kuzatiladiki, asosiy klassda ikkilamchi konstruktor aniqlangan. Ikkilamchi konstruktor aniqlangan holatlarda voris klassda ham ikkilamchi konstruktordan foydalanish maqsadga muvofiq. Bunga misol sifatida quyidagi dasturni ko‘rib chiqish mumkin:

```
fun main() {  
    val bob = Employee("Bob")  
}
```

```

        bob.printName()
    }
    open class Person{
        val name: String
        constructor(userName: String){
            name = userName
        }
        fun printName(){
            println(name)
        }
    }
    class Employee(empName: String): Person(empName)

```

Voris klassni kengaytirish

Voris klass asosiy klassdan metodlarni meros qilib oladi, lekin o‘z metodlarini ham aniqlashi mumkin. Bunga misol sifatida quyida dastur keltirilgan:

```

fun main() {
    val bobur = Employee("Bobur", "TATU FF")
    bobur.printName()
    bobur.printCompany()
}
open class Person(val name: String){
    fun printName(){
        println(name)
    }
}
class Employee(empName: String, val company: String):
    Person(empName) {
        fun printCompany(){
            println(company)
        }
    }
}

```

Yuqorida keltirilgan dasturda **Employee** klassi voris klass bo‘lib, u o‘zida xodimning ish joyini saqlaydigan *company* nomli xususiyat va *printCompany* metodini aniqlaydi.

Kotlin dasturlash tili voris klass yaratishda bitta klassdan voris olishi mumkin. Ya’ni voris klass bir vaqtning o‘zida bir nechta klassga voris bo‘la olmaydi.

Kotlin dasturlash tilida barcha klasslar **Any** klassining vorisi hisoblanadi, ya’ni **Any** klassi asosiy klass sifatida aniq ko‘rsatilmasa ham. Shuning uchun har qanday klassda **Any** klassida aniqlangan barcha xususiyatlar va metodlari mavjud bo‘ladi. Yaratilgan barcha klasslar *equals*, *toString*, *hashCode* kabi metodlarga ega bo‘ladi.

5.2. Xususiyatlarni va metodlarni boshqarish

Kotlin dasturlash tili asosiy klass aniqlagan funksiyalar va xususiyatlarni bekor qilish yoki qayta yozish imkonini beradi. Asosiy klass **open** kalit soʻzi yordamida yozilganligi sababli unda yozilgan metodlarni qayta yozish mumkin. Agar asosiy klassning biror metodi qayta yozish kerak boʻlsa, metodning oldiga **override** kalit soʻzi yoziladi.

Xususiyatlarni qayta yozish

Xususiyatlar voris klassda qayta yozish uchun **open** kalit soʻzi yordamida taʼriflanadi:

```
open class Person(val name: String) {  
    open var age: Int = 1  
}
```

Yuqorida keltirilgan klassdagi *age* xususiyatini qayta yozish mumkin. Agar xususiyat asosiy konstruktor orqali yaratilgan boʻlsa, **open** kalit soʻzi uni eʼlon qilish vaqtida uning oldin yoziladi:

```
open class Person(val name: String, open var age: Int = 1) {  
}
```

Voris klassda qayta yoziladigan xususiyatdan oldin **override** kalit soʻzidan yoziladi.

```
open class Person(val name: String, open var age: Int = 1) {  
}  
open class Employee(name: String) : Person(name) {  
    override var age: Int = 18  
}
```

Bu yerda *age* xususiyatining boshlangʻich qiymatini bekor qilish va unga boshqa boshlangʻich qiymat berish koʻrsatib oʻtilgan. Xususiyatlarni qayta yozish deganda ularning boshlangʻich qiymatini oʻzgartirish tushuniladi. Xususiyatni voris klassning konstruktorida ham qayta yozish mumkin:

```
open class Person(val name:String, open var age:Int = 1) {}  
open class Employee(name:String, override var age:Int = 18) :  
    Person(name, age) {}
```

Quyidagi dasturda xususiyatni qayta yozish koʻrsatib oʻtilgan:

```
fun main() {  
    val tohir = Person("Tohir")  
    println("Ismi: ${tohir.name}   Yoshi: ${tohir.age}")  
    val bobur = Employee("Bobur")  
    println("Ismi: ${bobur.name}   Yoshi: ${bobur.age}")  
}
```

```

open class Person(val name:String, open var age:Int = 1)
open class Employee(name:String, override var age:Int = 18):
    Person(name, age)

```

Yuqorida keltirilgan dasturning konsoldagi natijasi quyidagidan iborat:

```

Ismi: Tohir   Yoshi: 1
Ismi: Bobur   Yoshi: 18

```

Getter va setter metodlarini qayta yozish

Xususiyatlarni qayta yozish kabi uning **getter** va **setter** metodlarini ham qayta yozish mumkin. Bu metodlarni qayta yozish uchun voris klassda qayta yozilayotgan xususiyatning oldiga **override** kalit soʻzini yozish va unga tegishli boʻlgan metodni koʻrsatish kifoya.

```

open class Person(val name: String) {
    open val fullInfo: String
        get() = "Shaxs: $name - $age"
    open var age: Int = 1
        set(value) {
            if(value > 0 && value < 110)
                field = value
        }
}

open class Employee(name: String): Person(name) {
    override val fullInfo: String
        get() = "Xodim: $name - $age"
    override var age: Int = 18
        set(value) {
            if(value > 17 && value < 110)
                field = value
        }
}

fun main() {
    val tohir = Person("Tohir")
    tohir.age = 14
    println(tohir.fullInfo)
    val bobur = Employee("Bobur")
    bobur.age = 14
    println(bobur.fullInfo)
}

```

Yuqoridagi dasturda **Employee** klassidagi *fullInfo* va *age* xususiyatlari qayta yozilgan xususiyat va bu xususiyatlarga tegishli boʻlgan metodlar ham qayta yozilgan metodlardir.

Metodlarni qayta yozish

Asosiy klassning qayta yozilishi mumkin boʻlgan metodlarining oldiga ham **open** kalit soʻzi yozilib, bu kalit soʻz yordamida

xususiyatlardagi kabi voris klassda metodni qayta yozishni bildiradi. Voris klassda qayta yozilishi kerak bo'lgan metodning oldiga **override** kalit so'zi yoziladi:

```
open class Person(val name: String) {
    open fun display() {
        println("Ismi: $name")
    }
}
class Employee(name: String, val company: String) :
    Person(name) {
        override fun display() {
            println("Ismi: $name    Ish joyi: $company")
        }
    }
fun main() {
    val tohir = Person("Tohir")
    tohir.display()
    val bobur = Employee("Bobur", "TATU")
    bobur.display()
}
```

Yuqorida keltirilgan dasturda voris klassda *display* nomli metod qayta yozilmoqda.

Voris klassi iyerarxiyasidagi ustunlik

Kotlin dasturlash tilida metodlar vorislik iyerarxiyasi davomida qayta yozilishi mumkin. Bu shuni anglatadiki, ko'p qatlamli vorislik bajarilganda, eng yuqoridagi klassning xususiyati yoki metodi eng quyidagi klassda ham qayta yuklanishi mumkin. Quyidagi dasturda **Person** nomdagi klassdan **Employee** nomli klass yaratilgan bo'lib, bu klassdan esa **Manager** klassi hosil qilingan. **Manager** klassida **Person** klassiga mansub bo'lgan *display* metodi qayta yozilmoqda. Shu metodni o'zi **Employee** klassida ham qayta yuklangandi. Ammo **Employee** klassida bu metodga qayta yozish uchun buyruq berilmagan. **Employee** klassi **Person** klassiga mansub bo'lganligi sababli va **Person** klassida bu metodni qayta yozish uchun ruhsat berilgan. **Person** klassi eng yuqorida turganligi sababli bu metodni **Manager** klassida ham qayta yozish mumkin.

```
open class Person(val name: String) {
    open fun display() {
        println("Ismi: $name")
    }
}
open class Employee(name: String, val company: String) :
    Person(name) {
```

```

        override fun display() {
            println("Ismi: $name   Ish joyi: $company")
        }
    }
}
class Manager(name: String, company: String):Employee(name,
company) {
    override fun display() {
        println("Ismi: $name   Ish joyi: $company   Lavozimi:
Manager")
    }
}

```

Qayta yozishni ta'qiqlash

Kotlin dasturlash tilida vorislik yordamida klasslar hosil qilish vaqtida qayta yoziluvchi xususiyatlar va metodlarni qayta yozishni ta'qiqlash ham mumkin. Buning uchun qayta yoziladigan va shu klassdan voris olinganda, voris klassda xususiyat yoki metodni qayta yozishni ta'qiqlash uchun **final** kalit so'zidan foydalaniladi.

```

open class Person(val name: String) {
    open fun display() {
        println("Ismi: $name")
    }
}
open class Employee(name: String, val company: String):
Person(name) {
    final override fun display() {
        println("Ismi: $name   Ish joyi: $company")
    }
}
class Manager(name: String, company: String):Employee(name,
company) {
    // display metodini qayta yozish mumkin emas.
}

```

Voris klassdan asosiy klass a'zolariga murojaat

Shunday vaziyatlar bo'lib turadiki, unda voris klass a'zolaridan asosiy klassning a'zolariga murojaat qilish kerak bo'ladi. Bunday hollarda **Kotlin** dasturlash tilida asosiy klassni bildiruvchi **super** kalit so'zidan foydalaniladi. Bu kalit so'z klass o'zidan oldingi voris olingan klassni bildiradi.

```

open class Person(val name: String) {
    open val fullInfo: String
        get() = "Ismi: $name"
    open fun display() {
        println("Ismi: $name")
    }
}

```

```

open class Employee(name: String, val company: String):
    Person(name) {
        override val fullInfo: String
            get() = "${super.fullInfo} Company: $company"
        final override fun display() {
            super.display()
            println("Company: $company")
        }
    }
}

```

5.3. Mavhum klasslar va metodlar

Mavhum klasslar – bu **abstract** kalit soʻzi yoki modifikatori bilan aniqlangan klasslardir. Mavhum klasslarning oʻziga xos asosiy xususiyati shundan iboratki, bu klassdan obyekt hosil qilib boʻlmaydi. Quyidagi dasturda **Human** nomli mavhum klass aniqlangan.

```

abstract class Human(val name: String)

```

Mavhum klass oddiy klasslar singari xususiyatlarga va metodlarga ega boʻlishi mumkin, lekin uning konstruktoriga toʻgʻridan-toʻgʻri murojaat qilib obyekt hosil qilish mumkin emas. Ammo oʻzgaruvchi eʼlon qilish mumkin.

```

val jamol: Human // toʻgʻri
val aziza: Human = Human("Aziza") // xato

```

Mavhum klassdan faqat voris olib foydalanish mumkin. Quyidagi dasturda mavhum klassdan voris olish koʻrsatib oʻtilgan:

```

abstract class Human(val name: String) {
    fun hello() {
        println("Mening ismim $name")
    }
}

class Person(name: String): Human(name)

```

Oddiy klasslardan voris olishda ishlatiladigan **open** kalit soʻzini mavhum klasslarda ishlatish shart emas. Chunki bu klassdan obyekt hosil qilib boʻlmasligi sababli mavhum klass faqat vorislik uchun ishlatiladi. Mavhum klass yordamida eʼlon qilingan oʻzgaruvchiga mavhum klassdan voris sifatida olingan ixtiyoriy klassning obyektini berilishi mumkin. Masalan,

```

abstract class Human(val name: String) {
    fun hello() {
        println("Mening ismim $name")
    }
}

class Person(name: String): Human(name)

fun main(args: Array<String>) {

```



```

    val kamol: Person = Person("Kamol")
    val salim: Human = Person("Salim Mirzo")
    kamol.hello()
    salim.hello()
}

```

Mavhum klasslar mavhum xususiyat va mavhum metodlarga ega bo‘lishi mumkin. Mavhum metodlar va mavhum xususiyatlar **abstract** kalit so‘zi bilan aniqlaniladi. Mavhum metodlarning vazifasi ko‘rsatilmaydi, ya’ni metodning tana qismi mavjud emas, mavhum xususiyatlar ham qiymatga ega bo‘lmaydi yoki qiymati ko‘rsatilmaydi. Mavhum metodlarni va mavhum xususiyatlarni oddiy klasslarda aniqlashning imkoni yo‘q, mavhum a‘zolar faqat mavhum klasslarda aniqlanishi mumkin:

```

abstract class Human(val name: String) {
    abstract var age: Int
    abstract fun hello()
}
class Person(name: String): Human(name) {
    override var age : Int = 1
    override fun hello() {
        println("Mening ismim $name")
    }
}

```

Agar klass mavhum klassning vorisi bo‘lsa, u holda mavhum klassda ko‘rsatilgan mavhum a‘zolari o‘zida qayta yozishiga to‘g‘ri keladi. Shuning uchun mavhum klassdan yaratilgan voris klassda mavhum klassning mavhum a‘zolarini barchasini qayta yozish kerak bo‘ladi. Yuqoridagi dasturda **Person** klassi voris klass bo‘lganligi sababli uning tana qismida **Human** klassida ko‘rsatilgan *hello* metodini va *age* xususiyatini qayta yozish kerak bo‘ladi. Qayta yozishni ko‘rsatish uchun **override** kalit so‘zidan foydalaniladi. Mavhum xususiyatlarni voris klassning asosiy konstruktorda ham qayta yozish yoki e‘lon qilish mumkin:

```

abstract class Human(val name: String) {
    abstract var age: Int
    abstract fun hello()
}
class Person(name: String, override var age : Int):
    Human(name) {
        override fun hello() {
            println("Mening ismim $name")
        }
    }
}

```

Nega dasturlashda mavhum klasslar kerak? Klasslar odatda qandaydir haqiqiy obyektни ifodalaydi. Ammo bu obyektlarning ba'zilari to'g'ridan-to'g'ri ko'rinishga ega bo'lmagan mavhumlikni ifodalaydi. Masalan, geometrik shakllar tizimini olaylik. Aslida bunday geometrik shakl yo'q. Doira, to'rtburchak, kvadrat shakllar mavjud, lekin shakllarning hech qanday qiymati yo'q. Biroq, doira ham to'rtburchak ham umumiy qilib aytganda shakldir. Bunday holda, bu shakllar uchun mavhum **Figure** nomli klass belgilash mumkin. Bu mavhum klassdan voris olingan holda boshqa barcha shakllar uchun klasslar yaratish mumkin:

```
import kotlin.math.PI as M_PI
abstract class Figure {
    abstract fun perimeter(): Float
    abstract fun area(): Float
}
class Rectangle(val width: Float, val height: Float) :
    Figure() {
    override fun perimeter(): Float{
        return width * 2 + height * 2;
    }
    override fun area(): Float{
        return width * height;
    }
}
class Circle(val radius: Float):Figure(){
    override fun perimeter(): Float {
        return (2 * M_PI.toFloat() * radius)
    }
    override fun area(): Float {
        return (M_PI.toFloat() * radius * radius)
    }
}
```

Yuqoridagi dastur qismida bitta mavhum klass va ikkita mavhum klass yordamida yaratilgan voris klasslar keltirilgan. Mavhum klassning mavhum metodlari voris klasslarda qayta yozilib, yaratilgan klass uchun kerakli bo'lgan ma'lumotlarni qaytarish xususiyatiga ega. Dasturning yuqori qismida, ya'ni birinchi qatorida **Kotlin** dasturlash tilining paketlaridan biri bo'lgan matematik funksiyalar bilan ishlaydigan paket chaqirilib, undagi aynan matematik kattalik bo'lgan π sonini chaqirish va uni belgilash ko'rsatilgan.

5.4. Interfeyslar

Interfeys – bu obyekt bajarishi mumkin bo'lgan harakatlar tavsifi. Interfeys – bu klassga ma'lum xususiyatlarni kiritishga imkon beradigan

dasturlash kattaligi. Interfeyslar klass bajarilishi kerak bo'lgan shartlarni ifodalaydi. Interfeyslar xususiyatlar va metodlar deklaratsiyasini, shuningdek ularning standart bajarilishini o'z ichiga olishi mumkin. Interfeys aniqlashda **interface** kalit so'zidan foydalaniladi. Masalan,

```
interface Movable{
    var speed: Int
    fun move()
    fun stop(){
        println("To'xtatish")
    }
}
```

Yuqorida keltirilgan **Movable** nomli interfeys mashinalarni harakatini ifodalaydi. Bu interfeys ikkita metod va bitta xususiyatdan iborat. Interfeysning *move* metodining xatti-harakati mavjud emas, ya'ni tana qismi yo'q. Ikkinchi *stop* metodining tana qismi mavjud yoki standart tanaga ega. Interfeyslarda xususiyatlarni aniqlash jarayonida ularning boshlang'ich qiymatlari berilmaydi. Interfeys orqali to'g'ridan-to'g'ri obyekt yaratish mumkin emas, chunki interfeyslar konstruktorga ega emas. Shunchaki interfeyslar klassga mos tushadigan andozani ifodalaydi. Quyidagi dastur qismida interfeys orqali yaratilgan ikki oddiy klasslarni ko'rish mumkin. Bu klasslarni yaratishda yuqorida keltirilgan interfeysdan foydalanilgan.

```
class Car : Movable{
    override var speed = 60
    override fun move(){
        println("Mashina $speed km/soat tezlikda
harakatlanmoqda")
    }
}
class Aircraft : Movable{
    override var speed = 600
    override fun move(){
        println("Samolyot $speed km/soat tezlikda uchmoqda")
    }
    override fun stop(){
        println("Qo'nish")
    }
}
```

Interfeysdan foydalanish uchun yaratilayotgan klass nomidan so'ng, ikki nuqta qo'yilib, foydalanilayotgan interfeys nomi ko'rsatiladi. Interfeysdan foydalanilganda klass interfeysda aniqlangan xususiyatlar va metodlarni qayta yozishni amalga oshiradi. Agar interfeysda metodning tana qismi yozilgan bo'lsa, bu metodni xatti-harakatini o'zgartirish

xususiyatiga ega. Qayta yozilishi kerak bo'lgan xususiyat va metodlarning oldiga **override** kalit so'zi yoziladi.

Yuqorida keltirilgan klasslarda mashinani ifodalaydigan **Car** klassi, **Movable** interfeysidan foydalanilgan holda yaratilgan. Interfeysda *move* metodi aniqlangan, ammo bu metodning tana qismi mavjud emas. Shu sababli metodning xatti-harakatini **Car** klassida yozish kerak. Xuddi metodlarda bo'lgani kabi xususiyatlarni ham qayta yozish kerak. Xususiyatlarni qayta yozish uchun boshlang'ich qiymat berish kerak bo'ladi. Lekin *stop* metodini **Car** klassida qayta yozish shart emas, chunki bu metodning xatti-harakati interfeysning o'zida aniqlangan.

Yuqoridagi klasslarda **Car** klassidan tashqari samolyotlarni ifodalaydigan **Aircraft** klassi ham mavjud. Bu klass ham **Movable** interfeysi orqali yaratilgan. Bu klassda interfeysning ikki metodini qayta yozish ko'rsatib o'tilgan.

Interfeys yordamida o'zgaruvchi e'lon qilib, uning qiymatlariga yuqorida berilgan ikki klassning obyektlarini qiymat sifatida uzatish quyidagi dasturda ko'rsatib o'tilgan. Shuni esdan chiqarmaslik kerakki, interfeys orqali obyekt yaratish mumkin emas.

```
fun main() {  
    val m1: Movable = Car()  
    val m2: Movable = Aircraft()  
    m1.move()  
    m1.stop()  
    m2.move()  
    m2.stop()  
}
```

Bu dastur bajarilganda konsolda quyidagi natija chiqadi:

```
Mashina 60 km/soat tezlikda harakatlanmoqda  
To'xtatish  
Samolyot 600 km/soat tezlikda uchmoqda  
Qo'nish
```

Klasslar bir vaqtning o'zida bir nechta interfeyslardan foydalanishi mumkin. Bu holda interfeyslar vergul bilan ajratib yoziladi. Yuqorida keltirilgan **Car** klassidan foydalanilgan holda unga **Info** nomli interfeys qo'shib, klassning imkoniyatini kengaytirish mumkin. **Info** interfeysi o'zida ikki xususiyat *model* (obyektning modeli) va *number* (obyektning raqami) ni o'zida saqlasin. Interfeysning ko'rinishi quyidagicha:

```
interface Info{  
    val model: String  
        get() = "Noma'lum"  
    val number: String
```

```
}
```

Bu interfeysdan foydalanish quyidagicha:

```
class Car(override val model: String, override var number:
String) : Movable, Info{
    override var speed = 60
    override fun move(){
        println("Mashina $speed km/soat tezlikda
harakatlanmoqda")
    }
}
```

Berilgan interfeyslar va klassdan foydalanish quyidagi dasturda berilgan:

```
fun main() {
    val malibu: Car = Car("Malibu", "40 A 222 AA")
    println(malibu.model)
    println(malibu.number)
    malibu.move()
    malibu.stop()
}
```

Kotlin dasturlash tilida vorislik asosida yaratilayotgan klasslarda interfeyslardan foydalanish mumkin. Bunda asosiy klassdan keyin vergul bilan ajratilgan holda, foydalaniladigan interfeyslar nomlari ko'rsatiladi. Shunday vaziyatlar bo'ladiki, unda asosiy klassning metodini nomi bilan interfeysdagi metodning nomi ustma-ust tushadi ya'ni, nomlari bir xil bo'ladi. Bu holatda qaysi metodni bajarishni quyidagicha ko'rsatish mumkin:

```
open class Video {
    open fun play() { println("Play video") }
}
interface AudioPlayable {
    fun play() { println("Play audio") }
}
class MediaPlayer() : Video(), AudioPlayable {
    override fun play() {
        super<Video>.play()
        super<AudioPlayable>.play()
    }
}
```

MediaPlayer nomli klassda *play* nomli metod qayta yuklanmoqda. Bu metodni tana qismida asosiy klass va interfeysining metodlariga murojaat qanday bo'lishi ko'rsatib o'tilgan.

5.5. Ichki klasslar va interfeyslar

Kotlin dasturlash tilida klasslar va interfeyslarni yaratishda boshqa klass va interfeyslarni ular tarkibida aniqlash mumkin. Bunday klasslar (ichki klass yoki nested classes) deb yuritiladi. Ichki klasslar shu klass yoki interfeys uchun yordamchi vazifani bajaradi. Masalan, quyidagi holatda ichki klass aniqlangan:

```
class Person{
    class Account(val username:String, val password:String){
        fun showDetails(){
            println("UserName: $username")
            println("Password: $password")
        }
    }
}
```

Bu yerda **Account** klassi ichki klass bo'lib, bu klass joylashgan **Person** klassi esa tashqi klass deb yuritiladi. **Account** ichki klassning ko'rinish huquqi **public** bo'lib, bu klass dasturning istalgan joyida ko'rinadi. Lekin ichki klassga murojaat qilish uchun tashqi klass nomidan foydalanish kerak. Masalan, ichki klass obyektini yaratish uchun:

```
fun main() {
    val userAcc = Person.Account("admin", "123456");
    userAcc.showDetails()
}
```

Agar ichki klass **private** ko'rinish huquqi bilan aniqlangan bo'lsa, unda ichki klassning xususiyatlariga qiymat berish uchun, ichki klassning obyektini tashqi klassda aniqlangan bo'lishi kerak va kerakli ma'lumotlar tashqi klassdan foydalanilgan holda uzatiladi:

```
class Person(username: String, password: String){
    private val account: Account = Account(username,
password)
    private class Account(val username: String, val password:
String)
    fun showAccountDetails(){
        println("UserName: ${account.username}")
        println("Password: $account.password")
    }
}
fun main() {
    val tohir = Person("admin", "123456");
    tohir.showAccountDetails()
}
```

Klasslar ichki interfeyslarni o'z ichiga olishi mumkin. Bundan tashqari interfeyslar ham klasslarni o'z ichiga olishi mumkin.

```

interface ExternalInterface {
    class NestedClass
    interface NestedInterface
}
class ExternalClass {
    class NestedClass
    interface NestedInterface
}

```

inner klasslar

Ichki klasslar tashqi klassning xususiyatlari va metodlaridan foydalana olmaydi. Masalan, quyidagi dasturda buni ko‘rish mumkin. Agar ichki klass tashqi klassning metodidan foydalanish uchun harakat qiladigan bo‘lsa, xatolik beradi:

```

class BankAccount(private var sum: Int) {
    fun display(){
        println("sum = $sum")
    }
    class Transaction{
        fun pay(s: Int) {
            sum -= s           // bu xato
            display()          // bu xato
        }
    }
}

```

Bu yerda **BankAccount** nomli klass bank hisobini anglatuvchi klass hisoblanadi, u *sum* nomli xususiyatga ega. Bundan tashqari bank hisobidagi qiymatni ko‘rsatish uchun, bu klassda *display* nomli metod mavjud.

BankAccount klassining hisob operatsiyasini ifodalovchi, ushbu klass ichida joylashgan **Transaction** nomli klass berilgan. **Transaction** klassi ushbu hisobdan to‘lov qilish uchun *pay* nomli metodni o‘z ichiga oladi. Biroq **Transaction** klassi **BankAccount** klassning xususiyatlari va metodlaridan foydalana olmaydi.

