

# Univerzális programozás

---

Igy neveled a programozód!

Ed. BHAX, DEBRECEN,  
2019. február 19, v. 0.0.4

Copyright © 2020 Deák Ruben

Copyright (C) 2019, 2020, Norbert Bátfai Ph.D., batfai.norbert@inf.unideb.hu, nbatfai@gmail.com,

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.3 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled "GNU Free Documentation License".

<https://www.gnu.org/licenses/fdl.html>

Engedélyt adunk Önnek a jelen dokumentum sokszorosítására, terjesztésére és/vagy módosítására a Free Software Foundation által kiadott GNU FDL 1.3-as, vagy bármely azt követő verziójának feltételei alapján. Nincs Nem Változtatható szakasz, nincs Címlapszöveg, nincs Hátlapszöveg.

<http://gnu.hu/fdl.html>

A tananyag elkészítését az EFOP-3.4.3-16-2016-00021 számú projekt támogatta. A projekt az Európai Unió támogatásával, az Európai Szociális Alap társfinanszírozásával valósult meg.

DRAFT

**COLLABORATORS**

	<i>TITLE :</i>		
	Univerzális programozás		
<i>ACTION</i>	<i>NAME</i>	<i>DATE</i>	<i>SIGNATURE</i>
WRITTEN BY	Bátfai, Norbert, Bátfai, Mátyás, Bátfai, Nándor, Bátfai, Margaréta, Ákos Deák, Ruben	2020. november 30.	

**REVISION HISTORY**

NUMBER	DATE	DESCRIPTION	NAME
0.0.1	2020-02-12	Az iniciális dokumentum szerkezetének kialakítása.	nbatfai
0.0.2	2020-02-14	Inciális feladatlisták összeállítása.	nbatfai
0.0.3	2020-02-16	Feladatlisták folytatása. Feltöltés a BHAX csatorna <a href="https://gitlab.com/nbatfai/bhax">https://gitlab.com/nbatfai/bhax</a> repójába.	nbatfai
0.0.4	2020-02-19	A Brun tételes feladat kidolgozása.	nbatfai
0.0.5	2020-02-27	Helló, Turing! csomag kész.	deakruben
0.0.6	2020-03-13	Helló, Chomsky! csomag kész.	deakruben
0.0.7	2020-03-20	Helló, Caesar! csomag kész.	deakruben

**REVISION HISTORY**

NUMBER	DATE	DESCRIPTION	NAME
0.0.8	2020-03-27	Helló, Mandelbrot! csomag kész.	deakruben
0.0.9	2020-04-03	Helló, Welch! csomag kész.	deakruben
0.1.0	2020-04-10	Helló, Conway! csomag kész.	deakruben
0.1.1	2020-04-24	Helló, Schwarzenegger! csomag kész.	deakruben
0.1.2	2020-05-01	Helló, Chaitin! csomag kész.	deakruben

DRAFT

# Ajánlás

„To me, you understand something only if you can program it. (You, not someone else!) Otherwise you don't really understand it, you only think you understand it.”

—Gregory Chaitin, *META MATH! The Quest for Omega*, [[METAMATH](#)]

DRAFT

# Tartalomjegyzék

<b>I. Bevezetés</b>	<b>1</b>
<b>1. Vízió</b>	<b>2</b>
1.1. Mi a programozás? . . . . .	2
1.2. Milyen doksikat olvassak el? . . . . .	2
1.3. Milyen filmeket, előadásokat nézzek meg, könyveket olvassak el? . . . . .	3
<b>II. Tematikus feladatok</b>	<b>5</b>
<b>2. Helló, Turing!</b>	<b>7</b>
2.1. Végtelen ciklus . . . . .	7
2.2. Lefagyott, nem fagyott, akkor most mi van? . . . . .	8
2.3. Változók értékének felcserélése . . . . .	10
2.4. Labdapattogás . . . . .	11
2.5. Szóhossz és a Linus Torvalds féle BogoMIPS . . . . .	13
2.6. Helló, Google! . . . . .	14
2.7. 100 éves a Brun tétele . . . . .	16
2.8. A Monty Hall probléma . . . . .	16
2.9. Minecraft-MALMÖ bevezető . . . . .	18
2.10. Vörös Pipacs Pokol/csiga folytonos mozgási parancsokkal . . . . .	18
<b>3. Helló, Chomsky!</b>	<b>20</b>
3.1. Decimálisból unárisba átváltó Turing gép . . . . .	20
3.2. Az $a^n b^n c^n$ nyelv nem környezetfüggetlen . . . . .	21
3.3. Hivatalos nyelv . . . . .	22
3.4. Saját lexikális elemzõ . . . . .	22

3.5. Leetspeak . . . . .	23
3.6. A források olvasása . . . . .	24
3.7. Logikus . . . . .	27
3.8. Deklaráció . . . . .	28
3.9. Vörös Pipacs Pokol/csiga diszkrét mozgási parancsokkal . . . . .	29
<b>4. Helló, Caesar!</b>	<b>31</b>
4.1. double ** háromszögmátrix . . . . .	31
4.2. C EXOR titkosító . . . . .	33
4.3. Java EXOR titkosító . . . . .	35
4.4. C EXOR törő . . . . .	36
4.5. Neurális OR, AND és EXOR kapu . . . . .	38
4.6. Hiba-visszaterjesztéses perceptron . . . . .	42
4.7. Vörös Pipacs Pokol/írd ki, mit lát Steve . . . . .	43
<b>5. Helló, Mandelbrot!</b>	<b>44</b>
5.1. A Mandelbrot halmaz . . . . .	44
5.2. A Mandelbrot halmaz a std::complex osztályjal . . . . .	47
5.3. Biomorfok . . . . .	50
5.4. A Mandelbrot halmaz CUDA megvalósítása . . . . .	54
5.5. Mandelbrot nagyító és utazó C++ nyelven . . . . .	57
5.6. Mandelbrot nagyító és utazó Java nyelven . . . . .	58
5.7. Vörös Pipacs Pokol/fel a láváig és vissza . . . . .	61
<b>6. Helló, Welch!</b>	<b>62</b>
6.1. Első osztályom . . . . .	62
6.2. LZW . . . . .	66
6.3. Fabejárás . . . . .	68
6.4. Tag a gyökér . . . . .	71
6.5. Mutató a gyökér . . . . .	79
6.6. Mozgató szemantika . . . . .	80
6.7. Vörös Pipacs Pokol/5x5x5 ObservationFromGrid . . . . .	90

<b>7. Helló, Conway!</b>	<b>91</b>
7.1. Hangyaszimulációk . . . . .	91
7.2. Java életjáték . . . . .	109
7.3. Qt C++ életjáték . . . . .	118
7.4. BrainB Benchmark . . . . .	118
7.5. Vörös Pipacs Pokol/19 RF . . . . .	126
<b>8. Helló, Schwarzenegger!</b>	<b>127</b>
8.1. Szoftmax Py MNIST . . . . .	127
8.2. Mély MNIST . . . . .	132
8.3. Minecraft-MALMÖ . . . . .	132
8.4. Vörös Pipacs Pokol/javíts a 19 RF-en . . . . .	133
<b>9. Helló, Chaitin!</b>	<b>134</b>
9.1. Iteratív és rekurzív faktoriális Lisp-ben . . . . .	134
9.2. Gimp Scheme Script-fu: króm effekt . . . . .	135
9.3. Gimp Scheme Script-fu: név mandala . . . . .	139
9.4. Vörös Pipacs Pokol/javíts tovább a javított 19 RF-edén . . . . .	139
<b>10. Helló, Gutenberg!</b>	<b>140</b>
10.1. Programozási alapfogalmak . . . . .	140
10.2. C programozás bevezetés . . . . .	145
10.3. C++ programozás . . . . .	146
10.4. Python nyelvi bevezetés . . . . .	148
<b>III. Második felvonás</b>	<b>150</b>
<b>11. Helló, Berners Lee!</b>	<b>152</b>
11.1. Java és C++ OOP programozási nyelvek: . . . . .	152
11.2. Bevezetés a mobilprogramozásba . . . . .	158
<b>12. Helló, Arroway!</b>	<b>161</b>
12.1. OO szemlélet . . . . .	161
12.2. Gagyí . . . . .	164
12.3. Yoda Conditions . . . . .	166
12.4. EPAM: Java Object metódusok . . . . .	167

