A Kotlin egy multiparadigmanyelv. Statikusan van beírva, ami azt jelenti, hogy sok hibát lehet észlelni fordításkor, nem pedig futásidőben. Egyesíti az ötleteketobjektumorientált és funkcionális nyelvekből, ami segít elegáns kód írásábanés további hatékony absztrakciók felhasználásában. Hatékony módot kínál aszinkron kód írására, ami számos fejlesztési területen fontos. Csak e rövid leírások alapján talán már van egy intuitív elképzelése arról, hogy milyen típusú nyelv a Kotlin. Nézzük meg ezeket a kulcsfontosságú tulajdonságokat részletesebben. Először is nézzük meg, milyen alkalmazásokat készíthet a Kotlinnel.

Az **aszinkron kód** (asynchronous code) egy olyan programozási minta, ahol bizonyos műveleteket (pl. hálózati hívás, fájlművelet) **nem blokkoló módon** végzünk el. Ez azt jelenti, hogy a program **nem áll meg** ezeknél a műveleteknél, hanem közben más dolgokat is tud végezni, amíg az adott művelet be nem fejeződik.

Kotlinban az aszinkron kód általában így néz ki:

Kotlinban az `async`, `await`, `suspend` és a `coroutine` kulcsszavakat használjuk:

Egyszerű példa:

```
import kotlinx.coroutines.*

fun main() = runBlocking {
    launch {
        delay(1000)
        println("1 másodperc után ez lefutott.")
    }
    println("Ez előbb fog megjelenni.")
}
```

Magyarázat:

- `runBlocking` elindít egy blokkot, ahol korutinokat futtathatsz.
- `launch` egy új korutint indít el (párhuzamosan fut).
- `delay(1000)` nem blokkolja a fő szálat – vár 1 másodpercet **aszinkron módon**.

Miért hasznos az aszinkron kód?

- Nem blokkolja a felhasználói felületet (pl. mobil appban a gombok nem fagynak le).
- Hatékonyabb több szálon végzett műveletekhez.
- Jó választás **hálózati kérésekhez**, **adatbázis műveletekhez**, **időzített feladatokhoz**.

Összefoglalva:

> **Az aszinkron kód** Kotlinban azt jelenti, hogy a program nem várakozik passzívan egy hosszabb műveletre, hanem közben más feladatokat is végezhet. A Kotlin ezt **korutinokkal** oldja meg, amik könnyű szálként működnek.
>
> Továbbá

A Kotlin célja meglehetősen széles. A nyelv nem egyetlen problémára összpontosít, és nem foglalkozik egyetlen típusú kihívással, amellyel a szoftverfejlesztők ma szembesülnek. Ehelyett átfogó termelékenységi javulást biztosít a fejlesztési folyamat során felmerülő összes feladathoz, és kiváló szintű integrációt céloz meg az adott tartományokat vagy programozási paradigmákat támogató könyvtárakkal. A Kotlin gyakori használati esetei a következők: Android-eszközökön futó mobilalkalmazások készítéseKiszolgálóoldali kód (általában webalkalmazások háttérprogramjai) létrehozása.

A Kotlin kezdeti célja az volt, hogy tömörebb, termelékenyebb, biztonságosabb alternatívát nyújtson a Java számára, amely minden olyan környezetben alkalmas, ahol a Java használható. Ez magában foglalja a környezetek széles skáláját, a kis peremeszközök futtatásától a legnagyobb adatközpontokig. Mindezekben a felhasználási esetekben a Kotlin tökéletesen illeszkedik, és a fejlesztők kevesebb kóddal és kevesebb bosszúsággal végezhetik munkájukat. De Kotlin más kontextusban is dolgozik. A Kotlin Multiplatform használatával többplatformos alkalmazásokat hozhat létre asztali számítógépre, iOS-re és Androidra – ésmég a Kotlint is futtathatja a böngészőben.

