Kotlin alapú szoftverfejlesztés vizsgálata egy peer-to-peer fájlmegosztó alkalmazás elkészítésével

Pásztor Dániel
Automatizálási és Alkalmazott Informatikai Tanszék
Budapesti Műszaki és Gazdaságtudományi Egyetem
danim1130@gmail.com
Konzulens: Ekler Péter

Abstract. A Java nyelv első verzióját 1995-ben adták ki azzal a "Write once, run anywhere" szlogennel, melyet egy absztrakt számítógép specifikálásával, a Java Virtual Machine (JVM) segítségével érték el. Az azóta eltelt 22 év alatt 9 újabb verziót adtak ki, melyekkel sikerült egy még mai szempontok szerint is modern nyelvet létrehozni úgy, hogy végig kompatibilis maradt a régebben írt kóddal is. Ennek köszönhetően hatalmas sikere, ma már az egyik legkeresettebb programozási nyelvnek minősül.

A JetBrains nevű cseh cég első terméke, az IntelliJ is egy Java fejlesztő környezetként kezdte életét, mely az elsők között biztosított fejlettebb funkciókat a programozók számára. A folyamatos fejlesztéseknek, illetve a nyílt forráskódúság támogatásának köszönhető a jó hírnevük, mára már a legismertebb cégek közé tartozik a fejlesztők körében, több mint 600 dolgozóval és 21 termékkel. Termékük szoros kapcsolatban állva az Android operációs rendszerrel is.

Mivel a szoftverük nagy részét ők is Java nyelven írták, rengeteg problémába, illetve hiányosságba futottak bele. Ennek megoldására 2011-ben jelentették be, majd 2012-ben adták ki a saját JVM alapú nyelvüket, a Kotlint, mely egy mindenki számára ingyenesen használható nyílt forráskódú projekt lett. Azóta is aktív fejlesztés alatt áll, számos újabb funkcióval kiegészítve, mely még tovább növelte az emberek bizalmát a nyelvben.

Teljesen kompatibilis a Javaban fejlesztett kóddal, a kettő között egyszerű a kommunikáció, ez ideálissá tette az Android alapú fejlesztésre. Bár a Google által fejlesztett operációs rendszerre általában Java alapon fejlesztettek, az nem a Java által specifikált JVM-re fordult, ezért nem volt garantálva, hogy egy későbbi módosítás nem lehetetleníti-e el a Kotlinban írt alkalmazásokat. Az ilyen félelmek miatt a Google 2017-ben bejelentette, hogy a Kotlin az Android hivatalosan is támogatott nyelvévé válik, ezzel is bíztatva az ilyen irányú törekvéseket.

Dolgozatomban a Kotlin nyelv alapú fejlesztést fogom összehasonlítani más modern nyelvekkel, különös figyelmet fordítva a Java-ra. Ezt egy általam megírt nyílt forráskódú peer-to-peer fájlcserélő (BitTorrent) könyvtár, illetve alkalmazás segítségével teszem. Megvizsgálom, milyen plusz funkciókkal rendelkezik a nyelv, ezeknek előnyeit, illetve hátrányait. Különböző metrikák alapján vizsgálom a Javaval szembeni teljesítményét Android, illetve számítógépes környezetben.

Tartalomjegyzék

1. Bevezetés

A dolgozatom aktualitását a Kotlin nyelv adja, mely mint egy relatív új programozási nyelv, felvet bizonyos kérdéseket a hatékonyságáról, illetve a rendelkezésre álló eszközök minőségéről.

Ahhoz, hogy alaposabban meg tudjam ismerni a Kotlin nyelv felépítését, szintaktikáját, úgy döntöttem, hogy egy BitTorrent könyvtárat fogok implementálni rajta, melyet később nyílt forráskódúvá teszek. Mivel Kotlin nyelven írt és karbantartott BiTorrent könyvtárat nem találtam, ez az implementáció képes lenne egy hiány kitöltésére is a már meglévő könyvtárak mellett.

A BitTorrent könyvtár írása emellett jó alapot fog szolgálni a különböző metrikák mérésére, mint például fordítási idő, vagy kód méret. Annak érdekében, hogy össze lehessen hasonlítani, ugyanezt a BitTorrent könyvtárat implementálom Java nyelven is.

A Kotlin egyik fő felhasználási területe az Android alapú szoftverfejlesztés, miután a Google 2017 tavaszán bejelentette, hogy a Kotlin bekerül a hivatalosan támogatott Android fejlesztői nyelvek közé. Emiatt célszerűnek látom bizonyos androidos mérések megvalósítását is.

A dolgozatom további része az alábbi struktúrát követi:

- A 2. fejezet bemutatja a Kotlin, a Java, illetve a C# programozási nyelvek rövid történetét, illetve ismertetem a Kotlinnal kapcsolatos eddigi eredményeket.
- A 3. fejezet példákon keresztül mutatja be a Kotlin nyelv szintaktikáját, kitérve a torrentezéshez elengedhetetlen hálózatkezelési problémára is.
- A 4. fejezet a konkrét BitTorrent könyvtár implementációját írja le mind a Kotlin, mind a Java programozási nyelv esetében.
- A 5. fejezet ismerteti a két projekttel kapcsolatos tapasztalatokat, illetve a rajtuk végzett méréseken keresztül az eredményeket.
- A 6. fejezet kitekintést ad a dolgozat témáján túlra, hogy milyen területei vannak még a Kotlin nyelvnek.
- A 7. fejezet lezárja a dolgozatot, összefoglalja a tanulságokat.

2. Irodalomkutatás, felhasznált technológiák

Ebben a fejezetben bemutatom a dolgozatomban említett modern programozási nyelvek, illetve a BitTorrent történetét. Ismertetem a dolgozatomban felhasznált technológiákkal kapcsolatos eredményeket.

2.0.1. Java

A Java nyelv eredete az 1990-es évek elejéhez köthető. Ekkor a Sun Microsystem amerikai vállalatnál a jövőt a hálózati infrastruktúra terjedésében, illetve a különböző elektromos eszközök fejlődésében látták [1].

Ennek kiaknázására hozták létre a James Gosling által vezetett "Green Team" csapatot, melynek célja az új technológiák kutatása volt. Eredetileg a C++ nyelvet tervezték használni, de ezt végül a következő indokok miatt elvetették:

- a komplexitásából származó hibák,
- a memória kezelését a C++ a fejlesztőkre hagyta, mely már közepes projektek esetén is nehéz feladat volt, rengeteg hibalehetőséggel, melyeket sokszor még tesztelés közben se tudtak előidézni,
- olyan egységesített funkciók hiánya, mint a párhuzamos programozáshoz szükséges szálak
- nehézkes multi-platform fordítás

Eredetileg Gosling a C++ nyelvet akarta módosítani, illetve kiegészíteni, de végül inkább egy új nyelv mellett döntött, melyet az iroda előtti tölgyfa miatt *Oak*-nak nevezett el. A nyelv első alkalmazása a *Star7* nevű PDA volt [2], mely a korát megelőzve érintőképernyőjével rengeteg funkciót bizotsított, de nem váltotta be a hozzá füzőtt gazdasági reményeket. Ezután még próbálkoztak a kábel-televízió szolgáltatoknál, de az érdeklődés hiánya miatt ezen a piacon is megbuktak.

1994-ben egy három napos brainstorming után úgy döntöttek, hogy megcélozzák az internetet. Egy évvel rá, 1995-ben kiadták a saját böngészőjüket (*HotJava*) mely képes volt az interneten található Java programokat (úgynevezett *appleteket* futtatni. Ezzel egyidőben jelent meg publikusan is a nyelvük, melyeket jogvédelmi okokból *Java*-ra változtattak, illetve az időben legelterjedtebb böngésző, a Netscape is bejelentette a Java appletek támogatását.

A megjelenése óta már 9 nagyobb frissítést is kiadtak a Java-hoz, mellyel törekedtek arra, hogy a nyelv minél modernebb legyen, könnyítsenek a használatán. Emellett viszont tartották a verziók közötti kompatibilitást is (a régi Java kódok ugyanúgy lefordulnak a modern fordítókon is), ez pedig sokszor kompromisszumos megoldásokat eredményezett.

A Java újítása közé tartozott a *Java Virtual Machine* (JVM). A JVM egy olyan szigorúan specifikált absztrakt számítógép [3], mely képes a Java kódból a Java fordítóval készített program (*byte-kód*) futtatására. Ez a virtuális gép

általában egy önálló programként fut az adott platformon, de léteznek már olyan mikrokontrollerek is, melyek képesek közvetlen a Java utasításokat értelmezni és végrehajtani.. Ez a virtuális gép biztosítja azt, hogy a Java-ban megírt program minden olyan platformon képes futni, ahol a JVM implementálva van.

A megjelenésekor sokan kifogásolták a JVM teljesítményét, mivel ennek futtatása általában a valós alkalmazástól vette el a teljesítményt. A számítástechnika fejlődésével, illetve a JVM további fejlesztéseivel viszont a mai napra sikerült elérni, hogy csak különleges alkalmazási területeken legyen érezhető a teljesítménykülönbség.

A JVM elterjedése remek környezetet biztosított az új programozási nyelvek fejlődésének is. Ahelyett, hogy minden platformra külön kelljen fordítót tervezni, implementálni és fenntartani, elég egy JVM fordítót karbantartani, mely így az összes JVM által támogatott platformon futtatni tudja a kódot. Ilyen nyelv például a Scala, a Groovy, vagy a dolgozatom témája is, a Kotlin.

2.0.2. C#

A Java mellett hasonlóan elterjedt programozási nyelv a Microsoft által fejlesztett C#, melynek első verzióját 2000-ben adták ki. Létrejötte annak köszönhető, hogy a Microsoft a Java implementálása közben saját operációsrendszer-specifikus függvényekkel és szolgáltatáskkal bővíttete ki az SDK-t, mely szembe ment a Java által kitűzött multiplatform támogatással. A Sun végül beperelte a Microsoftot, melyet megnyertek, így a Microsoft köteles volt eltávolítani a saját Java verziójukat a rendszerükből [4]. Ennek eredményeképpen saját maguk fejlesztették ki a keretrendszerüket, a *.NET Framework*-öt, mely elsősorban a Windows rendszereken fut, és ehhez adták ki programnyelvként a C# első verzióját.

A C# a kiadásakor sok közös vonással rendelkezett a Java-val. Ide tartozik például az objektum-orientáltság, az automatikus memória-kezelés, illetve a hasonló C-szerű szintaxis.

Pont emiatt érdemes megfigyelni, hogy két nyelv, melyek hasonló tulajdonságokkal rendelkeztek megalkotásukkor, hogyan fejlődtek az idő folyamán, hogyan oldottak meg bizonyos feladatokat, problémákat, mint például a generikus függvények, illetve tárolók kezelése.