<b>13. Helló, Liskov!</b>	<b>170</b>
13.1. EPAM: Interfész, Osztály, Absztrakt Osztály . . . . .	170
13.2. EPAM: Liskov féle helyettesíthetőség elve, öröklődés . . . . .	173
13.3. Liskov helyettesítési elv sértése . . . . .	175
13.4. Szülő-gyerek osztály . . . . .	177
<b>14. Helló, Mandelbrot!</b>	<b>180</b>
14.1. EPAM: OO modellezés . . . . .	180
14.2. Forward engineering UML osztálydiagram . . . . .	182
14.3. EPAM: Neptun tantárgyfelvétel modellezése UML-ben . . . . .	183
14.4. EPAM: Neptun tantárgyfelvétel UML diagram implementálása . . . . .	184
<b>15. Helló, Chomsky!</b>	<b>190</b>
15.1. EPAM: Bináris keresés és Buborék rendezés implementálása . . . . .	190
15.2. EPAM: Order of everything . . . . .	192
15.3. EPAM: HashMap implementáció . . . . .	195
15.4. Full Screen . . . . .	201
<b>16. Helló, Stroustrup!</b>	<b>204</b>
16.1. JDK osztályok . . . . .	204
16.2. EPAM: It's gone. Or is it? . . . . .	205
16.3. EPAM: Kind of equal . . . . .	207
16.4. EPAM: Java GC . . . . .	209
<b>17. Helló, Gödel!</b>	<b>212</b>
17.1. EPAM: Mátrix szorzás Stream API-val . . . . .	212
17.2. EPAM: LinkedList vs ArrayList . . . . .	216
17.3. EPAM: Refactoring . . . . .	220
17.4. Gengszterek . . . . .	225
<b>18. Helló, (Unknown)! Helló, EPAM!</b>	<b>227</b>
18.1. EPAM: Titkos üzenet, száll a gépben! . . . . .	227
18.2. EPAM: ASCII Art . . . . .	235
18.3. OOCWC Boost ASIO hálózatkezelése . . . . .	238
18.4. BrainB . . . . .	239

<b>19. Helló, Lauda!</b>	<b>240</b>
19.1. EPAM: Kivételkezelés . . . . .	240
19.2. Port scan . . . . .	242
19.3. Junit teszt . . . . .	243
19.4. AOP . . . . .	244
<b>20. Helló, Calvin!</b>	<b>246</b>
20.1. Android telefonra a TF objektum detektálója . . . . .	246
20.2. MNIST . . . . .	247
20.3. EPAM: Back To The Future . . . . .	252
20.4. EPAM: AOP . . . . .	254
<b>IV. Irodalomjegyzék</b>	<b>258</b>
20.5. Általános . . . . .	259
20.6. C . . . . .	259
20.7. C++ . . . . .	259
20.8. Lisp . . . . .	259

# Ábrák jegyzéke

4.1. A double ** háromszögmátrix a memóriában . . . . .	33
5.1. A Mandelbrot halmaz a komplex síkon . . . . .	44
6.1. Polargenteszt.cpp futtatása . . . . .	65
6.2. Inorder fa bejárás . . . . .	69
6.3. Preorder fa bejárás . . . . .	70
6.4. Postorder fa bejárás . . . . .	71
7.1. Hangya szimuláció . . . . .	92
7.2. Hangya szimuláció . . . . .	109
7.3. Életjáték . . . . .	117
7.4. Életjáték . . . . .	117
7.5. BrainB Benchmark . . . . .	125

# Előszó

Amikor programozónak terveztem állni, ellenezték a környezetemben, mondván, hogy kell szövegszerkesztő meg táblázatkezelő, de az már van... nem lesz programozói munka.

Tévedtek. Hogy egy generáció múlva kell-e még tömegesen hús-vér programozó vagy olcsóbb lesz alkalmi igény szerint pár robot programozót a felhőből? A programozók dolgozók lesznek vagy papok? Ki tudhatná ma.

Minden esetre a programozás a teoretikus kultúra csúcsa. A GNU mozgalomban látom annak garanciáját, hogy ebben a szellemi kalandban a gyerekeim is részt vehessenek majd. Ezért programozunk.

## Funnily about programming laguages from Anupam Chugh

**Copyright Anupam Chugh, Independent iOS Developer**

C

I might be the oldest, but you still have to allocate me space. Pointing at others is my favorite hobby.

C++

You can hate us, despise us, but we'll find our ways into your framework. We believe in friends at work.

Java

We're getting old and were always verbose. But pay us and we'll give you long term support.

Python

We've made it possible to write pseudo-code in real even if you don't understand it. Not a semicolon or a bracket but extra whitespace is all it takes to cause chaos.

## Hogyan forgasd

A könyv célja egy stabil programozási szemlélet kialakítása az olvasóban. Módszere, hogy hetekre bontva ad egy tematikus feladatesokrot. minden feladathoz megadja a megoldás forráskódját és forrásokat feldolgozó videókat. Az olvasó feladata, hogy ezek tanulmányozása után maga adja meg a feladat megoldásának lényegi magyarázatát, avagy írja meg a könyvet.

Miért univerzális? Mert az olvasótól (kvázi az írótól) függ, hogy kinek szól a könyv. Alapértelmezésben gyerekeknek, mert velük készítem az iniciális változatot. Ám tervezem felhasználását az egyetemi programozás oktatásban is. Ahogy szélesedni tudna a felhasználók köre, akkor lehetne kiadása különböző korosztályú gyerekeknek, családoknak, szakköröknek, programozás kurzusoknak, felnőtt és továbbképzési műhelyeknek és sorolhatnánk...

## Milyen nyelven nyomjuk?

C (mutatók), C++ (másoló és mozgató szemantika) és Java (lebutított C++) nyelvekből kell egy jó alap, ezt kell kiegészíteni pár R (vektoros szemlélet), Python (gépi tanulás bevezető), Lisp és Prolog (hogy lássuk másat is) példával.

## Hogyan nyomjuk?

Rántsd le a <https://gitlab.com/nbatfai/bhax> git repót, vagy méginkább forkolj belőle magadnak egy sajátot a GitLabon, ha már saját könyvön dolgozol!

Ha megvannak a könyv DocBook XML forrásai, akkor az alább látható **make** parancs ellenőrzi, hogy „jól formázottak” és „érvényesek-e” ezek az XML források, majd elkészíti a dblatex programmal a könyved pdf változatát, íme:

```
batfai@entropy:~$ cd glrepos/bhax/thematic_tutorials/bhax_textbook/
batfai@entropy:~/glrepos/bhax/thematic_tutorials/bhax_textbook$ make
rm -f bhax-textbook-fdl.pdf
xmllint --xinclude bhax-textbook-fdl.xml --output output.xml
xmllint --relaxng http://docbook.org/xml/5.0/rng/docbookxi.rng output.xml ←
    --noout
output.xml validates
rm -f output.xml
dblatex bhax-textbook-fdl.xml -p bhax-textbook.xls
Build the book set list...
Build the listings...
XSLT stylesheets DocBook - LaTeX 2e (0.3.10)
=====
Stripping NS from DocBook 5/NG document.
Processing stripped document.
Image 'dblatex' not found
Build bhax-textbook-fdl.pdf
'bhax-textbook-fdl.pdf' successfully built
```

Ha minden igaz, akkor most éppen ezt a legenerált `bhax-textbook-fdl.pdf` fájlt olvasod.



### A DocBook XML 5.1 új neked?

Ez esetben forgasd a <https://tdg.docbook.org/tdg/5.1/> könyvet, a végén találod az informatikai szövegek jelölésére használható gazdag „API” elemenkénti bemutatását.