A statikus gépelés teszi a Kotlint teljesítővé, megbízhatóvá és karbantarthatóvá. A statikusan tipizált programozási nyelvek számos előnnyel járnak, mint példáulteljesítmény, megbízhatóság, karbantarthatóság és eszköztámogatás. A statikusan tipizált nyelv legfontosabb előnye, hogy a programban lévő minden kifejezés típusa ismert a fordításkor. A Kotlin egy statikusan tipizált programozási nyelv; A Kotlin fordító ellenőrizheti, hogy az objektumon elérni kívánt metódusok és mezők valóban léteznek-e. Ez segít kiküszöbölni a hibák egész osztályát – ahelyett, hogy futásidőben összeomlana, ha egy mező hiányzik, vagy a függvényhívás visszatérési típusa nem felel meg a vártnak, ezeket a problémákat a fordításkor fogja látni, lehetővé téve a fejlesztési ciklus korábbi szakaszábanjavíthatja ki őket. A statikus gépelés néhány előnye: Teljesítmény – A metódusok hívása gyorsabb, mert nem kell meghatározni, hogy melyik metódust kell futásidőben meghívni. Megbízhatóság – A fordító típusokat használ a program konzisztenciájának ellenőrzésére, így kevesebb az esélye az összeomlásnak futás közben. Karbantarthatóság – Az ismeretlen kóddal való munka egyszerűbb, mert láthatja, hogy a kód milyen típusokkal működik. Eszköztámogatás – A statikus gépelés megbízható refaktorálást, precíz kódkiegészítést és egyéb IDE-funkciókat tesz lehetővé.

Ez ellentétben áll a dinamikusan beírt programozási nyelvekkel, mint például a Python vagy a JavaScript. Ezek a nyelvek lehetővé teszik olyan változók és függvények definiálását, amelyek bármilyen típusú adatot tárolhatnak vagy visszaadhatnak, és futásidőben feloldhatják a metódus- és mezőhivatkozásokat. Ez rövidebb kódot és nagyobb rugalmasságot tesz lehetővé az adatstruktúrák létrehozásában, de hátránya, hogy az olyan problémák, mint a hibásan írt nevek vagy a függvényeknek átadott érvénytelen paraméterek, nem észlelhetők a fordítás során, és futásidejű hibákhoz vezethetnek.

A Kotlin nem követeli meg, hogy minden változó típusát explicit módon megadja a forráskódban. Sok esetben a változó típusa automatikusan meghatározható a környezetből, így kihagyhatja a typedeclaration-t. Íme a lehető legegyszerűbb példa erre:

```
val x: Int = 1
val y = 1   //a forditó automatikusan felismeri
```

A változó típusát explicit módon megadhatja ! . De gyakran nem kell megadnia. Deklarál egy változót, és mivel egész értékkel van inicializálva, a Kotlin automatikusan meghatározza, hogy a típusa Int.

Ha megnézzük a Kotlin típusrendszerének sajátosságait, sok olyan fogalmat találunk, amelyek más objektumorientált programozási nyelvekből ismertek. Az osztályok és interfészek például az elvárásoknak megfelelően működnek. És ha történetesen Java fejlesztő vagy, a tudásod különösen könnyen átvihető a Kotlinba.

A Kotlin nullázható típusok támogatása jele : [`?`] (? elvis operator jelentése = null), amely lehetővé teszi, hogy megbízhatóbb programokat írjon a lehetséges nullmutató kivételeket a fordításkor, ahelyett, hogy futásidőben összeomlás formájában tapasztalná őket.

Multiparadigma programozási nyelvként a Kotlin ötvözi az objektum-orientált megközelítést a funkcionális programozási stílussal. A funkcionális programozás kulcsfogalmai a következők:

Első osztályú függvények – A függvényekkel (viselkedés darabjaival) értékként dolgozik. Tárolhatja őket változókban, paraméterként átadhatja őket, vagy visszaadhatja őket más függvényekből. Megváltoztathatatlanság – Nem módosítható objektumokkal dolgozik, ami garantálja, hogy az állapotuk nem változhat meg a létrehozásuk után. Nincsenek mellékhatások – Tiszta függvényeket ír, olyan függvényeket, amelyek ugyanazt az eredményt adják vissza ugyanazon bemenetek mellett, és nem módosítják más objektumok állapotát, és nem lépnek kapcsolatba a külvilággal.

Milyen előnyökkel járhat, ha funkcionális stílusban ír kódot? Először is, ott van a tömörség előnye. A funkcionális kód elegánsabb és tömörebb lehet imperatív megfelelőjéhez képest: a változók mutálása, (megváltoztatása) valamint a hurkok és a feltételes elágazások helyett a függvényekkel való értékként való munka sokkal nagyobb absztrakciós erőt biztosít. A funkcionális programozási stílus alkalmazásával elkerülheti a kód duplikációját is. Ha hasonló kódtöredékei vannak, amelyek hasonló feladatot valósítanak meg, de néhány kisebb részletben különböznek, akkor könnyen kivonhatja a logika közös részét egy függvénybe, és a különböző részeket argumentumként adhatja át. Ezek az érvek maguk is függvények lehetnek. A Kotlinban ezeket a lambda-kifejezések tömörszintaxisával fejezheti ki. A funkcionális kód második előnye a biztonságos egyidejűség. A többszálú programok egyik legnagyobb hibaforrása ugyanazon adatok módosítása több "szereplőből" (gyakran több szálból) megfelelő szinkronizálás nélkül.

