Témalabor dokumentáció
Változatos programozási nyelvek és környezetek megismerése

Szalai Dávid
AGGRW2

konzulens:
Kövesdán Gábor

2018. december 8.

Tartalomjegyzék

1. fejezet

Bevezetés

1.1. Téma leírása

A szoftverfejlesztésben általában egy fő platformra specializálódunk, és ebben fejlesztünk. Ez praktikus döntés, hiszen a terület olyan gyorsan fejlődik, hogy nem lehet minden nyelven és platformon mély és naprakész tudást fenntartani. Emiatt azonban a fejlesztők hajlamosak nagyon egyoldalúan gondolkodni, csak a saját maguk által preferált környezet kínálta megoldásokat, technikákat használni és elfogadni.

A téma lehetőséget kínál az érdeklődő hallgatóknak, hogy több, jelentősen különböző nyelvekbe is belekóstoljanak, és ezáltal jobban megismerjék a manapság elérhető programozási nyelvek kínálta technikákat és lehetőségeket. A választott nyelveken olyan programokat kell elkészíteni, amelyek elég egyszerűek ahhoz, hogy legyen idő több programozási nyelven is megírni őket, ugyanakkor elég bonyolultak is legyenek, hogy a választott nyelvek eszköztárát alkalmazni lehessen rajtuk. Ilyen lehet például a black jack kártyajáték vagy a game of life.

A munka kimenete, hogy a hallgató bár nem feltétlen lesz "profi" egyik nyelvben sem, egy magas szintű rálátással rendelkezik majd az egyes nyelvekre és platformokra. Ez szolgálhat alapul a későbbi választáshoz, illetve akár konklúziók is megfogalmazhatók, hogy melyik nyelv milyen problémákban erős.

Javsolt nyelvek: Java, Kotlin, Python, JavaScript, Scala, Haskell, Erlang, Lua. A feladatnak az is része, hogy a használható fejlesztőkörnyezeteket és library-kat is feltérképezzük, ezért a feladatokat TDD megközelítéssel kell elvégezni.

A téma továbbvihető későbbi félévekre, sőt, igényes kidolgozása több félévet igényel.

1.2. A projekt célja

Alapvetően azért választottam ezt a témát, mert szerettem volna kicsit elrugaszkodni az eddig használt nyelvek világától és megismerni más, akár típusban eltérő, nyelveket is. A másik oka, hogy ezt a témát választottam, pedig az, hogy több nyelv megismerésével talán konkretizálódik, hogy mivel szeretnék foglalkozni a jövőben. Úgy gondolom, ha egy programozási nyelv megtetszik, akkor sokkal könnyebb témát keresni hozzá, hiszen ismertek a lehetőségei.

A célom pedig az volt, hogy tágítsam a látókörömet és ezáltal talán kialakul majd egy kép bennem, hogy milyen irányban induljak el a jövőben. A témalaborig csupán csak azokkal a nyelvekkel ismerkedtem és foglalkoztam, amiket az egyetemi évek alatt oktattak. Fontos volt számomra, hogy tágítsam a látókörömet. Persze már a választásomkor tudtam, hogy egy félév nem lesz elegendő, hogy megismerjek minden nyelvet, amit szeretnék. De úgy vélem, hogy kiválasztva a két legszimpatikusabbat, azért jelentős tapasztalatokkal gazdagodhatok, akkor is, ha tudom, hogy még rengeteg más nyelv van, amit még érdemes kipróbálni.

Ahogy a témaleírás1.1 is írja, ez a téma tipikusan továbbvihető a későbbi félévekre, hiszen nagy rész teljesen önálló munka. Én úgy érzem a lényegét mégis átadta, hiszen bár sok különböző nyelv van, azért az egyes típusokban sok a hasonlóság (gondolok itt például az OO vagy a szkript nyelvek világára, akár a funkcionális programozásra). Alapvetően ez egy remek téma és lehetőség volt, hogy betekintést nyerjek más típusú nyelvek világába és megismerhessem sajátosságaikat.

1.3. Választott nyelvek

Már a témalabor választásakor elgondolkoztam, hogy melyik nyelvek legyen azok, amelyekkel egy féléven át foglalkozni szeretnék. Több elképzelésem is volt, de nem akartam túlzásba esni. Ezért két szempont szerint elkezdtem szortírozni a lehetőségeket.

Először is, mindenképpen szerettem volna választani egy OO nyelvet, amit még sosem használtam. A választásom pedig azért a Kotlin3 lett. Ennek több oka is volt. A barátaim közül többen is javasolták, hogy válasszam ezt, mert ők egy nyári projekt alkalmával kipróbálták és rendkívül tetszett nekik. Ösztönöztek, hogy próbáljam ki én is. A másik ok, pedig egy kis utánajárás következtében alakult ki. Pár bemutatóvideó után nekem is megtetszett ez a nyelv, mert bár Java alapokra van helyezve, mégis szinte minden hibáját kijavították benne. Sőt, ha kicsit pongyolán akarok fogalmazni, akkor összegyúrták a Java és a C# nyelvet és az ő gyermekük lenne a Kotlin. Tehát ezzel mindenképp meggyőzött, hogy ez legyen az egyik választott nyelvem.

A másik szempontom pedig, hogy válasszak egy nem OO nyelvet. Először még nem tudtam eldönteni, hogy szkript vagy deklaratív nyelvet válasszak, így csak egy sorrendet tudtam kialakítani, hogy előbb lenne egy szkript nyelv és aztán egy deklaratív.

A szkript nyelvek közül nehéz volt választani. Bár szerencsére sok a hasonlóság, azért mindegyiknek megvannak a maga sajátosságai. Ilyenkor persze mindenkinek először a JavaScript jut eszébe és én épp ezért gondolkodtam valami másban. JavaScript betekintőt a félév során egy másik tantárgyból úgyis kapunk, ezért is gondoltam úgy, hogy eltérek a "mainstream" választástól. Épp emiatt a második nyelvemnek a Lua-t2 választottam.

Volt egy másik oka is, amiért ez lett a választás. Gyakran játszok ugyanis egy MMORPG játékkal, amelyben használhatok külső, kisegítő eszközöket (ún. Addonokat), és mindig is érdekelt, hogy ezeket hogyan csinálják, mert én is szeretnék csinálni számomra testre szabottakat vagy akár a meglévőket módosítani. Amikor utánajártam, hogy ezeket az Addonokat Lua-ban írják, akkor gondoltam úgy, hogy szeretnék foglalkozni ezzel a nyelvvel, ha lesz rá időm. És ezért esett a választás erre a nyelvre.

Mindazok a nyelvek, amiket nem választottam ki, de a munkám során vagy korábbi évek alatt előkerültek (Java, C, C++, C#, JavaScript, Pascal, Python), szeretném majd (ha tudom) referenciaként vagy akár "harmadik szemszögként" felhasználni. Így a nyelvek bemutatását még több perspektívával tudom ellátni és bemutatni, hogy mennyi apróbb különbség/hasonlóság van a nyelvekben.

Deklaratív nyelvet azért is raktam a végére, mivel a 7. félévben lehetőség lesz egy elágazó tantárgy kereteiben tanulni róla. Mindemellett úgy gondoltam, ha jut rá időm, szeretném kipróbálni a Haskell-t. Sajnos azonban ez a terv nem tudott megvalósulni a félév alatt.

1.4. A projektek bemumtatása

Először bemutatom a választott nyelveket vázlatosan. Az egyes példánál a projektekből idézek, illetve megpróbálok minél több referenciát felhozni, hogy más nyelvekben milyen hasonlóságok és különbségek vannak.. Majd ezután egy összehasonlítást 4 mutatok be mind a BlackJack, mind a GameOfLife projektekkel kapcsolatban, melyben a két nyelv különbségeit és hasonlóságait mérem össze.

2. fejezet

A Lua nyelv és sajátosságai

2.1. Kicsit a Lua-ról

A Lua egy programozási nyelv, amit 1993-ban Roberto Ierusalimschy, Luiz Henrique de Figueiredo és Waldemar Celes fejlesztett a Pontifical Catholic University of Rio de Janeiro egyetemen Brazíliában. A Lua portugálul holdat jelent és a helyes kiejtése "LOO-ah".

A Lua előnyei a bővíthetőség, egyszerűség, hatékonyság és hordozhatóság. Könnyű hozzá új metódusokat/funkciókat és modulokat. Gyakori szkript nyelv játék programokhoz. Az egyszerűsége ellenére a Lua nagyon erős ún. "multi-paradigm" programozási nyelv. Ezálzal több stípust is támogat, legyen az imperatív, funkcionális vagy objektum-orientált. Emellett a Lua-ban nem kell foglalkozni a memóriakezeléssel, ugyanis egy nagyon jó inkrementális garbage collector-ral rendelkezik. A Lua Pascal szerű szintaxist követ.

A másik nagy előnye a Lua-nak, hogy rendkívül gyors. A virtuális gép, amin a Lua 5.1-t használnak, az egyik leggyorsabb az olyan programoknál, amik szkript nyelveket használnak. Van még egy "just-in-time" fordítja is (a Lua-t a számítógép natív gépi kódjára fordítja le miközben fut a program), amely az x86 architektúrák számára lett tervezve és emiatt még gyorsabb.

2.2. Adattípusok és változók

2.2.1. Változók

A változó jelentése itt is annyit tesz, hogy egy érték tárolható benne, amire a neve segítségével lehet hivatkozni pl:

```
done = false
```

Az értékadás hasonló, mint a legtöbb nyelv esetében. Változó név szinte lehet bármi, de hasznos, ha beszédes nevet kapnak. Vannak azonban kulcs szavak a Lua-ban is, amelyeket nem szabad használni:

and	for	or
break	function	repeat
do	if	return
else	in	then
elseif	local	true
end	nil	until
false	not	while

Ami még érdekes (bár ez majd a többi rész után egyértelmű lesz), hogy ezeken a kulcsszavakon kívül a Lua nem használ mást. Nagyon érdekes, hogy ezekkel szinte mindent le tudunk írni. Emellett a változók kis és nagybetű érzékenyek, ami szintén hasznos tulajdonság.

2.2.2. Típusok

A Lua-ban alapértelmezetten 7 típus van (mondhatni ezek a beépített típusok): **nil, number, string, boolean, table, function és thread**. Szintén érdekesség, hogy ezzel a 7 típussal szinte mindent le tud fedni. Gyorsan vegyük sorra az egyes típusokat és jellemzőiket.

nil

A nil típus lényegében azt jelenti, hogy valaminek nincs hasznos értéke. Például, ha egy változó nincs inicializálva, akkor az alapértelmezett típusa nil.

```
...
 person
 print(typeof(person)) --> nil
 person = 5
 print(typeof(person)) --> number
...
```

Egy másik eset például egy függvény meghívásánál, ha nem adunk át elég paramétert egy függvénynek (pl: 3 paramétert vár és csak 2-t kap), akkor az a változó nil típusú lesz. Ez a típus tényleg a "különleges állatfajok" világába tartozik, ugyanis nem egyenértékű például a Java vagy a C++ null típusával. Ugyanakkor Nem is tekinthető default value-nak, mivel a nil típus csak a nil értéket értelmezi.

Talán a JavaScript tudná a legjobban körülírni. Ott ugyanis egy közeli fogalom létezik: **undefined**. Az undefined az inicializálatlan változót jelenti. A nil vegyíti a null és az undefined fogalmát és használatát (mondhatni ez a két fogalom nem vált ketté a Lua-ban).

number

A number is szintén egy érdekesen viselkedő típus. Míg más nyelvekben, mint például a C++ vagy a Java külön típusok vannak az egész és a tört számokra, addig a Lua-ban ezek mondhatni egyesítve vannak a number típusban. Ha kell Integer-ként viselkedik, ha kell, akkor Double vagy Float-ként. Alapértelmezésben minden szám Double ($-1.79 * 10^{308}$ és $1.79 * 10^{308}$ közötti értékek).

Akárcsak a Java-ban vagy a C#-ban, itt is lehetőség van szövegből számot csinálni. Erre való a **tonumber(str)** függvény.

```
...
print(tonumber("5")) --> 5
print(tonumber("-100")) --> -100
print(tonumber("0.5")) --> 0.5
print(tonumber("-3e5")) --> -300000
...
```

string

A string teljesen hasonlóan viselkedik a Java vagy a C# stringjéhez képest. Azokhoz hasonlóan itt is "immutable"-ö, azaz módosíthatatlanok. Itt is a Lua a szövegeket egy string pool-ban tárolja. A változó sosem tárolja az értéket, csupán a referenciát a string poolra. Az escape karakterek teljesen ugyanúgy működnek, mint más nyelvekben. Talán ez a típus a legsablonosabb mind közül, hisz a Lua-ban nincs plusz feladata.

Fontos még kitérnem egy tipikus tulajdonságra, amely sok nyelvbe eltérő: két string összeadása. Maga az a tény, hogy egy új string jön létre, az majdnem minden nyelvben ugyanaz (a Lua-ban is), viszont az összeadás módja eltér. Nem véletlenül használom az 'összeadás' kifejezést, hiszen a legtöbb nyelvben (pl. Java, Kotlin, C#) két stringet az összeadás operátorral kötünk össze. Lua-ban viszont ez eltér. Itt érdekes módon nem szokás az összeadás operátort használni. Helyette a **..** operátort használja:

```
str = "hello".."world"
print(str) --> hello world
```

Valamint fontos megjegyezni, hogy a Lua-ban ez az operátor impliciten átalakítja a nem string típusú változókat string-gé (ha lehet, pl. number-t igen). Így mindenféle probléma nélkül lefut a következő kód is:

```
str = "hello␣"..1.."␣world"
print(str) --> hello 1 world
```

Ez igen kényelmes és hasznos funkció, amely sok nyelvben gyakran plusz függvények meghívásával tudunk elérni. Illetve komplexebb típusoknál (kifejezetten table objektumokra értem), ha felül van definiálva a **__tostring** operátor (erről a 2.9.1. fejezetben fogok részletesebben beszélni.) , akkor automatikusan átalakítja string típusúvá.

Utoljára még megemlíteném a hossz operátort, amely Lua-ban kicsit eltér a más nyelvekben megszokottaktól. Java-ban vagy C++-ban egy külön függvényt definiálunk erre és nem operátort. Míg C#-ban property segítségével tudjuk lekérdezni egy string hosszát. Lua-ban erre a # operátort használjuk.

boolean

Érdemben teljesen hasonlóan működik ez a típus, mint minden más nyelvben. Két értéke lehet: a **true** és a **false**.

function

A function típus követi a szkript nyelvek közös vonásait. Függvény kétféleképpen lehet megadni:

```
function foo(arg1, arg2)
        print(arg1)
        print(arg2)
end
```

vagy

```
foo = function(arg1, arg2)
        print(arg1)
        print(arg2)
end
```

Igazából az első példa csak egy egyszerűbb leírása a másodiknak. Valamint az első példa alapján azt feltételeznénk, hogy van egy függvény, amit foo-nak hívnak. Valójában a második példa alapján jól látszik hogy egy változót hozunk létre, amely tárol egy függvényt. Magának a függvénynek nincs neve, mert a Lua-ban az összes függvény anonymus function.

```
foo = function(arg1, arg2)
        print(arg1)
        print(arg2)
end

bar = foo
```

Így már két változó mutat ugyanarra a függvényre. De fontos, hogy a függvény soha nem másolódik, csupán a referencia (az ilyen dolgot megszokottak pl. a Java és a C# világában, hiszen ott csak referencia típusú átadás van, de a C és C++-ban előfordul érték szerinti átadás). A függvények viselkedéséről még a későbbiekben (2.8) kitérek.

thread

A thread név ennek a típusnak kicsit csalóka lehet, ugyanis merőben eltér a Java vagy a C# Thread típusaitól. Lua-ban ugyanis nem valósul meg a többszálúság... Egy sokkal jobb megnevezés a **corutine** (cooperating routine). Egy corutine tulajdonképpen egy függvénynek felel meg 1 különbséggel Két plusz tulajdonsággal rendelkezik: tudnak **yield**-elni és vissza lehet tölteni őket. A yield függvény nagyjából a Java-ban a Sleep metódusnak felel meg, míg a visszatöltés automatikusan megtörténik, azzal nem kell foglalkozni.

A különbségre a magyarázat már a névből is kikövetkeztethető valamint a számítógépek ütemezőivel lehet jellemezni. Kétféle típusú lehet egy ütemező: preemptív, azaz elveheti egy száltól a futás jogát (ezzel megszakítva a jelenlegi futását), vagy kooperatív, ebben az esetben az ütemezőnek nincs joga elvenni a futás jogát.

Igazából a Lua-ban az utóbbi valósul meg, tehát nem igazán beszélhetünk többszálúságról, hiszen a szálak nem küzdenek a processzorért, ugyanakkor, ha épp nincs szükség, átadhatják a futás jogát (yield) és majd visszatérnek, ha nincs más várakozó. A corutine-ok menedzselésére a Lua-nak van egy beépített könyvtára: **corutine** névvel. Egy példa segítségével megpróbálom gyorsan bemutatni, hogy mik is ezek a corutine-ok:

```
foo = function()
        for i = 1, math.huge do
                corutine.yield(i)
        end
end

local co = corutine.wrap(foo)
print(co()) --> 1
print(co()) --> 2
print(co()) --> 3
```

Ismételten hasznos megjegyezni, hogy nincs többszálúság, az a fogalom, hogy a főprogram szála, nem létezik. Ugyanakkor meg lehet valósítani többszálúsági problémákat, mint például egy consumer-producer problémát, bár a Lua nem erre lett kitalálva. Azonban jó látni, hogy ha kell, erre is nyújt egyfajta lehetőséget a nyelv (sajnos elég gyorsan túlkomplikált és nehezen érthető lesz miattuk a kód).

table

A végére marad a Lua talán legérdekesebb és leghasznosabb, valamint legfontosabb típusa. A table típus segítségével tudunk implementálni bármiféle adatstruktúrát. Az alapokat itt felvázolom, de a későbbi részekben (2.9) kitérek még a sajátos viselkedésére.

Nagyon leegyszerűsítve a table egy kulcs-érték párokat tároló tömb. Nagyon hasonlít a map-re és az array-re, de valójában egyik sem és mindkettő egyszerre. Rendkívül különleges már csak az adattárolási lehetőségei.

```
table = {"a", 7, false, -1}
```

Ha egy table-t szeretnénk megadni, akkor azt evvel = { } a módon adhatjuk meg. A példában jól látható, hogy nem számít mit tárolunk egy table-ben. Ugyanakkor működik természeten az index operátor:

```
print(table[1]) --> a
print(table[3]) --> false
print(table[5]) --> nil
```

Itt két érdekesség figyelhető meg. Az első, hogy a számozás nem 0-tól, hanem 1-től indul. Ez tipikusan zavaró tud lenni a C, C++, Java vagy akár Kotlin nyelvek után, de nem egyedi (pl.: FORTRAN, SASL, MATLAB, Erlang). A másik, hogy míg csak 4 elem volt a táblában meg tudtam hívni az 5. indexűt. Mivel értelmes értéket nem tároltam benne így nil típusú az az érték, amit visszaadott és az értéke is nil.

Ami viszont érdekesség, az a következő példán látszik:

```
table[1000] = "z"
print(table[1000]) --> z
```

Ekkor azt gondolnánk, hogy a table újraméretezte magát, de valójában ben ez történt. És itt mutatkozik meg a table szépsége és sokoldalúsága. Ekkor ugyanis a table szétválik két részre. Egy iterálható és egy tisztán kulcs-érték párokon alapuló részre. Hogy mit is értek kulcs-érték párokon, azt a következő példa megmutatja:

```
table = { key1 = "a", ["egy␣kulcs␣szokozokkel"] = "b",
          [5] = 1, [print] = false }
```

A táblám tartalmaz egy key1 kulcsot, aminek az értéke 'a', ez még szinte alap is, de a többi példa már érdekesebb. A [5] nem ugyanaz, mintha egy szöveg lenne. Így tudok explicit indexet létrehozni. De akár függvénynevet (azaz egy változó nevet) is megadhatok kulcsként.

```
print(table["key1"]) --> a
print(table.key1) --> a
print(table[print]) --> false
```

A kulcs-érték párokat a fenti példán látva kétféle módon tudom meghívni. Míg az iterálható elemeket a szokásos módon. Visszatérve a korábbi példára, ahol az 1000-ik indexet adtam meg, már jobban érthető, hogy hogyan is lehetséges ez és hogy valójában mit is jelent.