I. rész

Bevezetés

DRAFT

# 1. fejezet

## Vízió

### 1.1. Mi a programozás?

Ne cifrázzuk: programok írása. Mik akkor a programok? Mit jelent az írásuk?

Magam is ezeken gondolkozok. Szerintem a programozás lesz a jegyünk egy másik világba..., hogy a galaxisunk közepén lévő fekete lyuk eseményhorizontjának felületével ez milyen relációban van, ha egyáltalán, hát az homályos...

### 1.2. Milyen doksikat olvassak el?

- Kezd ezzel: <http://esr.fsf.hu/hacker-howto.html>!
- Olvasgasd aztán a kézikönyv lapjait, kezd a **man man** parancs kiadásával. A C programozásban a 3-as szintű lapokat fogod nézegetni, például az első feladat kapcsán ezt a **man 3 sleep** lapot
- C kapcsán a [**KERNIGHANRITCHIE**] könyv adott részei.
- C++ kapcsán a [**BMECPP**] könyv adott részei.
- Az igazi kockák persze csemegéznek a C nyelvi szabvány [ISO/IEC 9899:2017](#) kódcsipeteiből is.
- Amiből viszont a legeslegjobban lehet tanulni, az a [The GNU C Reference Manual](#), mert gcc specifikus és programozókra van hangolva: szinte csak 1-2 lényegi mondat és apró, lényegi kódcsipetek! Aki pdf-ben jobban szereti olvasni: <https://www.gnu.org/software/gnu-c-manual/gnu-c-manual.pdf>
- Az R kódok olvasása kis általános tapasztalat után automatikusan, erőfeszítés nélkül menni fog. A Python nincs ennyire a spektrum magától értetődő végén, ezért ahhoz olvasd el a [?] könyv 25-49, kb. 20 oldalas gyorstalpaló részét.

### 1.3. Milyen filmeket, előadásokat nézzek meg, könyveket olvassak el?

A kurzus kultúrájának elvezéséhez érdekes lehet a következő elméletek megismerése, könyvek elolvasása, filmek megnézése.

Elméletek.

- Einstein: A speciális relativitás elmélete.
- Schrödinger: Mi az élet?
- Penrose-Hameroff: Orchestrated objective reduction.
- Julian Jaynes: Breakdown of the Bicameral Mind.

Könyvek.

- Carl Sagan, Kapcsolat.
- Roger Penrose, A császár új elméje.
- Asimov: Én, a robot.
- Arthur C. Clarke: A gyermekkor vége.

Előadások.

- Mariano Sigman: Your words may predict your future mental health, <https://youtu.be/uTL9tm7S1Io>, hihetetlen, de Julian Jaynes kétkamarás tudat elméletének legjobb bizonyítéka információtechnológiai...
- Daphne Bavelier: Your brain on video games, <https://youtu.be/FktsFcooIG8>, az esporttal kapcsolatos sztereotípiák eloszlatására ( „The video game players of tomorrow are older adults”: 0:40-1:20, „It is not true that Screen time make your eyesight worse”: 5:02).

Filmek.

- 21 - Las Vegas ostroma, <https://www.imdb.com/title/tt0478087/>, benne a **Monty Hall probléma** bemutatása.
- Rain Man, <https://www.imdb.com/title/tt0095953/>, az [?] munkát ihlette, melyeket akár az **MNIST**-ek helyett lehet csinálni.
- Kódjátszma, <https://www.imdb.com/title/tt2084970/>, benne a **kódtörő feladat** élménye.
- Interstellar, <https://www.imdb.com/title/tt0816692/>.
- Middle Men, <https://www.imdb.com/title/tt1251757/>, mitől fejlődött az internetes fizetés?
- Pixels, <https://www.imdb.com/title/tt2120120/>, mitől fejlődött a PC?

- Gattaca, <https://www.imdb.com/title/tt0119177/>.
- Snowden, <https://www.imdb.com/title/tt3774114/>.
- The Social Network, <https://www.imdb.com/title/tt1285016/>.
- The Last Starfighter, <https://www.imdb.com/title/tt0087597/>.
- What the #\$\$\*! Do We (K)now!?, <https://www.imdb.com/title/tt0399877/>.
- I, Robot, [https://www.imdb.com/title/tt0343818.](https://www.imdb.com/title/tt0343818/)

Sorozatok.

- Childhood's End, <https://www.imdb.com/title/tt4171822/>.
- Westworld, <https://www.imdb.com/title/tt0475784/>, Ford az első évad 3. részében konkrétan meg is nevezi Julian Jaynes kétkamarás tudat elméletét, mint a hosztok programozásának alapját...
- Chernobyl, <https://www.imdb.com/title/tt7366338/>.
- Stargate Universe, <https://www.imdb.com/title/tt1286039>, a Desteny célja a mikrohullámú háttér struktúrája mögötti rejtély feltárása...
- The 100, <https://www.imdb.com/title/tt2661044/>.
- Genius, [https://www.imdb.com/title/tt5673782.](https://www.imdb.com/title/tt5673782/)

**II. rész**

**Tematikus feladatok**

DRAFT

**Bátf41 Haxor Stream**

A feladatokkal kapcsolatos élő adásokat sugároz a <https://www.twitch.tv/nbatfai> csatorna, melynek permanens archívuma a <https://www.youtube.com/c/nbatfai> csatornán található.

DRAFT

## 2. fejezet

# Helló, Turing!

### 2.1. Végtelen ciklus

Írj olyan C végtelen ciklusokat, amelyek 0 illetve 100 százalékban dolgoztatnak egy magot és egy olyat, amely 100 százalékban minden magot!

A végtelen ciklusban a feltétel minden teljesül (true) ezért az utasítások újra lefutnak, így nincs olyan feltétel ami miatt kilépne a ciklus. Például: for(;;)

Egy mag 0 százalékban:

```
int
#include <stdio.h>
int
main ()
{
    while(true) {
        sleep(100);

        return 0;
}
```

Magyarázat: Az unistd header tartalmazza a sleep() függvényt, ezért kell include-olni az stdio.h header (standart input/output) mellett. Az int main() a fő függvényünk, a while() pedig a ciklus. A ciklusba a feltételt a ()-ban adjuk meg és amíg ez igaz addig a {}-ban megadott utasítások végrehajtódnak és a ciklus újra és újra lefut. A példában a ciklus feltétele "true" ami azt jelenti, hogy a feltétel igaz, tehát a ciklus minden újraindul, amíg ki nem lőjük. A sleep(100) függvény pedig azért kell, mivel ez altatja a processzor folyamat szálát. A függvényben megadott érték jelenti azt, hogy hány másodpercig altatja a processzort, jelen esetben 100 ms-ig.

Egy mag 100 százalékban:

```
#include <unistd.h>
#include <stdio.h>

int main(){
```

```
while(true) {  
}  
return 0;  
}
```

Magyarázat: Ez az előző példához hasonló. Az include-ok és a ciklus magyarázata megegyezik, az előző példáéval. Itt annyi a különbség, hogy nincs benne a sleep() függvény, azaz a szál nincs altatva, így a végtelen ciklus 100%-ban dolgoztat 1 szálat.

Minden mag 100 százalékban:

```
#include <omp.h>  
#include <stdio.h>  
#include <omp.h>  
  
int main(void)  
{  
    #pragma omp parallel  
    {  
        for(;;)  
        {  
  
        }  
    }  
    return 0;  
}
```

A programunk, az előzőhez képest egy openmp-vel bővült. #pragma omp parallel sor adja azt az utasítást a gének, hogy a feladat az összes szálon fusson, vagyis párhuzamosan minden szálon. (Ezért a fordításnál -fopenmp kapcsoló szükséges még a parancsba.)

## 2.2. Lefagyott, nem fagyott, akkor most mi van?

Mutasd meg, hogy nem lehet olyan programot írni, amely bármely más programról eldönti, hogy le fog-e fagyni vagy sem!