Ha megváltoztathatatlan adatstruktúrákat és tiszta funkciókat használ, biztos lehet benneilyen nem biztonságos módosítások nem fognak megtörténni, és nem kell előállniabonyolult szinkronizálási sémák. Végül a funkcionális programozás könnyebb tesztelést jelent. A mellékhatások nélküli függvények elszigetelten tesztelhetők anélkül, hogy sok beállítási kódra lenne szükség a teljes környezet felépítéséhez. Ha a függvények nem lépnek kapcsolatba a külvilággal, könnyebben érvelhet a kódjával kapcsolatban, és ellenőrizheti annak viselkedését anélkül, hogy állandóan egy nagyobb, összetettebb rendszert kellene a fejében tartania. Általában egy funkcionális programozási stílus sok programozási nyelvvel használható, és sok részét jó programozási stílusként támogatják. De nem minden nyelv biztosítja a könnyed használathoz szükséges szintaktikai és könyvtári támogatást. A Kotlin gazdag funkciókészlettel rendelkezik.

A Kotlin gazdag funkciókészlettel rendelkezik, amelyek a kezdetektől támogatjákfunkcionális programozást. Ezek a következők:

Függvénytípusok – Lehetővé teszi a függvények számára, hogy más függvényeket fogadja argumentumként vagy más függvények visszaadására Lambda-kifejezések – Lehetővé teszi a kódblokkok átadását minimális sablonelem Taghivatkozások – Lehetővé teszi a függvények értékként való használatát, és például argumentumként való átadását Adatosztályok – Tömör szintaxis biztosítása olyan osztályok létrehozásához, amelyek képesek megváltoztathatatlan

adatokat tárolniStandard könyvtári API-k – A szabványos könyvtár gazdag készlete az objektumok és gyűjtemények funkcionális stílusban történő használatához.

Az alábbi kódrészlet az aninput sequence-lel végrehajtandó műveletek láncolatát mutatja be. Egy adott üzenetsorozat birtokában a kód megtalálja az összes egyedi nem üres olvasatlan üzenetek feladóit nevük szerint rendezve:

```
messages.filter { it.body.isNotBlank() && !it.isRead
}.map(Message::sender).distinct().sortedBy(Sender::name )
```

`isNotBlank()` //ellenőrzi hogy van e benni szöveg és szóköz(whitespace)

`isBlank()` // Akkor igaz, ha a String üres, vagy csak whitespace karaktereket tartalmaz (pl. " ", "\n", "\t")

`distinc()` //Ez eltávolítja a listából a **duplikált feladókat**.

`sortedBy(Sender::name)` //sorba rendezi a feladókat(ABC)

A Kotlin szabványos könyvtár olyan funkciókat határoz meg, mint a szűrő, a térkép és a sortedBy az Ön számára. A Kotlin nyelv támogatja a lambda kifejezéseket és a taghivatkozásokat (például Message::sender), így az ezeknek a függvényeknek átadott argumentumok nagyon tömörek. Amikor kódot ír a Kotlinban, kombinálhatja az objektumorientált és a funkcionális megközelítéseket, és használhatja a megoldandó problémához legmegfelelőbb eszközöket; a funkcionális stílusú programozás teljes erejét megkapja a Kotlinban, és amikor szüksége van rá, dolgozhat változtatható adatokkal és írhat mellékhatásokkal rendelkező függvényeket, mindezt anélkül, hogy extra karikákon kellene átugrania. És természetesen az interfészeken és osztályhierarchiákon alapuló keretrendszerekkel való munka ugyanolyan egyszerű, mint amire számítani lehet.