Az iterálás lehetősége ezekben a table típusú objektumokban kétféle módon lehetséges. Az első a jól megszokott index szerinti iterálás. Csak ezzel van egy kis probléma. Ugyanis, ha az indexek valahogy így néznek ki: 1,2,3,5,7,1000... akkor mi is a szabály. Ez pedig nagyon frappánsan van kitalálva a Lua-ban.

Amíg tud egyesével iterálni, azaz vannak indexek, amiken végig tud menni, addig azokon az elemeken végigmegy, amíg tud. Ha azt akarom, hogy az első 5 elemet iterálja, akkor a 4. indexűnek is értéket kell adni. Ekkor már el tudok iterálni az 5. indexűig is. Ekkor felmerül a kérdés, hogy mi lesz a többi elemmel? Például azokkal, amik kulcs-értékként vannak tárolva. Mivel a table hátterében igazából egy hash map áll, így azokon az elemeken is végig tudunk menni. Ez a példa bemutatja először az index majd a hash map szerinti iterálást:

```
table = {1, 2, 3, key1 = "a", ["egy␣kulcs␣szokozokkel"] = "b",
         [5] = 1, [print] = false, [1000] = "k" }

for k,v in ipairs(table) do
        print(k, v) --> [1] 1, [2] 2, [3] 3
end

for k,v in pairs(table) do
        print(k, v) --> [key1] a, [2] = 2, ...
end
```

Az első for ciklusnál jól látható, hogy csak az indexelt elemek jelennek meg. A többit így nem tudjuk kilistázni. Ahhoz a másik for ciklusra van szükség. A kettő szinte teljesen megegyezik 1 különbséggel: az elsőben ipairs, míg a másikban pairs a függény neve. A kettő lényegében, annyiban különbözik, hogy az ipairs az iterálható elemek tömbjén megy végig, míg a sima pairs a hash mapen. Azért nem írtam a másodikhoz a teljes kimenetet, mert ez előre nem meghatározható. Ugyanis ez a hash sorrendjében járja be a table elemeit és az a sorrend nem feltétlenül egyezik meg az én deklarációmmal.

Az értékadás hasonlóan működik, akárcsak a stringeknél. Itt is referencia szerinti átadás történik. Fontos megjegyezni, hogy ettől még a Lua-ban nem csak referencia szerinti átadás van. Például a number típusú változók értékeit másolja. Mondhatnánk, hogy a number primitív típus, de ezt már megtárgyaltuk, hogy ez nem teljesen lenne igaz.

2.2.3. Összegzés

Annyit még mindenképpen meg kell említenem, hogy az egyes változók a nincsenek egy konkrét típushoz kötve. Ez azt jelenti, hogy a Lua egy "dynamically typed" nyelv. Ellentétben a C, C++, Java vagy Kotlin nyelvektől, amik erősen típusosak. A dinamikus típusosság jellemző a szkript nyelvekre, amely rendkívül nagy rugalmasságot biztosít, ugyanakkor megvannak a hátulütői. Ugyanis nagyon könnyű elveszni, hogy egy változó mögött mi is áll (egyen talán áll-e valami). Szerencsére a Lua-nak van egy egész hasznos error generálója, amely értelmes üzenetek tud adni és egy backtrace-t is nyújt.

2.3. Kifejezések

A kifejezés nagyjából annyit jelent, hogy operátorokat használ különböző értékeken és kiértékeli egy értékké. Lua-ban bárhol lehet használni kifejezéseket, ahol a Lua egy értéket vár. Pl. változó

értékadásakor vagy függvény paramétereként.

2.3.1. Aritmetikai operátorok

Lua-ban is megvannak az alap aritmetikai operátorok: `+` összeadás, `-` kivonás, `*` szorzás, `^` hatványozás, `/` osztás és a `%` moduló operátorok. Ezek csak number2.2.2 típusú változókkal és értékekkel működnek. Ezen kívül még létezik a negációs operátor, amely szinte teljesen megegyezik a mínusz operátorral, de ennek csak egy paramétere lehet:

```
a = 5
print(0 - a) --> -5, normal minusz
print(-a) --> -5, unary
```

Elsőre ennek nincs jelentősége, de a precedenciáknál fontos különbségnek számít.

2.3.2. Relációs operátorok

Lua-ban a relációs operátorok nagyjából teljesen megegyeznek más nyelvek operátoraival: `==` egyenlőség, `<=` kisebb-egyenlő, `>` nagyobb, stb. Ugyanúgy boolean2.2.2 értéket adnak vissza. Egyedül a nem egyenlő operátor tér el Java-tól vagy a C, C++-tól megszokottaktól (ami a `!=` forma). Itt ugyanis `~=` a jelölés. De ez csak jelölésben tér el, használatban teljesen azonos.

2.3.3. Logikai operátorok

Az operátorok harmadik típusába három logikai operátor tartozik: `and, or` és `not`. Ami érdekes, hogy az `and` és az `or` operátor bármilyen típus között működik és bármilyen típust vissza tud adni, míg a `not` csak boolean típust ad vissza. Ezek az operátorok minden értékhez igazat rendelnek, kivéve a false-t és a nil-t. Ez a szkript nyelveknél gyakori, de abban általában eltérnek, hogy melyik értéket tekintik false-nak és melyiket true-nak. Pl a JavaScript-ben a 0 false-ként viselkedik és az üres string is, míg a Lua-ban ezek mind true-ként. De talán a legérdekesebb a 0 értelmezése, hiszen a legtöbb nyelv (min pl a C) gyakran kihasználja, hogy a 0-t false-nak tekinti.

Még egy érdekesség a "logikai rövidzár" fogalma. Azaz a logikai operátor második tagját nem értékeljük ki, ha nincs rá szükség. Ez az `and` operátornál azt jelenti, hogy ha az első értéke false, akkor a másikat már ki sem kell értékelni, hiszen a kifejezés biztos false lesz. Az `or` esetében, ha az első kiértékelés után true-val tér vissza, akkor szintén nem kell a másikkal foglalkozni, hiszen a kifejezés értéke biztos igaz lesz. Ez a legtöbb nyelvben ugyanúgy jelen van és gyakran ki is használják, a Lua-ban pedig különösen.

Az alapértelmezett érték (default value) fogalmát a legtöbb nyelv kihasználja. Ennek egy tipikus esete a függvények meghívásánál jön elő. A lényeg, hogy a függvény egyes paramétereinek értékét nem feltétlenül szükséges megadnunk vagy szinte minden esetben ugyanazok, emiatt hasznos dolog lenne, hogy egy alapértelmezett értéket be tudjunk nekik állítani, így a függvény meghívásánál ha nem szükséges, akkor nem kell megadni.

A legtöbb nyelv ezt támogatja, de megvalósítása gyakran eltérő. C++ és C# egyszerűen a változó mellé egy egyenlőségjel után odaírjuk az alapértéket, addig például a Java-ban erre nincs lehetőség. Ott function overload-ot használják ki, ami nagyjából abból áll, hogy meghívják a kevesebb paraméterű függvényt, ami továbbhívja a teljes paraméterlistával rendelkező alapfüggvényt, csak már kiegészítve a hiányzó paraméterek alapértékével. De vannak olyan nyelvek, mint például a C, ami nem támogatja ad default értéket függvényeknél. Ott külön függvényeket kell létrehozni minden esetre.

A Lua-ban alapértelmezett értéket a függvény fejlécében nem tudunk megadni, de ezzel ellentétben ki tudjuk használni a logikai operátorokat és az eddigi ismereteinket a függvények paramétereiről. Ha egy függvény paraméterének nem adunk értéket, akkor az attól még meghívódik. Ekkor felmerül a kérdés, hogy mi lesz annak a paraméternek az értéke. Erre használjuk tipikusan a nil típust. Azaz minden "undefined" paraméter értéke nil lesz. Ezt pedig a logikai operátor használja ki. Ugyanis a nil értéke false-nak minősül.

```
function new(value, color, number, hidden)
        local newCard {}
        newCard.value = value
        newCard.color = color
        newCard.number = number
        newCard.hidden = hidden or false
        return newCard
end

card = new("KING", "SPADES", 11)
```

Ebben az esetben a hidden értéke nil lesz. Amit az értékadásnál a logikai operátor kihasznál, így bármi is áll a jobb oldalon, az az érték fog beleíródni a table ”hidden” attribútumába. Ha a függvényt 4 paraméterrel hívjuk meg, akkor pedig a logikai rövidzár miatt az átadott paraméter hívódik meg. Itt fontos megjegyezni, hogy egy kivételes esetre komolyan fel kell készíteni a függvényt. Ha false értéket adunk át, jól látszik, hogy a rövidzár akár kellemetlenül is hathat az értékre. Ezt egyedül jó meggondolással lehet csak kiküszöbölni (azaz mit írjunk az kifejezés jobb oldalára).

2.4. Vezérlő struktúrák

2.4.1. Elágazások

A Lua elágazások és ciklusok terén eléggé szűkre szabott. Elágazásokból az if-else állítás értelmezett, de körülbelül ennyi. A gyakran használt switch-case megoldásra nincs külön típus, sajnos esle if-ek segítségével lehet csak megoldani. Mindazonáltal fontos megemlíteni a szintaxist.

```
i = 5

if i > 2 then
 --> do smoething
end

if i < 4 then
 --> if statement is true
else
 --> if false
end
```

Látható, hogy nem kapcsos zárójeleket használ, hanem `then - end` párokat. Valamint, ha a kifejezés nem összetett, akkor a feltétel zárójelezése is elhagyható.

Fontos különbség azonban, hogy az `else if` és az `elseif` nem egyezik meg. Ugyanis az `else if` egy új blokkot nyit, míg az `elseif` az eredetit használhatja:

```
i = 5

if i > 2 then
--> do smoething
end

if i < 4 then
-->  if statement is true
elseif i > 6 then
--> ha hamis
                        --> nincs 'end' tag
end

if i < 4 then
-->  if statement is true
else --> az 'else if' szebb alakja
        if i > 6 then
                --> ha hamis
        end --> kell 'end' tag
end
```

Így látható, hogy a switch-case megoldást a Lua az `elseif` segítségével tudja megvalósítani:

```
if(result == ResultType.WIN) then
        print("You␣win!")
        playerOdds = 2
        bankOdds = -1
        self.winCount = self.winCount + 1
elseif(result == ResultType.WINBYJACK) then
        print("BlackJack!␣You␣win!")
        playerOdds = 2.5
        bankOdds = -1.5
        self.winCount = self.winCount + 1
elseif(result == ResultType.TIE) then
        print("It's␣tie!")
        playerOdds = 1
        bankOdds = 1
elseif(result == ResultType.LOSE) then
        print("You␣lost!")
        playerOdds = 0
        bankOdds = 1
        self.loseCount = self.loseCount + 1
elseif(result == ResultType.LOSEBYJACK) then
        print("You␣lost!")
        playerOdds = -0.5
        bankOdds = 1.5
        self.loseCount = self.loseCount + 1
else print("error")
end
```

2.4.2. Ciklusok

A Lua-ban jelen van a három megszokott ciklus: a **for, while, do – while**. Habár az utóbbi szintaxisa kicsit eltér a megszokottól. Itt is, mint az elágazásoknál, kicsit rendhagyó a szintaxis a megszokott kacsos zárójelektől, ami tipikusan jellemző a C, C++, Java vagy akár a Kotlin világában.

```
for i = 1, i <= 10, 1 do
        print(i)
end
```

Itt is felvehetünk egy ciklusváltozót, amely lokálisan lesz jelen a ciklus törzsében, akárcsak a megszokott módon pl. C++-ban. Itt is a második a megállási feltétel, míg a harmadik az iteráció nagysága. A zárójelezést a ciklusok esetében a **do – end** páros alkotja. Valamint fontos megjegyezni, hogy az elválasztások itt nem ;-t használnak, mint a legtöbb nyelvben (C, C++, Java), hanem csak sima vesszőt.

Így néz ki a tipikus for ciklus, de persze megannyi módon tudjuk változtatni. Alapesetben, ha egyesével iterálunk növekvő számokkal, akkor elhagyható a harmadik tag:

```
for i = 1, i <= 10 do
        print(i)
end
```

Emellett még egyszerűsíteni is lehet a megállási feltételen, ha a ciklusváltozó és a feltétel típusa megegyezik:

```
for i = 1, 10 do
        print(i)
end
```

Legtöbbször persze a for ciklust arra használjuk, hogy egy listán/tömbön végig iteráljunk. Egy korábbi példában már utaltam arra, hogy ez lehetséges:

```
for key, value in pairs(table) do
        print(key, value)
end
```

Ez a példa tipikusan a ”foreach” megfelelője. A ”pairs” függvény visszaad egy kulcs-érték párokat tartalmazó table-t és az ”in” kulcsszó segítségével kulcs-érték párokat ad értékül a ”key” és ”value” változóknak. Ezt annyiszor teszi meg, ahány elemmel tér vissza a pairs függvény által visszakapott table. Azt, hogy a két értéknek hogyan adhat így értéket, arra a későbbiekben visszatérünk2.5.

A **while** ciklus teljesen hasonlóan viselkedik a C, C++ vagy a Java-s barátaihoz képest:

```
i = 1
while i <= 10 do
        print(i)
end
```

Azonban a **do** − **while** kicsit eltér. Lua-ban ugyanis a **repeat** − **until** kulcsszavakat használjuk. Ezen kívül a logika tejesen megegyezik a megszokott viselkedéssel:

```
i = 1
repeat
        print(i)
until i > 10
```

Fontos megjegyezni, hogy itt nincsen `do -- end` blokk.

Még egy fontos dolog a ciklusokkal kapcsolatban, hogy értelmezzük a **break** fogalmát Lua-ban is. Teljesen hasonlóan viselkedik, mint a többi nyelvben: megszakítja a ciklust és kilép belőle az adott ponton:

```
table = {"a", "b", 5, 10, 100}
for k, v in pairs(table) do
        if(v == 5) then
                key = k
                break
        end
end
print(key) --> 3
```

Itt két dolgot érdemes megfigyelni. Az egyik, amely a legszembetűnőbb, hogy nincsenek pontosvesszők a sorok végén. Ez már korábban is előjött, de talán itt érzem már fontosnak megemlíteni, hogy miért is lehet elhagyni őket. Mivel a Lua egy szkript nyelv, a fordítása interpretálva történik, azaz sorról sorra (pontosabban kifejezésről kifejezésre) futtat. Így egy sor vége elég jelzés , nem kell külön pontosvesszővel jelezni.

A másik, nem annyira szembetűnőbb, hogy a ”key” változót ott hoztam létre a cikluson belül, mégis látszik azon kívül. Ennek okára szintén később kitérek2.6.

2.5. Értékadás és visszatérési érték

Az értékadásra azért térnék ki részletesebben, mert a Lua támogat egy plusz ”feature”-t a többi nyelvekhez képest. Ez pedig a többszörös értékadás fogalma. Fontos megjegyeznem, hogy ez a fogalom nem idegen a C, C++ és Java nyelveknél sem, de ott a jelentése, hogy ott ugyanazt az értéket több változónak egyszerre megadhatjuk. Lua-ban viszont kicsit eltér a jelentése. Ugyanis Lua-ben több változónak több különböző értéket adhatunk meg egyszerre.

```
var1, var2, var3 = 1, "a", true
print(var1)  --> 1
print(var2)  --> a
print(var3)  --> true
```

Ha több értéket akarunk megadni a kevesebb változónak, akkor egyszerűen a fel nem használt értékek elvesznek. Nem íródnak be sehova. Ha több a változó, mint az érték, amit értékül adnánk, akkor a változó értéke alapértelmezetten nil lesz.

```
var1, var2 = 1, 1+1, 1+2, 1+3
print(var1)  --> 1
print(var2)  --> 2


var1, var2, var3 = 1, "a"
print(var1)  --> 1
print(var2)  --> a
print(var3)  --> nil
```

És ez hol tud hasznos lenni? Hát például a függvények visszatérési értékeinél. A legtöbb nyelvben egy függvény csak egy dolgot tud visszaadni, legyen primitív típus vagy egy objektum (C#-ban például data objectet is visszaadhatunk, ami adatbázis entitások lekérdezés eredményhalmazát alkothatja, amelyeket dinamikusan is létrehozhatunk, előre nem definiált adatstruktúraként), de attól az még 1 elemnek számít. Lua-ban viszont ez nem kikötés. Felfoghatjuk, hogy értékek listáját adja vissza egy függvény.

Használata rendkívül kényelmes:

```
function foo()
        return 1,2,3
end

var1, var2, var3 = foo()
print(var2, var3) --> 2  3
```

Egy dologgal azonban tisztában kell lenni: hogy az a függvény hány paraméterrel tér vissza. Erre általában a függvény magírója figyelmezteti a használót, így elkerülendő a rossz/nem-kívánt használat.

Viszont emellett vannak további kényelmi funkciói. Például, ha nincs szükségünk minden értékre, amit visszaad egy függvény. Ezeket kétféle módon is megoldhatjuk. Az egyik eset, ha a nemszükséges értékek az utolsók. Ebben az esetben egyszerűen nem veszünk fel annyi változót. Csak a szükségeseket:

```
function foo()
        return 1,2,3
end

var1, var2 = foo()
print(var1, var2) --> 1   2
```

Ebben az esetben a '3' elveszik.

A másik eset, amikor a köztes értékekre nincs szükségünk. Ekkor persze tisztában kell lennünk a visszatérési értékek sorrendjével. De emellett a Lua úgy oldja meg a fölösleges értékek kezelését, hogy bevezet egy ún. **anonymus** változót (dummy varialbe). Ezt "_" szokás jelölni (ilyen jelölést a Kotlin lambda kifejezéseinél szokás használni a nem használt változók jelölésére).

```
function foo()
        return 1,2,3
end

var1, _, var3 = foo()
print(var1, var3) --> 1   3
```

Kiemelném még egyszer, hogy ez rendkívül hasznos tulajdonság mind változó értékadás, mind függvény visszatérési értékével kapcsolatban. Hiszen evvel lehetőség nyílik, hogy egy függvény több dolgot is végezzen egyszerre és a különböző (rész)eredményeket egyszerre adja vissza. Az eredményt pedig aszerint szortírozza a használó, amire éppen szüksége van (nem OO megközelítés, de mivel a Lua multi-paradigm nyelv2.1, ezért van ilyen lehetőség is).

2.5.1. Kiértékelés sorrendje

Muszáj kitérnem erre a részre is egy kicsit. Habár a Lua kiértékelési sorrendje teljesen megegyezik a megszokottakkal (először a jobb oldal értékelődik ki, majd a az értékadás a balnak), az előbbi téma tekintetében enne nagyon fontos jelentés van.

Legszebb példa erre két változó értékének a cseréje.

```
function shuffle()
        size = #deckPool --> deckPool is a table
        for i = size, 1, -1 do
                rand = math.random(size)
                deckPool[i], deckPool[rand] = deckPool[rand], deckPool[i]
        end
end
```

Ezen a példán nagyon szépen látszik a hasznossága a többszörös értékadásnak. Először a jobb oldal kiértékelődik és csak utána történik meg az értékadás. Ezzel lehetőségünk van az értékcserét egy sorban megoldni. Ez a legtöbb nyelvben kn. így nézne ki:

```
a = 1, b = 2
temp = a
a = b
b = temp
-- in lua:
a, b = b, a
```

2.6. Lokális Változók

Az eddigi példákban ügyeltem arra, hogy csak egyfajta változót mutassak be. Tehát az eddigi változók mind globális változóként voltak használva. Ez a fogalom teljesen azonos a C, C++ világában megszokottaktól. De persze ezeket tipikusan szeretik elkerülni csaknem minden program írásakor a programozók, ha lehet. Lua-ban is ez a tipikus helyzet.

Az általános "konvenció" erre az, hogy csak a könyvtárak és a közös használatra szán modulok legyenek globálisak. Minden más lehetőleg legyen lokális változó. Ennek okai kb. megegyeznek a C, C++ nyelv problémáival: szabadon írhatók, így az egyes objektumok állapotát elronthatják, stb. Ráadásul lua-ban még a típus sem ismert, így lehet, hogy egy teljesen más dolgot ír át az ember, mint amit akar (erre egy megoldás lehet a beszédes változónév, de még ez sem teljes megoldás).