2.0.3. Kotlin

A Kotlin nyelvet a cseh JetBrains nevű cég alkotta meg, mely régóta kapcsolatban áll a Java-val. 2001-ben adták ki első terméküket, az IntelliJ nevű fejlesztőkörnyezetüket, mely az elsők között rendelkezett fejlett Java kódgenerálás és módosítás funkciókkal. Az azóta eltelt 16 év alatt sikerült egy világszerte ismert vállalattá kinőni magukat a rengeteg nyelvet támogató fejlesztési és egyéb segédeszközeikkel.

A fejlesztések során sokszor futottak bele a Java nyelv hiányosságaiba, gyengébb pontjaiba, ezért elkezdtek egy új nyelvet keresni, mely képes lenne ezeket a hiányosságokat kiküszöbölni [5]. A kitűzött feltételeknek egyedül a *Scala* felelt

meg, ezt viszont a fordítási ideje miatt elvetették. Miután nem találtak megfelelő nyelvet, úgy döntöttek, hogy belekezdenek egy új fejlesztésébe, mely tanulva a modern nyelvek hibáiból, megpróbálja azokat kiküszöbölni. A kitűzött célok közé tartoznak a következőek:

- JVM alapú nyelv legyen, a fordítási idő egyezzen meg a Java kód fordításával.
- Teljes kompatibilitás Java nyelven írt kóddal, egy projektben akár mindkét nyelv szerepelhessen.
- Rugalmas, könnyen kiegészíthető nyelv legyen.
- Általános felhasználású, akár az iparban is használható legyen.

Bár az első véglegesített verzió 2016-ban jelent meg, a JetBrains már 2012-ben nyílt forráskódúvá tette az akkor még erősen fejlesztés alatt álló nyelvet. Maga a nyelv, illetve a hozzá tartozó könyvtárak mind ingyenesek, azzal a nem rejtett céllal, hogy a JetBrains által fejlesztett IntelliJ fejlesztőkörnyezet eladási számait növeljék. A nyelvnek már fejlesztése alatt sok rajongója lett, a teljes Java kompatibilitásnak köszönhetően az Android platformokra fejlesztők körében is elterjedt.

2017-ben a Google is elismerte mint hivatalosan támogatott Android fejlesztési nyelv, ami által még többen ismerték meg.

Bár eredetileg a 6. verziójú Java bytekódra fordult, a nyelv fejlesztése során bővítették a célplatformok számát, így ma már generálható 8. verziójú Java kód (mely optimalizálva van a 6. verzióhoz képest), emellett képes JavaScriptre, illetve kísérleti módban natív futtatható alkalmazásra is fordítani a Kotlin fordító.

A fejlesztések óta már rengeteg fejlesztő próbálta ki, így sok különböző szempont alapján tudták tesztelni. Egy weboldalas poszt azt találta, hogy a Kotlin fordító a leggyakoribb esetekben gyorsabb, mint a hasonló kódon futó Java fordító [6]

Született egy sorozat különböző Kotlinról fordított kódok teljesítményének összemérésére a Java megfelelőjükkel. Az itt mért adatok alapján a Kotlin legtöbb funkciója semmilyen jelentős teljesítménybeli csökkenést nem okozott [7].

2.1. BitTorrent protokoll

A BitTorrent protokollt 2001-ben adták ki az azt implementáló alkalmazással, forráskóddal és dokumentációval együtt, mellyel bárki elkészíthette a saját torrent alkalmazását, bármilyen platformra. A legfontosabb célkitűzései közé tartozott a minél gyorsabb, hatékonyabb és skálázhatóbb rendszer létrehozása. Ezeket a már akkor is létező technológiák ötvözéseiből sikerült elérni.

Azt a technikát már régebbi fájlmegosztók is ismerték, hogy ha egy nagy fájlt több kisebb csomagra bontják, és úgy töltik fel, sokkal hatékonyabb lesz a megosztás, ugyanis ilyenkor hiba vagy kapcsolatszakadás esetén elég csak a hibás

részt újra tölteni, nem kellett az egész fájlt elölről kezdeni. Ennek analógiájára a BitTorrent protokollban is a megosztásra váró fájlokat több részre (piece) bontjuk, melyeket megérkezésükkor egyben le is tudunk ellenőrizni.

A protokoll alapját a .torrent fájl képezi. Ez tekinthető egy leíró-fájlnak: tartalmazza többek között a megosztásra kerülő fájlok neveit, az egyes részek ellenőrző hash (lsd. később) kódját, magát a torrentet azonosító hash kódot, illetve a trackerek listáját. Ezt a fájlt a felhasználónak kell valahogy megszereznie, álltalában különböző torrent-weboldalakról.

A letöltés hatékonyságát a kölcsönös megosztás (tit-for-tat) elve teszi ideálissá. Egy felhasználó szívesebben osztja meg olyannal a fájl egy részét (piece-t), aki számára is küldött már adatot. Ezzel a technikával könnyű kiszűrni a kártékony letöltőket, akik nem akarnak visszatölteni a hálózatba, vagy szándékosan lassítva, akár hibás adatot osztanak meg.

A fájlok sok részre való osztása jelentősen felgyorsítja a fájlmegosztást is. Egy új torrent esetén egy résztvevő már abban a pillanatban megosztóvá tud válni, amint sikerül letöltenie az első piece-t. A BitTorrent hatékonyságát legjobban a rengeteg (többek között legális) felhasználása bizonyítja. Talán az egyik legjobb példa a világ legnagyobb MMORPG, a World of Warcraft javításainak (patch) letöltése. Egy ilyen patch a kisebb (heti) 100Mb-tól akár a 10Gb-os nagyságrendig is terjedhet, ezt pedig a több milliós aktív felhasználóbázishoz kell minél gyorsabban eljuttatni, melyet egy saját alkalmazáson keresztül tesznek meg, a BitTorrent protokoll segítségével.

3. Kotlin nyelv bemutatása

Ebben a fejezetben röviden összefoglalom a Kotlin nyelv felépítését, szintaktikáját, illetve pár különlegességét.

3.1. Alapok

Változók

Az egyik alapvető kérdés minden nyelv esetén a változókkal kapcsolatos tulajdonságok. Mint ahogyan a Java vagy a C#, a Kotlin is erősen típusos nyelv, vagyis a változók típusát egyértelműen meg kell adni annak létrehozásakor, ez pedig később nem módosítható.

Az említett nyelvek típusrendszerén viszont még erősebb garanciát is vállal: bevezeti a *Nullable* változótípust is, mellyel külön jelölni kell, ha egy változó felveheti a *null* értéket.

1. Listing. Nullable változó

```
var string : String
var nullableString : String?

string = "Test1"
nullableString = null
//string = null : error: null can not be value of a
    non-null type String
//string = nullableString : error: Type mismatch.
```

Nullable típusú változón nem lehet olyan függvényeket meghívni, melyek csak a *Non-nullable* típusú változón értelmezettek, ezzel elkerülve a legtöbb *NullPointerException* hibát. Azért, hogy ez ne jelentsen gondot a fejlesztőnek, a fordító képes tanulni a program kontextusából: ha egy *Nullable* típusú változó bizonyíthatóan nem tartalmazza a *null* értéket, akkor úgy viselkedik, mint a *Non-nullable* változata. Abban az esetben, ha a fejlesztő felül akarja írni ezt a viselkedést (mert például felismer egy olyan helyzetet, amikor a változó nem lehet *null*), a *non-null asserted* (!!.) operátorral írható felül.

Bizonyos helyzetekben hasznos még a *null-safe* (?.) operátor mely csak akkor hívja meg a jobb oldalon lévő függvényt, ha a bal oldalán lévő kifejezés nem *null*, különben *null* értékkel tér vissza.

2. Listing. Nullable változó kontextus.

```
var nullableString : String? = null
var string : String = "Test"

//nullableString.length : Error
nullableString!!.length //Okay, NullPointerException at
    runtime
var length = nullableString?.length //Okay, length = null
```

```
if (nullableString != null){
 println(nullableString.length) //Okay, smart cast to
     String
}

nullableString = string
println(nullableString.length) //Okay, smart cast to
    String
```

A Kotlin a C#-hoz hasonlóan támogatja a *type-interference* (típus származtatás) funkcióját, vagyis egy értékadással egybekötött deklarációnál nem kötelező kiírni a változó típusát:

3. Listing. Type-interference.

```
var string = "Test" //Okay, type is String
var number = 4 //Okay, type is Int
//var noType
//error: This variable must either have a type
    annotation or be initialized
```

A legtöbb nyelv támogatja valamilyen szinten a konstans/olvasható változók létrehozását (Java: final, C#: const). Ezek olyan módosítók, melyeket a változó létrehozásakor kell még hozzáírni. A Kotlin jobban megkülönbözteti ezeket a változókat: konstans értéket a *val*, míg változót a *var* utasítás deklarálja:

4. Listing. Konstans változó.

```
var string = "Test"
string = "AnotherTest"
val number = 4
//number = 4 : error: Val cannot be reassigned

val constString : String;
constString = "Hello" //Late-initialization allowed
```

A Kotlin támogatja a *String interpolation* műveletet, mellyel különböző változók, illetve kifejezések értékeit használhatjuk fel egy String típusú változó létrehozása közben. Ezt a *$<változó név>*, illetve *$kifejezés* struktúrával tehetjük meg.

5. Listing. String interpolation.

```
var string = "Hello World!"
println("$string hossza: ${string.length}")
```

Függvények

A Kotlinban az említett nyelvekkel ellentétben engedélyezett önmagában álló, nem osztályhoz rendelt függvény definiálása. Ilyenkor ez az adott package-hez tartozik, ugyanúgy adható láthatósági szint (*private, protected, public*). Egy függvény a következőképp néz ki:

6. Listing. Egyszerű függvény.

```
fun double(x: Int): Int {
    return 2*x
}
```

A Kotlin engedi a függvényekben az alapértelmezett értékek beállítását, illetve a függvény hívásakor a paraméterek nevesítését:

7. Listing. Alapértelmezett érték, paraméter nevesítés.

```
fun add(x: Int, y: Int = 1, z: Int = 0): Int {
    return x + y + z
}

add(1, z=3)
```

Ha a függvényem csak egy utasításból áll, akkor akár az egyenlőségjel operátort is használhatom, mely a visszatérési értéket is meghatározza:

8. Listing. Egy utasításos függvény.

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

fun addOne(x: Int) = add(x, 1)
```

Mivel a Kotlin egyik fő célja volt a könnyű kiegészíthetőség, engedélyezte az adott osztályok függvényekkel való kiegészítését. Ennek működése teljesen megegyezik a C#-ban található kiegészítő funkciókkal, valójában nem az osztályon módosítunk, a fordító fogja tudni, hogy ha egy metódus nem található egy osztályban, akkor egy kiegészítő metódust keressen. Ennek szintaktikája T típusú osztály esetén:

9. Listing. Osztály kiegészítés.