A párhuzamos és aszinkron kód természetessé válik, és korutinokkal strukturálódik Akár kiszolgálón, asztali gépen vagy mobiltelefonon futó alkalmazást készít, az egyidejűség – a kód több darabjának egyidejű futtatása – szinte elkerülhetetlen téma. A felhasználói felületeknek reagálniuk kell, miközben a hosszú ideig futó számítások futnak a háttérben. Az internetes szolgáltatásokkal való interakció során az alkalmazásoknak gyakran egyszerre több kérést kell benyújtaniuk. Hasonlóképpen, a szerveroldali alkalmazások várhatóan továbbra is kiszolgálják a bejövő kéréseket, még akkor is, ha egyetlen kérés a szokásosnál sokkal tovább tart. Mindezeknek az alkalmazásoknak egyidejűleg kell működniük, egyszerre több dologon kell dolgozniuk. Az egyidejűségnek számos megközelítése létezik, beleértve a szálakat, a visszahívásokat, a határidős ügyleteket, az ígéreteket, a reaktív bővítményeket és egyebeket. Kotlin az egyidejű és aszinkron programozás problémáját felfüggeszthető számításokkal, úgynevezett korutinokkal közelíti meg, ahol a kód felfüggesztheti a végrehajtását, és később folytathatja munkáját. Ebben a példában egy függvényt határoz meg, amely három hálózati hívást hajt végre az authenticate , a loadUserData és a loadImage meghívásával:

```
suspend fun processUser(credentials: Credentials) {
 val user = authenticate(credentials)   // hitelesítés a megadott adatok alapján

 val data = loadUserData(user)  // felhasználói adatok betöltése


 val profilePicture = loadImage(data.imageID) // kép betöltése
    // ...
 }

 suspend fun authenticate(c: Credentials): User { /* ... */
 }
suspend fun loadUserData(u: User): Data { /* ... */ }
 suspend fun loadImage(id: Int): Image { /* ... */ }
```

A `suspend` azt jelenti, hogy a függvény **függőben hagyható, vagyis szüneteltethető** (suspend = felfüggesztett).

Ezek a függvények **aszinkron módon működnek**, nem blokkolják a fő szálat, hanem amikor valami lassabb művelet van (pl. hálózat, adatbázis), akkor "elalszanak", majd később folytatódnak.

Ez a Kotlin **korutinok** `(coroutines)` egyik alapja, amivel könnyen írhatsz párhuzamos, nem blokkoló kódot.

A szál, amelyen ez a kód fut (és kiterjesztéssel maga az alkalmazás) azonban nincs blokkolva; amíg az eredményre vár processUser , addig más feladatokat is elvégezhet, például válaszolhat a felhasználói bevitelekre.

A következő példában két képet töltünk be egyidejűleg az `async` paranccsal , majd megvárjuk, amíg a betöltés befejeződik az `await`segítségével, visszaadva a képek kombinációját (pl. az egyiket a másikra) az eredményként:

```
 suspend fun loadAndOverlay(first: String, second: String):
 Image =
    coroutineScope {
val firstDeferred = async { loadImage(first) }
 val secondDeferred = async { loadImage(second) }
 combineImages(firstDeferred.await(), secondDeferre
 d.await())
    }
```

Megkezdi az első kép betöltését egy új korutinban. Megkezdi a második kép betöltését egy másik korutinban. Amikor mindkét kép betöltődik, átfedi őket, és visszaadja az eredményül kapott képet.

```
async { ... }   Két kép párhuzamos betöltése
await() Megvárja, hogy kész legyen a kép
combineImages(...)  Összerakja a két képet egy képpé
```

A Kotlin a Java kóddal interoperabilitásba van tehát hívható egy java alkalmazásból Kotlin kód és Kotlin alkalmazásból is hívható Java.a két nyelv interpolaritása zökkenőmentes.

A modern keretrendszerek, mint például a Spring (https://spring.io/), első osztályú támogatást nyújtanak a Kotlin számára.

példa:

```
 @SpringBootApplication
class DemoApplication
 fun main(args: Array<String>) {
    runApplication<DemoApplication>(*args)
 }
 @RestController
 class GreetingResource {
    @GetMapping
    fun index(): List<Greeting> = listOf(
        Greeting(1, "Hello!"),
        Greeting(2, "Bonjour!"),
        Greeting(3, "Guten Tag!"),
    )
 }
 data class Greeting(val id: Int, val text: String)
```

A Kotlin saját könyvtárainak folyamatosan növekvő ökoszisztémáját is élvezi, beleértve a szerveroldali keretrendszereket is. Például a Ktor (https://ktor.io/) a JetBrains által épített összekapcsolt alkalmazások keretrendszere a Kotlin számára, amely szerveroldali alkalmazások létrehozására és hálózati kérések készítésére használható kliens- és mobilalkalmazásokban.