Bunday holatlarni oldini olish uchun, ya‘ni ichki klass tashqi klassning xususiyatlari va metodlaridan foydalana olishi uchun **inner** kalit so‘zi yordamida belgilanishi kerak. Masalan:

```

fun main() {
    val acc = BankAccount(3400);
    acc.Transaction().pay(2500)
}
class BankAccount(private var sum: Int) {
    fun display(){
        println("sum = $sum")
    }
}

```

```

    }
    inner class Transaction{
        fun pay(s: Int){
            sum -= s
            display()
        }
    }
}

```

Bu yerda **Transaction** klassi **inner** kalit soʻzi yordamida belgilanganligi uchun **BankAccount** klassining xususiyatlari va metodlaridan toʻliq foydalana oladi. Agar ichki klass obyektidan foydalanish zaruriyati tugʻilsa, unda tashqi klassning obyektini yaratish kerak boʻladi. Tashqi klassning obyektini yaratish dasturning asosiy qismida berilgan. Bu quyidagidan iborat:

```

val acc = BankAccount(3400)
acc.Transaction().pay(2500)

```

Aʼzolar nomlarining ustma–ust tushishi

Ichki klassning xususiyatlari va metodlarini nomlari tashqi klassning xususiyatlari va metodlari nomlari bilan bir xil boʻlishi mumkin. Bunday holatlarda ichki klass tashqi klassning xususiyatlari va metodlariga quyidagi koʻrinishda murojaat qilinadi: *this@class_name.property_name* yoki *this@class_name.function_name*

```

class A{
    private val n: Int = 1
    inner class B{
        private val n: Int = 1
        fun action(){
            println(n)           // B klassining xususiyati
            println(this.n)      // B klassining xususiyati
            println(this@B.n)    // B klassining xususiyati
            println(this@A.n)    // A klassining xususiyati
        }
    }
}

```

Yuqorida berilgan **BankAccount** klassi bilan ichki **Transaction** klassini qayta yoziladigan boʻlsa, quyidagi koʻrinishga keladi:

```

fun main() {
    val acc = BankAccount(3400);
    acc.Transaction(2400).pay()
}
class BankAccount(private var sum: Int){
    fun display(){
        println("sum = $sum")
    }
}

```



```

    inner class Transaction(private var sum: Int) {
        fun pay() {
            this@BankAccount.sum -= this@Transaction.sum
            display()
        }
    }
}

```

5.6. Data klassi

Ba'zan klasslar faqat ba'zi ma'lumotlarni saqlash uchun ishlatiladi. **Kotlin** dasturlash tilida bu ko'rinishdagi klasslar ma'lumot klasslari (data classes) deb ataladi. Ma'lumot klasslari **data** kalit so'zi yordamida aniqlanadi:

```
data class Person(val name: String, val age: Int)
```

Ma'lumotlar klassini kompilyatsiya qilishda kompilyator avtomatik ravishda klassga o'ziga xos metodlarni qo'shadi, bunda klassning asosiy konstruktorida aniqlangan xususiyatlarni hisobga oladi. Qo'shiladigan metodlar quyidagilar:

- equals – ikkita obyektning solishtirish metodi;
- hashCode – obyektning xesh kodini qaytarish metodi;
- toString – obyektning satrli ko'rinishini qaytaruvchi metod;
- copy – obyekt ma'lumotlarini boshqa obyektga ko'chiruvchi metod.

Masalan, **toString** metodi yuqorida aytilganidek satrli ko'rinishda ma'lumot qaytaradi. Bu metodni oddiy klasslarda qanday qiymat qaytishi quyidagi dasturda keltirilgan:

```

fun main() {
    val aziza: Person = Person("Aziza ", 24)
    println(aziza.toString())
}

class Person(val name: String, val age: Int)

```

Bu dasturni ishlatilganida konsolda Person@3b9a45b3 ko'rinishdagi natija hosil bo'ladi (@ belgisidan keyingi belgilar ketma-ketligi o'zgarishi mumkin).

Agar yuqoridagi dasturda berilgan klass ma'lumotlar klassida bo'lsa, inson tushunadigan ma'lumotlarni beradi. Buning uchun **Person** klassi ma'lumotlar klassida bo'lishi kerak. Ma'lumotlar klassi oddiy klassdan ko'rinish jihatidan farq qilmaydi. Faqat **class** kalit so'zidan oldin **data** kalit so'zini qo'shish kifoya. Dasturdagi klassni ma'lumotlar klassiga o'tkazish quyidagi dasturda berilgan:

```
fun main() {
    val aziza: Person = Person("Aziza ", 24)
    println(aziza.toString())
}
```

```
data class Person(val name: String, val age: Int)
```

Bu dasturdan natija `Person(name=Aziza, age=24)` ko‘rinishda bo‘ladi. Bu natija obyektini qaysi klassga tegishliligini va xususiyatlarining qiymatlarigacha ma’lumot beradi. Ma’lumotlar klassidagi obyektning satrli ko‘rinishini olish uchun **toString** metodini qayta yozish shart emas. Bu turdagi klasslarda avtomatik ravishda klassning satrli ko‘rinishi aniqlanadi. Agar dasturchi ma’lumotlar klassining **toString** metodining natijasidan qoniqmasa bu metodni qayta yozishi mumkin. Quyida ma’lumotlar klassining **toString** metodini qayta yozish ko‘rsatib o‘tilgan:

```
data class Person(val name: String, val age: Int) {
    override fun toString(): String {
        return "Ismi: $name Yoshi: $age"
    }
}
```

Ma’lumotlar klassini **copy** metodini ishlatish quyidagi dasturda ko‘rsatilgan:

```
fun main() {
    val aziza: Person = Person("Aziza", 24)
    val naima = aziza.copy(name = "Naima")
    println(aziza.toString())
    println(naima.toString())
}
data class Person(var name: String, var age: Int)
```

Ma’lumotlar klassi aniqlash uchun bir nechta shartlar mavjud. Bu shartlar quyidagilardan iborat:

- ma’lumotlar klassining asosiy konstruktorida kamida bitta parametrlar bo‘lishi shart;
- ma’lumotlar klassining asosiy konstruktoridagi barcha parametrlar **val** yoki **var** kalit so‘zlari yozilishi shart, ya’ni xususiyatlar sifatida belgilanishi kerak;
- ma’lumotlar klassining asosiy konstruktoridan tashqari aniqlangan xususiyatlar **toString**, **equals** va **hashCode** metodlarida ishlatilmaydi;
- ma’lumotlar klassi **open**, **abstract**, **sealed** yoki **inner** modifikatsiyasi bilan aniqlanmasligi kerak.

Ma’lumotlar klassining asosiy konstruktorida xususiyatlarni **val** yoki **var** kalit so‘zlari bilan aniqlash mumkin.

Umuman olganda ma'lumotlar klassidagi xususiyatlar **val** orqali aniqlash, ya'ni ularni o'zgarmas holga keltirish tavsiya etiladi, chunki ular asosida **HashMap** kabi to'plamda obyekt kaliti sifatida ishlatiladigan xesh kodini hisoblab chiqadi.

Kotlin dasturlash tilida ma'lumotlar klasslari xususiyatlari qiymatlarini o'zgaruvchilarga bo'lishish qobiliyati mavjud.

```
fun main() {  
    val aziza: Person = Person("Aziza", 24)  
    val (username, userage) = aziza  
    println("Ismi: $username Yoshi: $userage")  
}
```

```
data class Person(var name: String, var age: Int)
```

Bu dasturning natijasi quyida ko'rsatilgan:

```
Ismi: Aziza Yoshi: 24
```

5.7. Enum to'plami

Ro'yxatlar yoki to'plamlar mantiqiy bog'liq doimiy kattaliklar bo'lib, to'plamlarni aniqlash uchun **enum** kalit so'zidan foydalaniladi. **Kotlin** dasturlash tilida to'plamlar ham klass sifatida e'lon qilinadi. Masalan, hafta kunlaridan iborat bo'lgan to'plam quyidagicha e'lon qilinadi:

```
enum class Day{  
    MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY  
}
```

Bu to'plam yuqorida aytilganidek hafta kunlarini ko'rsatib beradi. To'plamning elementlari ya'ni, o'zgarmaslar vergul bilan ajratilgan holda yoziladi. Har bir element bitta o'zgarmas obyektни ifodalaydi. Yuqorida ko'rsatilgan to'plamdan foydalanish quyidagi dasturda ko'rsatilgan:

```
fun main() {  
    val day: Day = Day.FRIDAY  
    println(day)  
    println(Day.MONDAY)  
}
```

To'plam sinflari oddiy sinflar singari konstruktorga ega bo'lishi ham mumkin. Bunday holatda elementlar konstruktorni chaqirishi mumkin.

```
enum class Day(val value: Int) {  
    MONDAY(1), TUESDAY(2), WEDNESDAY(3),  
    THURSDAY(4), FRIDAY(5), SATURDAY(6), SUNDAY(7)  
}  
  
fun main() {  
    val day: Day = Day.FRIDAY  
    println(day.value)  
    println(Day.MONDAY.value)
```

```
}
```

Yuqoridagi dasturda konstruktor orqali o'zgarmaslarning qiymatlarini berish ko'rsatib o'tilgan. Bu to'plamdagi har bir konstantani qiymatlari *value* xususiyati yordamida beriladi. Ammo bu to'plam qiymatlar ro'yxati emas. To'plamlarda xususiyatdan tashqari metodlar ham bo'lishi mumkin. To'plamlarda metodlar mavjud bo'lsa, konstantalardan so'ng nuqtali vergul qo'yiladi.

```
enum class Day(val value: Int){
    MONDAY(1), TUESDAY(2), WEDNESDAY(3),
    THURSDAY(4), FRIDAY(5), SATURDAY(6), SUNDAY(7);
    fun getDuration(day: Day): Int{
        return value - day.value;
    }
}
fun main() {
    val day1: Day = Day.FRIDAY
    val day2: Day = Day.MONDAY
    println(day1.getDuration(day2))
}
```

Bu dasturda haftaning ikki kuni orasidagi farqni aniqlaydigan **getDuration** nomli metod berilgan.

Standart xususiyatlar va metodlar

Kotlin dasturlash tilidagi to'plamlarda standart xususiyatlari va metodlar mavjud bo'ladi. To'plamning standart xususiyatlari quyidagilar:

- name – to'plamdagi elementning nomini qaytaradi;
- ordinal – to'plamdagi elementning o'rnini qaytaradi. Elementlar 0 dan boshlab tartiblanadi.

```
enum class Day(val value: Int){
    MONDAY(1), TUESDAY(2), WEDNESDAY(3),
    THURSDAY(4), FRIDAY(5), SATURDAY(6), SUNDAY(7)
}
fun main() {
    val day1: Day = Day.FRIDAY
    println(day1.name)
    println(day1.ordinal)
}
```

Quyida to'plamdagi standart metodlar to'plam nomi bilan ishlatiladi. Bu metodlar quyida berilgan:

- valueOf(value:String) – satr ko'rinishda berilgan element nomini qaytaradi. Agar satrda ko'rsatilgan qiymat mavjud bo'lmasa, xatolik beradi.
- values() – to'plamdagi barcha konstantalarni qaytaradi.

```

fun main() {
    for(day in Day.values())
        println(day)
    println(Day.valueOf("FRIDAY"))
}

```

5.8. Anonim klasslar va obyektlar

Ba’zida dastur tuzish davomida dasturning boshqa joylarida ishlatilmaydigan ma’lum bir klassning obyektini yaratish zarur bo’lib qoladi. Ya’ni klass faqat bitta obyektни yaratish uchun kerak bo’ladi. Bunday holatda, odatdagidek bitta klass yaratib, unda obyekt yaratish mumkin. Ammo **Kotlin** dasturlash tili bunday holatlar uchun anonim klass obyektini aniqlash imkoniyatini beradi.

Anonim klasslarni aniqlash uchun **class** kalit so’zidan foydalanilmaydi. Anonim klasslarning nomi yo’q, lekin oddiy klasslar singari boshqa klasslardan meros olish yoki interfeyslardan foydalanishlari mumkin. Anonim klasslar obyektlari anonim obyekt deb ataladi. Anonim obyektlarni **object** kalit so’zi bilan aniqlanadi. Masalan, quyida anonim obyekt hosil qilingan:

```

fun main() {
    val tohir = object {
        val name = "Tohir"
        var age = 37
        fun sayHello(){
            println("Salom, mening ismim $name")
        }
    }
    println("Ismi: ${tohir.name}   Yoshi: ${tohir.age}")
    tohir.sayHello()
}

```

Anonim obyektни ifodalaydigan **object** kalit so’zidan keyin yaratilayotgan obyektning a’zolari figurali qavslar orasida huddi klasslardagi kabi aniqlanadi. Anonim obyektlarda ham klasslardagi kabi a’zolar mavjud, ya’ni xususiyatlar va metodlarni ifodalash mumkin. Anonim obyektlarning xususiyatlariga va metodlariga obyektни saqlab turuvchi o’zgaruvchining nomi bilan murojaat qilish mumkin.

Anonim obyektlarda vorislilik

Vorislilik yordamida anonim obyekt hosil qilish uchun **object** kalit so’zidan keyin ikki nuqta bilan ajratilgan holda voris olinayotgan klass, ya’ni asosiy klassning konstruktori ishlatiladi:

```

fun main() {
    val tohir = object : Person("Tohir"){

```

```

        val company = "Universitet"
        override fun sayHello(){
            println("Salom, mening ismim $name. Men $company
da ishlayman")
        }
    }
    tohir.sayHello()
}

open class Person(val name: String){
    open fun sayHello(){
        println("Salom, mening ismim $name")
    }
}

```

Yuqorida keltirilgan dasturda anonim obyekt **Person** klassidan meros oladi va uning *sayHello* nomli metodini qayta yozadi.

Anonim obyektlar funksiya parametri sifatida

Funksiyalarning parametriga qiymat berishda o‘zgaruvchi yoki o‘zgarmaslardan tashqari anonim obyektlardan ham foydalanish mumkin:

```

fun main() {
    hello(
        object : Person("Salim"){
            val company = "JetBrains"
            override fun sayHello(){
                println("Salom, meming ismim $name. Men
$company da ishlayman")
            }
        }
    )
}
fun hello(person: Person){
    person.sayHello()
}
open class Person(val name: String){
    open fun sayHello() = println("Salom, mening ismim
$name")
}

```

Bu dasturda anonim obyekt **Person** klassidan voris olganligi sababli, bu anonim obyektini **Person** turida parametri mavjud bo‘lgan funksiyaqa qiymat sifatida uzatish mumkin.

Anonim obyektlar funksiya qiymati sifatida

Kotlin dasturlash tilida funksiyalar anonim obyektlar ko‘rinishda qiymat qaytarishi mumkin.

```

fun main() {
    val tohir = createPerson("Tohir", "Institut")
}

```

```

        tohir.sayHello()
    }
    private fun createPerson(_name: String, _company: String) =
    object{
        val name = _name
        val company = _company
        fun sayHello() = println("Salom, Mening ismim $name. Men
        $company da ishlayman")
    }

```

Yuqorida berilgan dasturning ayrim kamchiliklari mavjud. Anonim obyektning xususiyatlari va metodlaridan foydalana olish uchun, anonim obyekt qaytaruvchi funksiya **private** ko‘rinishida e‘lon qilingan bo‘lishi kerak.

Agar funksiyaning **public** yoki **private inline** ko‘rinish huquqida e‘lon qilinsa, anonim obyektning xususiyatlari va metodlaridan foydalanib bo‘lmaydi. Agar funksiya bu ko‘rinish huquqlarida e‘lon qilingan bo‘lsa, vorislik yordamida kamchilikni bartaraf etish mumkin:

```

fun main() {
    val tom = createPerson("Tohir", "Institut")
    println(tom.name)
    // println(tom.company)
}
private inline fun createPerson(_name: String, _comp: String)
= object: Person(_name){
    val company = _comp
}
open class Person(val name: String)

```

Bu dasturda *createPerson* funksiyasi **private inline** ko‘rinish huquqida bo‘lganligi sababli bu funksiyadan qaytadigan anonim obyektning **Person** klassidan voris olgan holda olish mumkin. Shunda **Person** klassida aniqlangan xususiyatlar va metodlarga murojaat qilish huquqi beriladi.

Nazorat savollari:

1. Vorislik deganda nimani tushunasiz?
2. Ko‘rinish huquqlari haqida ma’lumot bering.
3. Mavhum klasslar nima?
4. Anonim obyektlarga misollar keltiring.
5. Anonim klass nima?
6. Data klassi bilan oddiy klassning farqini tushuntirib bering.
7. Ichki klasslar va ichki interfeyslar haqida ma’lumot bering.

6. Obyektga yo'naltirilgan dasturlashning qo'shimcha xususiyatlari

6.1. Istisnolarni boshqarish

Istisno – bu dastur bajarilish jarayonida ro'y beradigan va dasturni normal ishlashini buzadigan hodisa hisoblanadi. Masalan, fayllarni tarmoq orqali uzatish jarayonida tarmoqdagi uzilish va buning natijasida hosil bo'ladigan vaziyat istisno hisoblanadi. Agar istisno hosil bo'lganda uni bartaraf etilmasa, dastur buziladi va ishini tugatadi. Dastur ishini to'g'ri va sifatli tugallashi uchun hosil bo'lgan istisnoli vaziyatlarni hal qilish kerak.

Istisnoli vaziyatlarni hal etish uchun **try..catch..finally** operatorlar birlashmasidan foydalaniladi. Har bir operatori o'zining vazifasiga ega, **try** operatoridan so'ng buyruqlar bloki mavjud bo'lib, bu blok hosil bo'lishi mumkin bo'lgan istisnoli vaziyat o'z ichiga oladi, **catch** operatori tutish bloki bo'lib, hosil bo'lgan istisnoli vaziyatni tutish bilan shug'illanadi va uni bartarab etadi, **finally** operatori istisnoli vaziyatlarni boshqaruvchi operatorlar birlashmasida tugatish vazifasini bajaradi.

```
try {  
    // istisno hosil bo'luvchi qism  
}  
catch (e: Exception) {  
    // istisnolarni bartaraf etish  
}  
finally {  
    // istisnoli vaziyatning oxiri  
}
```

Istisnoni bartaraf etish uchun **catch** blokidan keyin oddiy qavslar ichida, ya'ni, parametr sifatida istisnoning turi beriladi. Ushbu parametrni yuzaga kelgan istisno haqida ma'lumot olish uchun ishlatish mumkin.

Istisnolarni boshqarish operatorida **finally** ixtiyoriy, ya'ni, bu blokni yozmasdan tashlab yuborish ham mumkin. Tutish blokini, ya'ni, **catch** blokini ham tushirib qoldirish mumkin, lekin **try** blokini yozmaslik mumkin emas, **try** bloki yozilganidan so'ng albatta undan keyin **catch** yoki **finally** bloklaridan birini yozish maqsadga muvofiq bo'ladi. Istisnolarni boshqarish blokidagi **try** operatorining buyruqlar blokida bir emas, bir nechta istisnolar yuzaga kelishi mumkin. Bunday hollarda tutish bloki bir emas, bir nechta bo'lishi kerak. Tutish bloki faqat istisnoli vaziyat yuzaga kelganda bajariladi, **finally** bloki esa istisno yuzaga

kelmagan holda ham bajariladi. Masalan, quyida 0 ga bo‘linganda yuzaga keladigan istisno ko‘rsatilgan:

```
fun main() {  
  
    try{  
        val x : Int = 0  
        val z : Int = 0 / x  
        println("z = $z")  
    }  
    catch(e: Exception){  
        println("Exception")  
        println(e.message)  
    }  
}
```

Bu dasturda 0 ga bo‘lishda bo‘ladigan istisno yuzaga keladi. Bu istisnoni **catch** bloki tutib, foydalanuvchiga xabar beradi. Har bir istisno o‘zining turiga ega. Bu dasturda istisnoni tushish uchun **Exception** klassidan foydalanilgan. Bu dasturning natijasi quyidagicha:

```
Exception  
/ by zero
```

Operatorning to‘liq holda asosan fayllar bilan ishlash vaqtida ishlatilsa, agar fayl bilan ishlashda istisno yuzaga kelsa, **finally** blokida faylni yopish maqsadga muvofiq bo‘ladi va fayl ochiq qolishidan saqlaydi. Yuqoridagi dasturda keltirilgan istisnoni boshqarish operatorining to‘liq holda ishlatilganligi quyida ko‘rsatilgan:

```
fun main() {  
  
    try{  
        val x : Int = 0  
        val z : Int = 0 / x  
        println("z = $z")  
    }  
    catch(e: Exception){  
        println("Exception")  
        println(e.message)  
    }  
    finally{  
        println("Dastur yakunlandi")  
    }  
}
```

Bu dasturning natija quyidagicha bo‘ladi:

```
Exception  
/ by zero  
Dastur yakunlandi
```

Istisno haqida ma'lumot berish

Asosiy istisno klassi bu **Exception** klassi bo'lib, istisnolar haqida turli xildagi ma'lumotlarni taqdim etadi. Bu klassning quyidagi xususiyatlari mavjud:

- `message` – istisno haqida ma'lumot berish xususiyati;
- `stackTrace` – istisno holatlari haqida ma'lumot beradigan massiv.

```
fun main() {  
    try{  
        val x : Int = 0  
        val z : Int = 0 / x  
        println("z = $z")  
    }  
    catch(e: Exception){  
        println(e.message)  
        for(line in e.stackTrace) {  
            println("at $line")  
        }  
    }  
}
```

Bir nechta istisnolardan foydalanish

Bitta dasturda bir vaqtning o'zida bir nechta istisnolarni yuzaga keltirish mumkin. Turli xildagi istisnolarni boshqarish uchun alohida-alohida **catch** blokini yozish lozim.

```
try {  
    val nums = arrayOf(1, 2, 3, 4)  
    println(nums[6])  
}  
catch(e:ArrayIndexOutOfBoundsException){  
    println("Indeksga noto'g'ri murojaat")  
}  
catch (e: Exception){  
    println(e.message)  
}
```

Yuqorida keltirilgan dasturda massiv indeksiga noto'g'ri murojaat qilish va unda hosil bo'ladigan istisno ko'rsatib o'tilgan. Massiv indeksiga noto'g'ri murojaat asosan ko'rsatilgan indeks yo'qligidan kelib chiqadi va **ArrayIndexOutOfBoundsException** turidagi istisno hosil bo'ladi. Tutish blokida bu istisno turini ko'rsatib, hosil bo'lgan istisnoni bartaraf etish mumkin. Agar dasturda **ArrayIndexOutOfBoundsException** turi ko'rsatilmasa, unda bu istisnoni bartaraf etadigan boshqa turga o'tkaziladi. Boshqa turdagi istisno asosan **Exception** turida bo'lib, bu tur barcha istisnolarni asosidir. Bu dasturning natijasi quyidagicha:

Indeksga noto'g'ri murojaat

Dasturda tutish blokidan foydalanish jarayonida ishlatiladigan istisno turlarini ko'rsatish vaqtida shuni tartibga solish kerakki, birinchi yozgan istisno turi undan keyin yozilgan istisnoni qamrab olmasligi shart. Agar bunday holat kuzatilsa, birinchi turgan tutish bloki vazifa bajarib, asl xatolikni ko'rsatmasligi mumkin. Yuqoridagi dasturning tutish bloklarini o'rnini almashtirilsa, natija umuman boshqacha chiqadi. Ya'ni dasturning ko'rinishi quyidagicha:

```
try {
    val nums = arrayOf(1, 2, 3, 4)
    println(nums[6])
}
catch (e: Exception){
    println(e.message)
}
catch (e: ArrayIndexOutOfBoundsException){
    println("Indeksga noto'g'ri murojaat")
}
```

Bu ko'rinishdagi dasturning natijasi:

Index 6 out of bounds for length 4

Istisno hosil qilish operatori (throw)

Dastur yaratish vaqtida dasturchi tomonidan istisnolar hosil qilish mumkin. Istisnoni hosil qilish uchun **throw** operatoridan foydalaniladi. Bu operatoridan so'ng istisnoning holati ko'rsatiladi. Masalan, shaxsning yoshi aniqlash jarayoni uchun quyidagi istisno hosil qiluvchi funksiyasi ko'rsatilgan.

```
fun checkAge(age: Int): Int{
    if (age < 1 || age > 110)
        throw Exception("Noto'g'ri qiymat. Yosh chegarasi 0
dan 110 gacha.")
    println("To'g'ri qiymat.")
    return age
}
```

Bu funksiyadan istisno hosil qilinadi. Hosil qilingan istisno tutish blokiga jo'natiladi. Istisnoni tutish qismida hosil qilingan istisnoning habarini ekranga chiqarish quyidagi dasturda to'liq berilgan va dasturning natijasi ham qo'rsatilgan:

```
fun main() {
    try {
        val checkedAge1 = checkAge(5)
        val checkedAge2 = checkAge(-115)
    } catch (e:Exception){
```

```

        println(e.message)
    }
}
fun checkAge(age: Int): Int{
    if(age < 1 || age > 110)
        throw Exception("$age - Noto'g'ri qiymat. Yosh
chegarasi 0 dan 110 gacha.")
    println("$age - To'g'ri qiymat.")
    return age
}

```

Natijasi:

5 - To'g'ri qiymat.

-115 - Noto'g'ri qiymat. Yosh chegarasi 0 dan 110 gacha.