```
fun T.add(x: Int) : Int{
//...
}
```

Osztályok

A Kotlin nyelvben is természetesen megtalálhatóak az osztályok, melyek az objektum-orientált programozás alapvető elemei, meghatározzák az egyes objektumok viselkedését.

Ennek érdekében a Kotlin fejlesztői alaposan átgondolták, milyen hiányosságok, hibák találhatóak elsősorban a Java nyelv osztályai között, és ezekből tanulva egy modernebb nyelvi megoldást kerestek. Ehhez bevezettek újabb koncepciókat, melyeket ebben az alfejezetben mutatok be.

Láthatóság A Kotlin nyelvben minden változónak, illetve függvénynek a láthatósága alapértelmezetten *public*. A lehetséges láthatósági típusok megegyeznek a C#-ban található lehetőségekkel:

- *public*, mindenhonnan látható a deklaráció,
- *protected*, csak egy osztályon belül érvényes; ekkor a deklaráció csak az osztályban és a leszármazottjaiban látható,
- *private*, csak az adott osztályban, illetve (osztály nélküli függvények deklarációjánál) fájlban látható,
- *internal*, csak az adott modulon belül látható

A modul definíciója függ a fejlesztő környezetétől is, alapvetően az IntelliJ modul komponensének feleltethető meg. Látható, hogy bár a Kotlin ugyanúgy használja a fájlok *package*-be való rendezését, nincsen a Java-hoz hasonló alapértelmezett *package* láthatósági osztály.

Fontos megjegyezni, hogy a osztályokban definiált publikus mezőknek automatikusan generálódnak *get()*, illetve *set()* függvények a Java-val való kompatibilitás miatt.

Leszármazás, felüldefiniálás Természetesen a Kotlin nyelvben is le lehet származni osztályokból, illetve egyes függvényeket felül lehet írni. A Java nyelvvel teljesen ellentétesen viszont itt alapértelmezetten minden osztály, illetve metódus *final*, vagyis nem lehet se leszármazni, se felülírni. Ezek engedélyezésére van az *open* módosítószó.

Elsődleges, másodlagos konstruktor Mint a legtöbb programozási nyelvben, az osztályok létrehozásakor a Kotlin nyelvben is konstruktorok biztosítják az objektum inicializálását. A Scala nyelvhez hasonlóan viszont itt is megkülönböztethető az elsődleges, illetve másodlagos konstruktor.

Elsődleges konstruktornak tekintjük azt a konstruktort, mely szoros kapcsolatban áll az osztály egy példányával, annak változóival. Az elsődleges konstruktor paraméterei szabadon felhasználhatóak a változók inicializálásánál, sőt, akár a konstruktor paraméterének deklarálásánál megadhatjuk, hogy az adott paraméterből hozzon létre egy mezőt is.

10. Listing. Elsődleges konstruktor.

```
class Person constructor(firstName: String, lastName:
    String,
    val gender: String){
```

```
 val fullName = "$firstName $lastName"
}

fun main(args: ArrayList<String>){
 val p = Person("Teszt", "Elek", "male")
    println(p.fullName)
    println(p.gender)
}
```

Az elsődleges konstruktor deklarálásánál a *constructor* utasítás elhagyható, ha a láthatóságán nem akarunk változtatni. Elsődleges konstruktorból legfeljebb egy deklarálható.

A másodlagos konstruktort már az osztályon belül kell hasonlóan a *constructor* kulcsszóval deklarálni. Ennek már nincs kitüntetett szerepe, funkcionalitásában megegyezik a Java, illetve C# konstruktoraival.

Adat osztály A szoftverfejlesztés során sokszor szükségünk van olyan osztályokra, melyek fő feladata a különböző forrásból származó információk logikus csoportosítása, tárolása. Nézznünk erre egy egyszerű példát:

Szeretnénk egy *Person* osztályban tárolni egy ember vezeték-, és keresztnevét. Szükséges, hogy az osztály példányait felhasználhassuk a *Map* osztály kulcsaként, illetve helyesen működjön a *Set* tárolóosztállyal. Legyen az osztály *immutábilis*, vagyis a létrehozása után ne tudjunk változtatni az értékein.

A specifikációnak megfelelő osztály így néz ki Java nyelven:

11. Listing. Person osztály Java nyelven.

```
public class Person {

    private String firstName;
    private String lastName;

    public Person(String firstName, String lastName) {
        this.firstName = firstName;
        this.lastName = lastName;
    }

    public String getFirstName() {
        return firstName;
    }

    public String getLastName() {
        return lastName;
    }

    @Override
    public boolean equals(Object o) {
```

```
        if (this == o) return true;
        if (o == null || getClass() != o.getClass())
            return false;

        Person person = (Person) o;

        if (firstName != null ?
            !firstName.equals(person.firstName) :
            person.firstName != null) return false;
        return lastName != null ?
            lastName.equals(person.lastName) :
            person.lastName == null;
    }

    @Override
    public int hashCode() {
        int result = firstName != null ?
            firstName.hashCode() : 0;
        result = 31 * result + (lastName != null ?
            lastName.hashCode() : 0);
        return result;
    }
}
```

Látható, hogy a változók deklarálásához képest nagyságrendekkel több kód szükséges ahhoz, hogy az objektumaink helyesen viselkedjenek a különböző tároló osztályokkal, ehhez ugyanis a *hashCode()*, illetve az *equals()* metódust is felül kell írni. Ezeket a fordító képes legenerálni, viszont egy paraméter változtatása esetén figyelni kell arra, hogy ezeket a kódokat is újragenerájuk, különben nehezen észrevehető hibák keletkezhetnek.

Kotlinban ugyanezt a feladatot a következő kód oldja meg:

12. Listing. Person osztály Kotlin nyelven.

```
data class Person (val firstName: String, val lastName:
    String)
```

A Kotlin megkönnyítette a feladatunkat azzal, hogy egy osztály deklarálásakor megadhatjuk, hogy az Adat osztály legyen. Ekkor az elsődleges konstruktorban deklarált változók felhasználásával automatikusan előállítja a következő függvényeket:

- *toString()*, a könnyebb kiíratás érdekében (leginkább tesztelés közben tud hasznos lenni).
- *equals()*/*hashCode()*, a Java specifikációban meghatározott szabályt betartva deklarálja az említett metódusokat, így használhatóak lesznek a különböző kollekciókkal.

- *copy()*, egy másoló függvény, mellyel egy olyan másolatot készíthetünk, mely csak a paraméterként megadott változók értékeiben tér el, a többit meghagyja.
- *componentN()*, a *destructuring* operáció támogatására, ezzel a dolgozatomban nem foglalkozom.

Mivel ezeket a függvényeket a fordító generálja, nem kell attól tartani, hogy egy későbbi módosítás miatt a függvényeink inkonzisztens állapotba kerülnek.

Zárt osztály A Java *enum* típusú változói eredetileg nem voltak részei a nyelvnek, csak később, az 5-ös frissítéssel kerültek bele. A C, illetve C++ nyelvekben található *enum*-hoz képest drasztikusan más megoldást választottak. Míg az előbbi nyelvek a felsorolt elemeket egy egész számmal helyettesítették, a Java fordító minden egyes elemhez egy új osztályt generált, melyeket futási időben példányosított, minden elemet egyszer. Tehát a Java megoldásban minden *enum* elem egy objektum volt, mely így akár mezőket és metódusokat is tartalmazhatott.

A zárt (*sealed*) osztályok ezt az ötletet általánosítják. Egy zárt osztály automatikusan *abstract* is, viszont minden leszármazott osztályt az adott fájlban kell deklarálni. Ennek egyik előnye, hogy így a fordító le tudja ellenőrizni, ha például egy (a legtöbb nyelvben *switch* szerkezetnek nevezett) *when* összes lehetséges ágát lefedtük egy értékadásnál.

Ennek felhasználására látható egy jó példa, melyet a Kotlin ismertető oldaláról emeltem át [8]:

13. Listing. Zárt osztály.

```
sealed class Expr
data class Const(val number: Double) : Expr()
data class Sum(val e1: Expr, val e2: Expr) : Expr()
object NotANumber : Expr()

fun eval(expr: Expr): Double = when(expr) {
    is Const -> expr.number
    is Sum -> eval(expr.e1) + eval(expr.e2)
    NotANumber -> Double.NaN
    // the ‘else‘ clause is not required because we’ve
        covered all the cases
}
```

3.2. Lambda paraméterek, függvények

A Kotlin egyik legerősebb pontja a *lambda* függvények. Ezek gyakorlatilag olyan függvény konstrukciók, melyeket a hagyományos változókhoz hasonlóan tudok kezelni, vagy akár paraméterként egy másik függvénynek átadni. A lambda változó deklarálásánál meg kell adni a paraméterek, illetve a visszatérési érték

típusát. Ezek a hagyományos változókhoz hasonlóan elhagyhatóak, ha a fordító képes levezetni a változók típusát. Fontos megjegyezni, hogy a lambda függvény visszatérési értékét nem a *return* kulcsszó adja meg, hanem az utolsó kifejezés visszatérési értéke.

A deklarálásnál mindig előbb a paramétereket adjuk meg, majd ezt követi egy nyíl, melyet típus deklarálásánál a visszatérési érték típusa követi, értékadásnál pedig a konkrét utasítás.

14. Listing. Lambda függvények.

```
//Lambda parameter
fun findInList(list: List<T>, predicate: (T) ->
    Boolean): T?{
 ...
}

//Lambda declaration
var sum: ((Int),(Int) -> Int)? = null
var mul = {x: Int, y: Int -> x * y}

var pred = {x: Int -> (x % 2) == 0}
findInList(list, pred)
```

A lambda függvény paraméterek átadásának szintaktikája bizonyos körülmények között egyszerűsödhet. Ezt példák segítségével fogom bemutatni. Határozzuk meg egy egész számokat tartalmazó lista páratlan elemeinek összegét. Ehhez először kiszűrjük a páratlan elemeket a listából, majd felhasználjuk a beépített összegző függvényt az így kapott listán:

15. Listing. Lambda paraméter 1.

```
var sum = list.filter({elem -> (elem % 2 == 0)}).sum()
```

Egy bemeneti paramétert tartalmazó lambda függvények esetén engedélyezet a paraméter nevesítésének elhagyása, ekkor azt az *it* kulcsszó segítségével érhetjük el:

16. Listing. Lambda paraméter 2.