Megoldás videó:

Megoldás forrása: tegyük fel, hogy akkora haxorok vagyunk, hogy meg tudjuk írni a Lefagy függvényt, amely tetszőleges programról el tudja dönteni, hogy van-e benne végtelen ciklus:

```
Program T100  
{  
  
    boolean Lefagy(Program P)  
    {  
        if(P-ben van végtelen ciklus)  
            return true;
```

```
    else
        return false;
}

main(Input Q)
{
    Lefagy(Q)
}
}
```

A program futtatása, például akár az előző v. c ilyen pszeudókódjára:

```
T100(t.c.pseudo)
true
```

akár önmagára

```
T100(T100)
false
```

ezt a kimenetet adja.

A T100-as programot felhasználva készítük most el az alábbi T1000-set, amelyben a Lefagy-ra épőlő Lefagy2 már nem tartalmaz feltételezett, csak csak konkrét kódot:

```
Program T1000
{

    boolean Lefagy(Program P)
    {
        if(P-ben van végtelen ciklus)
            return true;
        else
            return false;
    }

    boolean Lefagy2(Program P)
    {
        if(Lefagy(P))
            return true;
        else
            for(;;);
    }

    main(Input Q)
    {
        Lefagy2(Q)
    }
}
```

Mit fog kiírni erre a T1000 (T1000) futtatásra?

- Ha T1000 lefagyó, akkor nem fog lefagyni, kiírja, hogy true
- Ha T1000 nem fagyó, akkor pedig le fog fagyni...

akkor most hogy fog működni? Sehogy, mert ilyen Lefagy függvényt, azaz a T100 program nem is létezik.

Tanulság: Ha a T100-t és T1000-t létező programnak tekintjük és T1000-ben meghívjuk saját magát. A T100 alapján ha a programunkba van végtelen ciklus, akkor igaz (true) értéket ad a Lefagy program a Lefagy2 programnak. Tehát az is igaz (true) értéket fog adni, viszont ha a Lefagy hamis (false) értéket ad vissza akkor a Lefagy2 belép egy végtelen ciklusba és a program le fog fagyni. Olyan program tehát mint a T100, nem működik mivel ha egy olyan program érkezik bele amiben van végtelen ciklus, akkor a program leáll mert a ciklus nem áll meg.

## 2.3. Változók értékének felcserélése

Írj olyan C programot, amely felcseréli két változó értékét, bármiféle logikai utasítás vagy kifejezés násználata nélkül!

A feladat két változónak az értékeinek felcserélésére. Az  $x = 2$  és  $y = 3$  példában ez nem tűnik nehéznek ha egy segédváltozót használunk aminek megadjuk  $x$ -értékét aztán  $x$ -nek az  $y$ -értékét végül  $y$ -nak pedig értékül adjuk a segédváltozó értékét. A következő példában egy másfajta változócsere fogunk alkalmazni, amely számolással cseréli fel az  $x$  és  $y$  értékeit.

```
#include <stdlib.h>
#include <stdio.h>

int main() {
    int x = 2, y = 3;

    printf("%s\n%d %d\n", "kulonbseggel:", x, y);

    x -= y;
    y += x;
    x = y-x;

    printf("%d %d\n", x, y);

    return 0;
}
```

Magyarázat: A fejlécet már ismerjük a 2.1-ből. A printf() függvény a kiíratást végzi, benne az első argumentum a a kíratás formátuma, a többi pedig a változók kiíratása. A „%s” azt jelenti, hogy egy szöveget fogunk kiíratni, amit a "%d" követ amely, egész típusú változót jelent, a „\n” pedig a sortörést jelenti. Aztán egy kis matematikai számítás, végül újra egy kiíratás, hogy megmutassuk, hogy a változók felcserélődtek.

Elsősorban egy kis bevezető:  $x += y$  egyenlő  $x = x + y$  kifejezéssel (ez a formátum csak egy rövidítés, későbbi programozásban hasznos dolog lesz)

A lépések egyszerűbben a következők:  $x = 2, y = 3$   $x := 3$  így az  $x$  értéke  $-1$   $y += -1$  ez azt jelenti hogy  $y$ -hoz hozzáadjuk az  $x$ -et (a  $-1$ -t) így  $x = -1$  és  $y = 2$   $x = y - x$  azaz  $x = 2 - (-1) = 2 + 1 = 3$  tehát az  $x$  értéke 3 és az  $y$  értéke 2 lett. Így kész is van a csere.

## 2.4. Labdapattogás

Megoldás forrása: <https://bhaxor.blog.hu/2018/08/28/labdapattogas>

```
#include <stdio.h>
#include <curses.h>
#include <unistd.h>

int main ( void )
{
    WINDOW *ablak;
    ablak = initscr ();

    int x = 0;
    int y = 0;

    int xone = 0, yone = 0;

    int xmax, ymax;

    for(;;)
    {
        getmaxyx ( ablak, ymax, xmax );
        mvprintw ( y, x, "O" );
        refresh();
        usleep (100);

        x = x + xone;
        y = y + yone;

        if ( x >= xmax-1 ) {
            xone *= -1;
        }
        if ( x <= 0 ) {
            xone *= -1;
        }
        if ( y >= yone-1 ) {
            yone *= -1;
        }
        if ( y >= ymax-1 ) {
            yone *= -1;
        }
    }
}
```

```
    }

    return 0;
}
```

Magyarázat: Az új dolog ami a fejlécnél feltűnik az a curses.h header. Ez képernyő kezelési függvényeket tartalmaz, és a program megjelenítéséhez szükségünk van rá. A main() függvényben a "void" kifejezés azt jelenti, hogy csak megjelenítünk a képernyőn valamit.

```
WINDOW *ablak;
ablak = initscr();
```

Így formázzuk meg a kimenetet. Az initscr() függvény curses módba lépteti a terminált.

A deklarált x-en és y-on lesz a kezdő értékünk. Az xone és yone pedig a lépésközöt mutatja, jelen esetben 1. (lépésenként a koordináta rendszeren xone, yone-al való elmozdulást). Az xmax és ymax lesznek a határértékek, hogy a program csak az ablakon belül mozogjon.

A végtelen ciklus miatt a labda pattogás nem áll ki nem lőjük a programot. A getmaxyx() függvény meghatározza az ablak méretét, a refresh() függvény pedig az ablakot frissíti. A mvprint() függvény az x és y koordináta tengelyen megjeleníti jelen esetben az "O" karaktert. A usleep() függvény altatja a ciklust, azaz mennyi időn belül induljon újra a ciklus, tehát a labda pattogásának sebességét is ezzel megadjuk.

```
x = x + xone;
y = y + yone;
```

Megnöveljük az értékeket, minden ciklus lefutásnál (mozog a "labda"). A következő négy if-el pedig azt vizsgáljuk, hogy a labda az ablak szélén van e, ha igen akkor -1 -el szorozzuk így a labda irányt változtat. A fordításnál -Incourses kapcsolót is kell használnunk a fejlécben megjelenő curses.h miatt.

Egy másik megoldás az ""if" logikai feltételek használata nélkül:

```
#include <stdio.h>
#include <stdlib.h>
#include <curses.h>
#include <unistd.h>

int
main (void)
{
    int xj = 0, xk = 0, yj = 0, yk = 0;
    int mx = 80 * 2, my = 24 * 2;

    WINDOW *ablak;
    ablak = initscr();
    noecho();
    cbreak();
    nodelay(ablak, true);

    for (;;)
    {
        xj = (xj - 1) % mx;
```

```
    xk = (xk + 1) % mx;

    yj = (yj - 1) % my;
    yk = (yk + 1) % my;

    clear ();

    mvprintw (0, 0,
              " ↔
              -----  

              " );
    mvprintw (24, 0,
              " ↔
              -----  

              " );
    mvprintw (abs ((yj + (my - yk)) / 2),
              abs ((xj + (mx - xk)) / 2), "X");

    refresh ();
    usleep (1)
}
return 0;
}
```

Magyarázat: A prgoramunk ugyan azt csinálja mint az "if"-es változata. Csak ugye most logikai kifejezés, utasítás nélkül. A megoldáshoz szükségünk van matematikai számításokra, ehez deklarálunk egész tipusú változókat. A számításokat egy végtelen ciklusban számoljuk és mvprintw-vel íratjuk ki a képernyőre. A clear()-el minden egyes számítás előtt letisztítjuk az ablakot, az eslő kettő mvprintw-vel a felső és alsó határokat rajzoljuk ki, a harmadikkal pedig a "Labdát". Az usleep() függvény itt is a pattogás sebbeségét határozza meg.