Istisnoli vaziyatdan qiymat qaytarish

Istisnoli vaziyatlar qiymat qaytarish xususiyatiga ega. Bu jarayonni quyidagi dasturda ko'rish mumkin.

```

fun main() {
    val checkedAge1 = try { checkAge(5) } catch (e:
Exception) { null }
    val checkedAge2 = try { checkAge(-125) } catch (e:
Exception) { null }
    println(checkedAge1)
    println(checkedAge2)
}
fun checkAge(age: Int): Int{
    if(age < 1 || age > 110)
        throw Exception("$age - Noto'g'ri qiymat. Yosh chegarasi
0 dan 110 gacha.")
    println("$age - To'g'ri qiymat.")
    return age
}

```

Yuqorida keltirilgan dasturda *checkedAge1* o'zgaruvchisi *checkAge()* funksiyasining qiymatini qabul qiladi. Agar istisno yuzaga kelsa, *checkedAge1* o'zgaruvchisi **catch** blokida ko'rsatilgan qiymatni qabul qiladi, ya'ni, bu dasturda ko'rsatilgan qiymat **null** ga teng. Agar boshqa natija qaytishi kerak bo'lsa, boshqa qiymatni tutish blokiga joylashtirish mumkin.

```

fun main() {
    val checkedAge2 = try { checkAge(-125) } catch (e:
Exception) { println(e.message); 18 }
    println(checkedAge2)
}
fun checkAge(age: Int): Int{
    if(age < 1 || age > 110)

```

```

        throw Exception("$age - Noto'g'ri qiymat. Yosh
chegarasi 0 dan 110 gacha.")
        println("$age - To'g'ri qiymat.")
        return age
    }

```

Yuqoridagi dasturning natijasi quyidagi ko‘rinishda bo‘ladi:

```

-125 - Noto'g'ri qiymat. Yosh chegarasi 0 dan 110 gacha.
18

```

6.2. Nullable turi va null qiymati

Kotlin dasturlash tilida **null** kalit so‘zi maxsus qiymatni ifodalaydi, bu o‘zgaruvchining qiymati yo‘qligini bildiradi. Masalan, quyida bu qiymatdan foydalanish ko‘rsatib o‘tilgan:

```

val n = null
println(n)

```

Bunday qiymatdan foydalanish zarur hollarda amalga oshiriladi. O‘zgaruvchining qiymati aniq bo‘lmagan vaqtda foydalanish maqsadga muvofiq. Yana shuni ta’kidlab o‘tish kerakki, **null** qiymatini standart turdagi, ya’ni, **Int** yoki **String** turidagi o‘zgaruvchilarga qiymat sifatida uzatish xatolikni keltirib chiqaradi. Masalan,

```

var n : Int = null // Xatolik, Int turidagi o'zgaruvchi faqat
raqamlarni qabul qiladi

```

Kotlin dasturlash tilida faqat **Nullable** turini ifodalovchi turlarga **null** qiymatini uzatish mumkin. Oddiy turlarni **Nullable** turiga aylantirish uchun yozilgan turning nomidan keyin so‘roq (?) belgisi qo‘yiladi. Yuqorida keltirilgan o‘zgaruvchini e’lon qilib, unga **null** qiymatini uzatish uchun turdan so‘ng so‘roq belgisini qo‘yish kifoya. Unda o‘zgaruvchi e’loni quyidagicha bo‘ladi:

```

var n : Int? = null

```

Bu o‘zgaruvchi **Nullable** turiga mansub bo‘lganligi sababli, o‘zgaruvchiga **null** yoki ko‘rsatilgan turning qiymatlar oralig‘idagi ixtiyoriy qiymatni qabul qilishi mumkin. Bunga misol sifati quyidagi dasturning bir qismida e’lon qilingan o‘zgaruvchini ko‘rish mumkin:

```

var age : Int? = null
age = 34
var name : String? = null
name = "Tohir"

```

Dasturchi tomonidan ishlab chiqilgan turlarda ham **Nullable** turini ishlatish mumkin. Masalan,

```

fun main() {
    var bobur: Person = Person("Bobur")
}

```

```
// bobur = null // Xatolik!
var tohir: Person? = Person("Tohir")
tohir = null // To'g'ri murojaat
}
class Person(val name: String)
```

Yuqorida keltirilgan dasturdagi `String?` va `Int?` turlari oddiy `String` va `Int` turlari bilan mos emas. `Nullable` turida bir qator cheklovlar mavjud.

- **null** qiymatini to'g'ridan-to'g'ri **Nullable** turida bo'lmagan, ya'ni oddiy ko'rinishdagi turlarga uzatish mumkin emas;
- **Nullable** turiga mansub bo'lgan o'zgaruvchilar o'zida obyektlarni saqlasa, uning xususiyatlari va metodlaridan to'g'ridan-to'g'ri foydalanish mumkin emas;
- **Nullable** turidagi o'zgaruvchini qiymatini funksiyaga argument sifatida uzatish mumkin emas, funksiyalar ma'lum bir qiymatga ega bo'lgan argumentlar qabul qiladi.

?: operatori (Elvis operatori)

Kotlin dasturlash tilining afzalliklaridan biri shundaki, bu dasturlash tilining turlar tizimida **null** qiymatini dastur tuzish vaqtida emas, balki kompilyatsiya vaqtida aniqlash imkonini beradi. Masalan, quyidagi dastur kodini oling:

```
var name : String? = "Tohir"
val username: String = name // Xatolik!
```

Bu dasturda berilgan *name* o'zgaruvchisi o'zida *"Tohir"* qiymatga ega bo'lgan satrni saqlaydi. Bundan tashqari *username* o'zgaruvchisi **String** turiga mansub bo'lib, qiymati satr ko'rinishiga ega bo'lgan belgilar ketma-ketligini saqlay oladi, lekin shunga qaramay bu o'zgaruvchiga *name* o'zgaruvchisining qiymatini to'g'ridan-to'g'ri uzatish mumkin emas. Bunday holda, kompilyator *name* o'zgaruvchisining qiymati nimaga tengligini bilmaydi. Chunki, *name* o'zgaruvchisi **null** qiymatiga ham teng bo'lishi mumkin, ammo oddiy turga ya'ni, **String** turiga mos o'zgaruvchiga **null** qiymatiga ega bo'lgan o'zgaruvchini uzatish mumkin emas. Bunday holda dasturda xatolik kelib chiqadi. Buni bartaraf etish uchun *Elvis* operatoridan (?:) foydalanish mumkin. Bu operator agar **Nullable** turida ko'rsatilgan o'zgaruvchining qiymati **null** bo'lsa, muqobil qiymatni uzatish imkonini yaratadi. Bu operatorining ishlash jarayonini quyidagi dasturning bir qismida ko'rish mumkin:

```
var name : String? = "Tohir"
val username: String = name ?: "Noma'lum"
var age: Int? = 23
val userAge: Int = age ?: 0
```

Bu dastur qismida *name* o'zgaruvchisining qiymati **null** ga teng bo'lsa, *username* o'zgaruvchisiga "Noma'lum" qiymat uzatiladi. Dasturda ko'rsatilgan ikkinchi o'zgaruvchi ya'ni, *age* o'zgaruvchisining qiymati **null** ga teng bo'lsa, *userAge* o'zgaruvchisiga 0 qiymat uzatiladi.

Elvis operatori ikki operandni o'z ichiga oladi. Agar birinchi operandning qiymati **null** bo'lmasa, birinchi operandning qiymati, aks holda, ikkinchi operandning qiymati qaytariladi.

Elvis operatorining vazifasini dasturiy misolda ko'riladigan bo'lsa, quyidagidan iborat:

```
var name : String? = "Tohir"
val userName: String
if (name != null) {
    userName = name
}
```

Lekin **Kotlin** dasturlash tilidagi Elvis operatori dasturni qisqatirishga imkon beradi.

?. operatori (xavfsiz chaqiruv)

Kotlin dasturlash tilida xavfsiz chaqiruv (?.) operatori ko'rsatilgan obyektning qiymati **null** bo'lmasa, obyektning xususiyatlari va metodlaridan foydalanish imkonini beradi.

Masalan, **String** turiga mansub bo'lgan obyekt *length* xususiyatiga ega bo'lib, bu xususiyat satrda qatnashgan belgilar sonini, satr uzunligini qaytaradi. **String?** turida belgilangan o'zgaruvchi saqlayotgan satrning uzunligini topish uchun ham, *length* xususiyatidan foydalanish kerak bo'ladi. Agar **String?** turida belgilangan satr **null** qiymatga ega bo'lsa, unda bu satrning uzunligini olish imkoni mavjud emas. Bunday hollarda satrning uzunligini topish uchun ?. operatoridan foydalanish zarur:

```
var message : String? = "Hello"
val length: Int? = message?.length
```

Yuqorida keltirilgan misolda *message* o'zgaruvchining qiymati **null** bo'lsa, *length* nomli o'zgaruvchining qiymati ham **null** ga teng bo'ladi. Agar *message* o'zgaruvchisi qandaydir belgilar ketma-ketligini qiymat sifatida qabul qilgan bo'lsa, unda bu belgilar ketma-ketligidagi belgilar soni, ya'ni, satrning uzunligi qaytariladi. Bu dasturlash tilida *val length: Int? = message?.length* ifodasi quyida berilgan dastur bilan teng kuchli hisoblanadi:

```
val length: Int?
if (message != null) length = message.length
else
    length = null
```

Xavfsiz chaqiruv operatoridan foydalanib, ixtiyoriy obyektning har qanday xususiyati va metodlaridan foydalanish mumkin. Elvis va xavfsiz chaqiruv operatorlarini birlashtirib, dastur tuzish quyidagicha:

```
val message : String? = "Hello"
val length: Int = message?.length ?:0
```

Bu dasturda *length* o'zgaruvchisi **Nullable** turida emas, balki oddiy butun sonli **Int** turida bo'lib, bu o'zgaruvchiga *message* o'zgaruvchining qiymati qandaydir satrdan iborat bo'lsa, satrning uzunligi, aks holda, 0 qiymat berilishini ifodalamoqda.

Xavfsik chaqiruv operatoridan foydalanib, **null** qiymati tekshiruvlar zanjirini yaratish mumkin. Masalan:

```
fun main() {
    var tohir: Person? = Person("Tohir")
    val tohirName: String? = tohir?.name?.uppercase()
    println(tohirName)
    var bobur: Person? = null
    val boburName: String? = bobur?.name?.uppercase()
    println(boburName)
    var sadir: Person? = Person(null)
    val sadirName: String? = sadir?.name?.uppercase()
    println(sadirName)
}
class Person(val name: String?)
```

Bu dasturda **Person** klassi boshlang'ich konstruktorga ega bo'lib, u **Nullable** ko'rinishidagi **String** turiga mansub bo'lgan xususiyatga ega. Ya'ni, bu xususiyatning qiymati satr yoki **null** ko'rinishida bo'lishi mumkin. Bu dasturdagi **Nullable** ko'rinishidagi **String** turiga mansub bo'lgan o'zgaruvchilarni oddiy **String** turida ifodalash mumkin. Buning uchun Elvis operatoridan foydalanish kerak. Elvis operatoridan foydalanilganda dasturning ko'rinishi quyidagi ko'rinishga keladi:

```
fun main() {
    var tohir: Person? = Person("Tohir")
    val tohirName: String = tohir?.name?.uppercase() ?: ""
    println(tohirName)
    var bobur: Person? = null
    val boburName: String = bobur?.name?.uppercase() ?: ""
    println(boburName)
    var sadir: Person? = Person(null)
    val sadirName: String = sadir?.name?.uppercase() ?: ""
    println(sadirName)
}
class Person(val name: String?)
```


!! operatori

Kotlin dasturlash tilida **null** qiymatlar bilan ishlovchi yana bitta operatori bu – **!!** operatoridir. Bu operator “**null** qiymatga ega emas” tushunchasini beradi. Bu operator bitta operandga ega. Agar operandning qiymati **null** bo‘lsa, istisno yuzaga keladi, aks holda operandning qiymati ustida ko‘rsatilgan amallar yoki operandning xususiyatiga va metodiga murojaat amalga oshiriladi.

```
fun main() {  
    try {  
        val name : String? = "Tohir"  
        val id: String = name!!  
        println(id)  
        val length: Int = name!!.length  
        println(length)  
    }  
    catch (e: Exception) {  
        println(e.message)  
    }  
}
```

6.3. Yo‘naltiruvchi xususiyatlar va delegatlar

Yo‘naltirilgan xususiyatlar ko‘rsatilgan qiymatlarni boshqa klassga olish yoki jo‘natishga imkon beradi. Bu xususiyatlar bilan ishlashda ko‘rsatilgan qiymatlarga ishlov berish, kerakli natijada yuz beradigan kamchiliklarni aniqlab, ularni bartaraf etish uchun qo‘shimcha imkoniyatlarni qo‘shish imkonini beradi. Yo‘naltirilgan xususiyatlar yaratishda quyidagi sintaksisga amal qilinadi.

val/var xususiyat_nomi: turi **by** ifoda

Yuqorida keltirilgan sintaksisda xususiyatning turini ko‘rsatilgandan keyin **by** kalit so‘zi yozilgan. Bu kalit so‘zdan keyin xususiyatga o‘rnatilayotgan qiymatni tekshiruvchi ifoda ko‘rsatiladi. Bu ifoda shartli ravishda **delegat** deb ataladigan klassni ifodalaydi. Yo‘naltiruvchi xususiyatlarni yaratishda hech qanday interfeysdan foydalanish mumkin emas. Yo‘naltiruvchi xususiyatlarni yaratishda o‘zida *getValue* va *setValue* metodlaridan iborat bo‘lgan klassdan foydalaniladi. Xususiyatlardan qiymat olish va ularga qiymat uzatuvchi **get** va **set** metodlari, delegat klassidagi *getValue* va *setValue* metodlariga o‘z vazifalarini topshiradi. Yo‘naltiruvchi xususiyatlarni asosiy konstruktorda e‘lon qilish yoki ko‘rsatib o‘tish mumkin emas.

Delegat klassidagi o‘qish metodi

Faqat o‘qiladigan xususiyatlar, ya’ni **val** kalit so‘zi yordamida e’lon qilingan xususiyatlar uchun delegat klassidagi *getValue* metodining parametrlari quyidagidan iborat bo‘lishi kerak.

Birinchi parametr yo‘naltiruvchi xususiyatning turi yoki shu xususiyat joylashgan klass turida bo‘lishi kerak. Ikkinchi parametr **KProperty**<*> turida ko‘rsatilgan argument. **Kotlin** dasturlash tilida **KProperty** turi maxsus tur bo‘lib, bu turdan foydalanish uchun *kotlin.reflect.** paketida joylashgan **KProperty** klassi chaqirilishi kerak. Yo‘naltiruvchi xususiyatga qiymat qabul qilishda delegat klassidagi *getValue* metodining turi yo‘naltiruvchi xususiyatning turi bilan bir xil ko‘rinishda bo‘lishi kerak. Masalan:

```
import kotlin.reflect.KProperty
fun main() {
    val tohir = Person()
    println(tohir.name)
    val bobur = Person()
    println(bobur.name)
}
class Person{
    val name: String by LoggerDelegate()
}
class LoggerDelegate {
    operator fun getValue(t:Person, p:KProperty<*>):String {
        println("${p.name} nomli xususiyatga murojaat.")
        return "Tohir"
    }
}
```

Bu yerda **Person** klassining *name* xususiyati, yo‘naltiruvchi xususiyat sifatida belgilangan. Yo‘naltiruvchi xususiyatidan qiymat olishda **LoggerDelegate** klassidagi *getValue* metodidan qiymat oladi. Xususiyat **Person** klassida aniqlanganligi sababli delegat klassidagi *getValue* metodining birinchi parametri **Person** turida ifodalanadi. Agar kerak bo‘lsa, ushbu parametr yordamida obyekt haqida qo‘shimcha ma’lumotlarni olish mumkin. Xususiyat **String** turida bo‘lganligi sababli, metod ham **String** turida qiymat qaytaradi. Metodning tana qismida xususiyat uchun qaytarilayotgan qiymatni ko‘rish mumkin. Bu qiymat “Tohir” qatorini qaytaradi, ya’ni, **Person** turida e’lon qilingan obyektning *name* xususiyatiga har safar murojaat qilinganda “Tohir” qatori qaytariladi.

Yuqoridagi keltirilgan dasturdagi xususiyat har safar bir xil qiymat qaytarganligi sababli unga ozgina o‘zgartirish kiritish, ya’ni yaratilayotgan **Person** nomli klassga asosiy konstruktor qo‘shish mumkin. Unda dastur quyidagi ko‘rinishga keladi.

```
import kotlin.reflect.KProperty
fun main() {
    val tohir = Person("Tohir")
    println(tohir.name)
    val bobur = Person("Bobur")
    println(bobur.name)
}
class Person(_name:String){
    val name: String by LoggerDelegate(_name)
}
class LoggerDelegate(val personName:String){
    operator fun getValue(t:Person, p:KProperty<*>):String {
        println("${p.name} nomli xususiyatga murojaat.")
        println("$personName qiymati o'rnatildi.")
        return personName
    }
}
```

Bu dasturda **Person** klassining asosiy konstruktori yordamida *name* xususiyatiga qiymat uzatish ko‘rsatib o‘tilgan. Bu qiymatni yo‘naltiruvchi xususiyat delegat ya’ni, **LoggerDelegate** klassiga uzatadi. Uzatilgan qiymat delegat klassining metodi yordamida obyektning *name* xususiyatiga yo‘naltiriladi. Obyektning xususiyatiga yo‘naltirilishidan oldin kerakli bo‘lgan ma’lumotlar konsol oynasida aks etadi.

Delegat klassida yozish metodi

O‘zgaruvchan xususiyatlar ya’ni, **var** kalit so‘zi bilan aniqlangan xususiyatlarga qiymat o‘rnatish jarayonida delegat klassining *setValue* nomli metodi ish bajaradi. Bu metodni yozishda uchta parametr qatnashib, birinchi parametr *getValue* metodi kabi yo‘naltiruvchi xususiyatning turi yoki shu xususiyat joylashgan klass turida bo‘lishi kerak. Ikkinchi parametr **KProperty<*>** turida ko‘rsatilgan argumentdan iborat. Uchinchi parametr esa o‘zgaruvchi xususiyat turida yoki uning **super** klassi turida bo‘lishi kerak. Masalan,

```
import kotlin.reflect.KProperty
fun main() {
    val tohir = Person("Tohir", 37)
    println(tohir.age)
    tohir.age = 38
    println(tohir.age)
    tohir.age = -139
}
```

```

        println(tohir.age)
    }
    class Person(val name: String, _age: Int) {
        var age: Int by LoggerDelegate(_age)
    }
    class LoggerDelegate(private var personAge: Int) {
        operator fun getValue(t: Person, p: KProperty<*>): Int{
            return personAge
        }
        operator fun setValue(t: Person, p: KProperty<*>, v: Int){
            if (v > 0 && v < 110) {
                println("$v qiymati o'rnatildi.")
                personAge = v
            } else {
                println("$v qiymati o'rnatilmadi.")
                println("Yosh chegara (1;110).")
            }
        }
    }
}

```

Yuqoridagi dasturda **Person** klassi berilgan. Bu klassning *age* nomli xususiyatiga delegat klass sifatida **LoggerDelegate** klassi keltirilgan. Bu *age* xususiyatiga delegat klassining *getValue* va *setValue* nomli metodlari qiymat oʻrnatib va uning qiymatini olib berish xususiyatiga ega. Agar oʻrnatilayotgan qiymat delegat klassining *setValue* metodida keltirilgan shartga mos boʻlsa, u holda qiymat xususiyatga oʻrnatiladi, aks holda, bu qiymatni oʻrnatilmagani haqida axborot beriladi. Yuqorida keltirilgan dasturning natijasi quyida keltirilgan:

```

37
38 qiymati o'rnatildi.
38
-139 qiymati o'rnatilmadi.
Yosh chegara (1;110).
38

```

6.4. Turlarini oʻzgartirish metodlari va operatorlari

Dastur tuzish davomida bir turdan ikkinchi turga oʻtishga toʻgʻri keladi. Bunday vazifalarni bajarish uchun **Kotlin** dasturlash tilida bir nechta metodlar va operatorlar mavjud.

Tur oʻzgartiruvchi metodlar

Maʼlumotlar turini oʻzgartirish uchun bazaviy turda aniqlangan metodlardan foydalanish mumkin. Bazaviy turlarga **Int**, **Long**, **Double** va boshqalar kiradi (Bazaviy turdagi baʼzi metodlar farq qilishi mumkin).

- **toByte()**: o‘zgaruvchining qiymatini **Byte** turiga o‘giradi. Agar o‘girish mumkin bo‘lmasa, *NumberFormatException* ko‘rinishidagi xatolik yuzaga keladi;
- **toShort()**: o‘zgaruvchining qiymatini **Short** turiga o‘giradi. Agar o‘girish mumkin bo‘lmasa, *NumberFormatException* ko‘rinishidagi xatolik yuzaga keladi;
- **toInt()**: o‘zgaruvchining qiymatini **Int** turiga o‘giradi. Agar o‘girish mumkin bo‘lmasa, *NumberFormatException* ko‘rinishidagi xatolik yuzaga keladi;
- **toLong()**: o‘zgaruvchining qiymatini **Long** turiga o‘giradi. Agar o‘girish mumkin bo‘lmasa, *NumberFormatException* ko‘rinishidagi xatolik yuzaga keladi;
- **toFloat()**: o‘zgaruvchining qiymatini **Float** turiga o‘giradi. Agar o‘girish mumkin bo‘lmasa, *NumberFormatException* ko‘rinishidagi xatolik yuzaga keladi;
- **toDouble()**: o‘zgaruvchining qiymatini **Double** turiga o‘giradi. Agar o‘girish mumkin bo‘lmasa, *NumberFormatException* ko‘rinishidagi xatolik yuzaga keladi;
- **toChar()**: o‘zgaruvchining qiymatini **Char** turiga o‘giradi.

Bazaviy turlarning hammasida *toByte* metodi mavjud bo‘lib, bu metod orqali o‘zgaruvchida keltirilgan qiymatni yuqorida aytilganidek **Byte** turiga o‘girish mumkin. Bunga quyida keltirilgan dasturning qismi misol bo‘ladi:

```
var t: Int = 300
println(t.toByte())
```

Bu dasturning qismining natijasi 44 sonidan iborat bo‘lib, **Byte** turi -128 dan 127 gacha bo‘lgan sonli qiymat qabul qiladi. 300 sonini **Byte** ko‘rinishga keltirilganda, bu chegaradan katta bo‘lganligi sababli sanash eng kichik qiymatdan davom etib, 300 – o‘rinda turgan sonli qiymat qaytariladi. Agar o‘zgaruvchining qiymati manfiy bo‘lsa va u chegaradan katta bo‘lsa, sanash teskari tartibda amalga oshiriladi.

```
val s: String = "12"
val d: Int = s.toInt()
println(d)
```

Yuqorida keltirilgan dasturning qismida satr ko‘rinishda berilgan sonli qiymatni butun songa o‘tkazish ko‘rsatilgan. Agar satr ko‘rinishda keltirilgan sonli ketma–ketlik o‘rniga, boshqa belgilardan tashkil topgan qator berilgan bo‘lsa, u holda bu tur almashtirishda istisnoli vaziyat hosil bo‘ladi. Yuqorida aytilganidek *NumberFormatException* nomli istisno

yuzaga keladi. Istisnoli vaziyatni boshqarish operatori yordamida, istisnoni boshqarish mumkin. Masalan:

```
fun main() {  
    val s:String = "tatuff"  
    try {  
        val d: Int = s.toInt()  
        println(d)  
    }  
    catch(e: NumberFormatException){  
        println(e.message)  
    }  
}
```

is operatori

is operatori o'zgaruvchining qiymati ma'lum bir ma'lumotlar turiga tegishli yoki tegishli emasligini aniqlashga imkon beradi. Bu operatorni ishlatish sintaksisi quyidagicha:

o'zgaruvchi_yoki_qiymat **is** ma'lumot_turi

Agar operatorning chap tomonidagi o'zgaruvchi yoki qiymat o'ng tomonda ko'rsatilgan ma'lumot turiga tegishli bo'lsa, bu operator **true** (rost) qiymat qaytaradi. Agar ko'rsatilgan o'zgaruvchi yoki xususiyat turga tegishlilik tekshiruvidan o'tsa, qo'shimcha ko'rinishdagi ma'lumotlardan foydalanilmagan holda, bu operatorning o'zi o'zgaruvchini yoki xususiyatni shu turga o'giradi. Bu o'girishda operator **aqlli translyator** deb yuritiladi. Bu operatorlarni nafaqat bazaviy turlarda balki, dasturchi tomonidan yaratilgan klasslarda va interfeyslarda ham foydalanish mumkin.

```
fun main() {  
    val tohir = Person("Tohir")  
    val bobur = Employee("Bobur", "TATU FF")  
    checkEmployment(tohir)  
    checkEmployment(bobur)  
}  
fun checkEmployment(person: Person){  
    // println("${person.name} ${person.company} da  
ishlaydi.") - bu murojaat xato  
    if(person is Employee){  
        println("${person.name} ${person.company}da  
ishlaydi.")  
    } else {  
        println("${person.name}ning ish joyi ma'lum emas")  
    }  
}  
open class Person(val name: String)  
class Employee(name: String,val company:String):Person(name)
```

Yuqorida keltirilgan dasturda **Employee** klassi **Person** klassidan meros oladi. *checkEmployment* funksiyasi parametr sifatida **Person** klassining obyektini qabul qiladi. Bu funksiyaning tana qismida parametrda kelgan obyekt **Employee** turiga mansubligini tekshirilmoqda, chunki har bir **Person** obyekt **Employee** turiga mansub bo'lmashligi mumkin. Agar ko'rsatilgan obyekt **Employee** turiga mansub bo'lsa, ma'lumot chiqarilayotgan vaqtda uning ish joyini ko'rsatish mumkin, aks holda, ma'lumot chiqarish vaqtida ish joyi ma'lum emasligi haqida xabar beriladi. Agar ko'rsatilgan obyekt **Employee** turiga mansub bo'lsa, **is** operatori yordamida bu obyektning turi avtomatik ravishda **Employee** turiga o'tkaziladi.

is operatorining inkor shakli ham mavjud bo'lib, bu **!is** ko'rinishida yoziladi. Bu ko'rinishda tekshirilayotgan obyekt yoki qiymat ko'rsatilgan ma'lumotlar turiga tegishli bo'lmasa, **true** (rost) qiymat qaytaradi. Unda yuqorida keltirilgan dastur quyidagi ko'rinishda yoziladi:

```
fun main() {
    val tohir = Person("Tohir")
    val bobur = Employee("Bobur", "TATU FF")
    checkEmployment(tohir)
    checkEmployment(bobur)
}

fun checkEmployment(person: Person){
    // println("${person.name} ${person.company} da
ishlaydi.") - bu murojaat xato
    if(person !is Employee){
        println("${person.name}ning ish joyi ma'lum emas")
    } else {
        println("${person.name} ${person.company}da
ishlaydi.")
    }
}

open class Person(val name: String)
class Employee(name: String, val company:String):Person(name)
```

Agar xodimning ish joyi bo'sh bo'ladigan bo'lsa, masalan **val bob = Employee("Bobur", "")** ko'rinishda bo'ladigan bo'lsa, dasturda obyektning turi avtomatik **Employee** turini aniqlaydi va ma'lumot ekranda tushunarsiz holatda chiqarilishi mumkin. Buni bartaraf etish uchun **Employee** klassining *company* xususiyatini bo'sh bo'lmagan qatorga tekshirish kerak bo'ladi. Bunday holda obyektning ma'lumotlar turini tekshirish qismidan keyin ish joyi uchun ishlatiladigan xususiyatning satr uzunligini tekshirish kerak bo'ladi. Ish joyini uzunligini tekshirish quyidagicha amalga oshiriladi:

```
person.company.length > 0
```

Variant tanlash (**when**) operatori yordamida ham **is** operatoridan foydalanish mumkin. Unda dastur qismining ko‘rinishi quyidagicha bo‘ladi:

```
when (person) {  
    is Manager -> println("${person.name} boshqaruvchi.")  
    is Employee -> println("${person.name} hodim.")  
    is Person -> println("${person.name} inson.")  
}
```

is operatori aqlli translyator bo‘lganligi sababli, o‘zgaruvchiga aqlli o‘zgaruvchi deb murojaat qilish mumkin. Aqlli o‘zgaruvchilar quyidagi cheklovlarga ega.