Tehát ha lehet akkor használjunk lokális változókat. De hogy ez mit is jelent? A fogalom teljesen azonos a más nyelvekben használt lokális változó fogalmával (csak egy bizonyos blokkon/környezeten belül definiált és használható változó). Ide legtöbb esetben függvények scope-ját értjük. De Lua-ban értelmezhetünk saját scope-ot is a **do** − **end** szintaxis segítségével, ha szeretnénk.

```
local x = 5
if x == 5 then
        print(x) --> 5
        local x = 1
        print(x) --> 1
end
print(x) --> 5
```

Mivel Lua-ban sorról sorral olvasunk, így nem tudjuk minden esetben a függvény scope-ját kihasználni, hogy felismerje, hogy az ott létrehozott változó legyen lokális vagy sem. Ehelyett a **local** kulcsszóval mondjuk meg Lua-ban, ha egy változót lokálisnak tekintünk.

Jól látszik, hogy az if-en belül definiált 'x' értéke az if scope-ján kívül nem marad meg. Csakúgy, mint más nyelvekben:

```
if expression then
        local x = 1
end
print(x) --> nil
```

Egy másik nagyon fontos megjegyzés a függvények használata. Függvényt kétféle módon tudunk definiálni2.2 és azt is tudjuk, hogy az egyik igazából a másik egy egyszerűbb alakja. Viszont, van egy nagy eltérés, ha lokális függvényt definiálunk:

```
local function foo()
end

local foo = function()
end
```

Az első esetben a függvény neve a függvény scope-jába tartozik, míg a második esetben nem. Ez például egy rekurzív függvénynél rendkívül fontos dolog, amelyre figyelni kell. Hiszen, ha a második módon írjuk meg, akkor hibát fogunk kapni, mert a függvény törzsében a foo-t globális változónak veszi és mivel a jobb oldali érték előbb értékelődik ki, így a foo meghívása még egy olyan változó használatát jelentené, ami eddig lokálisan még nem létezik, így automatikusan globálisnak veszi. Ekkor viszont nem tudja, hogy ez egy függvényt tárol.

2.6.1. Változók listája

Ha már egyszer újra elővettük a változókat, akkor érdemes még egy nagyon fontos dolgot megemlíteni, ez pedig a váltorúk listája (varargs).

```
local function foo(...)
end
        print(...)
end
local a,b,c = 1,2,3
print(foo(a,b,c)) --> 1   2   3
```

Ez a fogalom létezik C++ világában, ott a template meta-programozásnál játszik fontos szerepet, de megjelenik Java-ban is. Ez igazából egy változó, ami mondhatni ismeretlen hosszúságú változók listáját kezeli egy elemként. Ahogy az említett nyelvekben, a Lua-ban is ugyanaz a szerepe. Lokális változóként van kezelve és a használata nagyban függ attól, hogy a programozó mit is szeretne vele csinálni. Ugyanis, ha csak ezt a paramétert kapja, akkor pontosan tudnia kell, hogy mit takar.

Kicsit hasonló az értékadáshoz2.5. Tekinthetjük úgy, mint egy hosszú listát, amiből szeretnénk lecsípni valamennyit. Erre segítségül a Lua egy ún. **select(i, ...)** függvényt ad. Ez az első 'i' darab változót adja vissza a **'...'**-ból, így már tudjuk, hogy hány változónak fogunk értéket adni. Viszont az még mindig programozási kérdés, hogy mik ezek a változók.

A leggyakoribb használata mégis a függvénydelegálásnál jelenik meg. Hiszen ekkor a meghívott függvénynek a célja nem feltétlenül kapcsolódik a megkapott paraméterekhez, így azokat minden további nélkül továbbadja egy másik függvénynek, aki majd használja őket és tudja is, hogy miket kap.

```
foo = function(table,...)
...
        --> do something
...
        return table:new(...)
end
```

A példán jól láthatjuk, hogy a kapott paraméteren meghívunk egy további függvényt, melynek továbbadjuk a nem használt paramétereket. A kettőspont operátorra majd később kitérünk (2.10.1), egyenlőre fogadjuk el, hogy egy table típusú változónak egy new kulcsú elemét hívtuk meg, mivel az egy függvény.

2.7. Lua standard könyvtárai

Korábban már volt szó arról, hogy mit használunk lokális2.6 és mit globális változóként. Akkor említettem, hogy a kivételes esetek közé tartoznak a Lua beépített könyvtárai. Ezekről említenék meg egy gyors összefoglalást.

Math library

Ez teljesen azonos a más nyelvekben használatos Math könyvárakhoz. Itt nem kell sem #include-olni (mint C++-ban) vagy importálni (mint Java-ban), hanem alapból használhatjuk a **math** globális változón keresztül.

Minden tipikus és gyakran használt függvény definiálva van:

- math.abs
- math.exp
- math.deg
- math.cos
- math.sin
- math.log
- ...

De vannak konstansok is, mint pl. a **math.pi**. De ide tartozik a **math.random(x, y)** függvény is (x és y között generál egy random egészet, ha x = 0 és y = 1, akkor pedig float-ot generál).

String library

A **string** típusú változókat a legkönnyebb és legpraktikusabb a string könyvtárral manipulálni. Ez is teljesen megegyezik pl. a C-ben használt tipikus függvényekkel:

- string.len
- string.lower
- string.upper
- string.match
- string.join
- string.reverse
- string.split
- ...

Egy nagyon gyakran használandó függvény a **string.sub(str, i, j)**. Ez természetesen a substring megfelelője, de azért fontos megemlíteni, mivel a string nem indexelhető, ezért az egyetlen módja, hogy visszakapjuk a string egy karakterét, ha ezt a függvényt alkalmazzuk (ebben az esetben az i = j). Később (2.9.1) még lesz szó az index operátorról, amikor is kis csellel és némi következmény árán megoldható egy string indexelése.

Table library

Talán ez egyik legfontosabb könyvár a **table** library, amellyel strukturálisan tudjuk manipulálni a table objektumokat. Mivel a table, mint korábban kiderült2.2.2, egy különös állatfaj, mégis leginkább egy list/map-hez lehet hasonlítani. Más OO nyelvek gyakran definiálnak tömb-manipuláló függvényeket és Lua-ban erre szolgál a table könyvtár:

- table.insert
- table.remove
- table.wipe
- table.sort
- ...

Ez a könyvár elsősorban az 'insert' és 'remove' függvények miatt hasznos. Ezek ugyanis elrejtik előlünk a táblaindexelés aggasztó ismeretét. Tehát nem kell tudni, hogy egy tábla meddig van indexelve és onnan folytassuk tovább a következő index megadásával. Ezek a függvények pont arra jók, hogy ne nekünk kelljen számon tartani az egyes táblák jelenlegi indexeit (ameddig végig iterálhatunk rajtuk index, nem pedig hash szerint). Így utána kényelmesen használhatjuk az **ipairs** függvényt, ami azért hasznos, mert ha egy table-t tényleg listaként szeretnénk értelmezni, a sorrend gyakran fontos lehet.

Debug és System és IO library-k

Pár szó erejéig érdemes megemlíteni ezt a két könyvárat. A **debug** library-t gyakran segítségül hívhatjuk egy problémás kód hibájának megfejtésére (pl. a debug.traceback függvény). Ez hasznos tud lenni, hiszen fordítási nincs lehetőség jelezni a programozónak, viszont ezzel sokat tud segíteni, ha a minimálisnál több adatot ír ki hibaüzenetként.

A másik a **os** könyvtár. Az os library-hoz fordulhatunk például az idő lekérdezése végett, de akár írhatunk eseményeket, amiket szeretnénk, hogy a program kilépése során hívjon meg, mint például a biztos fájl bezárás (**os.exit(files.close())**).

A harmadik és talán leghasznosabb könyvtár az **io** library. E könyvtár segítségével tudunk fájlból olvasni és fájlba írni. Valamint a konzolra tudunk vele írni és olvasni is.

- io.open
- io.close
- io.flush
- io.write
- io.read
- io.lines

A 'write' függvény annyiban tér el a megszokott print-től, hogy a print mindig rak egy sorvége karaktert ha lefutott, míg az io.write nem. Az io.read függvényről még annyit kell tudni, hogy string-ként olvas egy egész sort. Ez majdnem minden más nyelvben így van, ahol dinamikus az olvasás (C-ben például meg kell adni az elvárt bemeneteket típussal pontosítva). A 'lines' függvényt pedig gyakran használjuk fálj sorainak beolvasására. Fontos, hogy nem soronként olvasunk vele. A lines először beolvassa az egész fájlt, és visszaad egy listát, amin végigiterálhatunk.

2.8. A függvények további sajátosságai

2.8.1. Static scoping

A Lua-ban a függvények mindig a saját kontextusukra hivatkoznak. Hogy ez mit is jelent, tekintsük a következő példát:

```
do
        local x = 5
        function foo()
                print(x)
        end
end
x = 1 --> 'x' global variable
foo() --> 5
```

Jól látszik, hogy hiába létezik egy globális változó ugyanavval a névvel, a függvény a saját környezete változóját használja. Azaz sosem arra a környezetre hivatkozik, ahol meg lett hívva.

A függvényeknél általánosságban igaz, hogy mindig "belülről kifele" halad. Azaz először a saját környezetében keresi a változókat, amiket használ és csak utána a globálisakat. Más nyelvekben, mint például C++ vagy Koltin, is hasonló működést tapasztalhatunk.

2.8.2. Closure

Még mielőtt belekezdenénk ebbe a témába, muszáj kitérnem, még egyszer a függvények visszatérési értékére. Az eddig bemutatottak mellett kiegészítem avval a tulajdonsággal, hogy függvény képes visszaadni egy másik függvényt is. Elsőre ez nem meglepő, hiszen a 'function' is egy típus és eddigi tudásunk alapján a függvény bármilyen típus(oka)t visszaadhat.

Nézzük a következő példát:

```
function counter()
        local x = 0
        return function()
                x = x + 1
                print(x)
        end
end
```

Itt a függvényünk létrehoz egy lokális változót majd visszatér egy függvénnyel. Mint azt az előző témában is bemutattuk, egy függvény a saját kontextusát használja ki.

```
c1 = counter()
print(c1()) --> 1
print(c1()) --> 2
```

Ezen a példán tejesen jól látszik a static scoping jelentősége.

```
c2 = counter()
print(c2 == c1) --> false
```

Érdekes megfigyelni, hogy a két változó értéke nem ugyanaz. De vajon miért? Hiszen ugyanazt a függvényt kapták vissza nem? Igazából nem. A **function() ... end** meghívása után egy új függvény jön létre mindig. Fontos megjegyezni, hogy ez nem ugyanaz az eset, mint a következő:

```
c3 = counter
```

Ekkor ugyanis nem fut le a function, csak a referencia másolódik.

Tehát a függvény meghívása egy teljesen új függvényt hoz létre. És vajon a környezete is új?

```
print(c2()) --> 1
print(c1()) --> 3
```

Ezen a példán jól látszik, hogy a számláló is különbözik a két függvénynél. Tehát nem csak egy új függvényt, hanem egy új lokális változót is létrehozott a Lua. Mind a két függvény különböző környezetet lát. Azokat a függvényeket, amik megőrzik a kontextusukat, **closure**-nek hívjuk. És azok a változók, amik "megőrződnek" a closure-ök által az ún. **upvalues** a Lua-ban. Mivel igazából

a **function() ... end** meghívása után mindig egy új closure keletkezik, ezért valójában az összes függvény a lua-ban closure-ként viselkedik.

Térjünk újra vissza arra, hogy egy függvény akár több dolgot is visszaadhat (akár több függvényt is), vagy akár egy table típusú objektumot, aminek az elemei a függvényünkben létrehozott újabb függvények. Ezek mind ugyanazt a kontextust látják, azaz ugyanahhoz a scope-hoz vannak kötve. Erre a tulajdonságra később visszatérünk (2.10.3) és akkor bemutatom mekkora szerepe van ennek például az objektum-orientáltságban.

2.8.3. Hooking

A függvények egy további hasznos tulajdonsága a hooking. Nyersen lefordítva nagyjából azt jelenti, hogy "beleakaszkodik". Ahhoz, hogy eszt megértsük egy egyszerű példán bemutatom a működését és így egyből értelmet nyer a jelentés:

```
do

        local old = print
        print = function(...) --> redefine print command
                return old(os.date("%H:%M:%S", os.time()), ...) --> new print
        end

end

print("Hi") --> 12:54:15 Hi
```

Tehát a hookingot arra használhatjuk, hogy meglévő függvényeink feladatát kiegészíthessük, plusz funkciókkal bővítsük. Ha jól megfigyeljük, ezzel akár a 'function overload'-ot is megvalósíthatjuk.

És az a szép a Lua-ban, hogy ezt bárhol megírhatjuk, ahol épp szükségünk van rá. Csak elmentjük egy lokális változóban az eddig használt függvényt, újradefiniáljuk a hooking segítségével és ha már nem használjuk, akkor visszaállítjuk a régit. Vagy, ha csak egy helyen akarjuk kihasználni, akkor az még egyszerűbb, hiszen a lokális környezett miatt máshol úgy sem látja a Lua ezt a "kiegészítést", így ott az eredetit használja. Viszont fontos, hogy tényleg lokális környezetben (akár closure-ban) legyenek és nem globálisan írjuk meg. Ezt a módszert akár wrapper függvényként is értelmezhetjük, de inkább egy függvény viselkedésének kiegészítésére használjuk.

2.9. Meta-táblák és meta-metódusok

Lua-ban minden értéknek van egy úgynevezett **metatable**-je. A metatable egy tipikus table objektum, ami képes különböző dolgok tárolására. A metatable-ket arra használják, hogy az egyes értékek vielkedését módosítsák. HOgy ez mit is jelent? Minden típusnak megvannak a maga standard operátorai és, hogy ehhez milyen függvények tartoznak. Ezeket az információkat tartalmazzák a metatable-ök.

Magán a table típuson kívül viszont minden más típus metatable-je megegyezik. Ez annyit jelent, hogy ha egy bizonyos string típusú változó összeadó operátorát akarná az ember átírni, akkor összes string operátorát megváltoztatja. A table típusú változók esetében minden table-nek saját metatable-je van (emlékeztető: a t = , leírása mindig új table-t hoz létre, egyúttal új metatable-t is).

A Lua beépített függvényekkel teszi lehetővé a metatable-ök elérését: **setmetatable(table, metatable), és getmetatable(table)**. A függvények kinézete a C nyelvre emlékeztet leginkább, ahol még nem voltak önálló objektumok saját viselkedéssel (függvényekkel). A 'setmetatable' segítségével az első paraméterként megadott table metatable-jét állítjuk be a második paraméterként megadott table-lel. Sajnos jól látszik, hogy ilyenkor az egész metatable-t felülírja (kicseréli). Tehát érdemes odafigyelni, hogy nem lehet sorba fűzni újabb és újabb operátor overload-okat ezzel a módszerrel.

```
local Player = {}

setmetatable(Player, {
                __call = function(class,...)
                        local newPlayer = class:new(...)
                        newPlayer.currentDeckIndex = 1
                        newPlayer.numberOfUsedDecks = 1
                        newPlayer.decks = class:createDecks(newPlayer)

                        return newPlayer
                end
})
setmetatable(Player, {
                __index = Participant
})

print( Player() ) --> error, attempt to call a table value (upvalue 'Player')


local Player = {}

setmetatable(Player, {
        __call = function(class,...)
                        local newPlayer = class:new(...)
                        newPlayer.currentDeckIndex = 1
                        newPlayer.numberOfUsedDecks = 1
                        newPlayer.decks = class:createDecks(newPlayer)

                        return newPlayer
                end,
        __index = Participant
})
```

Jól látható a probléma az első esetben, hiszen a **__call** operátor nem volt inicializálva a második setmetatable után, így a függvényhívás hatására nem volt semmi sem beállítva, ami hibát eredményezett.

2.9.1. Alapértelmezett meta-metódusok

Mint ahogy az előbbiekben elmagyaráztam a metatable-ben vannak az ún. meta-metódusok (továbbiakban metamethod). A metatable-ben speciális elemek vannak, amelyek tárolják a megszokott operátorok alapvető viselkedéseit. Ezeket a viselkedéseket függvények formájában értjük és őket hívjuk **metamethods**-nak.

A speciális kulcsokat a table-ben két aláhúzás karakterrel kezdjük és aztán jön a neve, amely utal arra, hogy milyen alapvető viselkedést definiál. A Lua a következő alap metamethod-okat szolgáltatja (ezek a metódusok két paramétert várnak):

- _add: + operátor
- _concat: a .. operátor (összefűzés)
- _div: / operátor
- _eq: == operátor
- _lt: < operátor
- _le: <= operátor
- _mul: * operátor
- _mod: % operátor
- _pow: ^ operátor
- _sub: – operátor
- _unm: negálás operátor
- _add: + operátor
- ...

Ezeken kívül van még 4 különleges operátor, amiket érdemes külön-külön kicsit jobban megismerni:

- _len: a **#** operátor. Ahogy a string-nél is megmutattam ez a hossz operátor, ami a string-ek esetében a szöveg hosszát adja meg (valószínűleg a string-len függvényt használva). Viszont gyakran szokás table típusú változóknál is használni, ugyanis alapértelmezetten visszaadja egy table-ben tárolt kulcs-érték párok számát. Azonban van egy fontos szabály: table objektum metatable-jére nem használható, mivel metatable-öket csak table-ökhöz csatolva tudunk használni.
- _call: a függvényhívás operátor. Fontos tulajdonság, hogy ez a függvény első paramétereként a table-t kapja és utána a többit, amit a függvénynek átadtak (a fenti példában 2.9 ez szemléletesen látszik).
- _newindex: ez a metódus akkor hívódik meg, ha megpróbálunk értékül adni egy új értéket egy olyan mezőnek, ami nincs a table-ben. Ez persze nem hívódik meg újra, ha már létezik az a "kulcs". Fontos megjegyzés, hogy az új mező felvétele megtörténik mindenképp, ha eddig nem létezett. Ez a függvény mint egy event-ként viselkedik (hasonlót C#-ban láthattunk).
- _index: ez a metódus akkor hívódik meg, ha hozzá akarunk férni a table egy mezőjéhez, ami még nem létezik. Fontos különbség a '_newindex'-hez képest, hogy itt nem történik értékadás, azaz nem is hoz létre új kulcs-érték párt. Itt megpróbálja elérni a "kulcs" által meghatározott értéket, és ha nincs ilyen kulcs a table-ben, akkor hívódik meg.

2.10. Objektum-orientáltság

Mint tudjuk, a Lua-ban megvalósítható az objektum-orientált programozás. A Lua támogatja az ún. prototipikus objektum-orientált programozást. Ezt pedig az _index metametódus segítségével éri el.

Nézzük, hogy mit is értünk prototipikus objektum-orientáltságon. Először is definiálunk egy table (továbbiakban táblaként is hívom), amiben függvényeket és különböző értékeket (attribútumokat) fogunk tárolni. Ez a táblát tekintjük az ún. **prototípus**nak.

2.10.1. Kettőspont operátor

Még mielőtt belekezdenénk, hogy hogyan is lehet megvalósítani az OO stílust Lua-ban, először térjünk ki egy fontos dologra a táblákkal kapcsolatban. Ez pedig a kettőspont operátor. Az eddigiekben, ha szerettünk volna egy függvényt felvenni a táblába, akkor egy változóban eltároltuk és így hivatkoztunk rá:

```
table = {}
table.foo = function()
        print(Hello)
end

table.foo() --> Hello
```

Mint egy tábla adattagja. Viszont van egy másik módszer, hogy definiálni tudjunk egy függvényt egy tábla attribútumaként:

```
Table = {
        tableView
}
function Table:noticeUpdate()
        self.tableView:update()
end

Table:noticeUpdate()
```

Itt láthatjuk, hogy a másik módszerben már a kettőspont operátort használjuk. Ez egy plusz tulajdonsággal ruházza fel a függvényt. Ebben az esetben, amikor meghívjuk a függvényt, impliciten megkapja paraméterként a táblát, mint paraméter. És a **self** kulcsszó segítségével tudjuk felhasználni.