## 2.5. Szóhossz és a Linus Torvalds féle BogoMIPS

Ez a program a gépünk szó hosszát fogja kiírni (jelen esetben az intiger tipus méretét), azaz az int méretét. A BogoMIPS a processzorunk sebbeségét lemérő program amit Linus Torvalds írt meg, a BogoMIPS-ben használt while ciklus feltételeivel írjuk meg a programot.

```
#include <stdio.h>

int main()
{
    int x = 1;
    int bit = 0;

    do
        bit++;
    while (a<<=1);
```

```
    printf ("%d %s\n", bit, "bites a szohossz.");  
}
```

A fejlécet ismerjük, a main() függvény a fő függvényünk, amelyben megadunk egy változónak egy tetszőleges értéket, jelen esetben 1-t. A bit változó számolni fogja, hogy hányszor fut le a ciklus. A programot hárultesztelős ciklussal do{}while() ciklussal futtatjuk, mivel az előtetsztelő while() ciklus nem számolná bele az első lépést.

A ciklus addig fut újra és újra amíg az x értéke nem 0. Tehát az x értéke kezdetben 1, a bináris értéke pedig 0001, a << (bitshift) operátor csak annyit csinál, hogy egy 0-val eltolja az 1-et, tehát a 0001-ből egy lépés után 0010 lesz ami 2, továbbá a második lépés után 0100 ami 4. A ciklus tehát addig fut amíg csupa 0 érték lesz a gépünk szóhossza, azaz az 1-est kitolja a szóhosszból, így az értékünk 0 lesz, a while ciklus befejeződik és a printf kíratja a bit értékét vagyis hogy hányat lépett az 1-es (hányszor futott le a ciklus), ez az szám megadja hogy hány bites a szóhossz (jelen esetben az int típus 32 bites lesz),

## 2.6. Helló, Google!

A PageRank egy keresőmotor a Googleban. A programot két fiatal írta meg 1998-ban, nevét az egyik kitalálója Larry Page után kapta.

A következőben, egy 4 weblapból álló PageRank-ét fogunk megnézni. A lapok PageRank-ét az alapján nézzük, hogy hány oldal osztotta meg a saját honlapján az oldal hiperlinkjét.

```
#include <stdio.h>  
#include <math.h>  
  
void kiir (double tomb[], int db)  
{  
    int k;  
    for (k = 0; k < db; ++k)  
        printf ("%f\n", tomb[k]);  
}  
  
double tavolsag (double PageR[], double PageRmatrix[], int n)  
{  
    double osszeg = 0.0;  
    int i;  
  
    for (i = 0; i < n; ++i)  
    {  
        osszeg += (PageRmatrix[i] - PageR[i]) * (←  
            PageRmatrix[i] - PageR[i]);  
    }  
    return sqrt(osszeg);  
}  
  
int main (void)  
{  
    double Honlap[4][4] = { {0.0, 0.0, 1.0 / 3.0, 0.0},
```

```
    {1.0, 1.0 / 2.0, 1.0 / 3.0, 1.0},  
    {0.0, 1.0 / 2.0, 0.0, 0.0},  
    {0.0, 0.0, 1.0 / 3.0, 0.0} };  
  
double PageR[4] = { 0.0, 0.0, 0.0, 0.0 };  
  
double PageRmatrix[4] = { 1.0 / 4.0, 1.0 / 4.0,  
    1.0 / 4.0, 1.0 / 4.0 };  
  
int i, j;  
  
for (;;) {  
    for (i = 0; i < 4; ++i)  
    {  
        PageR[i] = 0.0;  
        for (j = 0; j < 4; ++j)  
        {  
            PageR[i] += (Honlap[i][j] * PageRmatrix[j]);  
        }  
    }  
}  
  
for (i = 0; i < 4; ++i)  
{  
    PageRmatrix[i] = PageR[i];  
}  
}  
  
kiir (PageR, 4);  
  
return 0;  
}
```

A math.h header tartalmazza a matematikai számításokhoz szükséges függvényeket. A main() fügvénnyben létrehozunk egy mátrixot, ami a lapok összeköttetését adja meg. Ha az érték 0 akkor a lap nincs összekötve az adott lappal és önmagával sincs. Ahol 1/2 vagy 1/3 az érték az oldal másik oldallal való összeköttetését jelenti.

Például az 1/2: Az oldal 2 oldallal van összekötve.

A PageR tömb fogja a PageRank értéket tárolni. A PageRmatrix tömb pedig a mátrixal való számításokhoz kell. A következő lépés egy végtelen ciklus, ez majd a számítások végén a "break" parancsal lép ki, ha a megadott feltétel teljesül.

A for ciklusban van maga a PageRank számítása ami a tavolság() függvényt is meghívja, és egy részszámolást tartalmaz. A végtelen cikluson belül lévő ciklusok azért 4-ig mennek mert 4 weblapot nézünk. A ciklusbol a "break" parancsal lépünk ki ha a tavolsag() függvényben kapott eredmény kisebb mint 0.00000001. A végén a kiir() függvény megkapja a PageR értékeit és az weblapok számát és kiíratja.

## 2.7. 100 éves a Brun téTEL

A téTEL kimondja hogy az ikerprímek reciprokösszege a Brun konstanthoz konvergál, ami egy véges érték. A téTEL Viggo Brun-ről nevezték el a téTELt aki bebizonyította 1919-ben.

Megoldás forrása: <https://github.com/RubiMaistro/Prog1/blob/master/burn.R>

Az R nyelvű matlab könyvtár telpítési parancsai:

```
sudo apt-get install r-base
```

```
sudo apt-get install libopenblas-base r-base
```

```
sudo apt-get install gdebi
```

```
cd ~/Downloads
```

A végtelen cikluson belül lévő ciklusok azért 4-ig mennek mert 4 weblapot nézünk. A ciklusbol a "break" parancsal lépünk ki ha a tavolsag() függvényben kapott eredmény kisebb mint 0.00000001. A végén a kiir() függvény megkapja a PageR értékeket és az weblapok számát és kiíratja. rstudio-xenial-1.1.379-amd64.deb

A számoláshoz kell egy matlab könyvtár. A program fő része az stp függvény, a függvény megkapjat. X egy szam lesz ami megmondja meddig kell a prímeket számolni. Ehez a primes függvényt használjuk. A primes(x) kiírja x-ig a prímeket. A diff vektorban eltároljuk a primes vektorban tárolt egymás melletti prímek különbségét. A számítást úgy végezzük, hogy a 2-es prímszámtól indulva kivonjuk a prímből az előtte lévő prímet. Az idx el vizsgaljuk meg, hogy mely prímek különbsége 2 és ezek hol vannak (a helyüket a which függvény adja meg). A t1primes vektorban elhelyezzük ezeket a prímeket. A t2primes vektorbapedig amiezeknélkett" ovelnagyobb (azaz ikerprímek). rt1plust2 vektorban végezzük a reciklopépzést és a pár reciprokát összeadjuk. A returnban pedig a sum függvénnnyel vissza adjuk ezek összegét. Végül a plot() függvényel lerajzoljuk grafikusan.