```
var sum = list.filter({it % 2 == 0}).sum()
```

Abban az esetben, ha egy függvény utolsó paramétere egy lambda függvény, az kihozható közvetlen a függvény-paramétereket tartalmazó zárójel után. Ha ez a függvény nem tartalmaz több paramétert, a zárójelek elhagyhatóak, így megkapjuk a következő kódot:

17. Listing. Lambda paraméter 3.

```
var sum = list.filter{it % 2 == 0}.sum()
```

A Kotlin ezzel a szintaktikával, illetve a kiegészítő függvénnyekkel érte el, hogy a nyelv minél rugalmasabb, kiegészíthetőbb legyen. Jó példa erre a *synchronized* kulcsszó. A Java platformon elengedhetetlen kellék, ha a programunkat több szálon szeretnénk futtatni. Szorosan kapcsolódik az objektumokhoz tartozó monitorokhoz, mellyel garantálni tudjuk a szálak kölcsönös kizárását. Ez felhasználható metódusok deklarálásánál (melynék akkor az adott osztály példányának monitorját zárólja), vagy utasításként, mellyel a paraméterben megadott objektum monitorját zárolja, és hajtja végre a megadott kódot.

A Kotlinban nincsnen ilyen kulcsszó deklarálva, helyette viszont létezik egy ugyanilyen nevű függvény, melynek deklarációja a következő:

18. Listing. Synchronized függvény.

```
public fun <R> synchronized(lock: Any, block: () -> R): R
```

Ezzel pedig könnyedén elérhető ugyanaz a funkcionalitás, mint a Java megfelelőjében:

19. Listing. Kotlin synchronize.

```
class Test{
 fun hello() = synchronized(this) { //synchronized
     member function
      println("Hello World!")
     }
}

val list = listOf(1,2,3)
fun test(){
 synchronized(list){
      ...
     }
}
```

Ennek a megoldásnak megvan az az előnye, hogy így tetszés szerint felülírhatom a *synchronize* függvényt egy általam írt függvénnyel, mely hasznos lehet például tesztelés közben különböző információk kinyerésére.

Egy másik típus *vevővel rendelkező lamda* függvény. Ez lehetővé teszi, hogy a lambda függvényünket úgy hívjuk meg, mintha az adott objektum egy metódusa lenne, melyben a *this* változó az adott objektumra fog mutatni.

Egyik felhasználása lehet például a listák elemekkel való feltöltése a konstruálásuk idején. A Java erre a *"Double brace initialization"* néven elhíresült megoldást biztosította:

20. Listing. Double brace inicializáció.

```
List<String> list = new ArrayList<>(){{
 add("Hello");
    add("World!");
}};
```

Bár struktúrája a kódnak könnyen olvasható, nem igazán jó megoldás.

- Az első kapcsos zárójellel egy új névtelen osztályt hozunk létre, mely felüldefiniálja az előtte specifikált ArrayList<> osztályt.
- A második kapcsos zárójel az osztály inicializáló függvényét definiálja felül, mely közvetlen a konstruktor meghívása után fog lefutni.

Ezzel szemben a Kotlin e vevővel rendelkező lamdával könnyedén biztosítja nekünk ezt a funkciót mindenféle hátrány nélkül. Pont erre a felhasználásra hozták létre az *apply* függvényt.

21. Listing. Kotlin apply.

```
    val list = mutableListOf<String>().apply {
        add("Hello");
        add("World")
    }
```

3.3. Hálózati forgalom kezelése korutinokkal

A Kotlin nyelv az 1. frissítése óta támogatja a korutinok (*coroutines*) létrehozását. Ennek teljes körű ismertetése túlnyúlna a dolgozatom határain, viszont a hálózatkezelésnél elengedhetetlen volt, ezért ebben a fejezetben röviden bemutatom a működését. Részletes leírást példákkal kiegészítve a JetBrains biztosít [9].

A különböző I/O műveletek, mint például fájl írása, vagy a hálózati kommunikáció, mindig is nagyságrendileg lassabbak voltak, mint a CPU, de még a memóriaműveleteknél is. Ez pedig a fejlesztőknek jelentette a legnagyobb problémákat: hogyan lehetséges megoldani az ilyen hosszan tartó műveletek kezelését?

Egy egyszerű példasorban szeretném bemutatni a Java különböző megoldásait az ilyen feladatok kezelésére. A feladat legyen egyszerű: Hozzunk létre egy TCP szervert, mely képes fogadni a rácsatlakozó klienseket, és beolvassa a küldött üzenetet.

A legegyszerűbb megoldás erre, mely gyakorlatilag követi a specifikációban leírtakat:

22. Listing. Egyszerű szerver (blokkoló).

```
ServerSocket server = new ServerSocket(8080);
Socket socket = server.accept();

byte[] buff = new byte[1024];
int pos = 0, read;
InputStream inputStream = socket.getInputStream();
while ((read = inputStream.read(buff, pos, buff.length -
    pos)) != -1){
```

```
 pos += read;
}
socket.close();
```

Látható, hogy bár ez a megoldás viszonylag könnyen olvasható, két blokkoló függvényt is meghív (*accept()*, *read()*), így a program egész addig nem tud semmi mást csinálni, ameddig a kliens rácsatlakozik és elküldi az adatot.

Ennek egy triviális megoldása lehet az, ha a fent látott kódhoz indítunk egy új szálat, így nem az eredeti főszál fog blokkolódni, az dolgozhat további feladatokon. Ekkor viszont egyrészt már ügyelni kell a szálak közötti szinkronizációra, ha valamilyen adatot kell cserélniük a szálaknak.

Egy másik problémát jelent, ha egyszerre sok kapcsolatot kell kezelnie a programunknak, ekkor ugyanis minden kapcsolathoz létre kell hozni egy új szálat, mely idő,- és erőforrás igényes folyamat. Ezen segíteni tudnak a különböző *ExecutorService* megvalósítások, melyek képesek a szálakat újrahasznosítani más feladatok számára, de akkor is igaz marad, hogy minden párhuzamos blokkoló utasításhoz létre kell hozni egy új szálat.

Ez egy torrentalkalmazás számára, ahol egy pillanatban akár több száz klienssel is kommunikálhatunk, nem tekinthető jó megoldásnak.

A Java készítői is érezték ezt a hiányosságot, ezért a 4.-es frissítésben bevezették a *NIO* (Non-blocking IO) osztályokat. Ez a csomag lehetővé tette, hogy egy úgynevezett *Selector* objektumhoz több csatornát is rendeljünk, majd az objektumon keresztül az összes hozzá regisztrált csatornán egyszerre tudjunk várni.

23. Listing. Egyszerű szerver (NIO).

```
ServerSocketChannel serverSocketChannel =
    ServerSocketChannel.open();
serverSocketChannel.bind(new InetSocketAddress(8080));
serverSocketChannel.configureBlocking(false);

Selector selector = Selector.open();
serverSocketChannel.register(selector,
    SelectionKey.OP_ACCEPT);

SocketChannel socketChannel = null;
ByteBuffer buffer = ByteBuffer.allocate(1024);

while (true){
    selector.select();
    for (SelectionKey selectionKey :
        selector.selectedKeys()){
        if (selectionKey.isAcceptable()){
            socketChannel = serverSocketChannel.accept();
            socketChannel.configureBlocking(false);
```

```
                socketChannel.register(selector,
                    SelectionKey.OP_READ);
            }
            if (selectionKey.isReadable()){
                if(socketChannel.read(buffer) == -1) {
                    break;
                }
            }
        }
        selector.selectedKeys().clear();
}
```

Bár a *Selector* objektum ugyanúgy blokkolja a szálat, ezzel egyszerre akár több száz csatornát is tudunk kezelni egyetlenegy szállal. Hátránya viszont, hogy ezzel a jól olvasható, szekvenciális kódunkat fel kellett bontanunk, ezzel nehezítve a kód megértését.

Több kapcsolat esetén érdemes egy általánosabb hálózatkezelő osztályt írni, mely képes általánosan kezelni ezeket a műveleteket, és úgynevezett *callback* függvényekkel tud visszajelezni a programnak egy művelet befejezésekor. Erre a Java fejlesztőcsapata is rájött, ezért a Java *NIO2* frissítésben bevezették az *aszinkron* csatornákat. Ezek képesek aszinkron módon várakozni eseményekre, melyek megtörténtekor meghívja a függvény hívásakor paraméterként megadott *CompletionHandler* megfelelő függvényét.

24. Listing. Egyszerű szerver (aszinkron).

```
AsynchronousServerSocketChannel serverSocketChannel =
    AsynchronousServerSocketChannel.open();
serverSocketChannel.bind(new InetSocketAddress(8080));

buffer = ByteBuffer.allocate(1024);

serverSocketChannel.accept(null, new
    CompletionHandler<AsynchronousSocketChannel, Void>() {
     @Override
     public void completed(AsynchronousSocketChannel
         result, Void attachment) {
          result.read(buffer, result, new
              CompletionHandler<Integer,
              AsynchronousSocketChannel>() {
               @Override
               public void completed(Integer result,
                   AsynchronousSocketChannel attachment) {
                    if (result == -1){
                        onCompleted(buffer);
                    } else {
```

```
                    attachment.read(buffer, attachment,
                        this);
                }
            }

            @Override
            public void failed(Throwable exc,
                AsynchronousSocketChannel attachment) {

            }
        });
    }

    @Override
    public void failed(Throwable exc, Void attachment) {

    }
});
```

(Ez nem egy optimális implementáció, de a megoldás lényegét átadja)

Ebben az esetben nekünk már nem kell foglalkoznunk a csatorna állapotaival, elég csak egy *Handler* objektumot átadnunk. A megoldás előnye, hogy itt már egy szál se fog blokkolni, viszont az alapvetően szekvenciális kódunk itt már teljesen szétesett, különböző helyeken található meg. Megjelenik a *JavaScript* világban elterjedt *callback hell* jelenség, amikoris a különböző callback függvények egymásba ágyazásával egyre nehezebben olvasható kódot kapunk. Ezen különböző technikákkal lehet javítani, de teljesen megoldani nem lehet.

A Kotlin korutin megoldása ezt a problémát oldja meg azzal, hogy a callback alapú függvények hívását képes visszaalakítani szekvenciális kódvégrehajtásra. A funkció bizonyos szinten hasonlít a C#-ban is megtalálható *async*/*await* megoldáshoz, ennek egy általánosított esete.

Ahhoz, hogy egy függvényből korutint csináljunk, a *suspend* kulcsszóval kell deklarálnunk. Suspend függvényt csak másik suspend függvény hívhat meg; egy suspend függvény indításához a Kotlin biztosít megoldásokat. Egy suspend függvényen belül a kód végrehajtása szekvenciálisan történik.

Különlegessége abból adódik, hogy bizonyos függvények meghívása esetén, melyeknek eredményére várni kell, képes a végrehajtást átadni a programkód más részeinek anélkül, hogy szálat váltana, így akár a főszálon is végrehatjhatjuk a műveletünket.

25. Listing. Egyszerű szerver (korutin).