Jól érezhető, hogy ez mennyire hasonlít az OO nyelvekben megszokott **this** kulcsszóra. Igazából lényegében azzal egyenértékű, hiszen így a táblát elérve hozzáférünk minden más függvényhez, illetve attribútumhoz. A példában pl. így el tudjuk érni a 'tableView' adattagot és az, hogy rajta is meghívunk egy függvényt a kettőspont operátorral, már következtethetünk arra, hogy az is egy tábla.

Talán már lehet érezni, hogy ez mennyire segít majd nekünk az OO-s szemlélet megvalósításában. Habár érdemes fejben tartani, hogy ez is csak egy egyszerűbb írásmód, semmi más. Valójában ennek a megfelelője:

```
function Table.noticeUpdate(self)
        self.tableView:update()
end

Table.noticeUpdate(Table)
```

Itt talán érdemes megjegyezni, hogy az ilyen "megoldások" mennyire hasonlítanak a C nyelvben megszokott "adjuk át paraméterként a ...' függvénymegadásokhoz. Ez nem véletlen. A Lua ugyanis nagyban kötődik a C nyelvcsaládhoz. Megfelelő libraryk segítségével akár a lua kódunkat C kódra is át tudjuk írni és ezek a megoldások a könnyű olvasást és "átírást" teszik lehetővé.

2.10.2. Konstruktor

Azt már a függvények visszatérési értékeinél 2.5 is megtudtuk, hogy egy függvény visszatérhet több dologgal is, gyakran szeretjük ezeket egy közös halmazban tenni ha az adatok úgyis összetartoznak. Erre kényelmesen használhatjuk a table típust. Így ezzel nagyjából már el is tudjuk

képzelni, hogy hogyan is néz ki egy konstruktorhívás. Kicsit pontosabban a C nyelvben megszokott módszer rövid összefoglalója volt ez. Lényegében bármilyen objektumot létre tudunk hozni és a függvény visszatérési értékét elmentjük egy változóba.

Azonban mi nem szeretnénk egy 'createXXX()' függvény írni, hanem a C++ és Kotlin nyelvekben megszokott módon szeretnénk létrehozni az új objektumokat:

```
cell = CreateCell() --> C style constructor

cell = Cell() --> C++, Kotlin, ... style constructor
```

A következőkben ezt az elképzelést valósítjuk meg.

2.10.3. Prototípus és objektum

Először is meg kell tervezni a prototípust, amit majd felhasználunk az objektumok létrehozásához.

```
local Cell = {
        state = "",
        changeState = 0
}

function Cell:die()
        if(self.changeState == ChangeType.DIE) then
                self.state = StateType.DEAD
        end
end

function Cell:birth()
        if(self.changeState == ChangeType.BIRTH) then
                self.state = StateType.ALIVE
        end
end

...
```

Most pedig a következő lépés: megcsinálni a konstruktorunkat:

```
function CreateCell(state)
        local newCell = {}
        newCell.state = state
        newCell.neighbors = {}
        return newCell
end
```

Ez így viszont még nincs teljesen kész. Hiszen ez még csak annyit csinál, hogy létrehoz egy táblát, hozzáad pár attribútumot és visszaadja azt. A prototípust még valahogy hozzá kell kötni ehhez az új táblához. Nézzük a következő kódot:

```
local metatable = {
        __index = Cell
}

function CreateCell(state)
local newCell = setmetatable({}, metatable)
newCell.state = state
newCell.neighbors = {}
return newCell
end
```

Itt ami elsőre szembetűnik, hogy az _index metamethod nem függvényt kapott hanem egy táblát (jelen esetünkben a prototípust). Ezt azért tehetjük meg, mert a Lua ad erre támogatást (a függvényhívás teljesítmény visszaeséssel járhat így ez egy egyszerűbb megoldás). De vajon mit is jelent ez valójában:

```
local metatable = {
__index = function(t, k)
                return Cell[k]
        end
}
```

Csupán ennek a függvénynek a rövidítését. Ahol a 't' az az a tábla, amin az _index metódus meghívódott és a 'k' pedig a kulcs, amivel meghívták.

így már nagyon egyszerű értelmezni ezt:

```
local metatable = {
        __index = Cell
}
```

Tehát, ha egy tábla egy nemlétező attribútumát (ami alatt függvényt tároló változót is érthetünk) hívjuk meg, akkor az _index metódus által megnézi a prototípusunkban és ha talál, akkor meghívja azt. Ezzel a példánkban elérhetővé váltak a prototípus függvényei és előre definiált attribútumai.

Most már csak meg kéne valósítani, hogy úgy viselkedjen, mint egy tipikus konstruktor. Erre felhasználhatjuk a _call metamethod-ot. Azaz beállítjuk, hogy ha valaki meghívja a táblán a () operátort, akkor hívja meg a konstruktorunkat. Ezzel elrejtve azt.

```
local Cell = {
        state = "",
        changeState = 0
}

setmetatable(Cell, {
        __call = function(class, ...)
                return CreateCell(...)
        end
})

local metatable = {
__index = Cell
}

function CreateCell(state)
        local newCell = setmetatable({}, metatable)
        newCell.state = state
        newCell.neighbors = {}
        return newCell
end

local cell = Cell()
```

Ezen jól látható, hogy a prototípusnak beállítottuk, hogy ha függvényhívás operátort használják, akkor hozzon létre egy új táblát és használja a prototípus függvényeit.

Ezzel nagyjából készen is lennénk, de elég sok plusz kódot kéne írni. Nem mellesleg maga a konstruktor nem tartozik magához a prototípushoz. És azt sem használtuk ki, hogy valójában a _call hívásánál mi átadtuk magát a prototípust. Kicsit alakítva a kódot a következő, komplexebb, de mindenképp szebb kódot kapjuk:

```
local Cell = {
        state = "",
        changeState = 0
}

setmetatable(Cell, {
        __call = function(class, ...)
                return class:CreateCell(...)
        end
})

function Cell:CreateCell(state)
        local newCell = setmetatable({}, self)
        self.__index = self
        newCell.state = state
        newCell.neighbors = {}
        return newCell
end

local cell = Cell()
```

Vizsgáljuk meg, hogy mi is történt. Először is kihasználtuk, hogy a _call metódus megkapja a prototípust. Ezt fel tudjuk használni, hogy a konstruktorunkat a prototípushoz kössük, mint függvényt, hiszen meg tudjuk hívni. Ráadásul tovább is tudjuk adni a kettőspont operátor segítségével. Így a konstruktor már rendelkezik a prototípussal és nem kell egy globális változót

csinálnunk, hogy elérjük azt. Ha viszont már megvan a prototípus, nem kell mást csinálni, csak beállítani az új objektum metatable-jeként. Így a prototípus metatable-je megegyezik az új objektum metatable-jével.

A következő sor elsőre furán olvasható, de ha egyszerűen odaképzeljük a self helyett a Cell-t, akkor máris megértjük, hogy a 'metatable' létrehozását oldjuk meg egy sorban. Ekkor persze a prototípus __index-ét is beállítjuk magára a prototípusra, de ez teljesen jó (hiszen így érjük el, hogy az új objektum elérhesse a prototípust). Nem csak az új objektum, de a prototípus is saját magát látja és az ő attribútumai között keres. Van még egy további haszna is, de ezt majd az öröklésnél kifejtem.

Végeredményként teljesen ugyanazt a kódok kapjuk, de mégis szebb és kompaktabb a második, hiszen kihasználunk minden lehetőséget amit a metametódusok és a kettőspont operátor biztosítanak.

Még egy fontos dolog, amire mindenképp oda kell figyelni, amikor a konstruktorokat írja az ember. Először is: a prototípus inkább a közös viselkedésre való. Azaz a függvények legyenek elsősorban a prototípusban. Az egyéb változókat, mivel úgyis új objektum jön létre, nyugodtan létrehozhatjuk ott is, nem kell feltétlen a prototípusban felvenni őket. Sőt, a table-öket kifejezetten tilos. Ugyanis, ilyenkor minden új objektum ugyanazt a table-t látja és írja/olvassa (úgy viselkedik, mintha static addatagja lenne egy osztálynak pl. C++-ban). Ezt inkább el szeretnénk kerülni. Így a megoldás, hogy ezeket mindig a konstruktorban, az új objektumnak hozzuk létre.

2.10.4. Öröklés

Az utolsó dolog az OO világában, amire ki kell térnem az az öröklődés. A Lua erre szintén biztosít lehetőséget, méghozzá egy nagyon egyszerű módon. Csupán az előbbi gondolatmenetet kell kiegészíteni.

Vegyünk fel egy ősosztályt:

```
local Participant = {}

setmetatable(Participant, {
        __call = function(class,...)
                return class:new(...)
        end
})

function Participant:new(amountOfMoney, dealer, behavior, inputHandler, observer)
        local newParticipnat = setmetatable({}, self)
        self.__index = self
        newParticipnat.amountOfMoney = amountOfMoney or 100
        newParticipnat.behavior = behavior
        newParticipnat.dealer = dealer
        newParticipnat.inputHandler = inputHandler
        newParticipnat.observer = observer
        return newParticipnat
end
```

Ez teljesen hasonló módon tesszük meg, mint az előző példa esetén. Ezután nézzük meg, hogy hogyan néz ki egy leszármazottja:

```
local Player = {}

setmetatable(Player, {
        __call = function(class,...)
        local newPlayer = class:new(...)
        newPlayer.currentDeckIndex = 1
        newPlayer.numberOfUsedDecks = 1
        return newPlayer
        end,

        __index = Participant
})

player = Player()
print(player.amountOfMoney) --> 100
```

Nézzük meg mi is történt itt valójában, illetve mik is a változások az előző kódhoz képest. Az első szembetűnő dolog, hogy amikor a _call metódust felülírjuk, akkor látjuk, hogy a meghívjuk a **new** függvényt, holott a Player prototípus-nak nincs is ilyen függvénye. Ezt azért teheti meg, mert az _index-et is felüldefiniáltuk, mégpedig az ős prototípusával. Így már meg tudja hívni a 'new' függvényt.

Viszont érdemes megfigyelni, hogy a 'new' meghívásánál a 'class' igazából maga a Player prototípus. Tehát a 'self' paraméter már nem a 'Participant' tábla lesz, hanem a Player. Beállítjuk , hogy az új játékos metatable-je legyen a Player prototípus. Utána pedig az _index-et is beállítjuk, hogy a Player prototípusra mutasson. Ugye ekkor felmerül a kérdés, hogy ez mind szép és jó, de hogyan tudom akkor használni a Participant adattagjait? Egyszerűen úgy, hogy a Player _index-e pedig a Participant prototípusra mutat. Ez ezt jelenti, hogy először a Player táblában keresi és, ha ott nincs, akkor megnézi a Participant táblában. Ez a példa pedig szépen bemutatja, hogy a prototipikus öröklés hogyan valósul meg.

Pár fontos észrevétel azonban elengedhetetlen. Az első és szembetűnőbb, hogy a Player objektum mennyire értelmezhető Participant-nak is egyszerre. Igazából egy új Participant objektum nem jön létre, csak egy Player, ami tudja használni az őse függvényeit/attribútumait. De a Participant _call metódusa meg sem hívódott. Más OO nyelvekben az ős konstruktora is lefut (sőt tipikusan meg kell hívni azt). Itt nem.

Ez a gondolatmenet továbbvíve felveti az absztrakció és az interfész hiányának problémáját egyaránt. Lua-ban nem tudunk ilyen dolgokat definiálni. Egy későbbi fejezetben megmutatok egy lehetséges "megoldást' 4.3.12, amellyel interfész szerű viselkedést el tudunk érni.

A másik fontos dolog pedig, hogy a Player prototípus metatable-jének beállítása során egyszerre két metódust is definiálni kell jelen esetben. Hiszen egy korábbi példán 2.9 bemutattam, hogy milyen következményei vannak, ha külön-külön próbálnánk beállítani őket.

Zárásul még egy dologgal szeretném kiegészíteni az öröklést, mégpedig, hogy a Lua-ban lehetőség van a többszörös öröklésre (ezt csak nagyon kevés nyelv tudja elmondani magáról pl. C++). És annak ellenére, hogy ahol ezt egyen talán meg lehet valósítani, ott rendkívül sok odafigyelése és szabályra kell odafigyelni, addig Lua-ban ez rendkívül egyszerű módon megoldható.

```
local Driver = {}

setmetatable(Driver, {
        __index = Driver
})

function Driver:canDrive(age)
        if(age > 17) then
                return true
        else
                return false
        end
end

local Player = {}

setmetatable(Player, {
__call = function(class,...)
local newPlayer = class:new(...)
newPlayer.currentDeckIndex = 1
newPlayer.numberOfUsedDecks = 1
return newPlayer
end,

__index = function(t, k)
                return Participant[k] or Driver[k]
        end
})

player = Player()
print(player.amountOfMoney) --> 100
print(player:canDrive(16)) --> false
```

Látható, hogy csak az _index metódus változott meg. Mivel nem egyértelmű, hogy melyik prototípust használjuk, így a rövidítés nem lehetséges, tehát egy függvényt kell megvalósítani. Ez viszont

egy elég egyszerű függvény és a megértése szinte evidens. Ha nem találja a meghívott attribútumot a Participant-ban, akkor megnézi a Driver-ben. Ilyen egyszerű az egész.

Persze felmerülhet olyan kérdés, hogy a logikai vesztegzár miatt, ha mind a kettő rendelkezik egy ugyanolyan nevű függvénnyel, akkor sosem érjük el a második függvényét. De ekkor inkább az a kérdés merül fel, hogy ha mind a kettő használ azonos függvényt, akkor vajon jó-e maga a modell. Ez alatt azt értem, hogy lehetséges, hogy a Participant és a Driver egy közös ősből származik és a kérdéses függvényt az öröklési szinten feljebb kellene tolni (tipikus OO szemlélet). A többszörös öröklődés lehetősége viszont nem szokott ösztönző példa lenni. Csak jó tudni, hogy ez is megvalósítható, méghozzá nagyon egyszerű módon.

2.10.5. Modularitás

Utoljára az OO stílussal kapcsolatban ki szeretnék térni a modulokra való bontás témájára. Lua-ban ugyanis hasonló lehetőségünk van a prototípusokat külön modulokba szervezni, akár külön szkript-be kiszervezni őket. Ezzel lehetőségünk van egy szebb strukturális kezelhetőségre, illetve átláthatóbb lesz a kód.

Ahhoz, hogy egy szkriptet egy másik szkriptben fel tudjunk használni, arra a **require** függvény lesz a segítségünkre.

```
local ColorType = require("type.ColorType")
local CardType = require("type.CardType")
local Card = require "entity.Card"

local Dealer = {}

setmetatable(Dealer, {
        __call = function(class,...)
                return class:new(...)
        end
})

function Dealer:createDeck()

local deck = {}

local usedDecks = self.numberOfUsedDecks

        for i = 1, usedDecks, 1 do
                for color, _ in pairs(ColorType) do
                        for value, number in pairs(CardType) do
                                table.insert(deck, Card(value, color, number))
                end
        end
end

return deck

end
...
```

Ez hasonlóan működik, mint a C++-ban megszokott '#include', vagy a Java-s 'import'. Egy hasznos dologra felhívnám a figyelmet. Látszik hogy a 'Card' modul csatolásánál a require függvénynél elhagytam a zárójeleket. Lua-ban az egy paraméteres függvények esetében a primitív típusok esetében (legfőképp string) elhagyható a zárójel.

Most már talán csak az a kérdés merülhet fel, hogy miért kell elmenteni ezt egy változóba? Erre akkor kapunk magyarázatot, ha megvizsgáljuk pl a Card modult:

```
local Card = {}

setmetatable(Card, {
        __call = function(class,...)
                return class:new(...)
        end
})

function Card:new(value, color, number, hidden)
        local newCard = setmetatable({}, self)
        self.__index = self
        newCard.value = value
        newCard.color = color
        newCard.number = number
        newCard.hidden = hidden or false
        return newCard
end

return Card
```

Látható, hogy az utolsó sor visszaadja a prototípust. Így lehet könnyedén modulokat kapcsolni másik modulokhoz. Egyúttal még egyszer kitérnék a 'require' függvény használati módjára. Ugyanis két dologra használhatjuk. Az egyik, amit a példa is mutat. Azaz, ha egy szkript visszatér pl. egy prototípussal, akkor a require-nek van visszatérési értéke. Azonban, ha nincs 'return' a modul végén, akkor úgy viselkedik, mintha az egészet bemásolta volna abba a modulba, ahol meghívták a függvényt (csak úgy, mint C++-ban az '#include').

A végére pedig egy érdekes dolgot hagytam. Lua-ban ugyanis lehetséges a privát adattagok megvalósítása. Ez elég furcsán hathat, hiszen nincs kulcsszó rá, és az eddigiekben egyszer sem került elő még a lehetősége sem. Valójában impliciten mégis használtuk. Ha emlékszünk még a függvények sajátosságaira, megbeszéltük a **closure** 2.8.2 fogalmát. Tulajdonképpen a closure segítségével tudjuk megvalósítani a privát adattagokat. Egyszerűen írunk egy függvényt, amelyben létrehozzuk a lokális környezetben a változókat, függvényeket és a visszatérési értékben csak azokat adjuk vissza, amelyeket szeretnénk publikusnak kezelni. Tipikusan egy table-t adunk vissza, hiszen azt szeretnénk, hogy egy egységként lehessen bánni vele, akárcsak egy osztály objektumával.

Fontos megjegyezni, hogy ebben az esetben, amit eddig az prototipikus OO-ról és öröklésről bemutattunk, azt jelen esetben nem használhatjuk.

```
local ConsoleInputHandler = {}

function ConsoleInputHandler:getInstance()
        local readNumber = function()
                return tonumber(read())
        end

        local readKey = function()
                local line = string.upper(read())
                local result

                if(line == "S") then result = ActionType.SPLIT
                elseif(line == "D") then result = ActionType.DOUBLE
                elseif(line == "Q") then result = ActionType.END
                elseif(line == "H") then result = ActionType.HIT
                elseif(line == "E") then result = ActionType.STAND
                elseif(line == "N" or line == "I") then result = ActionType.NEW
                else result = ActionType.ERROR
        end

        return result

        end

        return {
                readNumber = readNumber,
                readKey = readKey
        }
        end

return ConsoleInputHandler
```

Jól látható, hogy ez egy table-t ad vissza. Igazából, azzal, hogy a 'getInstance' függvényt meghívjuk már létrejön egy példány, így lényegében kiváltottuk a konstruktorunkat. És ahogy a closure témában már beláttuk, minden új objektum a saját, új környezetét látja, elszeparálva a másiktól. Ha az öröklés akarnánk megvalósítani, akkor nncs egyéb dolgunk, mint felvenni egy _index változót a visszaadandó table-be.

Látható, hogy ez talán egy egyszerűbb módja, hogy objektumokat hozzunk létre. Ugyanakkor lemondunk a 'konstruktoros' megoldásról egyaránt (valamit valamiért). Illetve lesz egy indirekt hívásunk is, hiszen a closure függvény még magában nem az, amit mi használnánk. Viszont nagyon kényelmesen megoldhatjuk ezt a require függvénynél:

```
local InputHandler = require("input.ConsoleInputHandler"):getInstance()

-- or like this
local InputHandler = require "input.ConsoleInputHandler":getInstance()
...
```

Így már tényleg a használni kívánandó objektummal dolgozhatunk. Azért érdemes megemlíteni, hogy ha több példányt szeretnénk létrehozni (itt jelen esetben inkább singleton objektumként van használva), akkor érdemes elmenteni a modult és ahol kell, meghívni a példányosító függvényt.

3. fejezet

A Kotlin nyelv és sajátosságai

3.1. Kicsit a Kotlin-ról

A Kotlin erősen típusos programozási nyelv, Java virtuális gépre és JavaScript kódra is lefordítható. A fő fejlesztői a Jetbrains szentpétervári csapata, a nyelv a Szentpétervár közelében található Kotlin-szigetről kapta a nevét. Bár a szintaxisa nem kompatibilis a Java programnyelvvel, a Kotlin együttműködik a Java-ban írt kóddal és épít a java programkönyvtár részeire, mint például a Collections keretrendszerre.

A Kotlin programnyelvet 2011 júliusában hozták nyilvánosságra, amit akkor már egy éve fejlesztettek. A fordítót és a hozzá tartozó programokat 2012 februárjában adták ki nyílt forráskódú szoftverként Apache 2.0 licenc alatt. A Jetbrains bevallott motivációja az új nyelv fejlesztésében az, hogy az növelje az IDEA fejlesztőeszköz eladásait. Andrej Breslav vezető fejlesztő szerint a Kotlin egy a Java-nál jobb, de azzal még mindig teljesen kompatibilis nyelv, amely lehetővé teszi a fejlesztőknek, hogy a Kotlin felé mozduljanak.