- **val** kalit so‘zi yordamida e‘lon qilingan bo‘lishi kerak (yo‘naltiruvchi xususiyatlar bundan mustasno);
- **open** kalit so‘zi bilan aniqlangan xususiyatlar (ya‘ni, qayta yuklanadigan xususiyatlar) va **getter** metod yozilgan xususiyatlarni aqlli o‘zgaruvchi sifatida ishlatish mumkin emas;
- lambda amallari yordamida qiymat oladigan o‘zgaruvchilarni aqlli o‘zgaruvchi sifatida ishlatish mumkin emas;
- **var** kalit so‘zi bilan e‘lon qilingan o‘zgaruvchilarni aqlli o‘zgaruvchi sifatida ishlatish mumkin emas.

Ko‘rsatilgan turga o‘girish yoki **as** operatori

Kotlin dasturlash tilida bir turdan boshqa turga o‘girishda metodlardan tashqari yana bir operator ham ishlatiladi. Bu operator **as** operatori bo‘lib, bir turdagi qiymatni boshqa turga aniq o‘tkazish uchun ishlatiladi. Bu operatorining sintaksisi quyidagicha:

```
o‘zgaruvchi_yoki_qiymat as ma‘lumot_turi
```

Operatorning chap tomonidagi o‘zgaruvchi yoki qiymat o‘ng tomonda esa, o‘girilishi kerak bo‘lgan ma‘lumot turi ko‘rsatiladi. Masalan, **String?** turidagi qiymatni **String** turiga o‘girish quyidagicha amalga oshiriladi:

```
fun main() {  
    val hello: String? = "Salom Kotlin"  
    val message: String = hello as String  
    println(message)  
}
```

Yuqorida keltirilgan dasturda *hello* o‘zgaruvchisi qandaydir belgilar ketma-ketligiga teng. Agar *hello* o‘zgaruvchisining qiymati **null** bo‘lsa, unda xatolik yuzaga keladi. Sababi bazaviy klasslar **null** qiymatini olmaydi. Shuning uchun *ClassCastException* nomli istisno hosil bo‘ladi.

Bunday holatda, ya'ni istisno qilmaslik uchun operatorning xavfsiz ko'rinishidan foydalanish kerak bo'ladi. Agar ko'rsatilgan ma'lumotni boshqa turiga o'girish muvaffaqiyatsiz yakunlansa, **null** qiymat qaytariladi.

```
val hello: String? = null
val message: String? = hello as? String
println(message)
```

Ushbu operator dasturchi tomonidan tuzilgan klasslar yoki turlarda ham foydalanish mumkin.

```
fun main() {
    val tohir = Person("Tohir")
    val bobur = Employee("Bobur", "TATU FF")
    checkCompany(tohir)
    checkCompany(bobur)
}

fun checkCompany(person: Person) {
    val emp = person as? Employee
    if (emp != null) {
        println("${emp.name} ${emp.company}da ishlaydi.")
    } else {
        println("${person.name} ishchi hodim emas.")
    }
}

open class Person(val name: String)
open class Employee(name:String, var company:String):
    Person(name)
```

Yuqorida keltirilgan dasturda *checkCompany* funksiyasi **Person** klassining obyektini oladi va olingan qiymatni **Person** klassidan voris olgan **Employee** klassiga o'girishga xarakat qiladi. Har bir **Person** klassidagi qiymat **Employee** klassidagi qiymat bo'la olmaydi. Agar ko'rsatilgan **Person** klassidagi qiymat **Employee** klassidagi qiymat bo'la olsa, ekranga uning nomi va ish joyi haqida ma'lumot beradi. Agar ko'rsatilgan qiymat **Employee** klassiga tegishli bo'lmasa, unda ekranga boshqa ma'lumot chiqariladi. Bu dasturning natijasi konsol oynasiga quyidagicha chiqariladi:

```
Tohir ishchi hodim emas.
Bobur TATU FFda ishlaydi.
```

6.5. Turlarni kengaytirish funksiyalari

Kotlin dasturlash tilining boshqa obyektga yo'naltirilgan dasturlash tillaridan farqi shundaki, bu dasturlash tilida bazaviy ma'lumotlar turlari uchun qo'shimcha metodlar yozish mumkinligidir. Qo'shimcha metodlar

yozish bilan bazaviy turlarni kengaytirish imkonini beradi. Turlarni kengaytirish uchun quyidagi sintaksis orqali amalga oshiriladi:

```
fun tur_nomi.metod_nomi(parametrlar):qaytuvchi_tur{  
    funksiyaning_tana_qismi  
}
```

Umuman olganda turlarni kengaytirish uchun yoziladigan metodlar oddiy funksiya kabi yoziladi. Faqat oddiy funksiyalardan farqli tomoni funksiyaning nomini yozishdan oldin qaysi bazaviy turga tegishli ekanligini yozish kerak. Bazaviy turdan so‘ng nuqta qo‘yilib, yaratilayotgan metod nomi yoziladi.

Bazaviy turlardan bo‘lgan **Int** va **String** turlarini kengaytirish maqsadida ularga qo‘shimcha yangi metodlar qo‘shilishi misol sifatida quyidagi dasturda keltirilgan:

```
fun main() {  
    val hello: String = "hello world"  
    println(hello.wordCount('l'))    // 3  
    println(hello.wordCount('o'))    // 2  
    println(4.square())               // 16  
    println(6.square())               // 36  
}  
fun String.wordCount(c: Char) : Int{  
    var count = 0  
    for(n in this)  
        if(n == c) count++  
    return count  
}  
fun Int.square(): Int{  
    return this * this  
}
```

Yuqorida keltirilgan dasturda **Int** bazaviy turi uchun kvadrat metodi aniqlangan. Yaratilgan metodda **this** kalit so‘zi ishlatilgan bo‘lib, bu kalit so‘z yordamida berilgan obyektning qiymati ifodalanmoqda. Masalan, *4.square()* ko‘rinishida 4 ni, *6.square()* ko‘rinishida esa 6 ni ifodalaydi.

String turi uchun satrda ko‘rsatilgan belgi nechtaligini topish uchun *wordCount* metodi aniqlangan. Bazaviy turlarni kengaytirishda yoziladigan metodlarga parametr yozish shu metod yordamida ko‘rsatilgan.

Bazaviy turlarni kengaytirish vaqtida obyektning har qanday umumiy xususiyatlariga va metodlariga murojaat qilish mumkin, lekin xususiy va himoyalangan modifikatorlar yordamida yozilgan xususiyatlar va metodlardan foydalanish mumkin emas.

Bazaviy turlarni kengaytirish metodlari shu bazaviy turning qaysidir metodining nomi bilan ustma-ust tushsa, ya'ni oldindan aniqlangan bo'lsa, oldindan aniqlangan metod vazifasini bajaradi. Dasturchi tomonidan aniqlangan metod e'tiborsiz qoldiriladi.

6.6. infix ko'rinishdagi funksiyalar

Kotlin dasturlash tilida klassning metodiga murojaat qilishning nuqta va qavslarsiz usuli mavjud. Bu usul **infiks** ko'rinishdagi funksiyalar deb yuritiladi. Infiks ko'rinishdagi funksiyalar **infix** kalit so'zi yordamida e'lon qilinadi.

```
infix fun funksiya_nomi(parametr:parametr_turi):  
funksiya_turi{  
    funksiyaning_tana_qismi  
}
```

Infiks ko'rinishdagi funksiyalar quyidagi talablarga javob berishi kerak:

- infiks funksiyalar klassning metodi yoki tur kengaytiruvchi metod bo'lishi kerak;
- infiks funksiya bitta parametrga ega bo'lishi shart;
- parametr aniqlanmagan qiymatlar to'plamida bo'lmasligi kerak.

Infiks funksiyalarini aniqlashning ikki turi mavjud bo'lib, klass metodi va tur kengaytiruvchi metod ko'rinishi yoziladi. Klassning metodi sifatida yozilishiga quyidagi dastur misol bo'la oladi:

```
fun main() {  
    val acc = Account(1000)  
    acc put 150  
    acc.put(150)  
    acc.printSum()  
}  
class Account(var sum: Int) {  
    infix fun put(amount: Int) {  
        sum = sum + amount  
    }  
    fun printSum() = println(sum)  
}
```

Yuqorida keltirilgan dasturda infiks ko'rinishdagi funksiyaга murojlat *acc put 150* ko'rinishida berilgan. Bu ko'rinish *acc.put(150)* ko'rinish bilan bir xil vazifa bajaradi.

Infiks funksiyanı tur kengaytiruvchi metod sifatida yozish quyidagi dasturda keltirilgan. Bunda yuqorida berilgan dasturga o'zgartirishlar kiritilgan.

```

fun main() {
    val acc = Account(1000)
    acc.put 150
    acc.put(150)
    acc.printSum()
}
infix fun Account.put(amount: Int) {
    this.sum = this.sum + amount
}
class Account(var sum: Int) {
    fun printSum() = println(sum)
}

```

Satr ko‘rinishda berilgan o‘zgaruvchi uchun infiks ko‘rinishdagi funksiya quyida keltirilgan. Bunda **String** bazaviy klassi uchun tur kengaytiruvchi metoddan foydalanilgan bo‘lib, bu metod satrda keltirilgan belgilar ketma–ketligi ichidan ko‘rsatilgan belgining sonini aniqlab beradi.

```

fun main() {
    val hello = "hello world"
    val lCount = hello.wordCount('l')
    val oCount = hello.wordCount('o')
    println(lCount)
    println(oCount)
}
infix fun String.wordCount(c: Char) : Int{
    var count = 0
    for(n in this)
        if(n == c) count++
    return count
}

```

Nazorat savollari:

1. Istisnolarni boshqarish deganda nimani tushunasiz?
2. Bazaviy turlar bilan Nullable turlarining farqini nimada?
3. Tur o‘zgartiruvchi metodlarni sanab bering.
4. Tur o‘zgartiruvchi operatorlar haqida ma’lumot bering.
5. Turlarni kengaytirishda dasturlashning qaysi kattaligidan foydalaniladi?
6. Yo‘naltiruvchi xususiyatlar deganda nimani tushunasiz?
7. Delegatlar nima?
8. Bazaviy turlarga null qiymatini uzatish mumkinmi?
9. Infiks ko‘rinishidagi funksiyalar deganda nimani tushunasiz?

7. Umumiy turlar

7.1. Umumiy turlar va funksiyalar

Kotlin dasturlash tilida umumiy klasslar yoki turlar bu shablonlarni ifodalaydi. Umumiy turlar bu obyektning turi ma'lum bo'lmagan tur yoki klass hisoblanadi. Umumiy turlardan foydalanishda obyektning turi ko'rsatilishi kerak. Umumiy turlar **Kotlin** dasturlash tili tomonidan aniqlangan turlar hisoblanib, foydalanuvchi tomonidan ham aniqlanishi mumkin.

Umumiy turlar

Umumiy turlar obyekt turini parametrlangan ko'rinishda ifodalaydi. Umumiy tur yaratish quyidagi ko'rinishda bo'ladi:

```
class Person<T>(val id: T, val name: String)
```

Bu ko'rinishda **Person** klassi **T** parametrdan iborat bo'lib, bu parametr klass nomidan keyin burchakli qavslar ichida ko'rsatiladi. Bu parametr klass aniqlangan vaqtda ma'lum bo'lmagan ba'zi turlarni ifodalash uchun ishlatiladi. Asosiy konstruktorning parametri sifatida keltirilgan xususiyatlardan biri bo'lgan *id* xususiyatini klass aniqlangan vaqtda turi ma'lum bo'lmagan. Shuning uchun bu turni klassning parametri sifatida ko'rsatilib, unga **T** nomli belgilash kiritilgan. **Person** klassidan foydalanish davomida bu parametrning o'rniga *id* xususiyatining turi keltirib o'tiladi. Masalan:

```
fun main() {  
    val tohir: Person<Int> = Person(367, "Tohir")  
    val bobur: Person<String> = Person("A65", "Bobur")  
    println("${tohir.id} - ${tohir.name}")  
    println("${bobur.id} - ${bobur.name}")  
}
```

```
class Person<T>(val id: T, val name: String)
```

Agar konstruktorda **T** parametridan foydalanilgan bo'lsa, u holda obyekt yaratish vaqtida qaysi turda ishlatishni ko'rsatish shart emas. Konstruktorga yozilgan o'zgaruvchining turini **T** parametrining o'rniga qo'yib obyekt yaratiladi. Masalan:

```
val tohir = Person(367, "Tohir")  
val bobur = Person("A65", "Bobur")
```

Klasslarda tur parametri klassning xususiyatlaridan tashqari klassning metodlarini ham aniqlashda keng qo'llaniladi. Bunga misol sifatida quyidagi dasturni ko'rish mumkin:

```

fun main() {
    val tohir = Person("qwert", "Tohir")
    tohir.checkId("qwert")
    tohir.checkId("q34tt")
}
class Person<T>(val id: T, val name: String) {
    fun checkId(_id: T) {
        if (id == _id)
            println("ID lar bir hil")
        else
            println("ID lar har hil")
    }
}

```

Yuqorida keltirilgan dasturda **Person** klassining *id* nomli xususiyati bilan *checkId* metodida keltirilgan *_id* nomli parametr qiymatlari tengligini tekshiradi. Klassning *checkId* metodida keltirilgan *_id* nomli parametr **T** turida ko‘rsatilgan bo‘lib, u klassning *id* nomli xususiyati turi bilan bir xilda yozilgan.

Kotlin dasturlash tilida aniqlangan umumiy turlardan biri bu – **Array** massiv turidir. Bu tur dasturlar yaratishda keng qo‘llaniladi. Bu turning tur parametri yaratilayotgan massivning turini aniqlashga yordam beradi. Bu umumiy turni ishlatish quyida ko‘rsatib o‘tilgan:

```

val people:Array<String> = arrayOf("Tohir", "Bobur", "Samad")
val numbers:Array<Int> = arrayOf(1, 2, 3, 4)

```

Bir nechta parametrlar turini qo‘llash

Umumiy turlarni yaratishda tur parametrlarini bir emas balki, bir nechtasidan ham foydalanish mumkin. Bunda tur parametrlari vergullar bilan ajratilgan holda yoziladi.

```

fun main() {
    var word1: Word<String, String> = Word("one", "bir")
    var word2: Word<String, Int> = Word("two", 2)
    println("${word1.source} - ${word1.target}")
    println("${word2.source} - ${word2.target}")
}
class Word<K, V>(val source: K, var target: V)

```

Yuqorida keltirilgan dasturda ikkita tur parametridan foydalanilgan bo‘lib, bu turlarning birinchisi **K**, ikkinchi tur parametrni **V** bilan belgilangan. Bu yerda birinchi tur parametrning o‘rniga satrli tur **String** ishlatilgan. Ikkinchi tur parametrning o‘rniga birinchi misolda **String** turi, ikkinchi misolda esa **Int** turi ishlatilgan. Bir nechta tur parametrlari ishlatilgan vaqtda, ular bir vaqtning o‘zida bir xil yoki har xil turda bo‘lishi mumkin.

Umumiy funksiyalar

Kotlin dasturlash tilida umumiy klasslar kabi funksiyalarni ham ishlatish mumkin. Bunday funksiyalar umumiy funksiyalar deb yuritiladi.

```
fun main() {  
    display("Hello Kotlin")  
    display(1234)  
    display(true)  
}  
fun <T> display(obj: T) {  
    println(obj)  
}
```

Bu dasturda `display` nomli funksiya berilgan. Bu funksiyaning parametri **T** turida bo'lib, funksiyaning nomini yozishdan oldin tur parametri ko'rsatib o'tilgan. Tur parametrini ko'rsatishda klasslardagi kabi burchakli qavslardan foydalanilgan. Bu funksiyaning vazifasi **T** turidagi qiymatni qabul qilib, uni konsolga chop etishdan iborat. Bu funksiyaning ishlatishda ixtiyoriy qiymat uzatish mumkin.

Umumiy funksiyalar tur parametri keltirilgan turda ham qiymat qaytarishi mumkin. Bundan tashqari **Kotlin** dasturlash tilidagi umumiy turlarda ham qiymatlar qaytarish xususiyatiga ega. Quyidagi dasturda **Array** turiga mansub qiymat qaytarishi ko'rsatib o'tilgan:

```
fun main() {  
    val arr1 = getBiggest(arrayOf(1,2,3), arrayOf(3,4,5,6,7))  
    arr1.forEach { item -> print("$item ") }  
    println()  
    val arr2 = getBiggest(arrayOf("Tohir", "Salim", "Bobur"),  
arrayOf("Karim", "Aziza"))  
    arr2.forEach { item -> print("$item ") }  
}  
fun <T> getBiggest(args1: Array<T>, args2: Array<T>):  
Array<T> {  
    if (args1.size > args2.size) return args1  
    else return args2  
}
```

Yuqorida keltirilgan dasturda *getBiggest* nomli funksiya parametr sifatida ikkita massiv oladi, shu bilan birga bu ikki massivni tashkil etuvchi qiymatlar bir turga mansub bo'lishi ta'minlangan. Bu funksiya berilgan massivlardan elementi ko'p bo'lganini funksiyaning qiymati sifatida qaytarish xususiyatiga ega.

7.2. Umumiy turlarda cheklovlar

Umumiy turlar bilan ishlashda turli xildagi cheklovlar mavjud. Bu cheklovlar tur parametriga bog‘liq bo‘lib, tur parametrda keltirilgan turga qarab turli ko‘rinishda bo‘lishi mumkin. Masalan, ikki obyektни solishtirish funksiyasini aniqlash kerak bo‘lsa, oddiy solishtirish belgilaridan foydalanish mumkin emas. Solishtirish uchun umumiy funksiya yaratish mumkin. Ammo uning tana qismida obyektlarni solishtirishda turli muammolar kelib chiqishi mumkin.

```
fun <T> getBiggest(a: T, b: T): T{  
    if (a > b) return a // Xatolik!  
    else return b  
}
```

Kompilyator bu funksiyanı kompilyatsiya qilmaydi, chunki **T** tur parametri o‘rniga turli xil turlar bo‘lishi mumkin. Turlar turli xil bo‘lganligi sababli solishtirish operatsiyasi bajarilmaydi. Biroq barcha turlar solishtirish interfeysini qo‘llab-quvvatlaydi. Solishtirish interfeysi **Comparable** nomli interfeys hisoblanadi. Solishtirish interfeysini ishlatish quyidagi dasturda ko‘rsatib o‘tilgan:

```
fun main() {  
    val result1 = getBiggest(1, 2)  
    println(result1)  
    val result2 = getBiggest("Tohir", "Salim")  
    println(result2)  
}  
fun <T: Comparable<T>> getBiggest(a: T, b: T): T{  
    return if (a > b) a  
    else b  
}
```

Cheklov tur parametr nomidan keyin ikki nuqta bilan ajratilgan holda, ko‘rsatiladi ya’ni, <T: Comparable<T>>. Bu yerda **T** turi solishtirish interfeysi bilan cheklangan hisoblanadi. Shuni ham aytib o‘tish kerakki, barcha turlarning bazaviy klassi **Any?** turi hisoblanib, tur parametri oddiy ko‘rsatilgan bo‘lsada, aslida u <T: Any?> ko‘rinishi bilan ustma – ust tushadi. Umumiy turlardan foydalanish davomida mahalliy, ya’ni dasturchi tomonidan yaratilgan klasslardan va interfeyslardan foydalanish mumkin. Masalan, shartli ravishda xabar yuborish funksiyasi berilgan bo‘lsin. Unda quyidagi dastur xabar yuborishni ifodalaydi:

```
fun <T:Message> send(message: T){  
    println(message.text)  
}  
interface Message{
```



```

        val text: String
    }
    class EmailMessage(override val text: String): Message
    class SmsMessage(override val text: String): Message

```

Yuqorida keltirilgan dasturning qismida bitta xususiyatga ega bo'lgan matnli va shartli xabarni bildiruvchi **Message** nomli interfeys berilgan. Bu interfeys orqali, ya'ni meros olgan holda ikkita klass yaratilgan. Yaratilgan klasslar nomi **EmailMessage** va **SmsMessage**. Bundan tashqari *send* nomli funksiya yaratilgan bo'lib, bu funksiya `<T: Message>` ko'rinishdagi cheklov o'rnatilgan. Ya'ni, **Message** interfeysi orqali yaratilgan har qanday turdagi obyekt qabul qilinishi ko'rsatilgan. Bu klasslar va interfeysni ishlatish uchun dasturning asosiy funksiyasi quyidagi ko'rinishda bo'lishi kerak:

```

fun main() {
    val email = EmailMessage("Salom, bu email xabari.")
    send(email)
    val sms = SmsMessage("Salom, bu SMS xabar.")
    send(sms)
}

```

Bir nechta cheklovlarni o'rnatish

Yuqoridagi dasturda solishtirish interfeysi amalga oshiradigan har qanday obyektlarni *getBiggest* funksiyasi yordamida solishtirish mumkin. Agar funktsiyani faqat sonlarni solishtirish uchun ishlatish kerak bo'lsachi? Barcha sonlar **Number** bazaviy klassidan meros olgan holda yaratilgan. Shunday ekan bu funksiya yana bitta cheklov joylashtirish mumkin. Shunda yaratilgan funksiya faqat sonlarni taqqoslash uchun ishlatiladi xolos.

```

fun <T> getBiggest(a:T,b:T):T where T:Comparable<T>,T:Number{
    return if(a > b) a
    else b
}

```

Agar funksiya bir vaqtning o'zida bir nechta cheklovlar joylashtirish kerak bo'lsa, funktsiyaning qaytarish turidan keyin **where** kalit yozilib, bu kalit so'zdan so'ng cheklovlar vergul bilan ajratilgan holda quyidagi sintaksisga asosan yoziladi.

```

tur_parametri: cheklovlar

```

Yuqorida keltirilgan funktsiyadan foydalanish quyidagi dasturning asosiy funksiyasida ko'rsatib o'tilgan:

```

fun main() {
    val result1 = getBiggest(1, 2)
    println(result1)
}

```

```

val result2 = getBiggest(1.6, -2.8)
println(result2)

// Xatolik! String turi Number klassining vorisi emas
// val result3 = getBiggest("Tohir", "Salim")
// println(result3)
}

```

Dasturchi tomonidan yaratilgan klasslarda ham bir vaqtning o'zida bir nechta cheklovlar o'rnatish mumkin. Bunda yuqorida keltirilgan dasturdagi kabi cheklovlar **where** kalit so'zidan keyin dasturchi tomonidan aniqlangan cheklov interfeyslari yoki klasslarida yoziladi. Bunga misol sifatida quyidagi dasturni keltirish mumkin:

```

fun main() {
    val email = EmailMessage("Salom, bu xabar.")
    send(email)
    val sms = SmsMessage("Salom, bu xabar.")
    send(sms)
}
fun <T> send(message: T) where T:Message, T:Logger{
    message.log()
}
interface Message{ val text: String }
interface Logger{ fun log() }
class EmailMessage(override val text: String):Message,Logger{
    override fun log() = println("Email: $text")
}
class SmsMessage(override val text: String):Message,Logger{
    override fun log() = println("SMS: $text")
}

```

Bu yerda *send* funksiyasi ikkita cheklovga ega. Bu cheklovlar dasturchi tomonidan yaratilgan **Message** va **Logger** interfeyslardir.