```
suspend fun doNetworking(server:
    AsynchronousServerSocketChannel){
    val socket = server.aAccept()
    val buffer = ByteBuffer.allocate(1024)
    do {
```

```
        val read = socket.aRead(buffer)
    } while (read != -1)
}
```

Látható, hogy a kód nagyban hasonlít a blokkoló megoldáshoz, mégis a korutinoknak köszönhetően nem fogja a megfogni a végrehajtó szálat.

Ennek a függvénynek a meghívására a Kotlin a *launch()* függvényt biztosítja, mely paraméterként vár egy *Context* objektumot. Ez az objektum határozza meg, hogy az adott hívás milyen kontextusban fusson (pl. egy vagy több szálú kiszolgálón).

26. Listing. Egyszerű szerver (korutin).

```
fun main(args: ArrayList<String>){
    val singleThread =
        newSingleThreadContext(name="Network")

    val server = AsynchronousServerSocketChannel.open()
    server.bind(InetSocketAddress(8080))

    launch(singleThread){doNetworking(server)}
    launch(singleThread){doNetworking(server)}
}
```

Ebben a példában ugyanazon a szálon fog lefutni a két hálózati kérés úgy, hogy a nem blokkolják egymás végrehajtását. Emellett pedig a kódunk struktúrailag megegyezik az eredeti Java kóddal, ugyanúgy szekvenciálisan követik egymást az utasítások, ezzel a kód olvashatóságán javítva.

4. BitTorrent tervezés és megvalósítás

A dolgozatom keretében, annak érdekében, hogy gyakorlati tapasztalatot is szerezzek a Kotlin nyelvről, megírtam egy könyvtárat, mely megvalósítja a BitTorrent protokollt.

Célom a nyelv megismerése, illetve különböző összehasonlításokra alkalmas kód írása volt, ezért a BitTorrentet érintő algoritmikus kérdéseket próbáltam minél egyszerűbben megoldani. Ebből kifolyólag a megírt könyvtár jelenlegi állapotában nem használható fel alkalmazás fejlesztésére.

Több okból is a BitTorrent protokollra esett a választás mint implementálandó protokoll:

- Mindig is érdekeltek a hálózatok, különösképp a maguktól szerveződő peer-to-peer rendszerek felépítése, működése.
- A BitTorrent protokoll az időnek ellenállva még ma is az legnépszerűbb fájlmegosztási protokoll.
- A Kotlin, mint relatív új nyelv, még nem rendelkezik nyílt forráskódú BitTorrent könyvtárral.
- Egy fájlmegosztó könyvtárhoz szükséges fájl,- illetve hálózati forgalom kezelése ideális környezetet biztosít egy új nyelv tesztelésére.
- A szakdolgozatom miatt Java nyelven már implementáltam egyszer a BitTorrent protokollt, így magával a protokollal kapcsolatban már sikerült elég tapasztalatot szereznem.

Mivel az implementáció során közvetlen a két nyelvet akartam összehasonlítani, csak az általuk biztosított könyvtárakat használtam fel a fejlesztés során.

Ebben a fejezetben a BitTorrent könyvtár implementálását foglalom össze a két nyelven. Mindkét projektnek a forráskódja megtalálható a GitHub oldalon: KotlinTorrent [10], JavaTorrent [11].

4.1. BitTorrent Kotlin nyelven

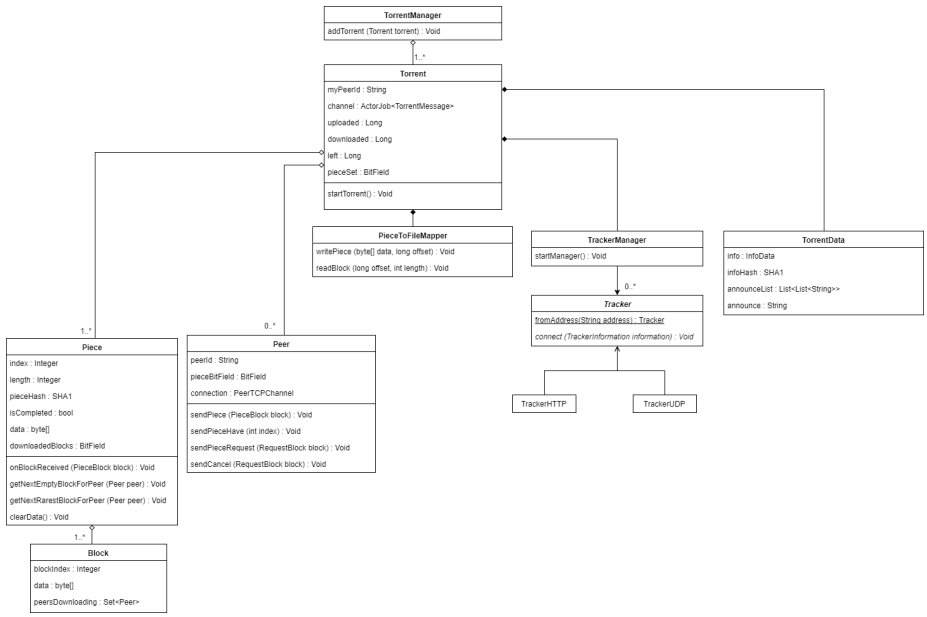
Először a Kotlin nyelven oldottam meg a feladatot. Ehhez az alábbi főbb osztályokat hoztam létre:

- *Torrent*: A könyvtár középpontjában áll, ez az osztály felel egy adott torrentért. Ő tárolja a tracker által talált peereket, kezdeményezi a csatlakozást a peerek felé, és kezeli a letöltés aktuális állapotát.
  - *TorrentData*: Ez az osztály tárolja az összes torrenttel kapcsolatos statikus adatot. Megfeleltethető a torrentet leíró .torrent fájllal.
  - *TorrentManager*: A Torrent példányok tárolására és keresésére létrehozott osztály.
  - *TorrentMessage*: A Torrent osztályhoz tartozó aktor üzenettípusai.

- *Peer*: Az egyes peerek állapotát, illetve a rajtuk végrehajtható műveleteket összefogó osztály.
  - *PeerTcpChannel*: A kommunkációhoz használt TCP csatornak kezelését megvalósító osztály.
  - *PeerMessage*: A Peer osztályhoz tartozó aktor üzenettípusai.
- *Piece*: A fájlmegosztás alapegységét, a *piece* elemek tulajdonságaiért felelős osztály.
- *BencodeEncode*/*BencodeDecode*: A protokollban használt úgynevezett *bEncode* kódolást implementáló függvények.
- *Tracker*: A tracker lekérdezéséért, illetve állapotának tárolásáért felelős osztály.

A hálózati kommunikációnál erősen kihasználtam a 3. fejezetben bemutatott korutinokat Ahhoz, hogy a elkerüljem a többszálúságból eredő szinkronizációt, illetve az ezzel együttjáró hibaforrásokat, a *Torrent*, *Peer*, illetve a *PeerTCPChannel* osztályok mind valamilyen formában tartalmazzák az *aktor* minta megfelelő variánsait.

Az elkészült Kotlin projekt felépítésének bemutatására rajzoltam egy UML diagrammot, melyet a 8. ábra mutat.



1. ábra. A Kotlin BitTorrent projekt felépítése UML diagramm segítségével. Nagyobb méretben megtalálható a függelékben.

Miután összehasonlítottam az így kapott Kotlin kódot az egy évvel ezelőtt írt Java implementációval, arra jutottam, hogy a kettő sokban eltér egymástól, ezért az összehasonlítás nem lenne megfelelő. Ebből kifolyólag úgy döntöttem, hogy követve a Kotlin verzióban hozott szoftvertervezési döntéseket, újraírom a könyvtárat Java nyelven is.

4.2. BitTorrent Java nyelven

Bár az implemetálás közben próbáltam minél pontosabban követni a Kotlin verziót, bizonyos pontokon el kellett térnem attól.

- A korutinok hiányából fakadóan a hálózati kommunkációhoz egy állapotgépet kellet létrehoznom, mely képes volt a különböző aszinkron hívások mellett számon tartani a protokoll aktuális állapotát.
- Mivel a Java nem biztosít támogatást az aktor mintára, azt saját megoldással kellett áthidalnom. Ezt végül a következő lépésekkel helyettesítettem:
  - Létrehoztam egy globálisan elérhető *ScheduledExecutorService* példányt, mely a neki küldött taskok listájával megfeleltethető az aktorok üzenet listájának.
  - Az aktor minden üzenet típusához létre hoztam egy publikus metódust, mely az adott üzenet feldolgozásáért felelt, és hozzátettem a synchronized módosítószót, ezzel elérve az üzenetfeldolgozás kölcsönös kizárását.
  - Egy aktornak küldött üzenet ezután megfeleltethető az ütemezőnek küldött taszkkal, mely meghívja az aktor megfelelő metódusát.

Bár az aktor minta lecserélése rengeteg szinkronizációt vont maga után, úgy éreztem, hogy az ebből származó performancia csökkenés nem számottevő egy alapvetően I/O limitált felhasználás esetén, cserébe viszont nem kellett egy külön osztály írnom az aktorok üzenetlistájának kezelésére.