## 2.8. A Monty Hall probléMA

Bevezetés a megértéshez: A kérdés vagy probléma egy vetélkedő játékból indul, amelyben van 3 ajtó és az egyik mögött egy értékes autó van, a másik kettő mögött 1-1 kecske, és amelyiket választja a játékos azt nyereményként megkapja. A versenyzőnek a 3 ajtó közül választania kell vagy sem.

Megoldás: Első ránézésre mindenki azt mondaná, hogy nem számít, hogy vált-e vagy sem mert 50-50% az esélye, hogy melyik ajtó mögött van az autó. Mivel már nem 3 hanem 2 ajtó közül kell választani, így már figyelembe se veszik azt a harmadik ajtót. De a megoldás az, hogy nagyobb az esélyünk akkor ha az előző döntésünket megváltoztatjuk és a másik ajtót választjuk.

Magyarázat: Kezdetben 3 ajtó közül 1 ajtót kell választanunk, azaz 1/3 az esélye, hogy eltaláljuk a jó megoldást és 2/3 hogynem. Ezek után a műsorvezető kinyit egy ajtót ami mögött nincs a nyeremény. Ez a valószínűségen nem változtat, úgyanúgy 1/3 eséllyel választottuk azt az ajtót ami mögött a nyeremény van. Viszont azok az ajtók közül ami mögött nincs semmi, már csak az egyik van csukva. Biztosra tudjuk, hogy a nyeremény a maradék két ajtó közül valamelyik mögött van. Tehát 2/3 az esélye annak, hogy a másik ajtó mögött van a nyeremény, mivel ha elsőre azt az ajtót választottuk amelyik mögött egy kecske van, és amikor megkapjuk a változtatásra a lehetőséget és élük a lehetőséggel és a másik ajtót választjuk, akkor biztosan az autót megnyerjük.

A problémával kapcsolatban egy R nyelvben írt szimuláció a következő:

```
kiserletek_szama=10000
kiserlet = sample(1:3, kiserletek_szama, replace=T)
jatekos = sample(1:3, kiserletek_szama, replace=T)
musorvezeto=vector(length = kiserletek_szama)

for (i in 1:kiserletek_szama)
{
  if(kiserlet[i]==jatekos[i])
  {
    mibol=setdiff(c(1,2,3), kiserlet[i])
  }

  else
  {
    mibol=setdiff(c(1,2,3), c(kiserlet[i], jatekos[i]))
  }
  musorvezeto[i] = mibol[sample(1:length(mibol),1)]
}

nemvaltoztatesnyer = which(kiserlet==jatekos)
valtoztat=vector(length = kiserletek_szama)
for (i in 1:kiserletek_szama)
{
  holvált = setdiff(c(1,2,3), c(musorvezeto[i], jatekos[i] ←
    []))
  valtoztat[i] = holvált[sample(1:length(holvált),1)]
}
valtoztatesnyer = which(kiserlet==valtoztat)

sprintf("Kiserletek szama: %i", kiserletek_szama)
length( nemvaltoztatesnyer )   length( valtoztatesnyer )
length( nemvaltoztatesnyer ) / length( valtoztatesnyer )
length( nemvaltoztatesnyer ) + length( valtoztatesnyer )

}
```

A kísérletet 10000x fogjuk eljátszani, a kísérlet vektorban 1 és 3 "ajtó" közül választunk 10000x. A replace=T-vel tesszük lehetővé, hogy egy eredmény többször is kijöhessen. A játékos valasztásait a játékos vektornál ugyan így meghatározzuk. A sample() fügvénnyel végezzük a kiválasztást. A műsorvezető vektort a length függvényel a kísérletek számával tesszük egyenlővé. Következik a for ciklus ami i=1 től a kísérletek számáig fut (10000). A ciklusban egy feltétel vizsgálat következik, az if-el megvizsgáljuk, hogy a játékos álltal választott ajtó megegyezik-e a kísérletben szereplő ajtóval. Ha a feltétel igaz egy vektorba bele tesszük azokat az ajtokat amiket a játékos nem választott, az else ágon pedig ha a feltétel nem igaz, akkor azt az ajtót eltároljuk amit nem ő választott és a nyereményt rejti ajtót.

A műsorvezető vektorban pedig azt az ajtót amit ki fog nyitni. A nemvaltoztat és nyer vektorban azok az esetek vannak amikor a játékos azt az ajtót választotta elsőre ami mögött az ajtó van és nem változtat a döntésén. A valtoztat vektorban pedig azt mikor megváltozatja a döntését és így nyer ezt egy for ciklussal vizsgáljuk. A legvégén kiíratjuk az eredményeket, hogy melyik esetben hányszor nyert.

## 2.9. Minecraft-MALMÖ bevezető

Egy kis bevezető tájékoztatást szeretnék adni a MALMÖ projektel kapcsolatban amellyel a továbbiakban nagyon érdekes és különleges feladatokat oldhatunk meg.

A **MALMÖ** egy kreativitást igénylő projekt melynek az alapja a nagy többség által ismert **Minecraft** nevű játék. A Minecraft Malmö projekt a Mojang fejlesztése, melynek első változtatát **2016** júliusában publikáltak, a projekt a mesterséges intelligenciára alapoz.

A projekt vezetői: **Katja Hofmann** fő kutató, **Andre Kramer** kutató mérnök és még sok más kutató. Céljuk a projekttel, hogy a Mesterséges Intelligencia ágát egy új környezetbe építve fejlesszék, kutassák és hogy sokan beszálljanak a projektbe, ezért is választották a Minecraft nevű játékot, ami nagy ismeretkörrel rendelkezik és ideális teret és körülményeket ad ennek a projektnek.

A projektben számtalan kreatív lehetőséggel rendelkezünk ahogyan ezt a projekt vezetői is említi. A lehetőségünk az ágens irányításával kezdve a különböző blockok azonosításával és a környezet felismerésével a blockok mozagtásán át, a különböző sziutációkban, az npc és mob közelség reakcióig és még tovább egy kreatív programozó és fan számára kimeríthatetlen lehet.

Ebben könyvben a MALMÖ projekt feladataiban csak a projekt egy minimális részét fogjuk érinteni.

A **Red Flower Hell** repóban különböző érdekes programkódok elérhetők.

Ebben az évben a DE-IK PTI karon a Red Flower Hell (RFH) MALMÖ projekttel foglalkozunk. Ennek célja, hogy a hallgatókat a mesterséges intelligencia kutatása és programozása felé ösztönözze, melynek nem tudhatjuk mennyire hasznos értéke lesz a mesterséges intelligencia tudomány fejlődésére nézve.

A **MALMÖ projekt** hivatalos oldala itt elérhető ahol több és részletesebb információt kaphatunk a projekttel kapcsolatban.

Valamint amikor már elérünk a Schwarzenegger fejezethez, kaphatunk egy rövid összefoglalót is az általunk kidolgozott projekt részekből.

## 2.10. Vörös Pipacs Pokol/csiga folytonos mozgási parancsokkal

Megoldás videó: <https://youtu.be/uA6RHxXH840>

Megoldás forrása elérhető a követekő linken:

<https://github.com/nbatfai/RedFlowerHell>

Link saját repóból: [Csiga mozgása folytonosan](#)

A feladat alapja, hogy Steve az ágens, csiga vonalban haladva jussok minél feljebb a tölcsérszerű aréna aljáról a tetejére ameddig a láva engedi ami ugyanis folyik fentről lefelé és Steve amint a lávát eléri meghal.

A megoldásához folytonos mozgást használunk. A **self.agent\_host.sendCommand(utasítás)** parancs szükséges ahol az utasítás helyen kell megadni Stevenek mit csináljon. Ezek a mozgást leíró utasítások, melyek paraméterét alapvetően, úgy kell használni mint a kapcsolókat, ha 0 az értéke nem, ellenben ha 1 akkor elvégezi a mozgást.

Minden szint falának elérésekor ugrik egyet a **jumpmove** parancsal, és jobbra fordulva megy tovább. minden újabb körben feljebb-feljebb jut Steve és így a körök is egyre hosszabbak lesznek. A mozgás időtartamát

a **time.sleep()** parancssal adhatjuk meg, azaz az argumentumában megadott másodpercig fut. minden kör egyértelműen 4 hosszú előre haladásból áll, ez a mozgás alapja, ezt kell kiegészíteni a fordulásokkal és ugrásokkal.