Klasslarda cheklovlar

Klasslar ham funksiyalar kabi cheklovlarni qabul qilishi mumkin. Masalan, klasslarda bitta cheklov o'rnatish quyidagi klassda keltirilgan:

```

class Messenger<T:Message>() {
    fun send(mes: T){
        println(mes.text)
    }
}

```

Klasslarda ham bir vaqtning o'zida bir nechta cheklovlar o'rnatish mumkin. Bunda funksiyalarga o'rnatilgan cheklovlar kabi **where** kalit so'zidan foydalanib, cheklovlar xuddi funksiyalardagi kabi vergullar yordamida ajratilgan holda yoziladi.

```

fun main() {
    val email = EmailMessage("Salom talabalar")
    val outlook = Messenger<EmailMessage>()
    outlook.send(email)
    val skype = Messenger<SmsMessage>()
    val sms = SmsMessage("Salom talabalar")
    skype.send(sms)
}
class Messenger<T>() where T: Message, T: Logger{
    fun send(mes: T){
        mes.log()
    }
}
interface Message{ val text: String }
interface Logger{ fun log() }
class EmailMessage(override val text: String):Message,Logger{
    override fun log() = println("Email: $text")
}
class SmsMessage(override val text: String): Message, Logger{
    override fun log() = println("SMS: $text")
}

```

Nazorat savollari:

1. Umumiy turlar deganda nimani tushunasiz?
2. Umumiy funksiyalar deganda nimani tushunasiz?
3. Umumiy turlar bilan oddiy turlarning farqi nimada?
4. Umumiy funksiyalar bilan oddiy funksiyaning farqini tushuntirib bering.
5. Umumiy turlarda cheklovlar qanday vazifa bajaradi?
6. Umumiy funksiyalarga cheklovlar joylashtirish mumkinmi?

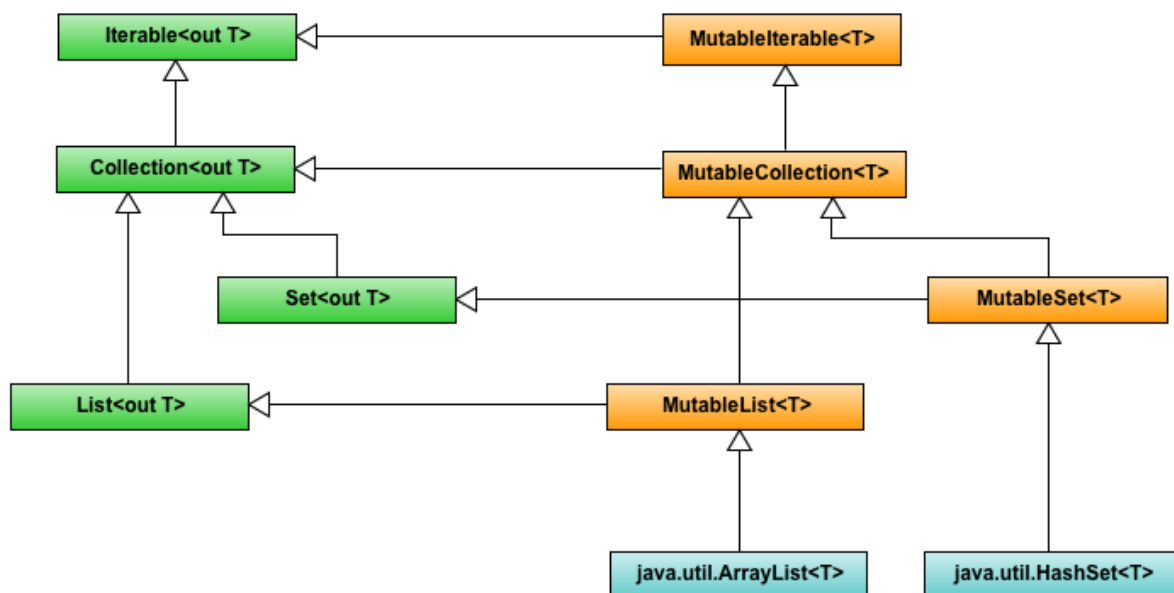
8. To‘plamlar

8.1. O‘zgaruvchan va o‘zgarmas to‘plamlar

To‘plamlar ma’lumotlarni saqlash uchun ishlatiladigan konteynerlarni ifodalaydi. To‘plamlardagi ma’lumotlar turlari bir–biridan ma’lumotlarni yig‘ish va ular ustida amallar bajarish bilan farq qiladi. **Kotlin** dasturlash tilining o‘ziga xos to‘plamlar kutubxonasi mavjud emas. Shuning uchun bu dasturlash tili **Java** dasturlash tilida taqdim etilgan to‘plamlardan to‘liq foydalanadi. Shu bilan birga **Kotlin** dasturlash tilidagi to‘plamlar qo‘shimcha funksiyalar bilan kengaytirilgan. **Kotlin** dasturlash tilida o‘zgaruvchan va o‘zgarmas to‘plamlar mavjud.

O‘zgaruvchan to‘plamlar ustida turli amallar bajarish mumkin, ya’ni, element qo‘shish, elementni o‘zgartirish, elementni o‘chirish mumkin. O‘zgarmas to‘plamlar ustida bunday amallarni bajarish mumkin emas.

Kotlin dasturlash tilidagi barcha to‘plamlar *kotlin.collections* paketida joylashgan. Bu paketda to‘plamlarga oid interfeyslar va klasslar aniqlangan.



8.1.1–rasm: Kotlin dasturlash tilidagi to‘plamlar

O‘zgarmas to‘plamlar

8.1.1–rasmda keltirilgan to‘plamlar iyerarxiyasining yuqori qismida takrorlash uchun iterator funksiyasini o‘z ichiga olgan **Iterable** interfeysi joylashtirilgan. To‘plamlar bilan ishlashga imkon beradigan asosiy interfeys **kotlin.collections** paketida joylashgan **Collection** nomli interfeys hisoblanadi. **Iterable** interfeysi o‘zida elementlarni mavjudligini tekshirish, elementlar ustida takrorlash, ma’lumotlarni o‘qish

funksiyalarini saqlaydi. Biroq elementlarni qo'shish, tahrirlash va o'chirish amallarini bajara olmaydi. Bu interfeysning asosiy metodlari quyidagilar hisoblanadi:

- **size**: ushbu xususiyat to'plamdagi elementlar sonini qaytaradi;
- **isEmpty()**: ushbu metod agar to'plam bo'sh bo'lsa, u holda rost qiymat aks holda, yolg'on qiymat qaytaradi;
- **contains(element)**: ushbu metodning parametrda ko'rsatilgan *element* qiymati mavjud bo'lsa, rost qiymat aks holda, yolg'on qiymat qaytaradi;
- **containsAll(collection)**: ushbu metodning parametrda ko'rsatilgan *collection* to'plami elementlari mavjud bo'lsa, rost qiymat, aks holda, yolg'on qiymat qaytaradi.

Kotlin dasturlash tilida o'zgarmas to'plamlarni ifodalovchi boshqa interfeyslar ham mavjud. Bu oddiy ro'yxatni ifodalovchi **List** va takrorlanmas elementlarga ega bo'lgan **Set** interfeyslaridir. Yana bir interfeys bu **Map** interfeysi bo'lib, bu interfeys kalit–qiymat juftligidan iborat bo'lgan elementlar to'plamidir.

O'zgaruvchan to'plamlar

Kotlin dasturlash tilida barcha o'zgaruvchan to'plamlar **MutableIterable** interfeysi tomonidan yaratiladi. Bu to'plamni elementlari ustida takrorlanish amalini bajarish uchun iterator funksiyalari mavjud. O'zgaruvchan to'plamlar interfeyslarining xususiyatlari va metodlari quyidagilar hisoblanadi:

- **add(element)**: ushbu metod ko'rsatilgan qiymatni to'plamga qo'shadi;
- **remove(element)**: ushbu metod ko'rsatilgan qiymat to'plamda mavjud bo'lsa o'chiradi. Agar ko'rsatilgan qiymat to'plamda bir nechta takrorlangan bo'lsa, birinchi kelganini o'chiradi;
- **addAll(elements)**: ushbu metod ko'rsatilgan elementlar to'plamini to'plamga qo'shadi;
- **removeAll(elements)**: ushbu metod ko'rsatilgan elementlar to'plamini to'plamdan o'chiradi;
- **clear()**: ushbu metod to'plamdagi barcha elementlarni o'chiradi, ya'ni to'plamni tozalaydi.

Kotlin dasturlash tilida o'zgaruvchan to'plamlarni ifodalovchi boshqa interfeyslar ham mavjud. Bu oddiy o'zgaruvchan ro'yxatni ifodalovchi **MutableList** va takrorlanmas elementlarga ega bo'lgan o'zgaruvchan to'plam **MutableSet** interfeyslaridir. Yana bir o'zgaruvchan

to‘plam interfeysi bu **MutableMap** interfeysi bo‘lib, bu interfeys kalit–qiymat juftligidan iborat bo‘lgan elementlar to‘plamidir.

8.2. List – to‘plami

List interfeysi elementlarning ketma–ket ro‘yxatini ifodalovchi to‘plamdir. **List** interfeysi o‘zgarmas to‘plamni ifodalaydi. **List** interfeysi yordamida yaratiladigan to‘plamlar elementlari asosan **listOf()** funksiyasi yordamida e‘lon qilinadi:

```
var numbers = listOf(1, 2, 3, 4, 5, null)
var numbers2: List<Int> = listOf(5, 6, 7)
```

List turi har xil metodlar yordamida elementlarini olish imkonini beradi. Bu metodlarning asosiylari quyidagilar:

- **get(index)**: ushbu metod ko‘rsatilgan indeksdagi elementni qaytaradi;
- **elementAt(index)**: ushbu metod ko‘rsatilgan indeksdagi elementni qaytaradi;
- **elementAtOrNull(index)**: ushbu metod ko‘rsatilgan indeksdagi elementni qaytaradi, agar bunday indeks bo‘lmasa, null qiymat qaytaradi;
- **first()**: ushbu metod ro‘yxatning birinchi elementni qaytaradi;
- **last()**: ushbu metod ro‘yxatning oxirgi elementni qaytaradi;
- **indexOf(element)**: ushbu metod ro‘yxatda birinchi kelgan elementning indeksini qaytaradi;
- **lastIndexOf(element)**: ushbu metod ro‘yxatda oxirgi kelgan elementning indeksini qaytaradi;
- **contains(element)**: ushbu metod agar ro‘yxatda ko‘rsatilgan element mavjud bo‘lsa, rost qiymat qaytaradi;

```
fun main() {
    val numbers : List<Int> = listOf(1, 2, 3, 4, 5)
    for (n in numbers)
        print("$n, ")
    println()
    println(numbers.get(1))
    println(numbers.indexOf(2))
    println(numbers.lastIndexOf(3))
    println(numbers.first())
    println(numbers.last())
    println(numbers.size)
    println(numbers.contains(4))
    println(numbers.elementAt(1))
    println(numbers.elementAtOrNull(9))
}
```

O'zgaruvchan ro'yxatlar

Kotlin dasturlash tilida o'zgaruvchan ro'yxatlar **MutableList** interfeysi bilan ifodalanadi. Bu interfeys orqali yaratilgan ro'yxat ustida elementlar qo'shish, elementlarni tahrirlash va elementlarni o'chirish amallarini bajarish mumkin. Bu amallar interfeysda metodlar yordamida aniqlab qo'yilgan. Bu interfeys orqali **ArrayList** klassi aniqlangan bo'lib, bu klass o'zgarmas ro'yxat yoki massiv tashkil etishga yordam beradi. O'zgaruvchan ro'yxat yaratish uchun ishlatiladigan funksiyalar quyidagilar hisoblanadi:

- **arrayListOf()**: ushbu funksiya **ArrayList** turiga mansub obyektlar yaratadi;
- **mutableListOf()**: ushbu funksiya **MutableList** turidagi obyektlar yaratish uchun xizmat qiladi.

Bu ikki funksiya orqali obyekt yaratish quyidagi dasturning qismida berilgan:

```
var numbers : ArrayList<Int> = arrayListOf(1, 2, 3, 4, 5)
var numbers2: MutableList<Int> = mutableListOf(5, 6, 7)
```

O'zgaruvchan ro'yxatlar ustida elementlar qo'shish, elementlarni o'chirish va turli amallarni bajaruvchi metodlar quyida keltirib o'tilgan. Bu usullar **MutableList** interfeysining metoddari hisoblanadi:

- **add(index, element)**: ushbu metod ko'rsatilgan indeksga *element* obyektning qiymatini qo'shadi;
- **add(element)**: ushbu metod ko'rsatilgan *element* obyektning qiymatini qo'shadi;
- **addAll(collection)**: ushbu metod ko'rsatilgan *collection* elementlar to'plamini ro'yxatga qo'shadi;
- **remove(element)**: ushbu metod ko'rsatilgan *element* qiymatini ro'yxatdan o'chiradi;
- **removeAt(index)**: ushbu metod ko'rsatilgan indeksdagi qiymatni o'chiradi;
- **clear()**: ushbu metod ro'yxatdagi barcha elementni o'chiradi yoki ro'yxatni tozalaydi.

```
fun main() {
    val numbers1:ArrayList<Int> = arrayListOf(1, 2, 3, 4, 5)
    numbers1.add(4)
    val numbers2: MutableList<Int> = mutableListOf(5, 6, 7)
    numbers2.add(12)
    numbers2.add(0, 23)
    numbers2.addAll(0, listOf(-3, -2, -1))
    numbers2.removeAt(0)
```

```

numbers2.remove(5)
for (n in numbers2){ println(n) }
}

```

8.3. Set – to‘plami

Kotlin dasturlash tilida elementlari takrorlanmas to‘plam yaratishda **Set** interfeysidan foydalaniladi. **Set** interfeysi o‘zgarmas to‘plamni ifodalaydi. **Set** interfeysi ham **Collection** interfeysidan meros olingan holatda yaratilgan. **Set** interfeysining ba’zi bir metodlari quyida keltirilgan:

- **contains(element)**: ushbu metod *element* o‘zgaruvchisining qiymati to‘plamda mavjud bo‘lsa, rost aks holda, yolg‘on qiymat qaytaradi;
- **isEmpty()**: ushbu metod to‘plam bo‘sh bo‘lsa, rost aks holda, yolg‘on qiymat qaytaradi;
- **minus(element)**: ushbu metod yordamida *element* o‘zgaruvchisining qiymatini to‘plamdan o‘chirib yangi to‘plam hosil qiladi. Bunda asl to‘plam o‘zgarishsiz qoladi;
- **plus(element)**: ushbu metod yordamida *element* o‘zgaruvchisining qiymatini to‘plamga qo‘shib yangi to‘plam hosil qiladi. Bunda asl to‘plam o‘zgarishsiz qoladi;

Kotlin dasturlash tilida o‘zgarmas to‘plam yaratish uchun **setOf()** nomli funksiyadan foydalaniladi. Quyidagi dasturda **setOf()** funksiya va **Set** interfeysidan foydalanish ko‘rsatib o‘tilgan:

```

fun main() {
    val items:Set<Int> = setOf(1, 2, 3, 4, 5)
    println(items.size)
    println(items.contains(4))
    println(items.isEmpty())
    println(items.minus(3))
    println(items.plus(7))
    for (n in items){ print("$n, ") }
}

```

Yuqorida keltirilgan dasturning natijasi quyidagi keltirib o‘tilgan:

```

5
true
false
[1, 2, 4, 5]
[1, 2, 3, 4, 5, 7]
1, 2, 3, 4, 5,

```

O‘zgaruvchan to‘plamlar

Kotlin dasturlash tilida o‘zgaruvchan to‘plamlar **Set** va **MutableCollection** interfeyslaridan vorislik yordamida yaratilgan

MutableSet interfeysi bilan yaratiladi. Bu interfeys bilan yaratilgan o'zgaruvchan to'plamlarni elementlarini **mutableSetOf()** nomli funksiya yordamida tashkil etish mumkin.

```
val numbers: MutableSet<Int> = mutableSetOf(35, 36, 37)
```

Kotlin dasturlash tilida o'zgaruvchan to'plamlarni quyidagi ko'rinishlari ham mavjud:

- **LinkedHashSet**: bu interfeys xesh jadvaliga bog'langan to'plamni bildiradi. Bu interfeysda yaratiladigan ro'yxatlar **linkedSetOf()** funksiyasi tomonidan tashkil etiladi;
- **HashSet**: bu interfeys xesh jadvalini ifodalaydigan to'plam bo'lib, bu to'plamning elementlarini **hashSetOf()** funksiyasi yordamida yaratish mumkin.

```
val numbers1: HashSet<Int> = hashSetOf(5, 6, 7)
```

```
val numbers2: LinkedHashSet<Int> = linkedSetOf(25, 26, 27)
```

```
val numbers3: MutableSet<Int> = mutableSetOf(35, 36, 37)
```

O'zgaruvchan to'plamlar yaratish uchun quyidagi dastur misol bo'la oladi:

```
fun main() {  
    val numbers: MutableSet<Int> = mutableSetOf(35, 36, 37)  
    println(numbers.add(2))  
    println(numbers.addAll(setOf(4, 5, 6)))  
    println(numbers.remove(36))  
    println(numbers)  
    numbers.clear()  
    println(numbers)  
}
```

Dasturning natijasi konsol oynasiga quyidagicha chiqadi:

```
true  
true  
true  
[35, 37, 2, 4, 5, 6]  
[]
```

8.4. Map – to'plami

Barcha dasturlash tillaridagi kabi bu dasturlash tilida ham **Map** nomli to'plam mavjud bo'lib, bu to'plamning har bir elementida kalit va shu kalitga bog'langan qiymat mavjud bo'ladi. Bu to'plamning barcha kalitlari takrorlanmas ko'rinishda yozilishi kerak. **List** va **Set** to'plamlaridan farqi **Map** interfeysi **Collection** interfeysining vorisi hisoblanmaydi. **Map** to'plamining obyektini yaratish uchun **mapOf()** nomli funksiyadan foydalaniladi. Bu to'plam o'zgarmas to'plam hisoblanadi.

```

val countries: Map<String, Int> = mapOf("AQSH" to 300,
"Fransiya" to 60, "Germaniya" to 81)
println(countries["AQSH"]) // 300
for(country in countries){
    println("${country.key} - ${country.value}")
}
println(countries)

```

Yuqoridagi dasturda to‘planning elementlari **mapOf** funksiyasi tomonidan e‘lon qilingan. Har bir element kalit va qiymatdan iborat bo‘lib, kalitga qiymatlar **to** kalit so‘zi bilan bog‘lanadi. Bu to‘plamda e‘lon qilingan obyektning o‘ziga tegishli bo‘lgan ba‘zi bir xususiyatlari mavjud. Bu xususiyatlardan asosiylari *key* va *value* bo‘lib, bu ikki xususiyat mos ravishda kalit va qiymatni anglatadi.

O‘zgaruvchan **Map** to‘plamini yaratish uchun **MutableMap** interfeysidan foydalaniladi. O‘zgaruvchan **Map** to‘plamini elementlarini yaratish uchun **mutableMapOf** funksiyasi yordamida ko‘rsatib o‘tiladi. O‘zgaruvchan **Map** to‘plamiga element qo‘shish uchun obyektga tegishli bo‘lgan **put** metodidan foydalanadi. Bu metod o‘zgaruvchan **Map** to‘plamiga ko‘rsatilgan kalit va shu kalit uchun qiymatni to‘planning so‘ngi elementi sifatida qo‘shadi.

```

val countries: MutableMap<String, Int> = mutableMapOf("AQSH"
to 300, "Fransiya" to 60, "Germaniya" to 81)
println(countries["AQSH"])
countries.put("Ispaniya", 33)
countries.remove("Fransiya")
for(country in countries){
    println("${country.key} - ${country.value}")
}
println(countries)

```

Map to‘plami bilan ishlash jarayonida to‘plamdagi keraksiz bo‘lgan elementni o‘chirish mumkin. O‘chirish to‘planning obyektiga tegishli bo‘lgan *remove* metodi yordamida amalga oshiriladi. Buning uchun elementning kalit so‘zidan foydalaniladi. Yuqorida keltirilgan dasturda “Fransiya” kalitiga ega bo‘lgan to‘planning elementi o‘chirilmoqda.

O‘zgaruvchan **Map** to‘plamining yuqorida keltirilgan ikki metodidan tashqari quyida keltirilgan xususiyatlari va metodlari ham mavjud.

- **entries:** ushbu xususiyat **Map** to‘plamining barcha kalit va qiymat juftliklarini qaytaradi;
- **values:** ushbu xususiyat **Map** to‘plamidagi barcha elementlarning qiymatlarini qaytaradi;

- **keys:** ushbu xususiyat **Map** to‘plamidagi barcha elementlarning kalitlarini qaytaradi;
- **size:** ushbu xususiyat **Map** to‘plamidagi elementlar sonini qaytaradi;
- **count():** ushbu metod **Map** to‘plamidagi elementlar sonini qaytaradi;
- **replace(key, value):** ushbu metod **Map** to‘plamidagi ko‘rsatilgan element mavjud bo‘lsa, uning qiymatini o‘zgartiradi;
- **set(key, value):** ushbu metod **Map** to‘plamidagi ko‘rsatilgan element mavjud bo‘lsa, uning qiymatini o‘zgartiradi;
- **get(key):** ushbu metod **Map** to‘plamidagi ko‘rsatilgan element mavjud bo‘lsa, uning qiymatini qaytaradi;
- **getValue(key):** ushbu metod **Map** to‘plamidagi ko‘rsatilgan element mavjud bo‘lsa, uning qiymatini qaytaradi;
- **isEmpty():** ushbu metod **Map** to‘plami bo‘sh bo‘lsa, rost qiymat qaytaradi;

Nazorat savollari:

1. To‘plamlar deganda nimatushuniladi?
2. Kotlin dasturlash tilidagi to‘plamlar turi sanab bering.
3. O‘zgaruvchan va o‘zgarmas to‘plamning farqini tushuntirib bering.
4. List to‘plami bilan Set to‘plamning farqini nimada?
5. List to‘plami bilan Map to‘plamning farqini nimada?
6. Set to‘plami bilan Map to‘plamning farqini nimada?
7. To‘plam yaratuvchi funksiyalarni sanab bering.

9. Korutinlar

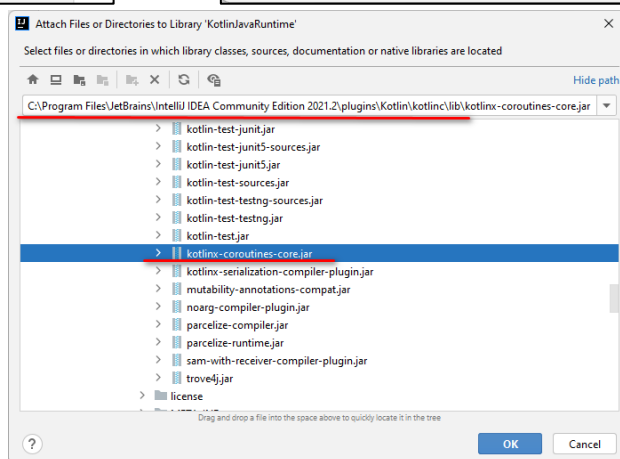
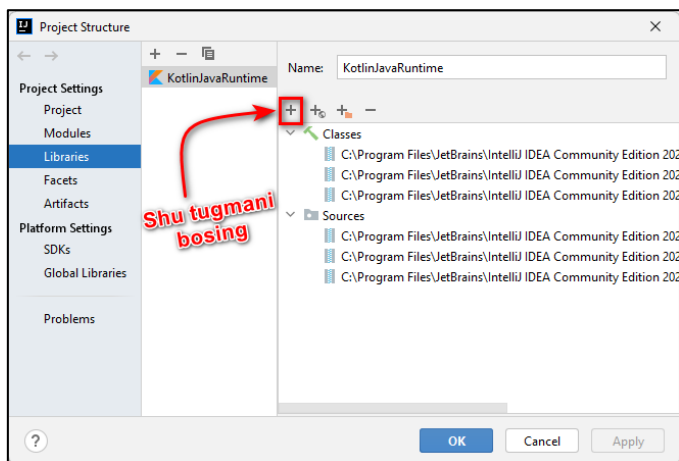
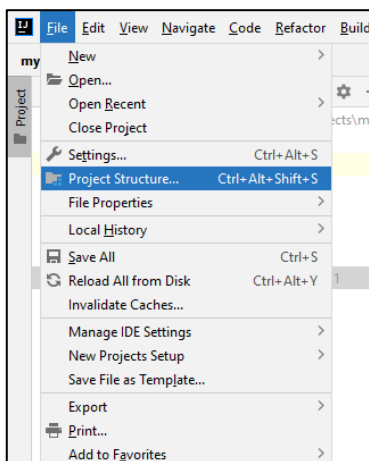
9.1. Korutinlar bilan tanishish

Hozirgi zamonda asinxron va parallel hisoblash ko‘plab dasturlash tillarning ajralmas xususiyatiga aylanib bormoqda. **Kotlin** dasturlash tili ham bundan mustasno emas. Nima uchun asinxron va parallel hisoblash kerak? Parallel hisoblash bir vaqtning o‘zida bir nechta vazifalarni bajarishga imkon beradi. Asinxron hisoblash esa uzoq vaqt talab qilinadigan vazifa bajarilayotganda asosiy dastur oqimini bloklamaslik uchun xizmat qiladi. Masalan, oddiy dastur yoki mobil ilova uchun grafik ma’lumotlarni chizishda, bundan tashqari qandaydir tugmani bosib internet resursiga so‘rov yuborish va undan kelgan ma’lumotlarni tahlil qilishda. Bu ikki ko‘rsatilgan misollar uzoq vaqt talab qilishi mumkin. So‘rov yuborilganida ilova osilib qolmasligi uchun internet resurslariga so‘rovlar asinxron tarzda yuboriladi. Asinxron so‘rovlar yordamida ilova internet resursidan javob kelguniga qadar kutib turmaydi, dastur o‘z ishini davom etadi va javob kelganida unga tegishli bo‘lgan buyruqlar ketma-ketligi bajariladi.