Ezeket leszámítva sikerült jól követni a Kotlin kódot, a metódusok többsége között megtalálható az egy-egy megfeleltetés, így úgy érzem, ez jó alapot ad a két nyelv összehasonlítására.

5. Eredmények

Ebben a fejezetben foglalom össze a két projekktel kapcsolatban mért eredményeimet.

5.1. Kód méret

Az első paraméter, melyeket meghatároztam, a forráskód méretei. Ez alatt értem a forráskódok teljes méretét (mely arányos a kódban található karakterek számával. Emellett mértem a sorok számát két módon is:

- Forrás sorok száma; azok a sorok, melyekben található kód.
- Összes sorok száma

A projekt méretének méréséhez az operációs rendszer beépített utasítását használtam. A sorok mérését egy, az IntelliJ fejlesztőkörnyezethez letölthető plug-in, a *Statistic* segítségével tettem meg. Az eredményeket a 1. táblázat foglalja össze.

1. táblázat. Kód méretek

Nyelv	Fájlok száma	Méret (byte)	Összes sor	Forrás sor
Kotlin	19	59 112	1 633	1 356
Java	27	87 953	2 683	2 183

Látható, hogy annak ellenére, hogy a projekt felépítése gyakorlatilag megegyezik, a Java fájlok száma lényegesen nagyobb a Kotlinnal szemben. Ennek indoka, hogy a Kotlin engedélyezi a tetszőleges számú publikus osztály deklarálását egy fájlban, míg a Java szigorúan egyre korlátozza ezt. Emiatt sok kisebb osztályt, mely véleményem szerint egy logikai egységet alkotott, a Kotlinban egy fájlba tettem, míg a Java-ban kénytelen voltam több fájlra osztani ezt.

Egy másik fontos megfigyelés tehető a méretekkel kapcsolatban. A fájlméret esetén 33%-al, míg a sorok száma 39%-al csökkent a Kotlin esetében. Ezt az eredményt árnyalja, hogy a Kotlin támogatást nyújtott a hálózati forgalom egyszerű kezelésére a korutinok segítségével, míg Java-ban erre egy külön állapotgépet kellett írni. Ha kivesszük a *PeerTCPChannel* osztályt az összehasonlításból, akkor pontosabb képet kaphatunk a javulásról. Ezt a 2. táblázat mutatja.

Ebben az esetben a fájlméret 27%-al, míg a sorok száma 35%-al csökkent, melyek az előző eredményekhez hasonlóan jelentős csökkenésnek minősülnek. Ebben az esetben a kódstruktúra pedig már szinte teljesen megegyezik, ezért a korrigált adatokból már jobban össze lehet hasonlítani a két nyelv hatását.

A Java kód egyértelműen bőbeszédűbb, például a következő indokok miatt:

2. táblázat. Kód méretek (korrigált)

Nyelv	Méret (byte)	Összes sor	Forrás sor
Kotlin	53 622	1 483	1 233
Java	73 664	2 289	1 845

- Változók deklarálásánál kötelező a típus deklarálása. A kotlin automatikus típus interferálása egyszerűsít ezen, és a véleményem szerint a kód olvashatóságán se ront.

- A különböző tárolókon végzett szűrések a lambda függvények optimalizálásának, illetve a kiegészítőfüggvényeknek köszönhetően a Kotlin nyelven kevesebb függvényhívással oldja meg ugyanazt a problémát.

- A Kotlin által végzett "okos" típuskonverzió, mely követi a kódban végzett típus-ellenőrzéseket.

A Kotlin tömörsége mellett a véleményem szerint általában nem nehezíti meg a kód olvashatóságát, ezzel ellentétben többször még segíti is azt.

Kifejezetten tetszett a konstans változók deklarálásának egyszerűsége, illetve az elsődleges konstruktor segítségével való inicializáció. Ezzel el lehet érni, hogy egy osztály mezőjének a deklarálásakor egyből inicializáljuk is, így a két utasítás egymás mellett fog állni, nem kell keresni az egyiket a másikhoz.

5.2. Fordítási idő

Nagyon fontos szempont egy nyelv választásánál, hogy a mellékelt fordító mennyi idő alatt tudja lefordítani a kódunkat. Részben a Kotlin nyelv is ennek köszönhető, mivel eredetileg a JetBrains csapata a Scala-ban megtalálta a számukra legtöbb hasznos funkciót, de végül a lassú fordítás miatt vetették el. Emiatt lett a Kotlin egyik fő célja, hogy legalább olyan gyorsan forduljon, mint a hasonló Java-ban írt kód.

Mivel a projekteket úgy alakítottam, hogy azok felépítése struktúrailag megegyezzenek, ideálisnak találtam a fordítási idők összehasonlításához.

Az összehasonlításhoz a *Gradle* build rendszert fogom használni. A *Gradle* az egyik legelterjedtebb Java fordítórendszer, az Android Studio fejlesztőkörnyezet is ezt használja. Előnye, hogy a különböző fordítási beállításokhoz egyszerű parancssoros hozzáférést biztosít, így könnyedén automatizálható a fordítás, illetve a statisztikák begyűjtése.

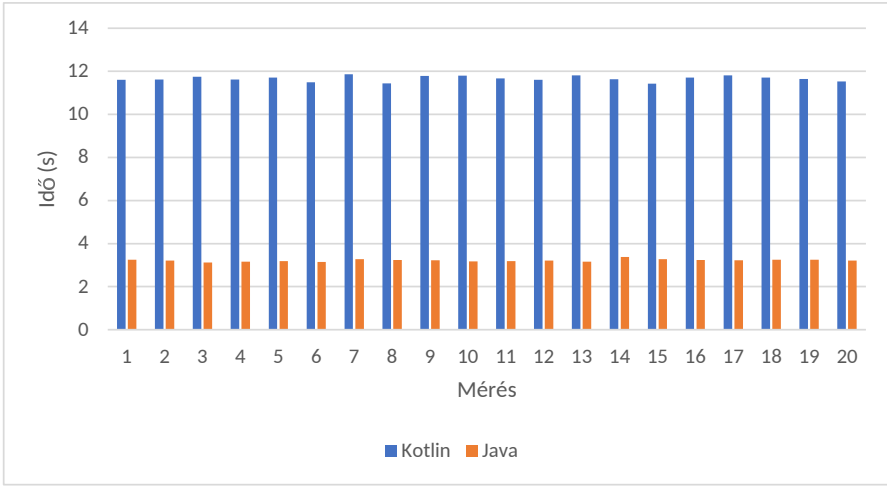
A Gradle-höz tartozik a *Gradle daemon*. Ez egy alapvetően a háttérben futó folyamat, mely a fordítások között se áll meg, ami így gyorsabb fordítást tud eredményez, ezért kikapcsolt és bekapcsolt állapotban is elvégzem a méréseket.

A projektek fordításának indítására, illetve a fordítási idő mérésére létrehoztam egy *Python* szkriptet, mely képes a mért adatokat egy CSV fájlba kiírni. A

szkript forráskódja megtalálható a GitHub oldalon [12]. A fordítási időt három különböző helyzetben mértem:

- Teljes fordítás. Minden fordítás előtt kitisztítom *clean* a projektmappát, így minden alkalommal a teljes projektet le kell fordítania.
- Üres fordítás. Ekkor egy, már lefordított projektet fordítok újra anélkül, hogy bármi is változna.
- Inkrementális fordítás. Ekkor egy fordítás előtt módosítok egy fájlon, így a fordítónak nem kell az egész projekten dolgoznia. Ehhez implementáltam még egy funkciót a torrent könyvtárba, a szkript pedig vagy hozzáadja, vagy kitörli ezt a funkciót futás előtt.

Először a *daemon* nélkül mérem a futási időket.



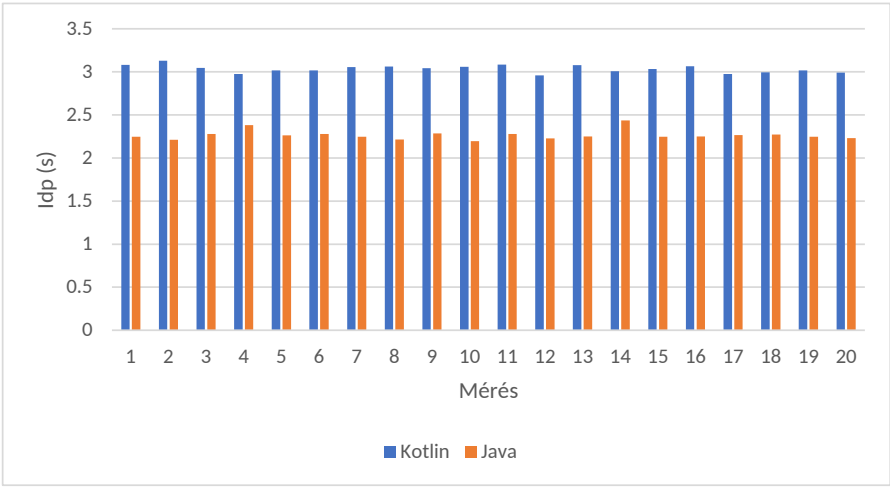
2. ábra. Teljes fordítási daemon nélkül

3. táblázat. Teljes fordítás daemon nélkül - statisztika

Nyelv	Átlag (s)	Szórás (s)
Kotlin	11.67	0.015
Java	3.22	0.003

Ahogy a 2 ábra, illetve a 3 mutatja, a Kotlin jelentősen rosszabbul teljesít, a Java átlagos 3.22 másodpercéhez képest a Kotlin fordítónak 11.67 másodpercre volt szüksége. Emellett a Java fordítási idő szórása is kisebb.

A második mérés volt az üres fordítás, melynél semmi nem változott a fordítások között (4 ábra, 4 táblázat). A Kotlin ekkor is rosszabbul teljesített, de itt a különbség annyira nem jelentős.



3. ábra. Üres fordítás daemon nélkül

4. táblázat. Üres fordítás daemon nélkül - statisztika