DRAFT

## 3. fejezet

# Helló, Chomsky!

### 3.1. Decimálisból unárisba átváltó Turing gép

Állapotátmenet gráfjával megadva írd meg ezt a gépet!

A gép a nevét Alan Truring után kapta 1936-ban. A gép decimális számrendszerből unáris számrendszerbe írja át a számot. Az unáris számrendszer másnéven egyes számrendszer. A gép úgy működik, hogy csak 1-eseket ír. Tehát például a 6-ost átírva 6 darab 1-est ír le unáris számrendszerbe átváltva, az 1-es helyett csak egy 1-est ír.

A Turing gép kódja c++ nyelven a következő:

```
#include <iostream>
using namespace std;
int main() {
    int a;
    int tiz=0, szaz=0;
    cout<<"Decimalis szam:\n";
    cin>>a;
    cout<<"A szam unarisban:\n";
    for (int i=0; i<a; i++) {
        cout<<"1";
        ++tiz;
        ++szaz;
    if (tiz==10) {
        cout<<" ";
        tiz=0;
    }
    if (szaz==100) {
        cout<<"\n";
        szaz=0;
    }
}
return 0;
}
```

A kódban egyszerűen csak egy fő függvényt használtunk a main()-t, ebben bekértünk egy tetszőleges számot és ezt alakítjuk unárisba. Egy for ciklust meghtároztuk, hogy 0-tól induljon ( $i=0$ ) és egészen addig fussen amíg eléri azt a számot amit megadtunk

( $i < a$ )

, persze minden lefutása után eggyel ( $i++$ ) növekedjen. A for ciklusban mindenki írunk egy 1-est, a két segédváltozót csak az áttekinthetőség miatt használjuk, tehát amikor már kiírtunk egymás után 10 darab 1-est íratunk egy szóközt, ha 100 darab 1-est akkor egy sortörést.

### 3.2. Az $a^n b^n c^n$ nyelv nem környezetfüggetlen

A generatív nyelvek kidolgozása Noam Chomsky nevéhez fűződik. A nyelveket osztályokba rendezzük, vannak erősebb és gyengébb osztályok és az erősebb osztály képes létrehozni gyengébb osztályt. Négy darab alapon fekszik a generatív nyelvtan:

- Terminális szimbólumok, a konstansok.
- Nem terminális jelek, a változók.
- Kezdőszimbólum, egy kijelölt szimbólum.
- Helyettesítési szabályok, ezzel a szavakat értelmezzük majd.

Legyenek a nyelv változói:

X Y Z

És legyenek a nyelv konstansai:

a b c

A helyettesítési szabályok:

$X \rightarrow abc, X \rightarrow aYbc, Yb \rightarrow bY, Yc \rightarrow Zbcc, bZ \rightarrow Zb, aZ \rightarrow aaY, aZ \rightarrow aa$

X ( $X \rightarrow aYbc$ )  
aYbc ( $Yb \rightarrow bY$ )  
abYc ( $Yc \rightarrow Zbcc$ )  
abZbcc ( $bZ \rightarrow Zb$ )  
aZbbcc ( $aZ \rightarrow aa$ )  
aabbcc

X ( $X \rightarrow aYbc$ )  
aYbc ( $Yb \rightarrow bY$ )  
abYc ( $Yc \rightarrow Zbcc$ )  
abZbcc ( $bZ \rightarrow Zb$ )  
aZbbcc ( $aZ \rightarrow aaY$ )  
aaYbbcc ( $Yb \rightarrow bY$ )  
aabYbcc ( $Yb \rightarrow bY$ )  
aabbbYcc ( $Yc \rightarrow Zbcc$ )  
aabbbZbcc ( $bZ \rightarrow Zb$ )  
aabZbbccc ( $bZ \rightarrow Zb$ )  
aaZbbbccc ( $aZ \rightarrow aa$ )  
aaabbbcc

Azt láthatjuk, hogy addig alkalmazzuk a helyettesítési szabályokat míg csak konstansaink lesznek. Azaz mindenkorban osztályt hozunk létre.

### 3.3. Hivatkozási nyelv

Ahogy a beszélt nyelv, úgy a programozási nyelv is fejlődik. Ennek a bemutatására az alábbi programot fogjuk használni:

```
#include <stdio.h>

int main()
{
    for(int i=0;i<1;i++)
        printf("Lefut");

}
```

A program egyszerű, kiíratja a for ciklus a "Lefut" szöveget. A kódot viszon több nyelvtanban is fordíthatjuk. Ha a C89-es nyelvtannal fordítjuk a kódot akkor "gcc -std=gnu89 fajlnev.c -o fajlnev"-et használom. Ekkor a program hibát fog kiírni a for ciklusnál, mert a C89-es nyelvtanban a for cikluson belül deklaráljuk az i változót, mert ebben a régebbi nyelvtanban még erre nincs lehetőségünk, csak cikluson kívül.

De viszont ha C99-es nyelvtannal fordítjuk "gcc -std=gnu99 fajlnev.c -o fajlnev"-et használom, akkor a kód hiba nélkül lefut, mivel ebben már szerepel az a lehetőség, hogy cikluson belül deklarálhatunk változókat a C89-es nyelvtannal ellentétben.

Tehát a tanulság, hogy egy programozási nyelven belül sem mindegy milyen tipusú nyelvtannal fordítjuk a kódunkat.

### 3.4. Saját lexikális elemző

A program a bemeneten megjelenő valós számokat összeszámolja. A lexikális elemző kódja:

```
%{
#include <string.h>
int szamok=0;
%}

%% [0-9] +
{++szamok;} %%

int main()
{
    yylex();
    printf("%d szam", szamok);
    return 0;
}
```

A számokat változóval számoljuk, hogy hányszor fordul elő szám a bemenetben. A programot a %-jelekkel osztjuk fel részekre. a [0-9]+ { ++szamok; }

Ez a sor adja azt, hogy 0-9 vagy nagyobb számot talál akkor növelte a "szamok" változót. A printf el pedig csak kiíratjuk hogy hány szám volt a bemenetben (ez az elemzés).

A yylex() a lexikális elemző a fordítás a következő:

```
flex program.l
```

ez készít egy "lex.cc.y" fájlt.

Ezt az alábbi módon futtatjuk:

```
cc lex.yy.c -o program_neve -lfl
```

A futtatáshoz pedig hozzá kell csatolni a vizsgált szöveget.

### 3.5. Leetspeak

Tutor: Kikina Dominik

Lexelj össze egy l33t ciphert!

```
% {  
  
    #include <string.h>  
    int szamok=0;  
} %  
  
%%  
"0" { printf("O"); }  
"1" { printf("I"); }  
"2" { printf("Z"); }  
"3" { printf("E"); }  
"4" { printf("A"); }  
  
"5" { printf("S"); }  
"6" { printf("b"); }  
"7" { printf("T"); }  
"8" { printf("B"); }  
"9" { printf("P"); }  
  
"O" { printf("0"); }  
"I" { printf("1"); }  
"Z" { printf("2"); }  
"E" { printf("3"); }  
"A" { printf("4"); }  
  
"S" { printf("5"); }
```

```
"b" { printf("6"); }
"T" { printf("7"); }
"B" { printf("8"); }
"P" { printf("9"); }

%%

int main()
{
yylex();
printf("%d szam", szamok);
retrun 0;
}
```

Ez a nyelv lefordítja a l33t nyelven írt szöveget vagy a l33t nyelvre írja át. A program működése hasonló az előzőhöz, csak itt a megadott számokat keresi és helyettük a l33t nyelvben a nekik megadott megfelelő betűt írja helyette. Az ellenkező esetben pedig ha l33t nyelvre akarjuk átírni a szöveget, akkor a megadott betűket keresi és alakítja át számokká.