Azért használják egyre többen a Kotlin nyelvet, mert mindent meg lehet valósítani, amit Java-ban, viszont a rossz tulajdonságait kiküszöböli, egyszerűsíti. A továbbiakban leginkább ezeket az új és hasznos megoldásokat emelem ki a nyelv bemutatása során (feltételezve, hogy Java-ban vagy akár a C#-ban megismert nyelvi elemek ismeretével rendelkezik az olvasó).

3.2. Adattípusok és változók

3.2.1. Változók

A Pascal-hoz és a Scala-hoz hasonlóan a Kotlin változó deklarációiban is a változó nevét követi a típus, a kettőt kettőspont választja el. Az utasítások végén a pontosvessző opcionális, általában egy új sor elegendő. Ellenben a Java-val, alapértelmezésben minden metódus publikus, a paramétereik pedig nem felülírhatóak a metódusból.

A Kotlin igyekezett kijavítani a Java nyelvben egy elég gyakran előforduló hibát, mégpedig a **nullPointerException**-t. Ez a jelenség legtöbbször az előre nem definiált változók miatt alakult ki. Ezért a Kotlin speciális szintaxissal rendelkezik a null értékek kezelésére. Azok a változók, amelyek felvehetnek null értéket, ? tipus deklarációval kell ellátni.

```
var str: String? = null
str = "text"
str = null  // null can be stored in it
var str2: String = null // error
```

Ha az ún. **nullable** típusokkal dolgozunk, akkor jól látszik, hogy azok nem egyenlők az egyszerűbb típusokkal (nem tudjuk egyszerűen összehasonlítani vagy értékül adni). Erre egy külön ellenőrzés szükséges, amely leggyakrabban a jól megszokott 'nullcheck'. Ezután a fordító **smart-cast**-olja nekünk a változót a 'nem-nullable' típusra és már használhatjuk a szokásos módon:

```
val line = readLine()
var isNumber = true

if(line == null || line == "") return null

line.forEach {
        if(!it.isDigit())  isNumber = false
}
```

Emellett az beépített típusok alapértelmezett függvényeit nem használhatkák a nullable típusú változók mindenféle ellenőrzés nélkül.

```
var str: String? = null
str = "text"
val num = str?.toInt()
```

Ez a **?.** operátor jól ismert a C# világából és a jelentése is ugyanaz: ha az érték 'null', akkor nem engedi meghívni a függvényt, hanem kivételt dob. Ha azonban szeretnénk jelezni a fordítónak, hogy a változó itt "biztos nem null", akkor erre is van lehetőségünk a **!!** operátor segítségével, ami viszont eltér a C#-tól:

```
var str: String? = null
str = "text"
val num = str!!.toInt()
```

Így már ismét úgy viselkedik, mint egy egyszerű típus.

De nem csak ez az egyik módja, hogy megpróbálják elkerülni a 'nullPointerException'-t. A kotlin-ban ugyanis minden változónak a létrehozása pillanatában értéket kell adni. Ezt általában kétféle módon tudjuk megtenni. Erre való a **val** és **var** kulcsszó. A 'val' jelentése, hogy, ha értéket adsz egy változónak, azt többé nem változtathatod meg (ez a 'const' kulcsszóra hasonlít legjobban). A másik a 'var', amely annyiban tér el a párjától, hogy ennek az értékét meg lehet változtatni. A példán láthatjuk a 'var' használatát.

Azért van egy kivétel, amely segítségével nem kell egyből értéket adni a változónak:

```
lateinit var str: String
// some code
str = "text"
```

Ezzel lehetőségünk lesz arra, hogy ne egyből definiáljuk az értéket. Ekkor persze csak a 'var' féle típus értelmezett. Leggyakrabban akkor használjuk, ha az osztály egy objektumának létrehozásánál még nem tudunk értéket adni, hanem majd egy függvényében tudunk először. Bár megjegyezném, hogy minél kevesebbszer fordul elő ennek a típusnak a használata, annál biztonságosabb a kód. És ez vonatkozik a 'var' használatára. A Kotlin ösztönözi a programozókat, hogy használják a lehető legtöbbször a 'val' kulcsszót. Ekkor ugyanis a legbiztonságosabb a program.

Még utoljára ki szeretnék térni a változók láthatóságára. Ezt azért itt taglalom, és nem az OO részben, mert a nyelv is maga objektum-orientált (azaz "mindent osztályokkal írunk le"). És itt még az egész változó koncepció ismert.

Ahogy már korábban is említettem, alapértelmezetten minden metódus publikus, és ez igaz a változókra. A Kotlin-ban a változók igazából **Property**-ként viselkednek (akárcsak a c#-ban), ezért publikusak. Azonban, ha szeretnénk, hogy privátok legyenek, egyszerűen azzá tehetjük őket:

```
private var isLowDeck = false
```

Igazából a háttérben a változó elérése továbbra is 'getter/setter' metódusokkal valósul meg (ha Java kódra fordítjuk ez látszik). De ezzel lehetőségünk van a sok ún. **boilerplate** kód elhagyására, ami szintén a Java egyik rossz tulajdonsága. Property-ket használva sokkal egyszerűbb és átláthatóbb lesz a kód. Azonban lehetőségünk van letiltani a publikus módosítást és ennek még egyszerűbb a módja, mint a C#-os megfelelője:

```
var winCount = 0
private set

var loseCount = 0
private set
```

3.2.2. Adattípusuk

Adattípusokra csak annyiban szeretnék kitérni, hogy mik a legfőbb eltérések a megszokottaktól (mivel nagyban épít a Java nyelvre, így az ott használtaktól való eltérésekre értem). A legszembetűnőbb talán már az előbbi példában is megfigyelhető volt, hogy a beépített típusokat nagybetűvel kezdjük. Nemcsak a 'String'-et (hisz az még Java-ban megszokott, de pl. C#-ban már kisbetűs), de mind az 'Int', 'Double' és 'Boolean'-t is nagybetűvel kezdjük.

Emellett a Kotlin-ban lehetőségünk van plusz funkciókkal ellátni az alapértelmezett típusokat, ha azt szeretnénk, hogy valami egyedi viselkedést meg tudjanak valósítani. Ezeket hívjuk **extension function**-nek.

```
fun Char.or(other: Char?): Char = this.toInt().or(other?.toInt()!!).toChar()
```

Itt például azt szeretnénk megvalósítani, hogy a 'Char' típusú változónak is legyen bitenkénti 'vagy' művelete. Alapvetően nincs a Char típusnak beépített 'or' függvénye, ezért ezzel a módszerrel lehetőségünk van írni. Látható, hogy működik a 'this' kulcsszó, amely azt a változót takarja, amin az 'or' függvényt meghívjuk (hasonló módszer ez, mint a C++-ban oly gyakran használt operátor overload, mikor megírjuk pl. a saját 'String' osztályunkat). A függvény szintaxisra a későbbiekben kitérünk 3.5.

3.3. Vezérlő struktúrák

Ebben a részben két dolgra szeretnék leginkább kitérni: az első, hogy megmutassam, hogy a Kotlin-ban mennyivel kompaktabb módon lehet pl. elágazásokat írni, illetve, hogy az elágazások Kotlin-ban kifejezések, így képesek értékkel visszatérni.

3.3.1. Elágazások

A két legismertebb elágazás az **if** és a **switch–case**. utóbbira azonban a Kotlin egy sajátos kulcsszót vezetett be: a **when**-t. A példánkban ezeknek a tipikus használatát mutatom be:

```
fun calculateResult(player: Deck, bank: Deck) : ResultType {

        val playerValue = evaluate(player.cards)
        val bankValue = evaluate(bank.cards)

        ...

        when {
                playerValue > 21 -> return ResultType.LOSE
                playerValue == 21 -> {
                        return if (bankValue == 21) ResultType.TIE
                        else ResultType.WIN
                }
                else -> return when {
                        bankValue == 21 -> ResultType.LOSE
                        bankValue > 21 -> ResultType.WIN
                        else -> {
                                when {
                                        bankValue < playerValue -> ResultType.WIN
                                        bankValue == playerValue -> ResultType.TIE
                                        else -> ResultType.LOSE
                                }
                        }
                }
        }
}
```

Ezen a komplex példán minden látható, amit az elágazásokkal lehet csinálni. Mint ahogy említettem, az elágazások kifejezések, így lehet visszatérési értékük, ami azt is jelenti, hogy függvény visszatérési értékének is megadhatjuk.

Mindemellett azt is láthatjuk, hogy a 'when' kulcsszót nemcsak a 'switch–case'-re lehet használni, hanem az 'elseif' többszörös elágazások helyettesítésére, ezáltal javítva az olvashatóságot és a kényelmesebb leírást (lehetőségünk van arra is, hogy először az 'elseif' módszert írjuk meg, majd az IDE átalakítja nekünk 'when' stílusú kódra, ami nagyon hasznos).

3.3.2. Ciklusok

A ciklusok szintaxisa és használata a Koltinban szinte teljesen hasonlóan működik, mint ahogy a Java-ban megszokhattuk. Azonban van egy-két sajátossága a 'for' ciklusnak Kotlinban. Alapvetően a 'for' ciklus képes végig iterálni minden olyan objektumon, aminek van 'iterátora'. Ezeken az ún. 'kollekciókon' való végigiterálás nagyon hasonlít a C#-ban használt **ForEach** ciklushoz:

```
        for (item in collection) print(item)

        ...

        for (item: Int in ints) {
          ...
        }
```

Amit még érdemes megfigyelni, hogy pl. az első esetben nem kellett megadni, hogy milyen típust tartalmaz a kollekció, ugyanis ezt nekünk kitalálja a fordító. Persze megadhatjuk a típust, ahogy a második példán is látszik, de ezt majd *később* látjuk, hogy ez szintén felesleges kódolás (még a C#-nál is egyszerűbb, hiszen ott a 'var' kulcsszót oda kell írni).

3.4. Kollekciók

Még egy fontos dolog, amire ki kell térnem, hogy a Kotlin-ban könnyen tudunk létrehozni kollekciókat, amiken végigiterálhatunk és akár használhatunk lambda kifejezéseket is (3.5.1-ben visszatérünk erre).

```
val cells = ArrayList<ArrayList<Cell>>()

(0..height-1).forEach { _ ->
        cells.add(ArrayList())
}

cells.forEach {
        (0 until width).forEach { _ ->
                it.add(Cell( if(random % chanceGen == 0) StateType.ALIVE else StateType.DEAD ))
        }
}
```

Láthatjuk, hogy kétféle módon is létre tudjuk hozni a kollekciónkat: a **..**, illetve az **until** kulcsszó felhasználásával. Az előbbi használata a kezdőértéktől a végértékig hoz létre egy kollekciót (integereket tartalmazó lista), míg az utóbbi annyiban különbözik, hogy az utolsó elemet nem veszi már bele (azaz, ha a width pl. 4, akkor a kollekció a 0,1,2,3 listának felel meg). Tipikusan az utóbbit gyakrabban használjuk. Akár 'for' ciklus-ba is beleágyazva:

```
for(j in 0 until width){
        val upperRow = if(i - 1 < 0) height - 1 else i - 1
        val lowerRow = if(i + 1 == height) 0 else i + 1
        val upperColumn = if(j - 1 < 0) width - 1 else j - 1
        val lowerColumn = if(j + 1 == width) 0 else j + 1

        cells[i][j].neighbors.addAll(arrayListOf(
                cells[upperRow][upperColumn],
                cells[upperRow][j],
                cells[upperRow][lowerColumn],
                cells[i]       [upperColumn],
                cells[i]       [lowerColumn],
                cells[lowerRow][upperColumn],
                cells[lowerRow][j],
                cells[lowerRow][lowerColumn]
        ))
}
```

Emellett ebben a példában azt is szerettem volna megmutatni, hogy kollekciók létrehozására a Kotlin is nyújt nekünk lehetőségeket. Erre az egyik legjobb példa az **arrayListOf** függvény, ami segítségével egy felsorolásból készít nekünk egy kollekciót. Annyira dinamikus, hogy felismeri az elemek típusát, amiket felsorolunk benne, így egy annak megfelelő kollekciót hoz létre. Úgy vélem, ez is egy nagyon szép, kompakt megoldás, ami szintén csökkenti a boilerplate mennyiségét és növeli az átláthatóságát a kódnak.

3.5. Függvények sajátosságai

A függvényekkel kapcsolatban már elmondtam egy pár jellemzőt, mint például, hogy alapértelmezetten publikusak Kotlin-ban, illetve fontos megjegyezni, hogy ún. **first-class** jellegűek. Ez annyit jelent, hogy el tudjuk őket tárolni változókban, illetve át tudjuk adni őket függvény paramétereként is valamint más **higher-order function** (olyan függvény, amely paraméterként vár függvényeket vagy függvénnyel tér vissza) visszatérési értékekeként is kaphatunk függvényt.

Mivel a Ktolin ún. **statically typed** nyelv, ezért a függvényeknek van egy elvárt szintaktikája. Ez teljesen hasonlóan kell elképzelni, mint a Java-ban megszokott függvény megadásának módját:

```
        (Int) -> String
```

Látható, hogy egy függvényt két dolog azonosít: a bemenő paraméterei és a visszatérési értéke.

Kotlin-ban azonban, ha egy függvénynek nincs visszatérési értéke, azt nem a 'void' kulcsszóval jelezzük, hanem a **Unit** típussal (Unit típusnak csak egyféle paramétere lehet: a Unit. Hasonlóan a 'nil' típushoz Lua-ban2.2.2).

```
(Int) -> Unit = ...
```

Persze, ahogy Java-ban is lehetséges, a paramétereket opcionális el lehet nevezni, ami segíti az értelmezést és a használatot is egyaránt.

```
(x: Int, y: Int) -> Point
```

Kotlin-ban minden függvényt a **fun** kulcsszóval deklarálunk és ugyanolyan módon tudjuk használni, mint más programozási nyelvekben. Emellett a Kotlin támogatja a default paramétereket is:

```
fun foo(bar: Int = 0, baz: Int) { ... }

foo(baz = 1) // The default value bar = 0 is used
```

Azonban, ha egy default paraméter megelőz egy olyan paramétert, aminek nincs alapértelmezett értéke, akkor csak úgy hívhatjuk meg a függvényt, hogy ha használjuk az ún. **named argument** opciót. Ezt persze nem csak ebben az esetben használhatjuk. Ha nem abban a sorrendben akarjuk megadni a paramétereket, ahogy eredetileg a függvény várná, akkor az argumentum megnevezés segítségével szinte bármilyen sorrendben meg tudjuk adni őket (viszont ekkor minden nem alapértelmezett paramétert meg kell nevezni).

Érdekesség, ha egy lambda kifejezést szeretnénk átadni egy függvénynek utolsó paraméterként, akkor azt megadhatjuk a függvény fejlécén kívül is:

```
fun foo(bar: Int = 0, baz: Int = 1, qux: () -> Unit) { ... }

foo(1) { println("hello") } // Uses the default value baz = 1
foo { println("hello") }    // Uses both default values bar = 0 and baz = 1
```

Mivel függvény visszatérési értéke bármilyen típus vagy akár kifejezés lehet, így Kotlin-ban van egy egyszerűbb mód, annak megadására, hogy ha egy szimpla kifejezést ad vissza a függvény. Ekkor akár a visszatérés típusa is elhagyható.

```
override fun showDeck(): Deck = deck
//or like this:
override fun hit(cards: ArrayList<Card>): Boolean = Calculator.evaluate(cards) <= 21
//or like this
override fun showDeck() = decks[currentDeckIndex]
```

Még utoljára kitérnék arra, hogy Kotlin-ban is lehetséges generikus függvényeket létrehozni és használni (teljesen hasonló módon, mint ahogy Java-ban megszokhattuk, csupán a létrehozás eltérő):

```
fun <T> singletonList(item: T): List<T> { ... }
```

3.5.1. Lambda kifejezések

Még mielőtt belekezdenék abba, hogy mi is az lambda, előtte elmagyaráznám, hogy miért is jó létrehozni egyszer használatos függvényimplementációkat. Talán a legegyszerűbb indok, hogy egy nagyon speciális függvényt szeretnénk megvalósítani, amire csak abban a pillanatban van szükségünk. Ezért feleslegesnek érezhetjük külön kiszervezni. Valójában talán ezért használják a legtöbbször őket.

Hogy létrehozzunk egy függvénypéldányt, arra a Kotlin sok módszert biztosít: lambda kifejezések, anonymous functions, vagy akár létező deklarációkra is hivatkozhatunk:

```
//lambda
val foo = { a, b -> a + b }
// anonymous fun.
val bar = fun(s: String): Int { return s.toIntOrNull() ?: 0 }
//ref.
val baz = String::toInt
```

Akár létrehozhatjuk **inline**-ként is őket, amely azt jelenti, hogy amikor meghívjuk őket, akkor a függvény törzse bekerül a hívó metódusba, így nem jön létre új objektum és megspórolható annak minden költsége (metódushívás, stack létrehozás, memória foglalás stb).

Akkor térjünk rá arra, hogy mik is azok a lambda kifejezések és miért is olyan hasznosak. Lényegében a lambda kifejezések anonymous függvények (absztrakt metódussal rendelkező interfész névtelen implementációja), amiket kezelhetünk értékként (át tudjuk adni függvény paramétereként, és vissza is térhetünk velük). Az a nagy eltérése a megszokott függvényektől, hogy nem lehet őket deklarálni.

A lambda kifejezéseket mindig kapcsos zárójelek között adjuk meg (így hozva létre a névtelen implementációt). Ezután következnek az opcionális bementeti paraméterek, majd a **->** jel. Ezt követi a függvény jelképes törzse. Ha a visszatérési érték nem 'Unit', akkor a lambda törzsében az utolsó kifejezést tekintjük a visszatérési értékként. De akár megadhatjuk specifikusan is a visszatérési értéket a 'return' kulcsszó segítségével.

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

val sumSub2nd: (Int, Int) -> Int = { x, y ->
                                        val s = sum(x,y) - y
                                        ...
                                        return  s }
```

Ahogy már korábban is említettem, ha egy függvény utolsó paraméterként egy lambda-t vár, akkor azt írhatjuk a zárójeleken kívül is:

```
max(strings) { a, b -> a.length < b.length }
```

Pár gyakran használt módszer a lambda kifejezések paraméterezésével: ha csak egy paramétert vár a lambda függvény, akkor használhatjuk az alapértelmezett nevét: az **it**-et.