Nyelv	Átlag (s)	Szórás (s)
Kotlin	3.03	0.002
Java	2.27	0.003

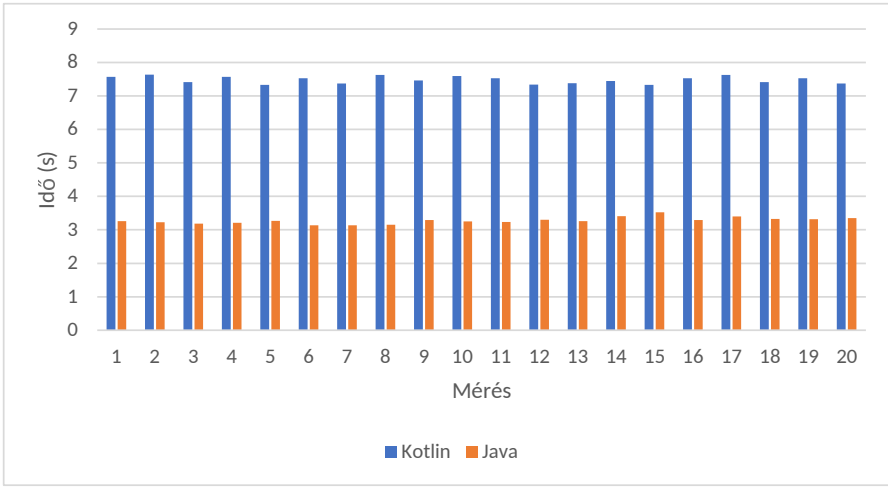
A harmadik mérés volt az inkrementális mérés, melynek eredményeit a 4 ábra mutatja, illetve a számolt statisztikai értékeket a 5. Érdekes módon a Java kód fordításánál nem látszódik érdemleges javulás a teljes fordításhoz képest. A Kotlin részéről viszont javulás tapasztalható, a teljes fordítás 12 másodpercéhez képest itt 7.5 másodperc szükséges a teljes folyamathoz, de még így is több a Java 3.25 másodperces fordításához képest.

A következő mérésekben bekapcsolva hagytam a Gradle *daemon* folyamatát, így arra lehet számítani, hogy a fordítási idő csökkeni fog attól függően, hanyadik mérést folytatom. Emiatt a statisztikákat itt már csak az utolsó 6 mérés eredményéből számolok.

Ismételten elvégzem a mérést a három különböző fordítási esetre.

5. táblázat. Inkrementális fordítás daemon nélkül - statisztika

Nyelv	Átlag (s)	Szórás (s)
Kotlin	7.84	0.36
Java	3.28	0.009



4. ábra. Inkrementális fordítás daemon nélkül

6. táblázat. Teljes fordítás daemon folyamattal - statisztika

Nyelv	Átlag (s)	Szórás (s)
Kotlin	3.12	0.001
Java	1.31	0.0002

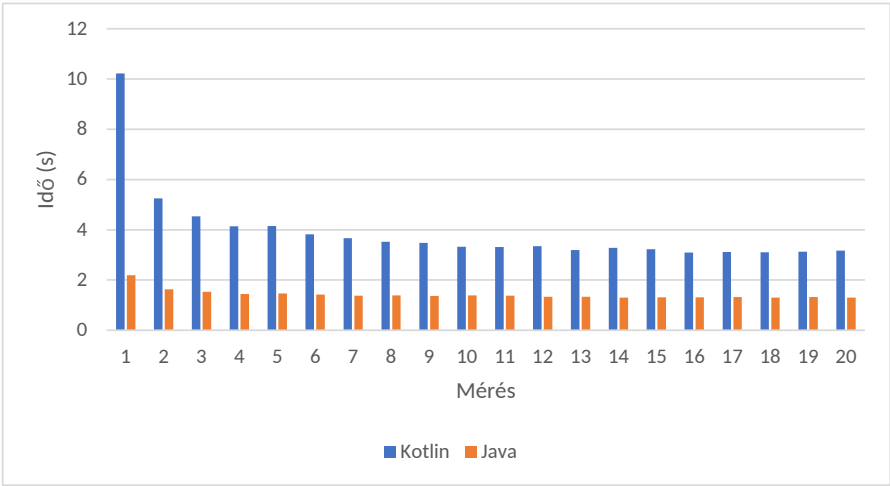
A 5 ábrán látható, hogy a *daemon* folyamat tényleg jelentősen javított a fordítási időkön mind a Java, mind a Kotlin esetében. Megfigyelhető, hogy a különböző optimalizációkhoz a Gradle-nek idő kell, Java esetében a 4. fordítás után nem tapasztalható javulás, míg Kotlin esetében egészen a 10. futtatásig stabilan gyorsult. A stabil állapotban a Kotlin még mindig kétszer lassabb a Java-hoz képest, ahogy ezt a 6 táblázat is mutatja.

7. táblázat. Üres fordítás daemon folyamattal - statisztika

Nyelv	Átlag (s)	Szórás (s)
Kotlin	1.17	0.0003
Java	1.17	0.0052

A 6 ábra az üres fordítás mutatja. A 7 táblázatról leolvasható, hogy a *daemon* futtatásával a Java és Kotlin ideje között gyakorlatilag semmi különbség nincs.

Az utolsó mérés, melyben inkrementális fordítást végeztem *daemon* folyamattal, eredményeit a 7 ábra illetve a 8 táblázat foglalja össze. Ez talán



5. ábra. Teljes fordítási daemon folyamattal

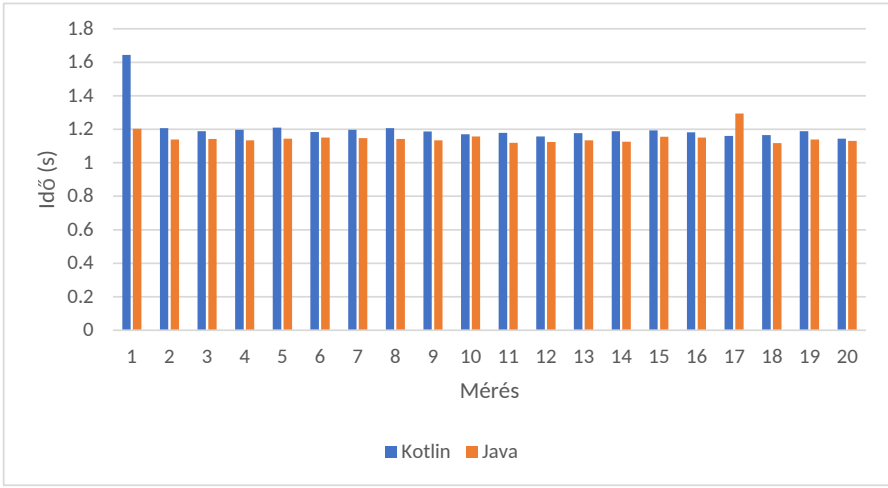
8. táblázat. Inkrementális fordítás daemon folyamattal - statisztika

Nyelv	Átlag (s)	Szórás (s)
Kotlin	1.81	0.004
Java	1.40	0.002

a legfontosabb mérés az eddigiek közül, ugyanis ez a leggyakoribb használati módja a fordításnak: kis változtatások gyakori fordítása a *daemon* futtatásával. Ebben a felállásban bár a Java még mindig egyértelműen jobban teljesít, a különbség már annyira nem jelentős, mint az eddigi mérésekben. A Java a mért adatok alapján 22%-al gyorsabb még mindig a Kotlin fordítójához képest.

A mérésekből megállapítható, hogy a Java fordítási ideje minden körülmény között jobb volt a Kotlinnál. Ezt az eredményt azonban a következő tények kis mértékben árnyékolják:

- A Kotlin még egy relatív új nyelvnek számít, mely állandó fejlesztés alatt áll. A fejlesztőcsapat is közölte, hogy a fordítón még számos optimalizációt terveznek végrehajtani, mely gyorsíthat a fordításon.
- Ezzel szemben a Java egy régebbi nyelv, melynek így rengeteg ideje volt tökélyre fejlesztenie a különböző fordítási lépéseket.
- Ezek a mérések egy relatív kis méretű projekten történtek egy számítógépen. Előfordulhat, hogy más projekt esetén, vagy jobb gép esetén (pl. 16GB RAM memória) jobban tudna teljesíteni a Kotlin.
- A Kotlin projektben előszeretettel használtam a korutinok által biztosított lehetőségeket. A "suspend" függvények miatt keletkező többletmunkát (a



6. ábra. Üres fordítás daemon folyamattal

szükséges állapotgépek kialakítása miatt) a Java projektben nem jelenik meg, hiszen ott nincsen ilyen lehetőség, ezért ezeket a részeket kézzel kellett megírnom.

5.3. Android kódmetrikák

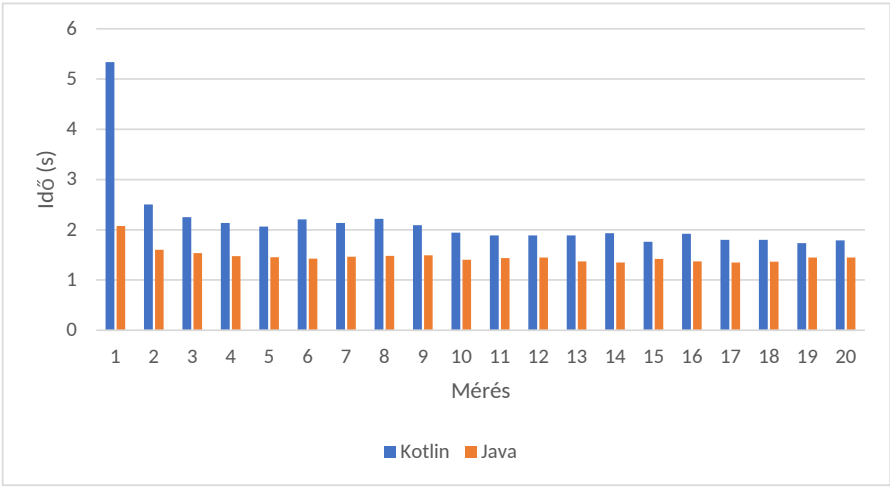
A Kotlin egyik legnagyobb célplatformja az Android operációs rendszer. A Java már már túlságosan is részletes kódja az Android esetében még tovább ront a helyzeten, ahol az alkalmazás rengeteg különböző állapota már így is megnehezíti a fejlesztők munkáját. Emiatt is ritka az olyan Android projekt, mely nem használ fel semmilyen külső könyvtárat az állapotok, illetve felületek kezelésének az egyszerűsítésére.

Kotlin esetében a nyelv rugalmassága miatt sokszor tisztább kódot lehet írni a Java-val szemben, és ugyanúgy JVM bájtkódra fog lefordulni, így nincsen meg az a teljesítmény csökkenés, amit a legtöbb multiplatform megoldás ad.

Emiatt tartottam fontosnak annak a vizsgálatát, hogy a Kotlin projekt hogyan teljesít Androidra való fordítás esetén.

Az Androidos világban a fájlméret mellett egy igen fontos száma az alkalmazásokban a deklarált metódusok száma. Ennek az az indoka, hogy az Android előír egy limitet a metódusok számára (65536), melyet ha átlép az alkalmazás, kötelező egy speciális osztályt használnia. Ez a teljesítményre is kihatással lehet, a limitet átlépő alkalmazások akár 20-30%-os lassulást is tapasztalhatnak.

A méréshez az IntelliJ fejlesztőkörnyezet által biztosított Android plug-int használtam. Mindkét nyelvnél létrehoztam egy új projektet, melyekben létrehoztam ugyanazt a példa projektet (egy felületet tartalmazó alkalmazás). Ezután bemásoltam a torrent forrásfájlokat, majd biztosítottam, hogy az Android kód elindítson egy torrentet.



7. ábra. Inkrementális fordítás daemon folyamattal

Az így létrejött projektet lefordítom *Debug*, illetve *Release* üzemmódban. Ezután a kész alkalmazások telepítőit megnyitottam egy analizátorban, mely képes megadni az osztályok, illetve a metódusok számát.

Azért, hogy meg tudjam mérni, hogy maga a BitTorrent könyvtár mekkora helyet foglal, készítettem két, gyakorlatilag üres projektet is, melybe nem másoltam be a torrent könyvtárat, és így fordítottam le. A rendes projekt, illetve az üres projekt fordításával megkaphatjuk, hogy a könyvtár használata mennyivel növelte az egyes értékeket. Ezeket tekintem korrigált értékeknek. Ez egy jó alapot fog biztosítani ahhoz, hogy megmérjem a fordító hatékonyságát.

Az így kapott eredményeket a 9. táblázat mutatja be. A korrigált értékeket a * oszlopok jelölik.

9. táblázat. Android metrikák