Kotlin dasturlash tilida asinxron va parallel hisoblashlar korutin shaklida amalga oshiriladi. **Korutin** – bu kodning qolgan qismi bilan parallel ravishda ishlashi mumkin bo‘lgan kodlar blokidir. Korutinlar bilan bog‘liq bo‘lgan asosiy funksiyalar **kotlinx.coroutines** paketida joylashgan.

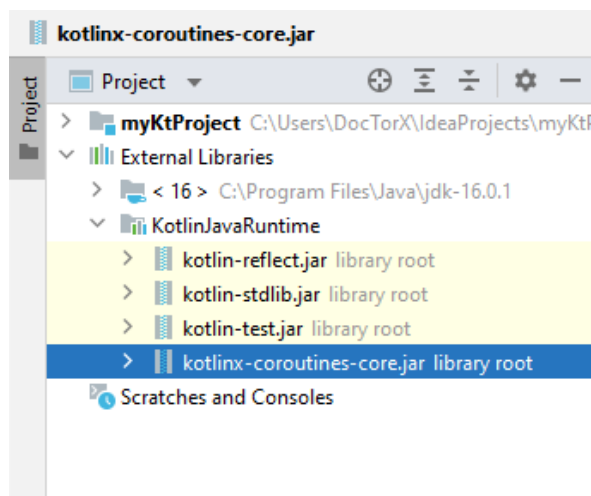
Korutin paketini dasturga qo‘shish

Korutin bilan ishlovchi paket yaratilayotgan dasturga to‘g‘ridan-to‘g‘ri bog‘lanmagan bo‘lib, bu paketni dasturchining o‘zi yaratayotgan dasturiga qo‘shib oladi. Agar dasturchi IntelliJ IDEA muhitidan foydalanayotgan bo‘lsa, unda yaratilgan loyiha uchun bu paketni qo‘shish quyidagicha amalga oshiriladi. File menyusidagi Project Structure bandi tanlanadi. Bu band tanlanganida Project Structure nomli oyna hosil bo‘ladi. Hosil bo‘lgan oynadagi Project Settings bo‘limidagi Libraries bandi tanlanadi. Libraries bandi tanlanganda oynaning o‘ng qismidagi + tugmasi bosilib, kerakli fayl qo‘shib olinadi. Qo‘shib olinadigan faylning nomi **kotlinx-coroutines-core.jar** bo‘lib, bu fayl **Kotlin** dasturlash tilining kompilyatori joylashgan papkaning ichidagi lib nomli papkada joylangan bo‘ladi (9.1.1–rasm).



9.1.1–rasm: kotlinx-coroutines-core.jar faylini ilovaga qo‘shish

Rasmda keltirilgan ketma–ketlik bajarilganida tashqi kutubxonalarni ro‘yxatga oluvchi External Libraries/ KotlinJavaRuntime dagi mavjud kutubxonalar bilan ro‘yxatga olinadi (9.1.2–rasm).



9.1.2–rasm: External Libraries/KotlinJavaRuntime bo‘limi

Boshqa turdagi muhitlarda kutubxonalarni bog‘lash bu ko‘rinishda bo‘lmashligi mumkin.

suspend ko‘rinishdagi funksiyani aniqlash

Korutinlar bilan ishlashda buyruqlar ketma–ketligini kutib turish vazifasini berish uchun **suspend** ko‘rinishdagi funksiya yaratish kerak bo‘ladi. Quyida keltirilgan dasturda funksiya **suspend** ko‘rinishida e‘lon qilingan bo‘lsa ham korutindan foydalanilmagan hol ko‘rsatilgan.

```
import kotlinx.coroutines.*
suspend fun main(){
    for(i in 0..5){
        delay(400L)
        print("$i, ")
    }
    println("\nHello Coroutines")
}
```

Bu yerda asosiy funksiya 0 dan 5 gacha bo‘lgan sonlar ketma–ketligini konsol oynasiga chiqarish keltirib o‘tilgan. Sonlarni konsol oynasiga chiqarish vaqtidagi simulyatsiya uchun **kotlinx.coroutines** paketidagi maxsus **delay()** funksiyasidan foydalanilgan. Bu funksiya kechiktirish ma‘nosini anglatib, kechiktirish millisekundlarda ifodalaniladi. Funksiyaga ko‘rsatilayotgan qiymat **Long** turiga mansub bo‘lishi kerak. Yuqoridagi dasturda funksiya ketma–ketlik elementlarini chiqarish uchun har bir murojaatda 400 millisekund kechiktirishni amalga oshiradi. Takrorlanish operatori ishini yakunlaganidan so‘ng ekranga “Hello Coroutines” qatorini chiqariladi.

delay() funksiyasini asosiy funksiyada ishlatish uchun asosiy funksiya **suspend** kalit so‘zi bilan e‘lon qilingan bo‘lishi kerak. Buyruqlar ketma–ketligi bajarilishini to‘xtatib turadigan va ma‘lum bir vaqtdan keyin davom etadigan funksiyalar albatta **suspend** kalit so‘zi yordamida aniqlangan bo‘lishi kerak. **delay()** funksiyasini ham **suspend** kalit so‘zi yordamida aniqlangan bo‘lib, **suspend** kalit so‘zi bilan aniqlangan har qanday funksiya shu kalit so‘z bilan aniqlangan funksiya bilan chaqirilishi lozim. Agar dasturni ishga tushirilsa, konsolda quyidagi ko‘rinishda natija hosil bo‘ladi.

```
0, 1, 2, 3, 4, 5,
Hello Coroutines
```

Yuqoridagi dasturda “Hello Coroutines” qatori takrorlanish tugashini kutadi. Ammo bunday ko‘rinish internet resurs bilan ishlash vaqtida xatolik keltirib chiqarishi mumkin yoki ma‘lumotlar uzatilishidagi kamchilik ko‘rinib qoladi.

Ma'lumotlar uzatilishidagi kamchiliklarni bartaraf etish uchun **suspend** ko'rinishdagi funksiyaga odatiy buyruqlar ketma-ketligi yozmasdan quyidagi ko'rinishda buyruqlar yozilishi kerak bo'ladi.

```
import kotlinx.coroutines.*
suspend fun main() = coroutineScope{
    launch{
        for(i in 0..5){
            delay(400L)
            print("$i, ")
        }
    }
    println("Hello Coroutines")
}
```

Birinchi navbatda korutinni aniqlash va bajarish uchun korutinning tana qismini aniqlab olish kerak, chunki korutinni faqat korutin doirasi ishlatish yoki chaqirish mumkin. Korutin doirasi *coroutineScope()* funksiyasining tana qismi hisoblanadi va bu funksiyaning tanasida korutin joylashadi. Bundan tashqari bu funksiyaning tanasida aniqlangan barcha korutintlarni bajarilishini kutadi. Shuni ta'kidlab o'tish kerakki, *coroutineScope()* funksiyasi faqat asosiy funksiya (main) ga ishlatiladi.

Korutin quruvchisi bo'lgan **launch** funksiyasi yordamida quriladi va ishga tushiriladi. Bu funksiya kodlar blokidan foydalanilgan holda korutin yaratadi. Yuqorida ko'rsatilgan dasturda korutin yaratuvchi **launch** funksiyasi tomonidan quyidagi korutin yaratilgan:

```
{
    for(i in 0..5){
        delay(400L)
        print("$i, ")
    }
}
```

Bu korutin dasturda berilgan boshqa kodlar bilan parallel ravishda bajariladi. Ya'ni bu korutin dasturning asosiy funksiyasida aniqlangan boshqa kodlardan mustaqil ravishda ishlash xususiyatiga ega. Natijada dasturni bajarish uchun buyruq berilganda konsolga quyidagi ma'lumotlar chiqariladi:

```
Hello Coroutines
0, 1, 2, 3, 4, 5,
```

Bu dasturda "Hello Coroutines" qatorini chiqarish uchun dastur takrorlanuvchi operator ishini tugatishiga qarab turmaydi. Balki u bilan parallel ravishda bajariladi.

Yuqorida ko'rsatib o'tilgan dasturlarda korutinlar asosiy funksiyaning tana qismida aniqlangan. Korutinni alohida funksiyada ham aniqlash mumkin.

```
import kotlinx.coroutines.*
suspend fun main()= coroutineScope{
    launch{ doWork() }
    println("Hello Coroutines")
}
suspend fun doWork(){
    for(i in 0..5){
        print("$i, ")
        delay(400L)
    }
}
```

Bu dasturda korutin buyruqlar ketma-ketligi *doWork()* funksiyasida aniqlangan. Bu funksiyada *delay()* funksiyasidan foydalanganligi sababli funksiya **suspend** kalit so'zi yordamida aniqlangan. Dasturning asosiy funksiyasi **main** ham **suspend** kalit so'zi bilan aniqlangan. Bu asosiy funksiyaning tana qismida **launch** funksiyasi yozilgan.

9.2. Korutin maydoni

Korutin faqat ma'lum bir korutin maydoni (coroutine scope)da bajarilishi mumkin. Korutin maydoni korutinlar ishlaydigan bo'shliqni ifodalaydi. Korutin maydoni ma'lum bir hayotiy sikli ega. Bu maydonning ichida aniqlangan korutinlar hayot siklini o'zi boshqarib boradi. **Kotlin** dasturlash tilida korutin maydonini yaratish uchun `CoroutineScope` interfeysi obyektini yaratuvchi bir nechta funksiyalardan foydalanish mumkin. Bu funksiyalardan biri – `coroutineScope()` nomli funksiya hisoblanadi. Bu funksiyani har qanday funksiya uchun qo'llash mumkin, masalan:

```
import kotlinx.coroutines.*
suspend fun main(){
    doWork()
    println("Hello Coroutines")
}
suspend fun doWork()= coroutineScope{
    launch{
        for(i in 0..5){
            print("$i, ")
            delay(400L)
        }
    }
}
```


Bir nechta korutinlarni ishga tushirish

Bir funksiyaning o'zida bir vaqtda bir nechta korutinlardan foydalanish va ularni ishga tushirish mumkin. Bunda barcha korutinlar bir vaqtning o'zida ishga tushib natija beradi. Masalan,

```
import kotlinx.coroutines.*
suspend fun main() = coroutineScope{
    launch{
        for(i in 0..5){
            delay(400L)
            println("$i - birinchi")
        }
    }
    launch{
        for(i in 6..10){
            delay(400L)
            println("$i - ikkinchi")
        }
    }
    println("Hello Coroutines")
}
```

coroutineScope() funksiyasi korutin maydonini yaratadi va bu maydonda aniqlangan barcha korutinlar bajarilishini nazarda tutadi. Yuqoridagi dasturda ikki korutin berilgan. Bu korutinlar faoliyatini tugatgandan so'ng asosiy dastur o'z ishini yakunlaydi. Bu dastur konsolga quyidagicha natija beradi.

```
Hello Coroutines
6 - ikkinchi
0 - birinchi
7 - ikkinchi
1 - birinchi
8 - ikkinchi
2 - birinchi
9 - ikkinchi
3 - birinchi
10 - ikkinchi
4 - birinchi
5 - birinchi
```

Ichki korutinlar

Bir korutin boshqa bir yoki bir nechta korutinlarni o'z ichiga olishi mumkin:

```
import kotlinx.coroutines.*
suspend fun main() = coroutineScope{
    launch{
        println("Tashqi korutin")
        launch{
```

```

        println("Ichki korutin")
        delay(400L)
    }
}
println("Asosiy dastur oxiri")
}

```

Ichki korutinlar tashqi korutinlar doirasida aniqlanadi. Bunda tashqi korutin bilan ichki korutin o‘z vazifasini bir vaqtda boshlaydi.

9.3. launch funksiyasi

Korutin yaratish uchun yuqorida keltirilganidek korutin quruvchi kerak bo‘ladi. Korutin quruvchi funksiya **kotlinx.coroutines** paketida joylashgan launch funksiyasi hisoblanadi. Bu funksiyaning ishlatish oldingi rejalarda ko‘rsatib o‘tilgandi. Bu rejada **launch** funksiyalarining ba’zi bir jihatlari ko‘rib chiqiladi.

Odatda korutindan natijaviy qiymat qaytmaganda va boshqa bir buyruqlar ketma–ketiligi bilan bir vaqtda bajarilishi kerak bo‘lganda **launch** funksiyasi ishlatiladi.

Korutinlarni **Job** interfeysidan olingan obyektning qiymati sifatida ishlatish mumkin. Bunga misol quyida berilgan dastur qismida ko‘rsatilgan:

```

val jasur: Job = launch{
    println("Salom")
    delay(400L)
}

```

Job interfeysidan olingan o‘zgaruvchilarning turli ko‘rinishdagi xususiyatlari va metodlari mavjud bo‘lib, bu xususiyatlar va metodlar ichidagi interfeysning hayotiy siklini ifodalovchi 3 ta xususiyatini alohida ko‘rsatish mumkin.

- isActive – korutinning bu xususiyati ishga tushirilganligini ifodalaydi. Agar korutin ishga tushirilgan bo‘lsa, rost qiymat qaytaradi;
- isCompleted – bu xususiyat korutinning ish faoliyatini tugaganligini bildiradi, agar korutinning ish faoliyati yakunlangan bo‘lsa, rost qiymat qaytaradi;
- isCancelled – bu xususiyat korutinning ish faoliyatini bekor qilinganligini yoki xatolik sodir bo‘lganligini bildiradi, agar korutin faoliyati bekor qilinsa yoki xatolik sodir bo‘lsa, rost qiymat qaytaradi.

Korutinni ish faoliyati bilan ishlash uchun metodlardan foydalaniladi. Bu metodlar quyidagilar hisoblanadi:

- `join()` – ushbu metod korutin tugaguncha kutish imkonini beruvchi metod hisoblanadi;
- `start()` – ushbu metod yordamida korutin ish jarayonini boshlaydi;
- `cancel()` – ushbu metod bajarilayotgan korutinni ishini tugatadi;
- `cancelAndJoin()` – ushbu metod bajarilayotgan korutin ishini to‘xtatib, korutin tugaguncha kutish ko‘rinishiga o‘tkazadi;
- `cancelChildren()` – ushbu metod korutinda joylashgan ichki korutintlarni to‘xtatish uchun xizmat qiladi.

O‘zgaruvchiga o‘zlashtirilgan korutinni ishga tushirish uchun yuqorida keltirilgan *join* metodidan foydalaniladi. Masalan,

```
import kotlinx.coroutines.*
suspend fun main() = coroutineScope{
    val job = launch{
        for(i in 1..5){
            print("$i, ")
            delay(400L)
        }
        println()
    }
    println("Salom")
    job.join() // korutinning tugashini kutish
    println("Hayr")
}
```

Keltirilgan dastur bajarilganida konsol oynasida quyidagi natija chiqadi:

```
Salom
1, 2, 3, 4, 5,
Hayr
```

Korutin ish faoliyatini kechiktirish

Korutin yaratilganida bajarilishi yoki berilgan buyruq yordamida ishga tushirilishi mumkin. Korutin yaratuvchi **launch** funksiyasi korutinni yaratadi va darhol ishga tushiradi. **Kotlin** dasturlash tilida korutintlarni ish faoliyatini kechiktirish mumkin. Korutintlarni ish faoliyati kechiktirish uchun korutinni yaratilayotgan vaqtda **launch** funksiyasiga **start = CoroutineStart.LAZY** qiymati o‘rnatiladi. Kechiktirilgan korutin bilan standart korutinning farqini bilish uchun quyida standart korutin berilgan:

```
import kotlinx.coroutines.*
suspend fun main() = coroutineScope{
    // korutin yaratish va ishga tushirish
    launch() {
```

```

        delay(200L)
        println("Korutin boshlash")
    }
    delay(1000L)
    println("main methodidagi boshqa harakatlar")
}

```

Yuqorida berilgan dastur ishga tushirilganda quyidagi natija chiqadi:

```

Korutin boshlash
main methodidagi boshqa harakatlar

```

Bu standart korutinning bajarilishi bo‘lib, **launch** funksiyasiga yuqorida keltirilgan buyruq kiritilib, ishga tushirilganda boshqa natija olish mumkin. Quyidagi dasturda shu o‘zgartirish kiritilgan holati ko‘rsatilgan:

```

import kotlinx.coroutines.*
suspend fun main() = coroutineScope{
    // kechiktirilgan korutin yaratish
    val job = launch(start = CoroutineStart.LAZY) {
        delay(200L)
        println("Korutin boshlash")
    }
    delay(1000L)
    job.start() // korutinni ishga tushirish
    println("main methodidagi boshqa harakatlar")
}

```

Bu dasturni ishga tushirilganda boshqa natija olish mumkin.

```

main methodidagi boshqa harakatlar
Korutin boshlash

```

9.4. **async va await operatori. Deferred interfeysi**

Korutinlarni ishga tushiruvchi **launch** funksiyasi bilan bir qatorda **kotlinx.coroutines** paketida yana bir korutin quruvchi funksiya **async** funksiyasi mavjud. Bu funksiya korutindan qandaydir natija olish uchun ishlatiladi.

async funksiyasi tomonidan boshqa korutinlar bilan parallel ravishda ishlaydigan alohida korutin yaratadi. Bundan tashqari **async** funksiyasi qiymat qaytaruvchi funksiya hisoblanib, yaratilgan korutin **Deferred** ko‘rinishidagi obyektни qaytaradi. **Deferred** interfeysi **Job** interfeysidan voris olib yaratilgan bo‘lib, **Job** interfeysining barcha xususiyatlari va metodlari bu interfeysda ham mavjud.

Deferred interfeysiga mansub bo‘lgan obyektдан natija olish uchun **await** metodi ishlatiladi. Quyidagi dastur bunga misol bo‘la oladi:

```

import kotlinx.coroutines.*
suspend fun main() = coroutineScope{
    val message: Deferred<String> = async{ getMessage() }
}

```

```

        println("Habar: ${message.await()}")
        println("Dastur tugadi")
    }
    suspend fun getMessage(): String{
        delay(500L)
        return "Hello"
    }

```

Yuqoridagi dasturda **async** funksiyasi va **await** metodi ishlatilishi ko'rsatib o'tilgan. Bu dasturning natijasi quyida ko'rsatib o'tilgan:

```

Habar: Hello
Dastur tugadi

```

Parallel ravishda bajariladigan bir nechta korutirlarni **async** funksiyasi yordamida yaratish ham mumkin:

```

import kotlinx.coroutines.*
suspend fun main() = coroutineScope{
    val numDeferred1 = async{ sum(1, 2) }
    val numDeferred2 = async{ sum(3, 4) }
    val numDeferred3 = async{ sum(5, 6) }
    val num1 = numDeferred1.await()
    val num2 = numDeferred2.await()
    val num3 = numDeferred3.await()
    println("number1: $num1 number2: $num2 number3: $num3")
}
suspend fun sum(a: Int, b: Int) : Int{
    delay(500L)
    return a + b
}

```

Kechiktiriladigan korutirlarni ham **async** funksiyasi yordamida yaratish mumkin. Kechiktiriladigan korutin **launch** funksiyasida qanday yaratilsa, **async** funksiyasida ham huddu shunday yaratiladi. Ya'ni, **async** funksiyasiga **start = CoroutineStart.LAZY** parametr sifatida beriladi.

```

import kotlinx.coroutines.*
suspend fun main() = coroutineScope{
    val sum = async(start = CoroutineStart.LAZY){ sum(1, 2) }
    delay(1000L)
    println("Korutin yaratilgandan keyingi harakatlar")
    sum.start()
    println("Natija: ${sum.await()}")
}
fun sum(a: Int, b: Int) : Int{
    println("Korutinni boshlash")
    return a + b
}

```

Bu dasturni ishlatilganidan so'ng konsol oynasida quyidagicha natija hosil bo'ladi.

Korutin yaratilgandan keyingi harakatlar
Korutinni boshlash
Natija: 3

9.5. Korutin dispatcheri

Korutinning tana qismi korutin dispatcheri kabi elementni o'z ichiga oladi. Korutin dispatcheri korutinni bajarish uchun qaysi oqim va oqimlar ishlatilishini aniqlaydi. Barcha korutin yaratuvchilar xususan, **launch** va **async** funksiyalari ixtiyoriy parametr sifatida **CoroutineContext** turidagi obyektini yaratadi.

Korutin yaratuvchi **launch** funksiyasi parametrlarsiz ishlatilganida korutin yaratiladi va yaratilgan korutin ishga tushiriladi. Yaratilgan korutin qaysi oqimda ishlayotganligini bilish uchun **currentThread** metodidan foydalaniladi. Bu metod **Thread** klassining statik metodi hisoblanadi. Oqimning nomini aniqlashda oqim obyektining **name** xususiyatidan foydalaniladi. Quyidagi dasturda oqim nomini olish ko'rsatib o'tilgan:

```
import kotlinx.coroutines.*
suspend fun main() = coroutineScope{
    launch {
        val k:String = Thread.currentThread().name
        println("Korutin oqimi: $k")
    }
    val m = Thread.currentThread().name
    println("main funktsiya oqimi: $m")
}
```

Bu yerda **Thread.currentThread().name** ko'rinishidan foydalanib oqimning nomini olish ko'rsatib o'tilgan. Bu dasturning ishga tushirib quyidagi natijani konsolda ko'rish mumkin:

```
main funktsiya oqimi: main
Korutin oqimi: DefaultDispatcher-worker-1
```

Natijadan ko'rinib turibdiki, asosiy funksiya "main" deb nomlangan oqimda bajariladi, asosiy funksiya ichida yaratilgan korutin **DefaultDispatcher-worker-1** deb nomlangan boshqa bir oqimda bajariladi.

Dasturning asosiy funksiyasi bo'lgan **main** funksiyasidagi korutinning tana qismi **coroutineScope** funksiyasi yordamida yaratilgan bo'lib, yaratilgan korutin uchun **Dispatcher.Default** turdagi standart dispatcherdan foydalanadi. Yuqorida ko'rstatilgan dasturdagi korutin bu turdagi dispatcher bilan ishlashi ko'rib chiqildi. Kotlin dasturlash tilida korutinlar bilan ishlovchi quyidagi dispatcherlar mavjud:

- **Dispatchers.Default** – agar dispatcher turi aniq ko'rsatilmagan bo'lsa, bu dispatcherdan foydalaniladi. Bu tur umumiy fonli

oqimlardan foydalanadi va kiritish–chiqarish (fayllar, ma’lumotlar bazasi va tarmoq bilan ishlash) oqimi bilan ishlamaydigan buyruqlarga javob beradi;

- **Dispatchers.IO** – kerak bo‘lganda hosil bo‘ladigan va kiritish–chiqarish buyruqlarini (fayllar, ma’lumotlar bazasi va tarmoq bilan ishlash) bajarishga mo‘ljallangan umumiy oqimlarda foydalaniladi;
- **Dispatchers.Main** – Android yoki JavaFX ilovalari kabi grafik ilovalarda ishlatiladi;
- **Dispatchers.Unfinined** – korutin aniq bir oqim yoki oqimlar bilan ta’minlanmagan holatlarda ishlatiladi. **Kotlin** dasturlar tilini ishlab chiqaruvchilar bu turdan foydalanishni tavsiya etmaydi;
- **newSingleThreadContext** va **newFixedThreadPoolContext** – bu ikki tur korutirlarni bajarish uchun oqimlarni qo‘lda sozlash imkonini beradi.

Korutin yaratuvchi funksiyalar, ya’ni **launch** va **async** ishga tushirish vaqtida yuqorida keltirilgan turlardan foydalanib, korutin dispatcherini o‘rnatish mumkin:

```
import kotlinx.coroutines.*
suspend fun main() = coroutineScope{
    launch(Dispatchers.Default) {
        val k:String = Thread.currentThread().name
        println("Korutin oqimi: $k")
    }
    val m = Thread.currentThread().name
    println("main funktsiya oqimi: $m")
}
```

Dispatchers.Unconfined turi buyruqlar ketma–ketligida korutinni birinchi marta to‘xtaguncha (**delay** funksiyasiga qadar) boshqaradi. Qayta tiklangan korutin yuqorida ko‘rsatilgan oqim turlaridan birida o‘z ishini davom ettiradi.

```
import kotlinx.coroutines.*
suspend fun main() = coroutineScope{
    launch(Dispatchers.Unconfined) {
        var k:String = Thread.currentThread().name
        println("Korutin oqimi (1): $k")
        delay(500L)
        k = Thread.currentThread().name
        println("Korutin oqimi (2): $k")
    }
    val m = Thread.currentThread().name
    println("main funktsiya oqimi: $m")
}
```

Yuqorida keltirilgan dasturning natijasi quyidagicha bo‘ladi:

```
Korutin oqimi (1): main
main funktsiya oqimi: main
Korutin oqimi (2): kotlinx.coroutines.DefaultExecutor
```

newSingleThreadContext turi qo‘lda ko‘rsatilgan nom bilan ishlaydigan oqimni ifodalaydi:

```
import kotlinx.coroutines.*
suspend fun main() = coroutineScope{
    launch(newSingleThreadContext("My Stream")) {
        val k:String = Thread.currentThread().name
        println("Korutin oqimi: $k")
    }
    val m = Thread.currentThread().name
    println("main funktsiya oqimi: $m")
}
```

Bu dasturni ishga tushirilganida konsol oynasiga quyidagicha natija chiqariladi:

```
main funktsiya oqimi: main
Korutin oqimi: My Stream
```

9.6. Korutinar bajarilishi bekor qilish

Ilovalar ishlayotganda korutirlarni bajarilishini bekor qilish mumkin. Masalan, internet resurs ma’lumotlarini yuklab olish uchun mobil ilovada korutin ishga tushiriladi, foydalanuvchi ilovaning boshqa sahifasiga o‘tish kerak bo‘lgan vaqtda ishlayotgan korutin kerak bo‘lmaydi. Bunday holda, tizim resursini isrof qilmaslik uchun korutin bajarilishini to‘xtatish yoki bekor qilish kerak bo‘ladi.