```
(0 until decks.count()).forEach {
        decks[it].cards.clear()
        decks[it].money = 0.0
}
```

Valamint, ha nem akarjuk használni a megadott paramétereket, akkor adhatunk egy jelzést a fordítónak:

```
  (1..behavior.getMaximumNumberOfDecks()).forEach { _ -> decks.add(Deck(ArrayList(), 0.0)) }
```

Látható, hogy a '_' jel arra szolgál, hogy ha nincs szükségünk a paraméter(ek)re, akkor így tudjuk ezt jelezni.

Talán az utóbbi példákból már látható, hogy hol is van a lambda kifejezéseknek haszna. Az a példa, hogy egy predikátum függvényt írunk és adjuk át a kiértékelő vagy filter függvénynek, már megszokott. Viszont a másik gyakori használati példa a kollekciókon való iterálás. Látható, hogy a **forEach** függvény vár egy lambdát, aminek segítségével végig mehetünk a kollekció egyes elemein és közben megmondhatjuk, hogy mit szeretnénk csinálni velük:

```
        private fun cellsBirth() {
                cells.forEach { i ->
                        i.forEach {
                                it.birth()
                        }
                }
        }

        fun livingCount(): Int{
                var counter = 0
                        cells.forEach { i ->
                                i.forEach {
                                if(it.state == StateType.ALIVE) counter++
                        }
                }
                return counter
        }
```

Akár a 'for' ciklust is meg tudjuk ezzel valósítani (ami bár szép, de teljesítményben lassabb, hiszen létre kell hozni egy kollekciót, amin végigiterálunk):

```
        (0 until decks.count()).forEach {
                decks[it].cards.clear()
                decks[it].money = 0.0
        }
```

Persze, ha nem lényeges a sebesség, akkor ez ugyanannyira olvasható, mint a 'for ciklus', azonban talán kicsit gyorsabb leírni.

Végezetül még annyit, hogy bár szintaktikailag hasonlítanak, a Kotlin és Java lambda-k teljesen különbözőek. Azaz, ha át akarjuk formázni java kódra, akkor a Kotlin-nak muszáj átalakítani a lambdáit olyan struktúrává, amit fel tudnak majd használni a JVM-en belül.

3.6. Objektum-orientáltság

A Kotlin alapvetően egy OO nyelv, tehát minden feladatot osztályok segítségével oldunk meg. Az OO világába tartozik pl a Java vagy a C# is, így ebben a részben csak azokat a nyelvi eltéréseket szeretném kiemelni, amelyek a Kotlin sajátosságait mutatják be.

3.6.1. Osztályok létrehozása

Egy osztály létrehozása Kotlin-ban lényegében a a Java és C# különböző megoldásainak ötvözése:

```
class Frame (
        val height: Int,
        val width: Int,
        val timeBetweenGens: Long = 250,
        val fillWithRandom: Boolean = true,
        val chance: Int = 4)
        : Runnable, GameObserver {}
```

Láthatjuk, hogy a C#-hoz (C nyelvekhez) megszokottan az öröklést és az interfészek implementációját a kettősponttal jelzi (a Java-ban az 'extends', 'implements' kulcsszóktól eltérően). Valamint az érdekesség még, hogy egyből az osztály nevének leírása után kell írni a konstruktort. Ez a legtöbb eddig referenciaként felhozott nyelvtől elérő sajátosság.

Lehetőség van másodlagos konstruktorok megadására is, ami a 'funtion overolad'-ot "próbálja" helyettesíteni:

```
class Bank (
        basicAmountOfMoney: Double,
        dealer: Dealer,
        behavior: Behavior,
        inputHandler: InputHandler,
        observer: PlayerObserver)
        : Participant(basicAmountOfMoney, dealer, behavior, inputHandler, observer){


constructor(
        basicAmountOfMoney: Double,
        dealer: Dealer,
        behavior: Behavior,
        observer: PlayerObserver)
        : this(basicAmountOfMoney, dealer, behavior, NoInputHandler(), observer)

}
```

Viszont az érdekesség, hogy ekkor is meg kell hívni az elsődleges konstruktort. Ezért emeltem ki, hogy csak próbálja helyettesíteni. Valójában nem váltja ki, csak lehetőségünk van alapértelmezett értékekkel meghívni az eredetit (ezt a megoldást Java-ban gyakran használják, mivel ott nincs default értékadás).

Amit még érdemes megfigyelni, hogy a 'Frame' konstruktorában használtam a 'val' kulcsszót. Ez azért hasznos sajátossága a Kotlin-nak, mert egyúttal az osztálynak is definiáltam ezen változóit (nincs szükség létrehozásra, majd a konstruktorban az ismétlődő értékadásra). Ez egyrészt ismét a boilerplate-et csökkenti, valamint biztosítja, hogy a változóknak ne lehessenek 'null'-ak, hiszen rögtön értéket kapnak.

3.6.2. Data klasszok

Gyakran kell olyan osztályokat definiálnunk, amelyek igazából csak adatok tárolására szolgálnak (tipikusan adatbázis entitások leképezést objektumokra). Ezeknek az adatoknak az elérése/mó-

dosítása tipikusan egyszerűnek kell hogy legyen. Java-ban ez rendkívül sok boilerplate-tel járt, hiszen a konstruktor írás, illetve a getter/setter metódusok írása sok kódolást vett igénybe. Persze az IDE lehetőséget biztosít függvénygenerálás segítségével, de igazából a kód létezik és felesleges sorokat kell akkor is megírni.

Erre találta ki a Kotlin az ún. **data class** fogalmát. Az előbbi példánkban látott osztály létrehozását annyival egészíti ki, hogy egy **data** kulcsszót használ az osztálydefiníció előtt:

```
data class Card(val value: CardType, val color: CardColor, var hidden: Boolean = false)
```

Ezzel jelezve az osztály feladatát. Függvényt nem tudunk így írni, de a fordításánál segíti az optimalizációt.

3.6.3. Object-ek

A másik nagyobb eltérés az osztály típusokkal kapcsolatban, hogy a Kotlin felvette külön nyelvi elemként a 'singleton' osztályt. Erre szolgálnak az ún. **object** osztályok:

```
object InputReader {

        val database = createDatabase("url")

        fun readNumber(): Int? {

                val line = readLine()
                var isNumber = true

                if(line == null || line == "") return null

                line.forEach {
                if(!it.isDigit())  isNumber = false
                }

                if(isNumber){
                return line.toInt()
                }

                return null
        }

}
```

Az a jellemzőjük, hogy nincs konstruktoruk, illetve az 'object' kulcsszót használjuk 'class' helyett. Ez a nyelvi sajátosság is szintén azért jött létre, hogy csökkentsük a fölösleges kódolást (Java-ban a konstruktor priváttá tétele, illetve a getInstance függvény egyszerűsítése).

3.6.4. Modularitás

Az OO témakör végén szeretnék kitérni még egy kicsit a modularitásra, illetve a Koltin projektek struktúrájára, ugyanis itt is van egy kis eltérés az eddigi OO nyelvekhez képest. Java-ban az egyes modulok mindig egy osztályt definiáltak. Nem lehetett egy fájlban több osztályt létrehozni (persze a belső osztályt nem értem ide). Kotlin-ban erre viszont lehetőségünk van. Alapesetben egy Kotlin fájl egy class leírásából áll, de ha létrehozunk újabb class-okat egy fájlon belül, akkor egy ún. **.kt** kiterjesztésű fájlt kapunk, ami jelzi, hogy ebben több osztály van definiálva. Ez megint csak hasznos lehet, hiszen így az egybe tartozó osztályokat egyként tudjuk kezelni és jelezni is.

Utoljára még az alkalmazás belépési pontjára térnék ki, azaz a **main** függvényre. Java-ban és C#-ban egy külön osztály kell neki létrehozni, amelyben van egy tipikusan 'static public Main' függvény. Ezzel ellentétben a Kotlin-ban nem kell külön belépési osztályt írni, ugyanis elég csak egy **main** függvény megírása, akár az egyik modulban is (ez kicsit hasonlít a C-ben megszokott 'int main' függvényhez, hiszen a fordító ott is egy ilyen függvényt keres). Ezzel akár több belépési pontot is definiálni tudunk, majd megmondjuk a fordítónak, hogy melyiket használja.

3.7. Többszálúság

A többszálúságra csupán csak azért szeretnék kitérni, mivel az életjáték projektben kihasználtam, hogy Kotlin-ban lehetőség van több szálat futtatni egyszerre. Így képes voltam a játékmenetet és a felhasználói inputot külön szálakon kezelni. A szálkezelés megvalósítása teljesen hasonló módon működik, mint a Java-ban megszokott módon.

```
class ManualInputHandler(private val observer: GameObserver) : Runnable {

        override fun run() {
                var done = false
                while (!done){

                        val command = readLine()

                        when (command) {
                                "" -> {
                                        observer.noticePause()
                                        println("c␣-␣continue\t␣s␣-␣stop")
                                }
                                "c" -> observer.noticeContinue()
                                "s" -> {
                                        observer.noticeStop()
                                        done = true
                                }
                        }
                }
        }

}

...

class Game(private val frame: Frame) {

        private val gameLoop = Thread(frame)
        private val inputWatcher = Thread(ManualInputHandler(frame))

        fun start(){
                frame.printResult()
                inputWatcher.start()
                gameLoop.start()
        }
}
```

Implementálnunk kell a 'Runnable' interfész 'run' függvényét, hogy megadjuk, hogy egy szál mit végezzen. Ezután a 'Thread' könyvtár segítségével létrehozzuk a szálainkat, majd elindítjuk őket. Innentől kezdve pár a processzor ütemezője lesz értük a felelős. Látható, hogy a megvalósítás egy az egyben a Java-s megvalósítás megfelelője. Ez tulajdonképpen nem véletlen. Többször is hangsúlyoztam, hogy a Kotlin nagyban épít a Java-ra. A legtöbb nyelvi sajátosság csupán egy egy kompaktabb leírás, amit meg lehet mind valósítani java-ban is, csupán ezt a Kotlin elrejti előlünk.

Azonban vannak olyan elemek, amelyeket Kotlin-ban sem tudunk egyszerűbben leírni (talán már alapból egy egyszerű szintaxis), ezért a használata szinte teljesen megegyezik a a Java-ban megszokottakhoz.

3.8. Összegzés

Összegzésként még egyszer kiemelném, hogy a Kotlin-t azért tartom egy nagyszerű OO nyelvnek, mert mindent, amit Java-ban meg lehet balósítani, azt itt is, de legtöbbször sokkal kompaktabb és egyszerűbb módon. Emellett e C#-ból merített hasznos sajátosságokat is remekül adaptálja. Viszonylag friss nyelvről van szó, ezért nem meglepő, hogy folyamatosan bővítik.

Viszont ami érdekes és szeretném megjegyezni, hogy a Kotlin alkotói törekszenek egy bizonyos 'Kotlin native' nyelven, amely célja, hogy versenyképes legyen a C, C++ nyelvek gyorsaságával, viszont sokkal kényelemesebb lenne a használata. Mindemellett már az elején is megjegyeztem, hogy a Kotlin 'open source' és remek a dokumentációja, ezáltal egy nagyon könnyen elsajátítható nyelv.

Nem véletlen, hogy a **Google** is felfigyelt rá. Illetve az android világában is egyre elterjedtebb és lassan kezdi kiszorítani a Java-t.

4. fejezet

Projektek összehasonlítása

4.1. Kicsit a projektekről

Még mielőtt elkezdeném a sajátos megvalósítások és különbségek összehasonlítását, szeretnék egy gyors összefoglalást mondani a projektek elkészülésével és sorrendjével kapcsolatban. Mivel alapvetően eddig csak OO nyelvekkel foglalkoztam eddigi tanulmányaim során, így teljesen egyértelmű volt, hogy a két nyelv közül melyikkel fogom kezdeni. Így nem csak könnyebben tudtam elképzelni a feladatok tervezését és megoldását, de egyúttal reméltem, hogy az OO szemléletben megírt projektek egy jó alapot biztosítanak majd a Lua projekteknél.

Emellett már az elején az volt a célom, hogy megvalósításban a két nyelv projektjei minél jobban hasonlítsanak egymásra. Így biztosítva, hogy ugyanazon feladatot megvalósító részek az egyes nyelvekben mennyire hasonlóak vagy éppen különbözőek lesznek. A végeredmény pedig mind a két feladatban szinte majdnem teljesen azonos lett. Ez azért figyelemre méltó és elsőre talán meglepő is, hogy a Lua is képes arra, hogy viszonylag teljesen azonos megoldást adjon ilyen típusú feladatokra (BlackJack, életjáték), mint a Kotlin.

A projektek bemutatásának szintén célja, hogy megismerhessük az egyes nyelvek előnyeit és hátrányait. Illetve az egyes részeknél arra is szeretnék majd kitérni, hogy a sajátosságokat és kényelmi szempontokat figyelembe véve az egyes helyzetekben melyik megoldást volt könnyebb megvalósítani/használni.

4.2. Életjáték

4.2.1. A játékról

Az életjáték egy egyszerű állapotgépet valósít meg. Játékként való megnevezése megtévesztő lehet, mivel ”nullszemélye” játék; és a ”játékos” szerepe mindössze annyi, hogy megad egy kezdőalakzatot, és azután csak figyeli az eredményt. Matematikai szempontból az ún. sejtautomaták közé tartozik.

A négyzetrács mezőit celláknak, a korongokat sejteknek nevezzük. Egy cella környezete a hozzá legközelebb eső 8 mező (tehát a cellához képest „átlósan” elhelyezkedő cellákat is figyelembe vesszük, feltesszük hogy a négyzetrácsnak nincs széle). Egy sejt/cella szomszédjai a környezetében lévő sejtek. A játék körökre osztott, a kezdő állapotban tetszőleges számú (egy vagy több) cellába sejteket helyezünk. Ezt követően a játékosnak nincs beleszólása a játékmenetbe. Egy sejttel (cellával) egy körben a következő három dolog történhet:

1. A sejt túléli a kört, ha két vagy három szomszédja van.

2. A sejt elpusztul, ha kettőnél kevesebb (elszigetelődés), vagy háromnál több (túlnépesedés) szomszédja van.

3. Új sejt születik minden olyan cellában, melynek környezetében pontosan három sejt található.

A sejt túléli a kört, ha két vagy három szomszédja van. A sejt elpusztul, ha kettőnél kevesebb (elszigetelődés), vagy háromnál több (túlnépesedés) szomszédja van. Új sejt születik minden olyan cellában, melynek környezetében pontosan három sejt található.

Fontos, hogy a változások csak a kör végén következnek be, tehát az „elhalálozók" nem akadályozzák a születést és a túlélést (legalábbis az adott körben), és a születések nem mentik meg az „elhalálozókat". A gyakorlatban ezért a következő lépéseket célszerű ilyen sorrendben végrehajtani:

1. Az elhaló sejtek megjelölése

2. A születő sejtek elhelyezése

3. A megjelölt sejtek eltávolítása

4.2.2. Sejtek

Lua-ban a sejt modul létrehozása:

```
local Cell = {
        state = "",
        changeState = 0
}

setmetatable(Cell, {
        __call = function(class, ...)
                return class:CreateCell(...)
        end
})

function Cell:CreateCell(state)
        local newCell = setmetatable({}, self)
        self.__index = self
        newCell.state = state
        newCell.neighbors = {}
        return newCell
end
```

Kotlin ban:

```
class Cell(var state: StateType){

        var change = ChangeType.NOTHING
        private set

        val neighbors = ArrayList<Cell>()
}
```

Mint azt gondolhattuk, Kotlin-ban sokkal kompaktabb módon tudunk leírni dolgokat. Ez egyen talán nem meglepő, hiszen, mint már említettem**??** ez a Kotlin egyik nagy előnye (minimális boiler plate). Azonban a Lua is minimalizálja a felesleges kódokat. Ezt leginkább a szkript nyelv sajátosságai miatt teheti meg, de ennek köszönhetően itt sem kell felesleges getter/setter metódusokkal telíteni az osztályt.

Mivel ez egy egyszerű osztály, a legtöbb dologban megegyezik mind a két megoldás: Lua-ban:

```
function Cell:die()
        if(self.changeState == ChangeType.DIE) then
                self.state = StateType.DEAD
        end
end

function Cell:birth()
        if(self.changeState == ChangeType.BIRTH) then
                self.state = StateType.ALIVE
        end
end
```

Kotlin ban:

```
fun die() {
        if(change == ChangeType.DIE)
        state = StateType.DEAD
}

fun birth() {
        if(change == ChangeType.BIRTH)
        state = StateType.ALIVE
}
```

Látható, hogy 1-2 apróságtól eltekintve a kettő szinte azonos. Azonban az állapotváltoztató függvényben van egy nagyobb eltérés: Lua-ban:

```
function Cell:checkState()
        local livingNeighbors = 0

        for _, cell in ipairs(self.neighbors) do
                if(cell.state == StateType.ALIVE) then
                        livingNeighbors = livingNeighbors + 1
                end
        end

        if(livingNeighbors > 3 or livingNeighbors < 2) then
                if(self.state == StateType.ALIVE) then
                        self.changeState = ChangeType.DIE
                else
                        self.changeState = ChangeType.NOTHING
                end
        else
                if(self.state == StateType.DEAD and livingNeighbors == 3) then
                        self.changeState = ChangeType.BIRTH
                else
                        self.changeState = ChangeType.NOTHING
        end
end

end
```

Kotlin ban:

```
fun checkState() {
        val livingNeighbors = neighbors.count { it.state == StateType.ALIVE }

        change = when {
         livingNeighbors > 3 || livingNeighbors < 2 -> {
                if(state == StateType.ALIVE) ChangeType.DIE
                else ChangeType.NOTHING
         }
         else -> {
                if(state == StateType.DEAD && livingNeighbors == 3) ChangeType.BIRTH
                else ChangeType.NOTHING
         }
        }

}
```

Itt jól látszik, hogy Lua-ban sokkal kisebb a lehetőségek tárháza. A **switch – case** megoldást Lua-ban csak 'ifelse'-ekkel lehet megvalósítani, míg Kotlinban már a switch-case többszörös elágazást is kompaktabb módon tudjuk felírni. Ezzel nemcsak, hogy rövidebb lesz a kód, de olvashatóbb is.

Összegezve, az egyszerűbb osztályokat Kotlin-ban sokkal könnyebb megvalósítani, bár Lua-ban sem bonyolult. Ezen az osztályon is remekül látszanak a Kotlin lényegi alapelvei: olvashatóság, kompaktság, egyszerűség. Bár a Lua is minimális kódmennyiséget használ, sajnos a szűk lehetőségei miatt nem tudja utolérni a Kotlin-t.

4.2.3. Váz

Megvalósításában szinte teljesen ugyanaz a két kód. Csupán pár helyen térnek el, de ott megvan a maga oka: Lua-ban:

```
function Frame:simulate()
        self:printResult()

        while(self.generationCounter < self.duration) do
                self:wait(self.timeBetweenGens / 1000)
                self:createNextGen()
        end

end

function Frame:wait(seconds)
        local time = os.clock()
        while os.clock()-time < seconds do  end
end
```

Kotlin ban:

```
override fun run() {

        state = GameState.RUNNING

        generationCounter = 0

        while (state != GameState.STOP){

                while (state == GameState.RUNNING)
                        createNextGen()

                Thread.sleep(1000)
        }
}
```

Jól látszik, hogy a Kotlin projektben egy állapotgép maga a játék, míg a Lua projekt egy egyszeri lefutású megvalósítás történik. Ennek az egyedüli oka a szálkezelés. Lua-ban nincs többszálúság, míg Kotlin-ban van. Emiatt nem tudtam egy örök ciklust írni Lua-ban, ott a while ciklus egy előre meghatározott ideig dolgozik, míg a Kotlin projektben a felhasználó állíthatja le a modellezést.

Emellett jól látható, hogy a ”szünet” megoldás a szálak segítségével mennyire egyszerű Kotlin-ban, míg a Lua-ban ehhez egy sajátos (nem épp számításigényes) megoldást kellett alkalmazni.

Ezen kívül viszont csak a kompaktabb leírásban és a sejtek létrehozásában tér el a két kód:

```
function Frame:init()

        math.randomseed(os.time())
        for i = 1, self.height, 1 do
                for j = 1, self.width, 1 do
                        local isAlive = (math.random(100000) % chanceGen) == 0

                        local cellState = StateType.ALIVE

                        if(isAlive == false) then
                        cellState = StateType.DEAD
                end

                table.insert(self.cells[i], Cell(cellState))
        end
end
```

Kotlin ban:

```
init{

        val random = Random()
        val chanceGen = if(chance < 2) 2 else chance

        cells.forEach {
         (0 until width).forEach { _ ->
                it.add(
                Cell( if(fillWithRandom && random.nextInt() % chanceGen == 0)
                        StateType.ALIVE
                        else
                        StateType.DEAD)
                )
         }
        }

        ...
}
```

4.2.4. Fájlból való olvasás

Ahogy vártuk, itt is a Kotlin kompaktabb lehetőségeket biztosít. Bár magában a két kód ugyanazt csinálja, ami külön érdem a Lua szempontjából, sajnos a plusz sorok és függvények miatt sokkal időigényesebb volt a megírása, annak ellenére, hogy akkor már megvolt a Kotlin-os változat és volt honnan kiindulni.