Verzió	Méret	Osztály	Metódus	Méret*	Osztály*	Metódus*
Java Debug	1.5 MB	1 673	13 401	0.1MB	86	485
Java Release	0.8 MB	704	5 431	0.1MB	84	433
Kotlin Debug	2 MB	2 600	20 335	0.1MB	99	559
Kotlin Release	0.9 MB	1 012	6 552	0.1MB	389	1 555

Egyértelműen a legérdekesebb eredmény, hogy a Kotlin *Release* fordítás alatt sokkal több osztályt, illetve metódust fordított a projekthez, mint *Debug* módban. Ez a Kotlinhoz szükséges könyvtárak (*stdlib*, *coroutines*) használatára vezethető vissza.

Debug üzemmódban a fordító nem távolítja el a felesleges osztályokat, metódusokat, így ebben a fordításban az saját és az üres projekt között egyedül az általam írt osztályok különböznek.

Release üzemmódban már be van kapcsolva a felesleges kódok eltávolítása, így az üres projekt esetén a Kotlin könyvtárak is törlődni fognak, ezzel tovább csökkentve az alkalmazás méretét.

Látható, hogy a Kotlin használata jelentősen megnövelte az osztályok, illetve metódusok számát, ez viszont leginkább a fordításhoz szükséges alapkönyvtárak okozzák. Magának a projektnek a fordításánál is a Java van fölényben, de itt már annyira nem számottevő a különbség (érdemes a *Debug* verziókat összehasonlítani).

A mérésekből megállapítható, hogy bár a Kotlin alapú Androidos szoftverfejlesztés megnöveli az alkalmazás méreteit, ez (főleg *Release* módban) annyira nem számottevő a Java verzióhoz képest.

6. Jövőbeli lehetőségek

A dolgozatom készítése során arra jutottam, hogy a Kotlin nyelv egy nagyon jó alternatívát ad a Java alapú fejlesztésre, ezért szeretnék jobban is megismerkedni a nyelv által nyújtott funkciókjal, lehetőségekkel.

Mindenképp tervezem befejezni a torrent könyvtárat, és azt nyílt forráskódúvá tenni. Ehhez jobban meg kell ismernem a külömböző ütemezési stratégiákat a peerekkel, illetve a trackerekkel kapcsolatban.

A torrent könyvtárat felhasználva szeretnék létrehozni egy androidos alkalmazást, természetesen ezt is Kotlin nyelven fejlesztve. Az androidos alkalmazást hasonlóan nyílt forráskódúvá tervezem tenni.

A [13] TornadoFx könyvtár lehetővé teszi, hogy a Java világból megismert *JavaFx* könyvtárhoz hasonló felhasználói alkalmazásokat hozzunk létre. Ennek a könyvtárnak a felhasználásával szeretném a torrent könyvtáramat a számítógépes világban is felhasználni.

Ezek mellett szeretném kipróbálni a Kotlin multiplatform adottságait. A Kotlin képes a Java környezeten kívül JavaScriptre, illetve kísérleti állapotban natív x86-os alkalmazásra fordulni, mely ideális lehet a projektkód újrahasználására a különböző platformok között.

7. Összefoglaló

A dolgozatomban ismertettem a torrentezés alapjait, illetve bemutattam a Kotlin nyelv alapvető szintaxisát, eredetét. Kitértem a kísérleti állapotban lévő korutinokra, mely jelentősen megkönnyítette a blokkolás nélküli hálózatkezelést.

A Kotlin nyelv ismertetése után bemutattam a BitTorrent könyvtár felépítését, majd kitértem a Java és Kotlin projektek közötti különbségek okaira, a megoldások módjára.

A két projekten mért különböző metrikák alapján az alábbi megállapításokat tettem:

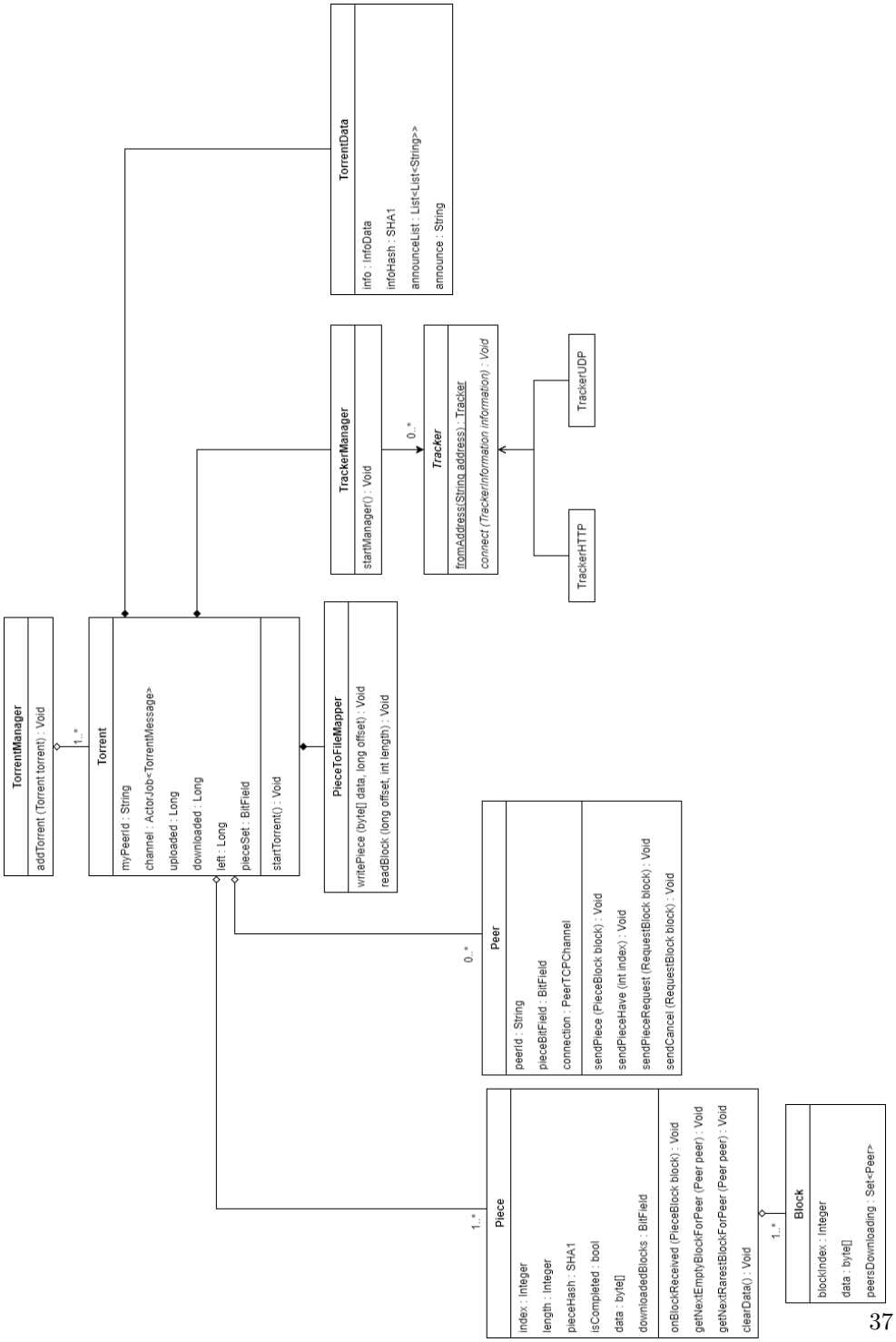
- A Kotlin kód kifejezőképessége sokkal erősebb a hasonló Java kódhoz képest. Emellett a kompaktsága, az immutábilis változók könnyű létrehozása, a kiegészítő funkciók és a lambdák használata kompaktabb és olvashatóbb kódot eredményez.
- A fordítási idők alapján minden körülmény között a Java volt előnyben. A leggyakrabban használt eset, amikor inkrementális fordítást hajtunk végre, a Gradle *daemon* pedig fut a háttérben, adja a legoptimálisabb fordítási időt, de a Kotlin ekkor is 30%-al le van maradva.
- Android platformon a Kotlin nagyobb kódot eredményez, de ez főleg a kötelező könyvtárak miatt van, az általam írt részek a fordítás után körülbelül ugyanakkora helyet foglaltak el.

A dolgozatban a Kotlin nyelvet már egy bonyolultabb, valós feladat megvalósítására használtam, melyben szükség volt hálózatkezelésre, fájlkezelésre, ütemezésre illetve különböző szűrési műveletekre kollekciókon.

Megállapítottam, hogy az ilyen, már összetettebb feladatok ellátására is képes a Kotlin, ezek alapján bátran ajánlom bármilyen új vagy meglévő projektnél a használatát.

A dolgozatom végén leírtam, milyen jövőbeli lehetőségeket tudok elképzelni a Kotlinnal kapcsolatban. Véleményem szerint a Kotlin egy nagyon jól sikerült programozási nyelv, még ha a fordító jelenleg nincs is egy szinten a Java fordítóval, de a jövőben várható ennek lényeges javulása.

8. Függelék



8. ábra. A Kotlin BitTorrent projekt felépítése UML diagramm segítségével

9. Hivatkozások

[1] Oracle, „The history of java technology.” http://www.oracle.com/technetwork/java/javase/overview/javahistory-index-198355.html, October 2017.

[2] Wikipedia, „Oak (programming language).” https://en.wikipedia.org/wiki/Oak_(programming_language), October 2017.

[3] Oracle, „The java® virtual machine specification.” https://docs.oracle.com/javase/specs/jvms/se7/html/index.html, October 2017.

[4] Wikipedia, „C sharp.” https://hu.wikipedia.org/wiki/C_Sharp, October 2017.

[5] JetBrains, „Why jetbrains needs kotlin.” https://blog.jetbrains.com/kotlin/2011/08/why-jetbrains-needs-kotlin/, October 2017.

[6] JetBrains, „Kotlin vs java: Compilation speed.” https://medium.com/keepsafe-engineering/kotlin-vs-java-compilation-speed-e6c174b39b5d, October 2017.

[7] D. McGregor, „The cost of kotlin language features.” http://oneeyedmen.com/cost-of-kotlin-preliminary-results-part1-baselines.html, October 2017.

[8] JetBrains, „Sealed classes.” https://kotlinlang.org/docs/reference/sealed-classes.html, October 2017.

[9] JetBrains, „Guide to kotlinx.coroutines by example.” https://github.com/Kotlin/kotlinx.coroutines/blob/master/coroutines-guide.md, October 2017.

[10] D. Pásztor, „Kotlin bittorrent library.” https://github.com/danim1130/kotlinTorrentGradle, October 2017.

[11] D. Pásztor, „Java bittorrent library.” https://github.com/danim1130/javaTorrentGradle, October 2017.

[12] D. Pásztor, „Gradle teljesítménymérés.” https://github.com/danim1130/GradlePerformance, October 2017.

[13] E. Syse, „Tornadofx: Javafx framework for kotlin.” https://github.com/edvin/tornadofx, October 2017.