Yaratilgan korutin **Job** interfeysining obyekt bo‘lib, bajarilayotgan korutinni bekor qilish uchun **cancel** metodidan foydalaniladi. Quyida keltirilgan dasturda korutinni bekor qilish ko‘rsatib o‘tilgan:

```
import kotlinx.coroutines.*
suspend fun main() = coroutineScope{
    val downloader: Job = launch{
        println("Fayllarni yuklashni boshlash")
        for(i in 1..5){
            println("$i - fayl yuklandi")
            delay(500L)
        }
    }
    delay(800L)
    println("Fayllar yuklanishi to'xtatildi")
    downloader.cancel() //korutinni bekor qilish
    downloader.join() //korutin tugashini kutish
    println("Dastur tugadi")
}
```



```
}
```

Yuqorida keltirilgan dasturda faylni yuklashga yo‘naltirilgan ilovaga taqlid qilingan korutin aniqlangan. Korutinning tana qismida 1 dan 5 gacha bo‘lgan sonlar ketma–ketligi keltirilib, shartli ravishda 5 ta faylni yuklash sifatida qaralgan. Korutin obyekt sifatida *downloader* nomli o‘zgaruvchi keltirilib, bu obyekt bilan korutin bajarilishini to‘xtatish uchun **cancel** metodidan foydalanilgan. Korutinni to‘xtatish buyrug‘idan so‘ng, **join** metodi yozilib, bu metod orqali korutin tugatishini kutish aks ettirilgan. Bu dasturning natijasi konsol oynasida quyidagicha ko‘rinadi:

```
Fayllarni yuklashni boshlash
1 - fayl yuklandi
2 - fayl yuklandi
Fayllar yuklanishi to‘xtatildi
Dastur tugadi
```

Yuqorida keltirilgan dasturdagi **cancel** va **join** metodlarini birlashtirib **cancelAndJoin** metodidan foydalanish mumkin.

```
import kotlinx.coroutines.*
suspend fun main() = coroutineScope{
    val downloader: Job = launch{
        println("Fayllarni yuklashni boshlash")
        for(i in 1..5){
            println("$i - fayl yuklandi")
            delay(500L)
        }
    }
    delay(800L)
    println("Fayllar yuklanishi to‘xtatildi")
    //korutinni bekor qilish va tugashini kutish
    downloader.cancelAndJoin()
    println("Dastur tugadi")
}
```

CancellationException istisnosi bilan ishlash

Korutinlar bilan ishlashga mo‘ljallangan **kotlinx.coroutines** paketida joylashgan **suspend** ko‘rinishidagi funksiyalarining barchasini to‘xtatish mumkin. Bu shuni anglatadiki, korutin uzilib qolganida tekshirish mumkin. Agar korutin uzilib qolsa, **CancellationException** istisnosi qaytariladi. Korutinning tanasida bekor qilingan korutinni istisnoli vaziyatlarini hal etish mumkin. Masalan:

```
import kotlinx.coroutines.*
suspend fun main() = coroutineScope{
    val downloader: Job = launch{
        try {
            println("Fayllarni yuklashni boshlash")
        }
    }
}
```

```

        for (i in 1..5) {
            println("$i - fayl yuklandi")
            delay(500L)
        }
    }
    catch (e:CancellationException){
        println("Fayllarni yuklab olish to'xtatildi")
    }
    finally {
        println("Yuklash tugallandi")
    }
}
delay(800L)
println("Fayllar yuklanishi majburan to'xtatildi")
//korutinni bekor qilish va tugashini kutish
download.cancelAndJoin()
println("Dastur tugadi")
}

```

Bu dasturda korutin tanasida istisnoli bartaraf etuvchi blok ishlatilgan. Agar korutin tashqi tomondan to'xtatilsa, korutinni to'xtaganligini bildiruvchi **CancellationException** turidagi istisno hosil bo'ladi va istisnoli bartaraf etish uchun xatolikni ushlash ya'ni, **catch** blokida ko'rsatilgan buyruqlar ketma-ketligi bajariladi va undan so'ng istisnoli tugashini bildiruvchi **finally** blokida keltirilgan buyruqlar ketma-ketligi bajariladi. Agar korutin vazifasini to'ltaligicha bajarib bo'lsa, unda istisnoli holat yuzaga kelmaydi va istisnoli boshqaruvchi blokning tugash qismi bajariladi.

Dasturda *download* nomli korutin obyekti **cancel** metodi yordamida to'xtatilib, korutin bekor qilingan. Bunda yuqorida aytilganidek istisno hosil bo'ladi. Natijada dastur quyidagi ko'rinishda konsolga qiymat chiqaradi:

```

Fayllarni yuklashni boshlash
1 - fayl yuklandi
2 - fayl yuklandi
Fayllar yuklanishi majburan to'xtatildi
Fayllarni yuklab olish to'xtatildi
Yuklash tugallandi
Dastur tugadi

```

async ko‘rinishidagi korutnlarni bekor qilish

async funksiyasi yordamida yaratilgan korutinlar ham bekor qilinishi mumkin. Bunday holatlardagi istisnoli vaziyatlarni bartaraf etish uchun odatda **await** metodi istisno hosil bo‘luvchi blok ichiga joylashtiriladi:

```
import kotlinx.coroutines.*
suspend fun main() = coroutineScope{
    val message = async {
        getMessage()
    }
    message.cancelAndJoin()
    try {
        println("Habar: ${message.await()}")
    }
    catch (e:CancellationException){
        println("Korutin bekor qilindi")
    }
    println("Dastur tugadi")
}
suspend fun getMessage(): String{
    delay(500L)
    return "Salom"
}
```

Ushbu dasturning natijaviy qiymati quyidagidan iborat:

```
Korutin bekor qilindi
Dastur tugadi
```

9.7. Channel interfeysi

Kanallar – ma’lumotlar oqimini uzatish imkonini beradi. **Kotlin** dasturlash tilida kanallar **Channel** interfeysi bo‘lib, uning ikkita asosiy metodi mavjud. Bu ikki metod quyidagilar hisoblanadi:

- **abstract suspend fun send(element: E):Unit** – jo‘natish metodi bo‘lib, kanalga *element* obyektini jo‘natadi;
- **abstract suspend fun receive():E** – ma’lumotlarni olish metodi bo‘lib, kanaldan ma’lumotlarni olish uchun ishlatiladi.

Quyida keltirilgan dasturda **Int** turidagi sonlarni uzatuvchi eng oddiy kanal ko‘rsatib o‘tilgan:

```
import kotlinx.coroutines.*
import kotlinx.coroutines.channels.Channel
suspend fun main() = coroutineScope{
    val channel = Channel<Int>()
    launch {
        for (n in 1..5) {
            // kanal orqali ma’lumot jo‘natish
            channel.send(n)
        }
    }
}
```

```

    }
}
// kanaldan kelgan ma'lumotlarni qabul qilish
repeat(5) {
    val number = channel.receive()
    print("$number, ")
}
println("\nDastur tugadi")
}

```

Dasturda ishlatilgan **Channel** interfeysi **Kotlin** dasturlash tilining *kotlinx.coroutines.channels* paketida joylashgan. **Channel** interfeysidan foydalanish uchun dasturga shu paketni import qilish kerak bo'ladi. Dasturda import qilish quyidagi ko'rinishda ko'rsatib o'tilgan:

```
import kotlinx.coroutines.channels.Channel
```

Dasturda kanalni ifodalovchi o'zgaruvchi yoki obyekt yaratilgan. Bu obyekt quyidagi ko'rinishda ifodalanadi:

```
val channel = Channel<Int>()
```

Dasturda yaratilgan kanal **Int** turidagi qiymatlarni uzatishi sababli, yaratilgan *channel* obyekt shunga mos ravishda **Int** turi bilan yozilgan. Kanal yaratilgandan so'ng kanalga kerakli bo'lgan qiymatlarni uzatish uchun korutindan foydalanilgan bo'lib, bu korutin kanalga 1 dan 5 gacha bo'lgan sonlarni uzatishga yordam beradi. Korutinning tana qismida yaratilgan *channel* obyektiga **send** metodi yordamida sonlar uzatish ko'rsatib o'tilgan. Kanalning **send** metodi asosan korutinda ishlaganligi sababli, bu metod korutin tanasida yozilishi shart.

Ushbu dasturda takrorlanishni ifodalovchi **repeat** funksiyasi ishlatilgan. Kanaldan ma'lumotlarni olish uchun funksiyaning parametri sifatida 5 soni yozilib takrorlanish soni ko'rsatilgan. Chunki kanalga dastur 5 ta qiymat uzatadi. Takrorlanish funksiyasining tana qismida **receive** metodi ishlatilgan bo'lib, bu metod kanal orqali qabul qilingan qiymatlarni qaytarish xususiyatiga ega. Yuqorida keltirilgan dasturning konsoldagi natijasi quyidagidan iborat.

```

1, 2, 3, 4, 5,
Dastur tugadi

```

Boshqa turdagi ma'lumotlarni ham kanal orqali jo'natish mumkin. Quyida keltirilgan dasturda satrdan tashkil topgan ma'lumotlarni kanal orqali jo'natish ko'rsatib o'tilgan:

```

import kotlinx.coroutines.*
import kotlinx.coroutines.channels.Channel
suspend fun main() = coroutineScope{
    val channel = Channel<String>()

```

```

launch {
    val users = listOf("Tohir", "Bobur", "Samiya")
    for (user in users) {
        println("Jo'natildi: $user")
        channel.send(user)
    }
}
repeat(3) {
    val user = channel.receive()
    println("Qabul qilindi: $user")
}
println("Dastur tugadi")
}

```

Ushbu dasturda kanal orqali 3 ta satr jo'natilishi va qabul qilinishi ko'rsatib o'tilgan. Dasturning natijasi konsol oynasida ko'rinishi:

```

Jo'natildi: Tohir
Qabul qilindi: Tohir
Jo'natildi: Bobur
Qabul qilindi: Bobur
Jo'natildi: Samiya
Qabul qilindi: Samiya
Dastur tugadi

```

Kanalni yopish

Kanalda boshqa ma'lumotlar yo'qligini ko'rsatish uchun kanalni **close** metodi yordamida yopish mumkin. Agar kanaldan ma'lumotlarni qabul qilish uchun **for** takrorlanish operatoridan foydalanilsa, kanalni yopish to'g'risidagi axborot qabul qilinganida takrorlanish operatori oldin qabul qilgan barcha ma'lumotlarni ustida ko'rsatilgan buyruqlar ketma-ketligini bajaradi:

```

import kotlinx.coroutines.*
import kotlinx.coroutines.channels.Channel
suspend fun main() = coroutineScope{
    val channel = Channel<String>()
    launch {
        val users = listOf("Tohir", "Bobur", "Samiya")
        for (user in users) {
            channel.send(user)
        }
        channel.close() // kanalni yopish
    }
    for(user in channel) { // ma'lumotlarni qabul qilish
        println(user)
    }
    println("Dastur tugadi")
}

```

produce va consume andozalari

Yuqorida keltirilgan misollarda asosan bir korutindan ikkinchisiga ma'lumotlarni uzatishni keng tarqalgan usuli ko'rsatib o'tilgan. Bunday ko'rinishdagi kodlarni soddalashtirish va yozishni osonlashtirish uchun **Kotlin** dasturlash tili bir qator qo'shimcha funksiyalarni taqdim etadi. Masalan, **produce** funksiyasi korutin yaratishni ifodalaydi va u kanalga ma'lumotlarni yuboradi. Bu funksiyadan foydalanib, ma'lumotlarni yuboradigan yangi korutin funksiyasini aniqlash mumkin:

```
fun CoroutineScope.getUsers():ReceiveChannel<String>=
produce{
    val users = listOf("Tohir", "Bobur", "Samiya")
    for (user in users)
        send(user)
}
```

Bu dasturda **getUsers** nomli metod aniqlangan bo'lib, u **CoroutineScope** interfeysini kengaytirish yordamida yaratilgan. Metod qiymat qaytaruvchi hisoblanib, **ReceiveChannel** turida qiymat qaytaradi. **ReceiveChannel** turi o'z navbatida satrli qiymatlardan tashkil topgan qiymatlar majmuasidan iborat bo'ladi.

getUsers metodi korutin yaratuvchi **produce** funksiya bilan ishlab chiqilgan. Yaratilgan metodda satrli ko'rinishda bo'lgan qiymatlarni kanalga yuborish ko'rsatib o'tilgan. Kanaldan ma'lumotlarni o'qib olish uchun **ReceiveChannel** turida hosil qilingan obyektning **consumeEach** metodini ishlatish mumkin, ushbu metod dasturda takrorlanuvchi jarayon hosil qiladi:

```
val users = getUsers()
users.consumeEach{ user -> println(user) }
```

Yuqorida keltirilgan dasturlarni birlashtirilsa, to'liq dastur quyidagi ko'rinishga ega bo'ladi:

```
import kotlinx.coroutines.*
import kotlinx.coroutines.channels.*
suspend fun main() = coroutineScope{
    val users = getUsers()
    users.consumeEach { user -> print("$user, ") }
    println("\nDastur tugadi")
}
fun CoroutineScope.getUsers(): ReceiveChannel<String> =
produce{
    val users = listOf("Tohir", "Bobur", "Samiya")
    for (user in users) {
        send(user)
    }
}
```

}

Ushbu dasturning konsoldagi natijasi:

Tohir, Bobur, Samiya,
Dastur tugadi

Nazorat savollari:

1. Korutin nima?
2. Korutin hosil qiluvchi funksiyalarni sanab bering.
3. Korutindan qiymat qaytarish mumkinmi?
4. Qaysi korutin yaratuvchi funksiya qiymat qaytara oladi?
5. Korutin maydoni deganda nimani tushunasiz?
6. Korutin dispetcheri qanday vazifani bajaradi?
7. Kanallarning vazifasi nimadan iborat?
8. Kanallar bilan ishlovchi interfeyslarni sanab bering.
9. Kanal hosil qilishda funksiyalardan foydalanish mumkinmi?
10. Kanallarni yopish mumkinmi?
11. Kanal yopuvchi metod qanday?

10. Asinxron oqimlar

10.1. Asinxron oqimlarga kirish

Korutin bitta qiymatni qaytarishga imkon beradi. Buning uchun yuqorida keltirilgan mavzudagi **asyn** funksiyasidan foydalaniladi. **Kotlin** dasturlash tili obyektlar to‘plamini qaytarish imkonini beradi. Obyektlar to‘plamini qaytarish uchun korutin **List** kabi elementlar to‘plamini qaytarishi kerak, masalan:

```
import kotlinx.coroutines.*
suspend fun main() = coroutineScope{
    launch {
        getUsers().forEach { user -> println(user) }
    }
}
suspend fun getUsers(): List<String> {
    delay(1000L)
    return listOf("Tohir", "Bobur", "Samiya")
}
```

Biroq bunday to‘plamlarning muammosi shundaki, ular bir vaqtning o‘zida barcha obyektlarni qaytaradi. Agar ro‘yxatda 1000 ta obyekt kutilsa, shunga mos ravishda **getUsers** funksiyasi 1000 ta obyektlar ro‘yxatini qaytargunga qadar dastur qaytayotgan ma’lumotlarni boshqara olmaydi. Bunday katta ma’lumotlarni asosan ma’lumotlar bazasidan yoki internet resurslaridan olish mumkin.

Kotlin dasturlash tilida bu muammoni aynan asinxron oqimlar hal qila oladi. Yuqorida keltirilgan dasturni asinxron oqim yordamida qanday hal qilinishini quyidagi dasturda berilgan:

```
import kotlinx.coroutines.flow.*
import kotlinx.coroutines.*
suspend fun main(){
    getUsers().collect { user -> println(user) }
}
fun getUsers(): Flow<String> = flow {
    val database = listOf("Tohir", "Bobur", "Samiya")
    var i = 1;
    for (item in database){
        delay(400L)
        print("$i-element: ")
        emit(item)
        i++
    }
}
```


Asinxron oqimlarni yaratish va ular bilan ishlashda **Flow** interfeysi qo'llaniladi. Ya'ni, asinxron oqim **Flow** turidagi obyektни qaytaradi. Ushbu interfeys elementlar to'plamidan iborat bo'lib, elementlarning turini ko'rsatish kerak bo'ladi. Yuqorida keltirilgan dasturdagi elementlar satrli ko'rinishda bo'lganligi sababli, **Flow** interfeysi **String** turidagi elementlardan tashkil topadi. Yuqorida keltirilgan dasturdagi **getUsers** nomli funksiya **String** turidagi elementlardan tashkil topganligi va asinxron ko'rinishda qaytarilishi uchun quyidagi ko'rinishda yozilgan:

```
fun getUsers(): Flow<String>
```

Asinxron oqim bilan ishlovchi funksiyalar **suspend** ko'rinishida yozilishi shart emas. Asinxron oqimlarda **Flow** turida ma'lumotlar olish uchun **flow** funksiyasi ishlatiladi.

```
fun getUsers(): Flow<String> = flow {  
    // flow funktsiyasi yordamida asinxron oqim yaratish  
}
```

Yuqorida keltirilgan dasturda shartli ravishda ma'lumotlar bazasi bilan ishlovchi dasturga taqlid qilinib, ma'lumotlar bazasi sifatida satrlar ro'yxati olingan. Takrorlanish jarayonida ushbu ro'yxatning har bir elementiga murojaat qilinadi va **emit** funksiyasi yordamida joriy element yoki obyekt oqimga yuboriladi.

Dasturning asosiy funksiyasi bo'lgan **main** da **getUsers** nomli funksiyaga murojaat qilish ko'rsatib o'tilgan. **Flow** interfeysida obyektlarni oqimdan boshqarish uchun bir qancha funksiyalar aniqlangan, ulardan biri **collect** funksiyasidir. Parametr sifatida keltirilgan obyekt oqimdan uzatilgan obyektни ifodalaydi. Bu funksiyaning ko'rinishi quyidagicha bo'ladi:

```
getUsers().collect { user -> println(user) }
```

Ushbu buyruq bajarilganida konsol oynasida quyidagicha natija chiqadi:

```
1-element: Tohir  
2-element: Bobur  
3-element: Samiya
```

Shunda dasturdagi **getUsers** funksiyasi barcha qatorlarni qaytarilishini kutmaydi. Quyida asinxron oqimdan qaytariladigan sonlari ko'rish mumkin:

```
import kotlinx.coroutines.flow.*  
suspend fun main(){  
    getNumbers().collect { number -> print("$number, ") }  
}  
fun getNumbers(): Flow<Int> = flow{
```

```

    for(item in 1..5){
        emit(item * item)
    }
}

```

Keltirilgan dastur yuqorida keltirilgan satrli ro'yxat asinxron oqim bilan yuborish bilan bir xil vazifa bajaradi. Bu dasturda **getNumbers** funksiyasi **Int** sonlarini qabul qiluvchi asinxron oqimni ifodalaydi. Obyekt sifatida oqimdan 1 dan 5 gacha bo'lgan sonlarning kvadratlari qaytariladi. Ushbu dasturning konsoldagi natijasi quyidagicha:

```
1, 4, 9, 16, 25,
```

Shuni ta'kidlash lozimki, qabul qilingan ma'lumotlar **collect** funksiyasi uchramagunga qadar asinxron oqimga o'tkazilmaydi.

```

import kotlinx.coroutines.flow.*
suspend fun main(){
    // oqim yaratish
    val numberFlow = getNumbers()
    println("numberFlow nomli oqim yaratildi ")
    println("collect funksiyasini ishga tushirish")
    // oqimni ishga tushirish
    numberFlow.collect { number -> print("$number, ") }
}
fun getNumbers() = flow{
    println("numberFlow nomli oqimni ishga tushirish")
    for(item in 1..5){
        emit(item * item)
    }
}

```

Ushbu dasturning natijasi konsol oynasida quyidagi ko'rinishda chiqariladi:

```

numberFlow nomli oqim yaratildi
collect funksiyasini ishga tushirish
numberFlow nomli oqimni ishga tushirish
1, 4, 9, 16, 25,

```

10.2. Asinxron oqimlar yaratish

Asinxron oqimlar yaratish uchun turli xildagi metodlardan foydalanish mumkin. Bu metodlar 3 ta bo'lib, ularning ko'rinishlari quyida berilgan: **flow**, **flowOf** va **asFlow**.

flow funksiyasi

flow funksiyasi asinxron oqimlar tashkil etilishida ishlatiladi. U funksiya yaratishga yoki funksiya sifatida qo'llanilishi mumkin. Oldingi rejada funksiya yaratish ko'rsatib o'tilgandi. Funksiya sifatida ishlatilishi quyidagi dasturda berilgan:

```
import kotlinx.coroutines.flow.*
suspend fun main(){
    val userFlow = flow {
        val usersList = listOf("Tohir", "Bobur", "Samiya")
        for (item in usersList)
            emit(item)
    }
    userFlow.collect({user -> print("$user, ")})
}
```

Yuqorida keltirilgan dasturda **Flow<String>** turidagi **userFlow** oʻzgaruvchisini yaratilgan va **flow** funksiyasi tomonidan ishlab chiqarilgan oqimni ifodalaydi. Bunda oqim satrlar roʻyxatida berilgan obyektlarni oqimga oʻtkazish koʻrsatib oʻtilgan. Ushbu dasturning natijasi konsol oynasida quyidagi koʻrinishda boʻladi:

Tohir, Bobur, Samiya,

flowOf funksiyasi

Asinxron oqim hosil qiluvchi funksiyalardan biri **flowOf** boʻlib, berilgan qiymatlar toʻplamida oqim hosil qiladi. Quyida ushbu funksiya yordamida yaratilgan asinxron oqim keltirilgan:

```
import kotlinx.coroutines.flow.*
suspend fun main(){
    val numberFlow : Flow<Int> = flowOf(1, 2, 3, 5, 8)
    numberFlow.collect{n -> print("$n, ")}
}
```

Yuqoridagi dasturda 5 ta butun sonli qiymatlaridan tashkil topgan asinxron oqim ishlab chiqaruvchi **flowOf** funksiyasidan foydalanilgan, shuning uchun hosil boʻladigan oqim **Flow<Int>** turiga mansub boʻladi. Koʻrsatilgan barcha qiymatlar avtomatik ravishda oqimga uzatiladi va ularni **collect** funksiyasi yordamida olish mumkin. Bu dasturning natijasi quyidagidan iborat:

1, 2, 3, 5, 8,

Satlardan tashkil topgan roʻyxatni ham shu koʻrinishda yaratish mumkin:

```
import kotlinx.coroutines.flow.*
suspend fun main(){
    val userFlow = flowOf("Tohir", "Bobur", "Samiya")
    userFlow.collect({user -> println(user)})
}
```

asFlow metodi

Kotlin dasturlash tilida standart to‘plamlar va ketma–ketliklar **asFlow** metodi bilan kengaytirilgan, bu metod to‘plamni yoki ketma–ketlikni oqimga aylantirish imkonini beradi:

```
import kotlinx.coroutines.flow.*
suspend fun main() {
    // ketma-ketlikni oqimga aylantirish
    val numberFlow : Flow<Int> = (1..5).asFlow()
    numberFlow.collect { n -> println(n) }
    // List<String> to‘plamini oqimga aylantirish
    val userFlow = listOf("Tohir", "Bobur", "Samiya").asFlow()
    userFlow.collect { user -> println(user) }
}
```

10.3. Oqim bilan ishlovchi operatorlar va funksiyalar

Kotlin dasturlash tilida oqimlar bilan ishlash uchun **Flow** interfeysida bir qancha funksiyalar aniqlangan. Ular ma’lum bir qiymatni yoki qayta ishlangan oqimni qaytarishiga qarab ikki turga bo‘linadi, bular terminal va oraliq funksiyalardir. Quyida bu ikki turdagi funksiyalar haqida ma’lumot keltirilgan.

Oqimning terminal funksiyalari

Oqimning terminal funksiyalari obyektlarni oqimdan to‘g‘ridan–to‘g‘ri qabul qilish yoki qandaydir yakuniy qiymat qaytarish imkonini beruvchi funksiyalarni ifodalaydi:

- collect() – uzatilayotgan qiymatlarni oqimdan oladi;
- toList() – qiymatlar oqimini **List** to‘plamiga o‘zgartiradi;
- toSet() – qiymatlar oqimini **Set** to‘plamiga o‘zgartiradi;
- first()/firstOrNull() – oqimdan birinchi obyektini oladi;
- single()/singleOrNull() – oqimdan bitta obyektini olishni kutadi;
- count() – oqimdagi elementlar sonini oladi;
- reduce() – oqim elementlarida ma’lum bir buyruqning natijasini oladi;
- fold() – reduce() funksiyasi bilan bir xil bo‘lib, oqim elementlarida boshlang‘ich qiymatni oladi.