```
...
if(instance.file_exists() == false) then return error("file␣not␣found.") end

instance.lines = instance.getLines()


local firstLine = instance.split(instance.lines[1], "␣")

local error = assert( #firstLine == 3,
        "The␣context␣of␣the␣file␣is␣incorrect!␣PLease␣make␣sure␣to␣give␣a␣correct␣matrix␣definition!")
if(type(error) == string) then
        print(error)
end
...

instance.split = function(s, delimiter)
        local result = {};
        for match in (s..delimiter):gmatch("(.-)"..delimiter) do
                table.insert(result, match);
        end
        return result;
end
```

Kotlin ban:

```
...
 val lines = if(relative) File("testMaps/$fileName.txt").readLines()
        else File(fileName).readLines()

val firstLine = lines[0].split("␣")

if(firstLine.count() != 3)
        throw Exception("Nem␣megfelelo␣a␣fajl␣tartalma!␣A␣matrix␣megadasa␣helytelen!")
...
```

Látszik, hogy a Lua-ban még külön függvényeket kellet létrehozni, hogy ugyanazt az eredményt elérjük. Első sorban kellett egy 'split' függvényt, amely a Kotlin-ban beépített volt. Emellett a fájlkezeléshez is kellett két plusz függvény, amelyeket még fel sem tüntettem: 'getLines' és a 'file_exists'. Ez a kettő teljesen hasonló dolgot végez, mint a Kotlinos megfelelője, de sajnos nem alapértelmezett, ezért szintén meg kellett írnom őket:

```
instance.getLines = function()
        if(instance.relative) then
                instance.fileName = "testMaps/".. instance.fileName ..".txt"
        end

        local lines = {}
        for line in io.lines(instance.fileName) do
                lines[#lines + 1] = line
        end

        return lines
end

instance.split = function(s, delimiter)
        local result = {};
        for match in (s..delimiter):gmatch("(.-)"..delimiter) do
                table.insert(result, match);
        end
        return result;
end
```

Nem mellesleg a a Lua-ban nincs lehetőség a fájlrendszer nézegetésére, mindenféle engedély nélkül (ez a szkriptnyelvek gyakori hátulütője). Emiatt persze egy egyszerű fájlkilistázás, mint Kotlin-ban, a Lua-ban már nemhogy nem egyszerű, hanem nem lehetséges. Tehát ez a kód/függvény csak a Kotlin projektben használható:

```
fun listMaps() { File("testMaps/").list().forEach { println(it) } }
```

És itt is látszik, hogy mennyire egyszerű és kényelmes használni.

4.2.5. Input olvasó

Végre elérkeztünk ahhoz a ponthoz, hogy kimondhassam, hogy a Lua valamiben praktikusabb. Ez pedig a különböző inputok beolvasása. A Kotlin projektben a többszöri beolvasás miatt célszerű volt ezt a feladatot egy singleton osztályra bízni, ami még az ellenőrzést is elvégzi:

```
object InputReader {
        fun readNumber(): Int? {

                val line = readLine()
                var isNumber = true

                if(line == null || line == "") return null
                        line.forEach {
                        if(!it.isDigit())  isNumber = false
                }
                if(isNumber){
                        return line.toInt()
                }
                return null
        }
}
```

Látható, hogy milyen komoly ellenőrzéseket kell végeznünk egy egyszerű szám beolvasása végett. Lua-ban ez ennyire egyszerűen megoldható:

```
 value = tonumber(io.read())
```

Ahogy a Lua-s részben már korábban bemutattam 2.2.2, a **tonumber** függvény egy string-et vár és visszaad vagy egy number-t vagy nil-t (ha nem lehet számmá alakítani). Tehát ezzel az egy sorral megoldottuk a problémát, amire a Kotlinban egy singleton osztályt kellett írni.

Egy dolgot azonban meg kell említenem. A szálkezelés hiánya miatt a Lua-ban a simuláció megkezdése után nincs lehetőség felhasználói inputra, míg a Koltin projektben van. Ezt szintén egy osztállyal oldottam meg:

```
class ManualInputHandler(private val observer: GameObserver) : Runnable {
        override fun run() {
                var done = false
                while (!done){

                        val command = readLine()
                        when (command) {
                                "" -> {
                                observer.noticePause()
                                println("c - continue\t s - stop")
                                }
                                "c" -> observer.noticeContinue()
                                "s" -> {
                                observer.noticeStop()
                                done = true
                                }
                        }
                }
        }

}
```

Látható, hogy ezt kifejezetten egy szál függvényének írtuk meg. Emellett, hogy képes legyen befolyásolni a játékot, egy 'observer'-t is tartalmaz. Ez teljesen hiányzik a Lua projektből.

4.2.6. Pálya (váz) létrehozása

Mindkettő projektben létrehoztam egy segédosztályt, ami a váz létrehozásáért felelős. Ebben valósul meg a fájlból való beolvasás alkalmazása. Mivel ezek egyszerű függvények, nem meglepő, hogy a két megoldás szinte teljesen azonos. Nem kellett semmilyen extrém dolgot megvalósítani (csak azokat az elemeket használják fel, amelyek megvalósítása azonban eltér, ahogy az előző részekben bemutattam). Például a váz létrehozása szinte szóról-szóra megegyezik:

```
instance.createIndividual = function(frame)
        print("\nI - Individual settings\t G - General settings")

  if(string.upper(read()) == "G") then
  frame = Frame(
          instance.getValue("Please enter the width of the testframe (integer value):"),
          instance.getValue("Please enter the height of the testframe (integer value):")
          )
  else
  frame = Frame(
        instance.getValue("Please enter the width of the testframe (integer value):"),
        instance.getValue("Please enter the height of the testframe (integer value):"),
        instance.getValue("Please enter the elapsed time between two generations (in millisec)!"),
        instance.getValue("Please enter the rate of the living cells (choose at least 2)!")
        )
  end

        return frame
end
```

Kotlin ban:

```
private fun createIndividual(): Frame {

  println("\nE - Egyeni beallitasok\t A - Altalanos")

  return if(readLine() == "a")
      Frame(  height = getValue("Kerem adja meg a vaz magassagat (egesz ertek):"),
              width = getValue("Kerem adja meg a vaz szelesseget (egesz ertek):"))

  else Frame(
   getValue("Kerem adja meg a vaz magassagat (egesz ertek):"),
   getValue("Kerem adja meg a vaz szelesseget (egesz ertek):"),
   setGenTime("Kerem adja meg, hogy milyen gyorsan jojjon letre az uj generacio (milisec-ben)!"),
   setLivingChance("Kerem adja meg, hogy kb. minden hanyadik sejt eljen (az ertek egesz es legalabb 2 legyen)")
   )
}
```

Még az ellenőrzés is szinte teljesen megegyezik (nem a szintaxisra értve):

```
instance.getValue = function(msg)
        local correct = false
        local value

        while(correct ~= true) do
                print("\n"..msg.."\t")
                value = tonumber(read())
                if(type(value) == type(0)) then
                        correct = true
                else
                   print("Type error! Please use integer value!")
                end
        end

return value
end
```

Kotlin ban:

```
private fun getValue(message: String): Int {
        var value: Int? = 0
        var correct = false

        while (!correct){
                print("\n$message\t")
                value = InputReader.readNumber()
                if(value != null)
                        correct = true
                else
                   println("Tipushiba! Kerem adjon meg egy egesz erteket!")
        }
        return value!!
}
```

Érdemes megfigyelni, hogy míg a Kotlin kódban **'null check'** van, addig a Lua a **'type check'** -et használja ki.

4.2.7. Játékmag

Utoljára pedig szeretnék kitérni, hogy hogyan is indul és működik a két projektben maga a játék. Mint már többször is említettem, a Lua-ban nem lehet többszálúságot megvalósítani, így ott egyszerűen a 'játékloop'-ot elindító függvényt meghívva már futtatjuk is az alkalmazást:

```
require("FrameMaker").getInstance().createFrame():simulate()
```

Érdemes azért megfigyelni, hogy mennyire kompaktul tudjuk azt megtenni. A 'require' függvény visszaadja az előbb említett váz csináló modult, amin rögtön meghívjuk a 'createFrame' függvényt, ami már a 'Frame'-et visszaadja. Majd ezen elindítjuk a játék loop-ot.

Kotlin-ban a többszálúság miatt a gameloop kezelését egy osztályra bíztam:

```
class Game(private val frame: Frame) {

        private val gameLoop = Thread(frame)
        private val inputWatcher = Thread(ManualInputHandler(frame))

        fun start(){

                frame.printResult()
                inputWatcher.start()
                gameLoop.start()

        }
}
```

Ennek tényleg csak az a felelőssége, hogy létrehozza, majd elindítja a szálakat. És ezzel együtt a gameloop-ot. A main itt is csak egy sorból áll:

```
fun main(args: Array<String>) {
        Game(FrameMaker.createFrame()).start()
}
```

Ez már kicsit jobban hasonlít a Lua-s megoldásra. Valójában a plusz dolog csak a szálas megoldás. Ennek hiányában tényleg megegyezne a két játékindítás.

4.2.8. Összegzés

A projektek készítése során alapvetően a Kotlin tűnt egyszerűbbnek és kényelmesebbnek. Ez persze nem meglepő, hiszen eddig csak OO nyelvekkel foglalkoztam, így egy viszonylag ismerős környezetben kellett újra alkotnom és emiatt könnyebb volt megcsinálni ezt a projektet.

Ugyanakkor mindenképp kiemelném, hogy az enyhe többletkód ellenére a Lua projekt írása is viszonylag kényelmesen és gyorsan haladt. Nagyon intuitív és könnyen tanulható nyelv. Egyszerűen kigondoltam, hogy hogyan kellene működnie Lua-ban az egyes elemeknek és már készen is volt. Persze ez több időt vett igénybe, hiszen meg kellett szokni egy teljesen más látásmódot. Ami viszont a Lua rovására írható, az egyértelműen a szkriptnyelvek tipikus hátránya. Mégpedig, hogy nincs fordítási idejő ellenőrzés. Rengetegszer estem abba a hibába, hogy egyszerűen elgépeltem egy szót, amit persze nem jelez hibának, így csak futás közben jöttek elő a kellemetlenségek. Valamint a teljes projektet át kellett látnom, hogy tudjam használni az egyes függvényeket. Ez persze így elég értelmetlenül hangzik, mert hát csak én dolgoztam vele, nekem kellene mindent tudnom. És ez teljesen igaz. De például, ha megkérném egy kollégámat, hogy egészítse ki 1-2 függvénnyel a kódom, akkor eléggé meg lenne lőve és jó ideig kellene bujkálni a kódban, hogy rájöjjön pl. "itt most mit is ad vissza?" vagy "van ilyen attribútuma?". És én is ugyanígy lennék vele. Tehát Lua-ban nagyon meg kell tervezni, hogy hogyan nézzen ki egy projekt, mert a "kiegészítem itt... kicsit írok még ide... nagyban rontják az átláthatóságot. Kotlin-ban azért erre kisebb az esély. Ez talán leginkább a "dynamic type"-nak köszönhető. Persze ez inkább annak okoz problémát, aki kevés Lua projektet látott. Tipikusan nem szoktak bonyolultak lenni és kis utánajárás után gyorsan meg lehet érteni mi mit csinál.

Nem mellesleg a Lua szkriptek mind "open source"-ok, szóval bárminek a kódját bármikor megnézheti az ember, így könnyen megértheti a működését és tanulhat is belőle. Összességében a kisebb projektek megvalósítására a Lua teljesen alkalmas, de egy komplexebb alkalmazást már nem igazán lenne célszerű ebben írni. És ezzel előreutalok a BlackJack projektek konklúziójára, mert ezt a meglátást ott is megtehetném.

4.3. BlackJack

4.3.1. A játékról

A BlackJack-nek rengeteg változata van, amiket a mia kaszinókban lehet játszani: klasszikus, Atlantic city, európai, prémium, stb... Az én alkalmazásomban a klasszikus póker szabályait használtam fel. Ezt egy-két helyen kiegészítettem vagy egy kicsit eltértem, de csupán azért, hogy megmutassam, hogy ezek alig pár sor megváltoztatását jelentik a kódban. A jelenlegi szabályok a következők:

A klasszikus blackjackben több, 52 lapos paklit használnak. A paklikat együtt keverik meg, mielőtt minden kezet kiosztanának, és az osztó nem kap zárt kártyát. Az osztóknak húzniuk kell, ha a lap összesített értéke 16 vagy kevesebb és ha blackjacked van, az 3:2 oddsszal fizet. A játékos lapja blackjackkel szemben 3:2 oddsszal veszít, viszont nem tud negatív összegbe menni a pénze.

A játékban a lapok értéke a számoknál megegyezik a szám értékével (tehát kilences = 9, kettes = 2), a figurák értéke mindig 11. Az ász kivételes eset, ugyanis tud egyként és tizenegyként is viselkedni. Alap esetben 11 az értéke, de dinamikusan megváltozik 1-re (ekkor mondjuk, hogy puha a kéz), ha a lapok összértéke meghaladja a 21-et. Amennyiben több ász van egy kézben és a lapok értéke több, mint 21, akkor addig váltanak az ászok 11-ről 1 értékűre, amíg 21 alá tudnak menni. 21 alatt az ász értéke 11 marad.

A játékosnak minden körben lehetősége van emelni. Maximum emelési lehetősége mindig az adott pénzösszege. Lehet tét nélkül is játszani. A játékosnak lehetősége van húzni, amíg a kezében a kártyáinak az értéke el nem éri a 21-et. Ezután nincs lehetősége további lapkérésre. A

játékos mindig duplázhat minden kéznél, ami azt jelenti, hogy megduplázhatod az eredeti tétedet. Amennyiben elfogy a pénze a duplázás miatt, akkor nem keletkezik plusz összeg, hanem all-in szerűen mindenét felteszi a játékos.

A játékos ketté is oszthatja a két kártyát, ha azok párt alkotnak. Tehát ha kilences-kilences, király-király vagy hetes-hetes kerül hozzád, két külön kezet csinálhatsz belőlük és az eddigi tétet megfelezve a két kéz külön játszhat. A játékosok kétszer oszthatják ketté a lapjukat, összesen három kezet létrehozva. A játékosok bármilyen kéznél duplázhatnak.

A játékosok nem „adhatják fel" a klasszikus blackjack asztaloknál. Csak megállni tudnak.

A játék maga kétszemélyes (a bank és a játékos), de a Kotlin projektben még gondoltam a bővíthetőségre, illetve, hogy ha esetleg nem csak klasszikus blackjack-et szeretnének játszani, hanem más változatot, akkor erre legyen felkészülve a modell (nagy figyelmet fordítva az OCP elvre). A Lua projekt kicsit egyszerűbb ebben a témában. Ott leginkább csak a klasszikus változatra összpontosítottam. Ennek elsődleges oka inkább az, hogy az OO tervezési elveket már nem olyan egyszerűen tudja megvalósítani. Másrészt viszont annak hátrányaival sem kell foglalkoznia (ezt majd a későbbi példák során jobban kifejtem).

4.3.2. A classic viselkedés

A klasszikus blackjack szabályait ún. 'behavior' szerűen valósítottam meg. Ez a **Strategy** tervezési mintát követi Kotlin-ban. Ezt egy behavior interfész implementációval értem el, amlyet jelen esetben egy 'classic' behavoir osztály valósít meg.

Kotlinban ezt nagyon szépen lehet megvalósítani:

```
interface Behavior {
        fun hit(cards: ArrayList<Card>) : Boolean
        fun double(cards: ArrayList<Card>) : Boolean
        fun split(cards: ArrayList<Card>, numberOfDecks: Int) : Boolean
        fun getMaximumNumberOfDecks() : Int
}

abstract class Classic : Behavior{
        override fun getMaximumNumberOfDecks(): Int = 1
        override fun hit(cards: ArrayList<Card>): Boolean = true
        override fun double(cards: ArrayList<Card>): Boolean = true
        override fun split(cards: ArrayList<Card>, numberOfDecks: Int): Boolean = true
}
```

Definiáltam egy alapértelmezést a klasszikus viselkedésnek és ha szükséges, a leszármazottak ezeket felül tudják írni, ha más viselkedést szeretnének. Igazából itt látszik, amit korábban említettem, hogy csupán 1-2 sort kell átírni és máris kicsit tudunk módosítani a szabályokon.

```
class ClassicBank : Classic(){
        override fun hit(cards: ArrayList<Card>): Boolean = Calculator.evaluate(cards) <= 16
        override fun double(cards: ArrayList<Card>): Boolean = false
        override fun split(cards: ArrayList<Card>, numberOfDecks: Int): Boolean = false
}
```

Ezzel a 'strategy' megoldással csupán a bővíthetőség lehetőségét akartam illusztrálni, hogy ha valaki más változatot játszana, akkor ennyire egyszerűen kiegészítheti a kódot.

Ugyanakkor a Lua-ban ezt a ”pattern”-t nem lehet ilyen szépen megvalósítani, habár nem lehetetlen. Csupán annyi a különbség, hogy nem tudjuk kikényszeríteni sem az interfész, sem az absztrakt osztály fogalmát. Egyszerűen, ehelyett csak definiáljuk az alap klasszikus viselkedést:

```
local ClassicBehavior = {}
...
function ClassicBehavior:getMaximumNumberOfDecks()
        return 1
end

function ClassicBehavior:hit()
        return true
end

function ClassicBehavior:double()
        return true
end

function ClassicBehavior:split()
        return true
end
```

Látható, hogy lényegén teljesen megegyezik a Kotlin-os megfelelőjének, csupán nem absztrakt, így létre lehet hozni. De ez csak alkalmazás kérdése. Ezt az osztályt ugyanis csak közös ősnek lett megírva, hogy aki akarja, azt felhasználhatja. Illetve, hogy ezt a négy függvényt használja minden 'behavior'.

A felhasznált viselkedések már jóval kifejezőbbek:

```
local ClassicBank = {}
setmetatable(ClassicBank, {
        __index = ClassicBehavior,
        __call = function(class, ...)
        return class:new(...)
        end
})
function ClassicBank:hit(cards)
        return Calculator.evaluate(cards) <= 16
end

return ClassicBank
```

Jól látszik az öröklés lehetősége. De ugyanakkor nem muszáj alkalmazni. Ha tudjuk, hogy csupán egy függvényt használunk majd az 'alap viselkedés'-ből, akkor azt egyszerűen megírhatjuk. Hiszen nincs Lua-ban function overload. Ha létezik a prototípusban a használandó függvény, akkor az ősben már nem keres és, ha más függvény nem is kell, akkor nincs is rá szükség. Vegyük észre, hogy ez mennyire kényelmes tud lenni. Hiszen csak a szükséges függvényeket definiáljuk, ami a Kotlin-os verzió esetén akár kényelmetlen 'override'-ok írásával jár. Lua-ban ”ha nincs ré szükség, akkor nem írom meg...” módszert lehet alkalmazni.

4.3.3. Definiált típusok és enum-ok

Kotlin-ban ad a nyelv lehetőséget enum-ok létrehozására:

```
enum class CardType {
        TWO,
        THREE,
        FOUR,
        FIVE,
        SIX,
        SEVEN,
        EIGHT,
        NINE,
        TEN,
        JACK,
        QUEEN,
        KING,
        ACE;
        fun number() =
        when(ordinal){
                in JACK.ordinal..ACE.ordinal -> 11
                else -> ordinal + 2
        }
}
```

Sőt, akár enum osztályokat is tudunk használni és így akár még függvényeket is írhatunk. Lua-ben erre egyszerűen egy table-t alkalmazhatunk:

```
local CardType = {
        TWO = 2,
        THREE = 3,
        FOUR = 4,
        FIVE = 5,
        SIX = 6,
        SEVEN = 7,
        EIGHT = 8,
        NINE = 9,
        TEN = 10,
        JACK = 11,
        QUEEN = 11,
        KING = 11,
        ACE = 11;
}
return CardType
```

Függvény írásának lehetősége fennáll, de nem olyan könnyű elérni vele azt az eredményt, amit szeretnénk. Ezért egyszerűen inkább kihasználjuk a kulcs-érték páros megoldást. A többi definiált típus esetében a is hasonló megoldásokat alkalmaztam.

4.3.4. Inputkezelés

Ahogy az életjátékos projektben is már láthattuk, a Lua a szöveg beolvasását és számmá alakítását sokkal egyszerűbben meg tudja valósítani.

```
local readNumber = function()
        return tonumber(read())
end
```

Kotlin-ban ezt az eredményt csak egy alapos ellenőrző függvény után tudjuk elérni:

```
override fun readNumber(): Double? {
        val line = readLine()
        var isNumber = true

        if(line == null || line == "") return null

        val numberOfDots = line.count { it == '.' }
        line.forEach {
        when {
                it == '.' && numberOfDots > 1 -> isNumber = false
                it == '.' && numberOfDots <= 1 -> isNumber = true
                !it.isDigit() -> isNumber = false
                }
        }

        if(isNumber){
                return line.toDouble()
        }
        return null
}
```

Persze adottak a kompakt megoldások, mint például a beépített 'count' függvény, de még így is láthatóan hosszabb a megoldás. A Kotlin projektben még muszáj kitérnem egy szükséges kiegészítésre. Láthatjuk, hogy az inputkezelést is interfészekkel oldottam meg, mivel a későbbiekben láthatjuk majd, hogy a két résztvevőnk ugyanúgy rendelkezik egy input olvasóval, de a banknak erre nincs szüksége ár igazából. Így számára külön definiáltam egy olyan olvasót, amely egy nagyon egyszerű implementációt valósít meg:

```
class NoInputHandler : InputHandler{
        override fun readNumber(): Double? = null
        override fun readKey(): ActionType = ActionType.ERROR
}
```

Ezt a Lua projektben teljesen kihagyhattam, mert ott nem köt a típusosságból származó kötelező viselkedés. Ezt úgy értem, hogy a banknak nem számít milyen input olvasót kap, hiszen nem ellenőriz típust és mivel nem is foglalkozik vele, így egyszerűen nem kap értelmes változót (ezt majd egy későbbi példában szemléltetem).

4.3.5. Kártya

A kártya tulajdonképpen nem tartalmaz mást, mint adatokat. Az ilyen típusú osztályok megvalósítására pedig igen kellemes megoldást nyújt a Kotlin:

```
data class Card(val value: CardType, val color: CardColor, var hidden: Boolean = false)
```

Csupán egy sor és az egész osztály kész. Nincs semmilyen 'boiler plate'. Lua-ban annak ellenére, hogy minimális a kód, ezt semmiféleképpen sem tudja utánozni:

```
local Card = {}
setmetatable(Card, {
        __call = function(class,...)
        return class:new(...)
        end
})
function Card:new(value, color, number, hidden)
        local newCard = setmetatable({}, self)
        self.__index = self
        newCard.value = value
        newCard.color = color
        newCard.number = number
        newCard.hidden = hidden or false
        return newCard
end
return Card
```

4.3.6. Pakli

A pakli esetében it teljesen hasonló az eset. A Kotlin ugyanúgy egy egyszerű data class-al megoldja, míg a Lua-ban hasonló módon írnánk meg az osztály prototípusát, mint a kártyánál.

4.3.7. Osztó

Az osztó szerepe lényegében a kártyalapok biztosítása. Egy asztalnak egy osztója van és az összes résztvevő ezt használja. A megvalósítása a két projektben szinte teljesen azonos:

```
function Dealer:new(numberOfUsedDecks, observer)
        local newDealer = setmetatable({}, self)
        self.__index = self
        newDealer.numberOfUsedDecks = numberOfUsedDecks or 1
        newDealer.observer = observer
        newDealer.maxCardNumber = newDealer.numberOfUsedDecks * 52
        newDealer.deckPool = newDealer:createDeck()
        newDealer:shuffle()
        return newDealer
end
```

Kotlin-ban:

```
open class Dealer(val numberOfDecksUsed: Int, val observer: DealerObserver) {

        private val maxCardNumber = 52 * numberOfDecksUsed
        protected var deckPool: ArrayList<Card> = createDeck()

        init{
                deckPool.shuffle()
        }
...
```

Amire viszont ki kell térnem, az az input beolvasásnál előjött probléma egy másik oldala. Ez bár egy összetettebb probléma, sajnos az OO nyelvek gyakori hátránya. A **dependancy injection** fogalma mindig is együtt járt az OO szemlélettel. Ugyanis jól látszik a Kotlin kódból, hogy egy 'DealerObserver' objektum szükséges a rendes működéshez, ami nem lesz más, mint maga az asztal. És emiatt az asztalnak már léteznie kell, mikor az osztót létre szeretnénk hozni. Hogy ez miért is probléma. Az OO egyik szépsége, hogy egy osztály elrejti az adattagjait, hogy kívülről ne lehessen módosítani (ezáltal elrontani a belső állapotát).

Ez pedig a tesztelésnél igen kellemetlen problémákat eredményez. Ugyanis létre kellett hoznom már most egy segédosztályt a helyes működés tesztelésére, aminek a célja, hogy a kártyákat és sorrendjüket előre meg tudjam határozni:

```
class TesterDealer(numberOfDecksUsed: Int, observer: DealerObserver)
: Dealer(numberOfDecksUsed, observer){

        override fun giveCard(): Card = deckPool.removeAt(0)
        fun createNewDeck(cards: ArrayList<Card>) { deckPool = cards }
}
```

Ez mind plusz kódolást jelentett, de nem csak itt, hanem a unit tesztek írása közben is. A Lua ennél sokkal egyszerűbb megoldást ad. Mivel ott a prototípus nem zárt a változtatásra, így egyszerűen definiáltam egy új osztót, amely azt a 'giveCard' függvényt megvalósítja, amit szintén tudok manipulálni.

```
local testDealer = {
        deckPool = {}
}
function testDealer:giveCard()
        return table.remove(self.deckPool, 1)
end
...
function testGameWithWinByBlackJackAndStopAndQuit()

        local table = Table( 5, ConsoleInputHandler)
        table.dealer = testDealer
...
```

Ez sokkal kényelmesebb és egyszerűbb megoldás és itt látszik meg, hogy az OO szemléletnek is vannak hátulütői.

4.3.8. A "játékban résztvevő"

Ez az osztály felel az egységes kezelésért. A közös ősosztály a résztvevők számára. Ebben definiáljuk, hogy mik az alapvető viselkedései az egyes résztvevőnek az asztalnál. Ez szintén a bővíthetőség az OO szemlélet miatt lett így létrehozva (akárcsak a viselkedésnél, a minta lényege teljesen hasonló). A Lua projektben is is megjelenik és a lényege itt is az, hogy egy alap prototípust biztosít (egy példát), hogy ha valaki bővítené a kódot, akkor ezekre a függvényekre és attribútumokra szüksége lesz.

4.3.9. A játékos

A játékos modelljét először Kotlinban írtam meg és ennek mintájára készült a Lua projekt is. Nem meglepő, hogy a megoldások rendkívül hasonlóak. Sőt, a szintaxistól eltekintve szinte semmilyen nagyobb különbséget nem lehet felhozni. Nagyon szépen lehet látni, hogy alapvetően a logikát ugyanúgy meg lehet valósítani mind a két nyelvben, ami alatt inkább azt értem, hogy az egyik nyelvben megírt gondolatokat szinte egy az egyben átültetve a mások kódba (a szintaxis persze változik), ugyanazt az eredményt kapjuk. És itt most nem a 'copy – paste'-re szeretnék utalni, hanem a két nyelv nem igényel ilyen szinten teljesen más gondolkodási módot.

4.3.10. A bank

Akárcsak a játékosnál, itt is lényegében teljesen megegyezik a két kód. Csupán a Kotlin kompakt használatának alkalmazásával tudunk 1-2 sort nyerni.

```
function Bank:showDeck()
        return self.deck
end

function Bank:preparation()
        self.deck.cards = {}
end
```

Míg kotlinban:

```
override fun showDeck(): Deck = deck

override fun preparation() { deck.cards.clear() }
```

Bár azért érdemes megfigyelni a sajátos megoldást a Lua-ban a kártya pakli törlésének. Míg Kotlinban a meglévő listát töröljük, de maga a váz megmarad, addig Lua-ban egyszerűen kap egy újat. Persze azért még mindig lényeges különbség, hogy a Kotlin-os megoldásban a lista saját függvényei nem vesznek el, de ha csak tényleg adattárolásra használjuk, akkor mind a két eset teljesen jó.

4.3.11. Az asztal

Az asztal maga a játéktér, hiszen ez valósítja meg azt az állapotgépet, ami által az alkalmazás működik. Ami viszont érdekes, hogy itt is szinte teljesen megegyezik a két projekt megoldása. Azonban egy dologra mindenképpen ki kell térni, mert lényegi különbség nagyjából csak itt jelenik meg. Ez pedig a konstruktor:

```
function Table:new(numberOfDecks, inputHandler)
        local newTable = setmetatable({}, self)
        self.__index = self
        newTable.inputHandler = inputHandler
        newTable.dealer = Dealer(numberOfDecks, self)
        newTable.player = Player(0, newTable.dealer, ClassicPlayer(), inputHandler, newTable)
        newTable.bank = Bank(0, newTable.dealer, ClassicBank(), nil, newTable)
        newTable.state = GameState.NEW
        newTable.isLowDeck = false
        newTable.tableView = ClassicTableView(newTable.player, newTable.bank)
        newTable.wincount = 0
        newTable.loseCount = 0
return newTable
end
```

Míg kotlinban:

```
class Table(numberOfDecks: Int, private val inputHandler: InputHandler, testing: Boolean = false)
        : DealerObserver, PlayerObserver {

        val dealer = if(!testing) Dealer(numberOfDecks, this) else TesterDealer(numberOfDecks, this)

        private val player = Player(0.0, dealer, ClassicPlayer(), inputHandler, this)

        private val bank = Bank(0.0, dealer, ClassicBank(), this)

        private var state = GameState.NEW

        private var isLowDeck = false

        private val tableView = ClassicTableView(player, bank)

        var winCount = 0
        private set
        var loseCount = 0
private set
...
```

Azt érdemes előtte tudni, hogy mind a Player, mind a Bank konstruktora ugyanazokat a paramétereket várja, ugyanabban a sorrendben. Leginkább a bank létrehozásában van lényegi eltérés. Itt láthatjuk ugyanis azokat a szemléletes ”több kód”-nak a hasznát.

A bank ugyanis a Kotlin projetben rendelkezik egy második konstruktorral:

```
constructor( basicAmountOfMoney: Double,
        dealer: Dealer,
        behavior: Behavior,
        observer: PlayerObserver)
        : this(basicAmountOfMoney, dealer, behavior, NoInputHandler(), observer)
```

Ezen látszik, hogy hol van szerepe a különleges inputkezelésnek.

A másik szembetűnő részlet az osztó létrehozása. Mint ahogy említettem ennek az oka, hogy a tesztelést lehetségessé tegyem. Ezzel a Lua projektben nem kell külön foglalkozni és ezért érdekes módon szebb OO megvalósítást tesz lehetővé (leszámítva persze, hogy a Lua-s Table minden attribútumát kívülről el lehet érni és megváltoztatni).

4.3.12. A játékmenet "figyelő"

Ami még talán feltűnt az előző példában, hogy a Kotlin projektben a Table megvalósít két interfész implementációt. Erre azért van szükség, hogy a kirajzolást **MVC** modell szerűen tudjuk megvalósítani, amellyel megint csak az OO felfogást erősítem. Hiszen így a logikát és a kirajzolást különválasztom és más-más osztály felelőssége lesz.

```
interface PlayerObserver {
        fun noticeNewGame()
        fun noticeEndGame()
        fun noticeUpdate()
}

Table{
...
    override fun noticeUpdate() { tableView.update() }
        override fun noticeNewGame() {state = GameState.NEW }
        override fun noticeEndGame() { state = GameState.END }
...

}
```

Valamint az **TDA** elvet követve létrehoztam egy 'DealerObserver' interfészt is, amely segítségével az osztó tud jelezni, hogy kevés a lapja, az asztalnak, aki erre tud reagálni:

```
interface DealerObserver {
        fun noticeLowDeck()
}
Dealer{
...
        open fun giveCard() : Card {
                if(deckPool.count() < maxCardNumber / 2) observer.noticeLowDeck()
                return deckPool.removeAt(0)
        }
...
}
Table{
...
        override fun noticeLowDeck() { isLowDeck = true }
...
        private fun play(){

                if(isLowDeck){
                        dealer.createNewDeck()
                        isLowDeck = false

                }
        ...
        }
...
}
```

Ez a megvalósítás megjelenik a Lua projektben is. Itt viszont nem tudjuk jelezni, hogy ezek interfész függvények (maximum kommentek segítségével).

```
Table = {}
...
-- Interface functions

function Table:noticeUpdate()
        self.tableView:update()
end

function Table:noticeNewGame()
        self.state = GameState.NEW
end

function Table:noticeEndGame()
        self.state = GameState.END
end

function Table:noticeLowDeck()
        self.isLowDeck = true
end
```

Ezek ugyanúgy viselkednek, mint a Kotlin-os megfelelőjük. Csak itt mind a résztvevők, mind a az osztó magát az egész Table objektumot látja, minden más adattagjával, hiszen nem interfészként van értelmezve. Itt akkor a példában bemutattam egy korábbi megjegyzésem (2.10.4) értelmezését.

4.3.13. Az eredményszámító

Ez az osztály felel a kártyák kiértékeléséért, illetve az eredmény meghatározását is ide szerveztem ki. Itt érvényesül a szabályzat másik része, ami a kártyákra vonatkozik.

A két megoldás itt is hasonló, azonban a definiált típusok használatában vannak eltérések.

```
local evaluate = function(cards)
        local sum = 0

        for _, card in ipairs(cards) do
                sum = sum + card.number
        end
...
```

Míg kotlinban:

```
fun evaluate(cards: ArrayList<Card>) : Int{
        var sum = 0
        cards.forEach { sum += it.value.number() }
...
```

Alapból már az feltűnik, hogy Lua-ban nincs += operátor, emiatt a basic megoldás. A másik, az enum-ok használata. Visszaemlékezve, hogy hogyan voltak definiálva külön-külön (4.3.3) egyből látszik, hogy hogyan is működik. Mégis azért egy nagy különbség, hogy a Kotlin-ban maga az enum osztály lehetőséget nyújt, hogy a típus és az értéke együtt használható legyen (és ezért csak egy attribútum felel), addig a Lua projektben a 'value' és a 'number' változó oldja meg ugyanazt a 'Card' osztálynál. Erre külön oda kell figyelni, ami a kényelmes használatot rontja és billenti az előnyt a Kotlin oldalára.

Az eremény kiszámolását végző függvény lényegében ugyanaz mind a két projektben, azonban szokás szerint a Kotlin sokkal kompaktabb módon tudja leírni a többszörös elágazást. Ez ismét csak egy pozitívum.

4.3.14. Kirajzolás

A kirajzolás igazából két részre tudjuk osztani. Egy kisebb és egy nagyobb egységre. A kisebb egység maga az egyes kártyák kirajzolása, amelyért mindkét projektben egy osztály a felelős: 'UnicodeMaker' illetve 'CardMaker'. Érdemes már a nevükben rejlő különbséget is megfigyelni. A Kotlin 'UnicodeMaker' osztály ugyanis a kártya unicode-ját adja vissza és a kirajzolást azt az arra

specializálódott, kirajzolásért felelős osztály majd megoldja. Mig a Lua-s megoldásban már maga a 'CardMaker' osztály felelős a kirajzolásért.

Ennek oka, hogy Lua-ban sokkal dinamikusabban lehet létrehozni unicode értéket:

```
io.write(utf8.char(tonumber(cardUnicodeString,16)))
```

Itt látszik, hogy egy string-ből alakít unicode-ot. Míg Kotlin-ban ez nem ilyen egyszerű. Ugyanis az escape karaktereket nem lehet csak úgy dinamikusan összefűzni. A megoldás eléggé összetett, bár szerencsére a Kotlin még segít nekünk ebben:

```
fun Char.or(other: Char?): Char = this.toInt().or(other?.toInt()!!).toChar()
...
        val firstChar = '\uD83C'
        val secondChar = colorChar.or(list[card.value])

        return "$firstChar$secondChar"
...
```

Itt lényegében bitműveletekkel összeszerkesztjük a kívánt unicode karaktert. Erre a Kotlin 'extension function' nyelvi elemmel nyújt segítséget. Ezáltal plusz viselkedést definiálhatunk a 'Char' típusnak, amely segítségével a kívánt hatást elérhetjük.

A kirajzolás másik nagyobb egysége, az egyes résztvevők megjelenítéséért felelős osztályok. A Kotlin itt is az MVC modellt követi, azaz rendelkezik egy 'updatable' interfésszel, amit a résztvevők nézetei implementálnak:

```
interface Updatable {
        fun update()
}

class ClassicBankView(private val bank: Bank) : ClassicView() {
        override fun update() {
                val deck =  bank.showDeck()
                deck.cards.forEach { print(UnicodeMaker.cardToUnicode(it)) }
        }
}
...
```

Ezekért azonban nem az egyes játékosok tartalmazzák, hanem a Table objektum és a 'PlyerObserver' segítségével jeleznek, hogy megváltozott az állapotuk. A table pedig erre kirajzoltatja az új állapotot:

```
class ClassicTableView(player: Player, bank: Bank) : ClassicView() {

        private val playerView = ClassicPlayerView(player)
        private val bankView = ClassicBankView(bank)

        override fun update() {

                bankView.update()
                println("\n")
                playerView.update()
        }
}
```

Ez teljesen hasonlóan van megvalósítva a Lua projektben is:

```
local ClassicBankView = {}

setmetatable(ClassicBankView, {
        __call = function(class, ...)
                return class:new(...)
        end
})

function ClassicBankView:new(bank)
        local newView = setmetatable({}, self)
        self.__index = self
        newView.bank = bank
        return newView
end

function ClassicBankView:update()
        for _, card in ipairs(self.bank:showDeck().cards) do
                CardMaker.printCard(card)
        end
        print("\n")
end

return ClassicBankView

...

local ClassicTableView = {}

setmetatable(ClassicTableView, {
        __call = function(class, ...)
                return class:new(...)
        end
})

function ClassicTableView:new(player, bank)
        local newView = setmetatable({}, self)
        self.__index = self
        newView.player = PlayerView(player)
        newView.bank = BankView(bank)
        return newView
end

function ClassicTableView:update()
        self.bank:update()
        self.player:update()
end

return ClassicTableView
```

4.3.15. A játékmag

Utoljára még magáról a játék indításáról szeretnék beszélni egy kicsit. A megoldás is itt hasonló, akárcsak az életjátéknál:

```
fun main(args: Array<String>) {

        Table(numberOfDecks =  8, inputHandler = ConsoleInputHandler()).startGame()

}
```

A Lua projektben:

```
local InputHandler = require "input.ConsoleInputHandler":getInstance()

local Table = require "entity.Table"

Table(6, InputHandler):startGame()
```

Ugye itt nem kell szálakkal foglalkozni, így mind a kettő megoldás egy állapotgépet használ a játék magjaként.

```
function Table:startGame()
print("\nWelcome␣to␣the␣table!␣Do␣not␣forget,␣the␣house␣always␣wins!")

        while(self.state ~= GameState.QUIT) do
                if(self.state == GameState.RUNNING) then self:play()
                elseif(self.state == GameState.NEW) then self:settings()
                elseif(self.state == GameState.END) then self:showStatistics()
                elseif(self.state == GameState.QUIT) then
                end
        end

        print("Thank␣you␣for␣choosing␣our␣BlackJack␣game!")
end
```

És a Kotlin projektben is teljesen hasonló a megoldás.

4.3.16. Összegzés

Összegzésként ugyanazokat a megfigyeléseket és tapasztalatokat szereztem, mint az életjáték projekt kapcsán4.2.8. Azonban itt szeretnék egy kicsit kitérni a unit tesztek írására. Ugyanis ez egy kicsit komplexebb játék volt az életjátékhoz képest és az OO megoldás megmutatta, hogy néhány hátránya hol tud nagyon kijönni. Alap esetben a Lua projekt unit tesztjeinek írása kényelmesebb volt, mint a Kotlin-é, annak ellenére, hogy az IDE is ad támogatást és a 'JUnit' használata is kényelmesebb, mint a Lua-s **luaunit** használata.

4.4. Vélemények

Így mind a négy projekt összehasonlítása után elmondhatom, hogy a Lua igazán figyelemreméltó nyelv, hiszen, bár alapból nem OO szemléletű nyelv, akkor is meg lehet benne valósítani szinte minden OO lehetőséget (kisebb-nagyobb eltérések híján). Persze az OO séma megvalósítása elég erős szabályokat és elveket takar, amiket nyilván nem tud betartani mindig a Lua (legalábbis nagy nehézségek árán), de kinézetre meg lehet valósítani, és ha valaki ezzel a hozzáállással használja, akkor a projektekből is látszik, hogy szinte ugyanaz a végeredmény.

Mégis konklúzióként az mondanám, hogy jobb volt Kotlin-ban dolgozni. Nemcsak az OO szemlélet, hanem az IDE támogatása és a szélesebb lehetőségek tárháza is nagy szerepet játszik ilyenkor. És ebben a Lua sajnos hátrébb szorul.