Oraliq funksiyalar

Oraliq funksiyalar oqimdagi qiymatlarni oladi va qayta ishlangan oqimni qaytaradi.

- combine() – oqimlardan kelgan ma’lumotlarga ko‘rsatilgan buyruqlarni qo‘llagan holda ikki oqimni birlashtiradi;

- `drop()` – oqim boshidan belgilangan miqdordagi qiymatlarni chiqarib tashlab qolgan natijaviy olingan oqimni qaytaradi;
- `filter()` – shartga mos keladigan elementlarni qoldirib oqimni filterlaydi;
- `filterNot()` – shartga mos kelmaydigan elementlarni qoldirib, oqimni filterlaydi;
- `filterNotNull()` – **null** bo'lgan barcha elementlarni olib tashlab, oqimni filterlaydi;
- `map()` – oqim elementlariga ko'rsatilgan funksiyani qo'llaydi;
- `onEach()` – oqim bilan ishlashdan oldin, oqim elementlariga ko'rsatilgan funksiyani qo'llagan;
- `take()` – oqimdan ma'lum miqdordagi elementlarni tanlaydi;
- `transform()` – oqim elementlariga ko'rsatilgan funksiyani qo'llaydi;
- `zip()` – ikki oqimning elementlariga ko'rsatilgan funksiyani qo'llagan holda bitta oqim yaratadi.

Yuqorida keltirilgan barcha funksiyalarning ishlash tamoyili keyingi rejalarda batafsil tushuntirib o'tilgan.

10.4. Oqim elementlari soni `count`, `take` va `drop` funksiyalari.

`count` funksiyasi

Ushbu funksiya oqimdagi obyektlar sonini qaytaradi:

```
import kotlinx.coroutines.flow.*
suspend fun main(){
    val userFlow = listOf("Tohir", "Bobur", "Samad").asFlow()
    println("Soni: ${userFlow.count()}")
}
```

Ushbu funksiya ko'rsatilgan mantiqiy shart asosida oqimning elementlar sonini o'zgartirib ko'rsatish xususiyatiga ega:

```
import kotlinx.coroutines.flow.*
suspend fun main(){
    val userFlow = listOf("Tohir", "Bobur", "Kamola",
"Samad", "Laziza").asFlow()
    val count = userFlow.count{ name -> name.length > 3 }
    println("Soni: $count")
}
```

Ushbu dasturda shartli ravishda foydalanuvchining nomini ifodalash uchun *name* o'zgaruvchi ko'rsatilgan. Bu o'zgaruvchi o'zida saqlayotgan foydalanuvchining nomi uzunligi 5 ta belgidan ko'p bo'lgan holdagi nomlarning sonini qaytaradi. Yuqorida ko'rsatilgan dasturning natijasi 2 ga teng bo'lib, bu natija foydalanuvchilar nomlaridagi belgilar soni aynan 2 foydalanuvchida 5 ta belgidan ko'pligini bildiradi.

take funksiyasi

Ushbu funksiya oqimdagi elementlar sonini cheklaydi. Parametr sifatida qabul qilingan sonli qiymat oqim boshidan qoldirilishi kerak bo'lgan elementlar sonini ifodalaydi:

```
import kotlinx.coroutines.flow.*
suspend fun main() {
    val userFlow = listOf("Tohir", "Bobur", "Kamola",
"Samad", "Laziza").asFlow()
    userFlow.take(3).collect { user -> print("$user, ") }
}
```

Yuqorida keltirilgan dasturdagi oqimning boshidagi 3 ta element natija sifatida ekranga chiqariladi:

Tohir, Bobur, Kamola

drop funksiyasi

Ushbu funksiya ko'rsatilgan sonli qiymatdagi elementlarni oqimning boshidan olib tashlaydi. Ko'rsatilgan sonli qiymat parametr sifatida keltiriladi:

```
import kotlinx.coroutines.flow.*
suspend fun main(){
    val userFlow = listOf("Tohir", "Bobur", "Kamola",
"Samad", "Laziza").asFlow()
    userFlow.drop(3).collect{user -> print("$user, ")}
}
```

Yuqorida keltirilgan dasturda birinchi uchta elementni olib tashlaydi, natijada oxirgi ikki element oqimda qoladi. Dastur konsolga quyidagicha natija beradi:

Samad, Laziza

10.5. first va single funksiyalari

first yoki firstOrNull funksiyalari

Ushbu funksiyalardan **first** funksiyasi oqimning birinchi elementini oladi.

```
import kotlinx.coroutines.flow.*
suspend fun main(){
    val userFlow = listOf("Tohir", "Bobur", "Kamola",
"Samad", "Laziza").asFlow()
    val firstUser = userFlow.first()
    println("Birinchi foydalanuvchi: $firstUser")
}
```

Ushbu funksiya parametr sifatida mantiqiy turga tegishli bo'lgan shart buyruqlarini olishi mumkin. Agar parametr joylashtirilsa, bu funksiya shartga mos keladigan oqimning birinchi elementini qaytaradi:

```
import kotlinx.coroutines.flow.*
suspend fun main(){
    val userFlow = listOf("Tohir", "Bobur", "Kamola",
"Samad", "Laziza").asFlow()
    val firstUser = userFlow.first{ name-> name.length > 5}
    println("Birinchi foydalanuvchi: $firstUser")
}
```

Ushbu dasturning natijasi quyidagi ko‘rinishda bo‘ladi:

Birinchi foydalanuvchi: Kamola

Oqim elementlarining hech biri shartga javob bermasa, xatolik yuzaga keladi. Ushbu xatolikni bartaraf etish uchun **Kotlin** dasturlash tilining xatolik bilan ishlovchi **try...catch** operatoridan foydalanish tavsiya etiladi. Ammo istisnoli vaziyat keltirib chiqarmaslik uchun **first** funksiyasining ekvivalenti bo‘lgan **firstOrNull** nomli funksiya ishlatilsa ham bo‘ladi. Bu funksiya oqim elementlarini shartga javob bermagan holda **null** qiymat qaytarishga asoslangan. Yuqoridagi dasturga quyidagicha o‘zgarishlar kiritilsa, natijada **null** qiymati qaytadi.

```
val firstUser = userFlow.first{ name-> name.length > 6}
```

Bu o‘zgartirishdan so‘ng dastur quyidagi ko‘rinishga keladi.

```
import kotlinx.coroutines.flow.*
suspend fun main(){
    val userFlow = listOf("Tohir", "Bobur", "Kamola",
"Samad", "Laziza").asFlow()
    val firstUser = userFlow.first{ name-> name.length > 6}
    println("Birinchi foydalanuvchi: $firstUser")
}
```

single yoki singleOrNull funksiyalari

single funksiyasi oqim faqat bitta elementdan iborat bo‘lsa, shu elementni qaytarishga asoslangan. Oqimda hech qanday element bo‘lmasa, **NoSuchElementException** istisnosi va agar oqimda bir nechta element bo‘lsa, **IllegalStateException** istisnosi yuzaga keladi. Quyida oqim bitta elementdan iborat bo‘lgan dastur keltirib o‘tilgan:

```
import kotlinx.coroutines.flow.*
suspend fun main(){
    val userFlow = listOf("Tohir").asFlow()
    try {
        val singleUser = userFlow.single()
        println("Bitta foydalanuvchi: $singleUser")
    }
    catch (e:Exception) { println(e.message) }
}
```

Ko'rsatilgan dasturda istisnoli vaziyat yuzaga kelganda shu vaziyat haqida habar beradigan **try...catch** operatoridan foydalanilgan. Oqim bo'sh bo'lganda yoki oqimda bittadan ko'p element bo'lganda istisno yuzaga keladi. Bitta elementdan iborat bo'lishi mumkin bo'lgan oqimlarda istisnoli vaziyatni boshqarish uchun **singleOrNull** funksiyasidan foydalanish mumkin. Bu funksiya oqim bo'sh yoki bir nechta elementdan iborat bo'lganda **null** qiymat qaytaradi.

```
import kotlinx.coroutines.flow.*
suspend fun main() {
    val userFlow = listOf<String>().asFlow()
    val singleUser = userFlow.singleOrNull()
    if (singleUser != null)
        println("Bitta foydalanuvchi: $singleUser")
    else
        println("Topilmadi")
}
```

Ushbu dasturning natijasi konsol oynasida quyidagi ko'rinishda bo'ladi:

Topilmadi

10.6. map va transform funksiyalari

map funksiyasi

map funksiyasi oqimdan kelgan ma'lumotlarni o'zgartiradi. Parametr sifatida oqimdan kelgan ma'lumotlarni o'zgartiruvchi funksiya ishlatiladi. Parametr sifatida keltirilgan funksiya oqimda kelgan obyektни o'zgartirib, o'zgartirilgan ma'lumotni qiymat sifatida qaytarishi mumkin.

```
import kotlinx.coroutines.flow.*
suspend fun main() {
    val peopleFlow = listOf(
        Person("Tohir", 37), Person("Samiya", 41),
        Person("Bobur", 21)
    ).asFlow()
    peopleFlow.map { person -> person.name }
        .collect { personName -> print("$personName, ") }
}
data class Person(val name: String, val age: Int)
```

Ushbu keltirilgan dasturning natijasi quyida berilgan:

Tohir, Samiya, Bobur,

Yuqoridagi dasturda oqim sifati keltirilgan ro'yxat **Person** turidagi elementlardan iborat. Bu dasturda **Person** turi insonni ifodalaydi. Ushbu dasturda ro'yxat elementlari oqimi hosil qilingan. **map** funksiyasi yordamida **Person** turidagi oqimni **String** ko'rinishiga keltirilmoqda va

oqimning **collect** funksiyasi yordamida **String** turidagi ma'lumotlar qabul qilinib uning ustida amallar bajarilishi ko'rsatilmoqda.

Yuqoridagi dasturga bir nechta o'zgartirishlar kiritib, voyaga yetgan yoki yetmagan insonlarini ko'rsatish mumkin.

```
import kotlinx.coroutines.flow.*
suspend fun main() {
    val peopleFlow = listOf(
        Person("Tohir", 37),
        Person("Bilol", 5),
        Person("Samiya", 14),
        Person("Bobur", 21),
    ).asFlow()
    peopleFlow.map { person ->
        object {
            val name = person.name
            val isAdult = person.age > 18
        }
    }.collect { user ->
        println("${user.name} voyaga ${if (user.isAdult)
"etgan" else "etmagan"} ")
    }
}
data class Person(val name: String, val age: Int)
```

Bu dasturning natijasi konsol oynasida quyidagicha chiqadi:

```
Tohir voyaga etgan
Bilol voyaga etmagan
Samiya voyaga etmagan
Bobur voyaga etgan
```

transform funksiyasi

Ushbu funksiya ham **map** funksiya singari oqimdagi ma'lumotlari o'zgartirishga asoslangan bo'lib, **map** funksiyasidan farqli tomoni shundan iboratki, bu funksiya o'zgartirilgan ma'lumotni **emit** funksiyasi yordamida oqimga o'tkazadi. Bunga quyida keltirilgan dastur misol bo'la oladi:

```
import kotlinx.coroutines.flow.*
suspend fun main() {
    val peopleFlow = listOf(
        Person("Tohir", 37),
        Person("Bilol", 5),
        Person("Samiya", 14),
        Person("Bobur", 21),
    ).asFlow()
    peopleFlow.transform { person ->
        if (person.age > 18) {
            emit(person.name)
        }
    }
}
```

```
        }.collect { personName -> println(personName) }
    }
data class Person(val name: String, val age: Int)
```

Bu dasturda **Person** turidagi obyektning *age* xususiyati 18 dan katta bo'lsa, bu obyektни **emit** funksiyasi yordamida yangi oqimga o'tkazadi. Dasturning natijasi konsol oynasida quyidagicha hosil bo'ladi:

```
Tohir
Bobur
```

Ushbu funksiyada **emit** funksiyadan bir necha marta foydalanish ham mumkin. Bunga quyida keltirilgan dastur misol bo'la oladi:

```
import kotlinx.coroutines.flow.*
suspend fun main(){
    val numbersFlow = listOf(2, 3, 4).asFlow()
    numbersFlow.transform{ n ->
        emit(n)
        emit(n * n)
    }.collect { n -> print("$n, ") }
}
```

Bu misolda ro'yxatda keltirilgan raqamlar va ularning kvadratlarini oqimga jo'natish ko'rsatib o'tilgan. Dasturning natijasi quyidagi ko'rinishda bo'ladi:

```
2, 4, 3, 9, 4, 16,
```

10.7. Ma'lumotlarni filtrlash

filter funksiyasi

Oqimlarda ishlatiladigan **filter** funksiyasi obyektlarni filtrlash uchun ishlatiladi. Funksiyaning parametri shartni ifodalovchi buyruqlar ketma-ketligini oladi. Ushbu funksiyaga misol tariqasida quyidagi dasturni ko'rish mumkin:

```
import kotlinx.coroutines.flow.*
suspend fun main(){
    val peopleFlow = listOf(
        Person("Tohir", 37),
        Person("Bilol", 5),
        Person("Samiya", 14),
        Person("Bobur", 21),
    ).asFlow()
    peopleFlow.filter{
        person -> person.age > 18
    }.collect { person ->
        println("Nomi:${person.name}\tYoshi:${person.age}")
    }
}
data class Person(val name: String, val age: Int)
```

Yuqorida keltirilgan dasturning natijasi konsol oynasida quyidagi ko‘rinishda gavdalanadi:

```
Nomi: Tohir    Yoshi: 37
Nomi: Bobur   Yoshi: 21
```

takeWhile funksiyasi

Kotlin dasturlash tili turli vaziyatlar uchun bir qator filtrlash funksiyalarini taqdim etgan. Shunday filtrlash funksiyalaridan biri **takeWhile** funksiyasi bo‘lib, ko‘rsatilgan shart to‘g‘ri bo‘lganda oqimdan elementni tanlash imkonini beradi:

```
import kotlinx.coroutines.flow.*
suspend fun main(){
    val peopleFlow = listOf(
        Person("Tohir", 37),
        Person("Aziza", 23),
        Person("Bilol", 5),
        Person("Samiya", 14),
        Person("Bobur", 21),
    ).asFlow()
    peopleFlow.takeWhile{ person -> person.age > 18}
        .collect { person ->
            println("Nomi:${person.name}\tYoshi:${person.age}")
        }
}
```

```
data class Person(val name: String, val age: Int)
```

Bu dasturning natijasi quyidagicha:

```
Nomi:Tohir    Yoshi:37
Nomi:Aziza    Yoshi:23
```

Natijaga e‘tibor berilsa, ro‘yxatda keltirilgan obyektlardan biri chiqarilmagan. **takeWhile** funksiyasining vazifasi shart yolg‘on bo‘lgan vaqtda oqimga uzatayotgan ma‘lumotlarni to‘xtatadi, ya‘ni shart birinchi marta yolg‘on holatga kelgandan so‘ng qolgan ma‘lumotlar tekshirilmaydi.

dropWhile funksiyasi

dropWhile funksiyasi **takeWhile** funksiyasiga qarama–qarshi vazifani bajaradi. Ya‘ni, ko‘rsatilgan shartni qanoatlantiruvchi va birinchi marta shartni qanoatlantirmagan obyektдан oldingi obyektlarni oqimdan olib tashlaydi:

```
import kotlinx.coroutines.flow.*
suspend fun main(){
    val peopleFlow = listOf(
        Person("Tohir", 37),
        Person("Aziza", 23),
        Person("Bilol", 5),
        Person("Samiya", 14),
    )
```

```

        Person("Bobur", 21),
    ).asFlow()
peopleFlow.dropWhile{ person -> person.age > 18}
    .collect { person ->
        println("Nomi:${person.name}\tYoshi:${person.age}")
    }
}
data class Person(val name: String, val age: Int)

```

Yuqorida keltirilgan dasturda **dropWhile** funksiyasi *person.age > 18* shartga mos kelgan birinchi va ikkinchi **Person** obyektini o'tkazib yuboradi. Ro'yxatda keltirilgan uchinchi element shartga mos bo'lmaganligi sababli, uchinchi elementdan boshlab ro'yxatning oxirgi elementigacha oqimga yuboradi. Dasturning natijasi quyidagi ko'rinishda bo'ladi:

```

Nomi:Bilol      Yoshi:5
Nomi:Samiya     Yoshi:14
Nomi:Bobur      Yoshi:21

```

10.8. reduce va fold funksiyalari

reduce funksiyasi

Kotlin dasturlash tilida **reduce** funksiyasi oqimdagi barcha elementlarini bitta qiymatga keltirish uchun xizmat qiladi. **reduce** funksiyasi ikki parametrdan tashkil topgan bo'lib, birinchi parametr qaytarilishi kerak bo'lgan qiymatni ifodalasa, ikkinchi parametr esa oqimdagi mavjud elementlarni ifodalaydi. Quyida sonli elementlardan tashkil etilgan oqimni ko'rish mumkin. Keltirilgan ro'yxatdagi elementlarni yig'indisini chiqarish uchun ushbu funksiyadan foydalanilgan.

```

import kotlinx.coroutines.flow.*
fun main() {
    val numberFlow = listOf(1, 2, 3, 4, 5)
    val reducedValue = numberFlow.reduce{ a, b -> a + b }
    println(reducedValue) // 15
}

```

Dasturning natijasi ro'yxatda keltirilgan sonli qiymatlarning yig'indisiga teng, ya'ni 15 ga teng bo'ladi.

Sonli qiymatlardan tashqari satri qiymatlar bilan ham shunday dastur tuzish mumkin. Bunga misol sifatida quyidagi dasturni ko'rish mumkin:

```

import kotlinx.coroutines.flow.*
fun main() {
    val users = listOf("Tohir", "Bobur", "Kamol", "Samiya",
"Aziza").asFlow()
    val reducedValue = users.reduce{ a, b -> a + " " + b }
}

```

```
println(reducedValue)
}
```

Bu dasturda **reduce** funksiyasi satrlarni birlashtirish uchun ishlatilmoqda.

fold funksiyasi

fold funksiyasi oqim elementlarini bitta joyga yoki o‘zgaruvchiga yig‘adi. Bu funksiya **reduce** funksiyasidan farqi shundaki, birinchi parametr sifatida dastlabki qiymatni oladi. Quyida ushbu funksiyaga misol keltirilgan:

```
import kotlinx.coroutines.flow.*
suspend fun main(){
    val users = listOf("Tohir", "Bobur", "Kamol", "Samiya",
"Aziza").asFlow()
    val foldedValue = users.fold("Users:", {a,b->a+" "+b })
    println(foldedValue)
}
```

10.9. Oqimlarni birlashtirish

Kotlin dasturlash tilida **zip** funksiyasi ikkita oqimni birlashtirish imkonini beradi. **zip** funksiyasi ikkita parametr oladi. Birinchi parametr – birlashtirilayotgan ma’lumotlar oqimi. Ikkinchi parametr – birlashtirish funksiyasi hisoblanadi. Birlashtirish funksiyasi ikkita oqimning mos keladigan elementlarini parametr sifatida oladi va ularni birlashtirish natijasini qaytaradi. Quyida ikki oqimni birlashtirish uchun dastur keltirib o‘tilgan:

```
import kotlinx.coroutines.flow.*
suspend fun main(){
    val english = listOf("red", "yellow", "blue").asFlow()
    val uzbek = listOf("qizil", "sariq", "ko'k").asFlow()
    english.zip(uzbek) { a, b -> "$a: $b" }
        .collect { word -> println(word) }
}
```

Bu yerda **zip** funksiyasi *english* va *uzbek* nomli oqimlarni takrorlaydi. **zip** funksiyasi ishlatilganida birinchi oqim elementi *a* parametriga, ikkinchi oqim elementi esa *b* parametriga uzatiladi. Birlashtirish funksiyasi ikkala elementni bir satrga birlashtiradi. Har bir ikki element satr ko‘rinishida yangi oqimning elementini tashkil etib, oqim elementi sifatida uzatiladi. Dasturning konsoldagi natijasi quyidagicha bo‘ladi:

```
red: qizil
yellow: sariq
blue: ko'k
```

Yuqorida keltirilgan dasturda bir turga tegishli bo'lgan ikkita oqim birlashtirilib, shu turga tegishli bo'lgan natijani qaytaradi. **zip** funksiyasi turli ko'rinishda bo'lgan ikki oqimni birlashtirib, umuman boshqa turdagi oqim sifatida ham qaytarishi mumkin.

```
import kotlinx.coroutines.flow.*
suspend fun main() {
    val names = listOf("Tohir", "Bobur", "Samiya").asFlow()
    val ages = listOf(37, 41, 25).asFlow()
    names.zip(ages) { name, age -> Person(name, age) }
        .collect { person ->
            println("Nomi: ${person.name} \tYoshi:
${person.age}")
        }
}
data class Person(val name: String, val age: Int)
```

Bu dasturda **String** oqimi va **Int** oqimi **zip** funksiyasi orqali birlashtiriladi va hosil bo'lgan natijani **Person** turidagi obyekt sifatida qaytaradi. Dastur natijasi quyidagicha:

```
Nomi: Tohir   Yoshi: 37
Nomi: Bobur   Yoshi: 41
Nomi: Samiya  Yoshi: 25
```

Nazorat savollari:

1. Asinxron oqim deganda nimani tushunasiz?
2. Oqim bilan ishlovchi funksiyalarni sanab bering.
3. Oqim ma'lumotlarni filtrlovchi funksiyalarni sanab bering.
4. Asinxron oqimlar yaratuvchi funksiyalar va metodlarni tushuntirib bering.
5. Oqimning terminal funksiyalarini sanab bering.
6. Oqimning oraliq funksiyalarini sanab bering.

Mundarija:

1. KIRISH	3
1.1. Kotlin nima?.....	3
1.2. Kotlin kompilyatorini o‘rnatish va sozlash	4
1.3. Kotlin tilida Hello World dasturini yaratish.....	6
1.4. Kotlin dasturlash tilining muhitlari	7
2. Kotlin dasturlash tilining asoslari	16
2.1. O‘zgaruvchilar.....	16
2.2. Ma’lumotlar turlari.....	17
2.3. Arifmetik amallar	20
2.4. Solishtirish belgilari va shartli ifodalar	25
2.5. Tarmoqlanuvchi jarayonlar	27
2.6. Takrorlanuvchi jarayonlar	31
2.7. Ketma–ketliklar.....	35
2.8. Massivlar	35
3. Funksional dasturlash	39
3.1. Funksiyalar va ularning parametrlari	39
3.2. O‘zgaruvchan sonli parametrlar. vararg kaliti.....	42
3.3. Qiymat qaytaruvchi funksiyalar. return operatori	44
3.4. Bir qatorli va mahalliy funksiyalar.....	46
3.5. Funksiyalarni qayta yuklash.....	48
3.6. Yuqori darajadagi funksiyalar	52
3.7. Anonim funksiyalar va ulardan foydalanish	54
3.8. Lambda amallari.....	56
4. Obyektga yo‘naltirilgan dasturlash	62
4.1. Klasslar va obyektlar.....	62
4.2. Xususiyatlar va metodlar.....	63
4.3. Konstruktorlar	65
4.4. Ko‘rinish huquqlari	70
4.5. Qiymat o‘rnatuvchi va qiymat oluvchi metodlar	73
4.6. Paketlar va ularni qo‘shib olish.....	77

5. Interfeyslar va vorislilik	80
5.1. Vorislilik	80
5.2. Xususiyatlarni va metodlarni boshqarish	84
5.3. Mavhum klasslar va metodlar	88
5.4. Interfeyslar	90
5.5. Ichki klasslar va interfeyslar	94
5.6. Data klassi	97
5.7. Enum to‘plami.....	99
5.8. Anonim klasslar va obyektlar.....	101
6. Obyektga yo‘naltirilgan dasturlashning qo‘shimcha xususiyatlari	104
6.1. Istisnolarni boshqarish	104
6.2. Nullable turi va null qiymati	109
6.3. Yo‘naltiruvchi xususiyatlar va delegatlar	113
6.4. Turlarini o‘zgartirish metodlari va operatorlari	116
6.5. Turlarni kengaytirish funksiyalari.....	121
6.6. infix ko‘rinishdagi funksiyalar.....	123
7. Umumiy turlar	125
7.1. Umumiy turlar va funksiyalar	125
7.2. Umumiy turlarda cheklovlar	128
8. To‘plamlar.....	132
8.1. O‘zgaruvchan va o‘zgarmas to‘plamlar	132
8.2. List – to‘plami	134
8.3. Set – to‘plami	136
8.4. Map – to‘plami.....	137
9. Korutinlar.....	140
9.1. Korutinlar bilan tanishish.....	140
9.2. Korutin maydoni	144
9.3. launch funksiyasi.....	146
9.4. async va await operatori. Deferred interfeysi.....	148
9.5. Korutin dispatcher.....	150

9.6. Korutinlar bajarilishi bekor qilish	152
9.7. Channel interfeysi	155
10. Asinxron oqimlar	160
10.1. Asinxron oqimlarga kirish.....	160
10.2. Asinxron oqimlar yaratish.....	162
10.3. Oqim bilan ishlovchi operatorlar va funksiyalar.....	164
10.4. Oqim elementlari soni count, take va drop funksiyalari.	165
10.5. first va single funksiyalari	166
10.6. map va transform funksiyalari.....	168
10.7. Ma'lumotlarni filtrlash	170
10.8. reduce va fold funksiyalari	172
10.9. Oqimlarni birlashtirish	173

X.Sh.MUSAYEV

KOTLIN DASTURLASH TILI

(O‘quv qo‘llanma)