### 3.6. A források olvasása

Hogyan olvasod, hogyan értelmezed természetes nyelven az alábbi kódcsipeteket? Például

```
if(signal(SIGINT, jelkezelő)==SIG_IGN)
    signal(SIGINT, SIG_IGN);
```

Ha a SIGINT jel kezelése figyelmen kívül volt hagyva, akkor ezen túl is legyen figyelmen kívül hagyva, ha nem volt figyelmen kívül hagyva, akkor a jelkezelő függvény kezelje. (Miután a **man 7 signal** lapon megismertem a SIGINT jelet, a **man 2 signal** lapon pedig a használt rendszerhívást.)

#### Bugok

Vigyázz, sok csipet kerülendő, mert bugokat visz a kódba! Melyek ezek és miért? Ha nem megy ránézésre, elkapja valamelyiket esetleg a **splint** vagy a **frama**?

```
if(
    signal(SIGINT, SIG_IGN) !=SIG_IGN)
    signal(SIGINT, jelkezelő);
```

A példában szereplő kód részlet ellentetje, azaz ha a SIGNT jel kezelése nem lett figyelmen kívül hagyva akkor, a jelkezelő függvény kezelje.

```
int i=0;

for(i=0; i<5; )
    printf("(%d) ", ++i)
```

```
(1) (2) (3) (4) (5)
Process returned 0 (0x0)    execution time : 0.069 s
Press any key to continue.
```

Ez egy for ciklus, a benne lévő i változó kezdőértéke 0, a ciklus addig fut le újra és újra amíg az i értéke kisebb mint 5. Ebben az esetben az i értékét még a lefutás előtt mindig növeljük 1-el, mivel " $++i$ "-ként használjuk, ebben az esetben az  $i=0$ -ra ha teljesül a feltétel akkor az  $i=1$ -el (tehát mindenkor a ráköveztkezőjére) fut le a ciklus.

```
int i=0;

for(i=0; i<5;)
    printf("(%d) ", i++)
```

```
(0) (1) (2) (3) (4)
Process returned 0 (0x0)    execution time : 0.100 s
Press any key to continue.
```

Ez a for ciklus hasonló az előzőhez, a különbség viszont az, hogy az i értékét csak a ciklus lefutása után növeljük 1-el, ebben az esetben pedig  $i=0$ -ra ha teljesül a feltétel akkor szintén  $i=0$ -val fut le a ciklus.

```
int i=0;
int tomb[6];

for(i=0; i<5; tomb[i] = i++)
    printf("%d ", tomb[i]);
}
```

```
4200816 0 1 2 3
Process returned 0 (0x0)    execution time : 0.062 s
Press any key to continue.
```

Ez a for ciklus egy tömböt feltölt az i értékével. Nagyon érdekes módon történik ez mivel a tömb első eleme memóriaszemét lesz, ez azért történik meg mert amikor  $i=0$ -tól indul a for ciklus a tomb is az első eleménél az az a 0.-nál tart. Bug: A for cikluson belül semmi nem történik ezért az első értékkadás a tombbe úgymond nem sikerül, majd végrehajtódik a for ciklus harmadik utasítása a léptetés, ahol értéket adunk a tombba és az i 1-el növekszik, ezért a for ciklus minden futásnál megkapja az aktuális i értékének az előző futás kezdőértéket, ezért 0-tól kezdve felvész minden i-t 0-tól indulva, de a tomb utolsó eleme 3 lesz, mert 5-ig alapból nem megy el.

```
int i=0, n=10, a=10;
int *d, *s;
*d = a;
```

```
s = d;
for(i=0; i<n && (*d++ = *s++); ++i)
{
    printf("%d %d %d\n", i, s, d);
}
```

```
0 2945028 2945028
1 2945032 2945032
2 2945036 2945036
3 2945040 2945040
4 2945044 2945044
```

```
Process returned 0 (0x0) execution time : 0.069 s
```

Ebben az esetben a for ciklusban két feltétel van, tehát akkor fut le a for ciklus ha mind a két feltétel teljesül. Az első feltétel, hogy  $i < n$ . A második feltétel, hogy  $d$  és  $s$  mutató egyenlő, és minden ciklusnál növeljük a értékeket. A két feltételt és operátorral füzzük össze. Bug: A második feltételt célszerűbb lenne a for cikluson belül if-ekkel vizsgálni, mert nem logikai feltétel.

```
int f(int a, int b){
    return a+b;
}
int main() {

    int a = 0;
    printf("%d %d %d", f(a,++a), f(++a,a), a);
    return 0;
}
```

```
4 2 2
Process returned 0 (0x0) execution time : 0.137 s
Press any key to continue.
```

A printf() függvény kiírja az összeget a képernyőre. Itt két egész típusú (%d) változót, az f() függvénnyel határozzuk meg a számot, a hiba csak az, hogy nem megfelelő a sorrend.

```
int f(int a){
    a+=a;
    return a;
}
int main() {

    int a = 1;
    printf("%d %d", f(a), a);
    return 0;
}
```

```
26 13
Process returned 0 (0x0)    execution time : 0.154 s
Press any key to continue.
```

A printf() függvény kiír két egész számot, az első számot az "a"-t a f() függvény határozza meg itt 2 lesz és a második pedig maga az "a" változó értéke az 1.

```
int f(int a){
    return a;
}
int main(){

    int a = 1;
    printf("%d %d", f(&a), a);
    return 0;
}
```

```
6422300 214
Process returned 0 (0x0)    execution time : 0.078 s
Press any key to continue.
```

A printf() függvény kiír két egész számot, amiben eltér az előző kódcsipettől, hogy itt az f() függvény az a változó memóriacímét kapja meg.

### 3.7. Logikus

Hogyan olvasod természetes nyelven az alábbi Ar nyelvű formulákat?

```
$ (\forall x \exists y ((x < y) \wedge (y \text{ prim}))) $

$ (\forall x \exists y ((x < y) \wedge (y \text{ prim})) \wedge (S y \text{ } \leftrightarrow \text{ prim})) $

$ (\exists y \forall x (x \text{ prim}) \supset (x < y)) $

$ (\exists y \forall x (y < x) \supset \neg (x \text{ prim})) $
```

A természetes nyelvet mi emberek megértjük és ennek függvényében gondolkodunk és cselekszünk, viszont létre kellett hoznunk egy nyelvelt jelen esetben az Ar nyelvet amellyel a számítógépünkkel is tudunk kommunikálni. Egyszerűen csak meg kell tanulnunk melyik parancs mit jelent, az Ar nyelv egy komplex nyelv. Vannak benne logikai összekötőjelek (például: és = \wedge, nem = \neg, vagy = \vee, implikáció(ha

A, akkor B) = \supset). Vannak kvantorok (például: "létezik" = \exists és a " minden" = \forall). Az "S" értéknövelés, a kiírást pedig a \text{-el végezzük.

### 3.8. Deklaráció

Vezesd be egy programba (forduljon le) a következőket:

- egész
- egészre mutató mutató
- egész referenciajá
- egészek tömbje
- egészek tömbjének referenciajá (nem az első elemé)
- egészre mutató mutatók tömbje
- egészre mutató mutatót visszaadó függvény
- egészre mutató mutatót visszaadó függvényre mutató mutató
- egészet visszaadó és két egészet kapó függvényre mutató mutatót visszaadó, egészet kapó függvény
- függvénymutató egy egészet visszaadó és két egészet kapó függvényre mutató mutatót visszaadó, egészet kapó függvényre

```
#include <iostream>
int main()
{
    int a;
    int *b=&a;
    int &r=a;
    int c[5];
    int (&tr)[5]=c;
    int *d[5];
    int *h();
    int *(*l)();
    int (*v(int c))(int a, int b);
    int (*(*z)(int))(int,int);
}
```

Mit vezetnek be a programba a következő nevek?

```
int a;
```

Egy egész tipusú változót deklarál.

```
int *b = &a;
```

Egy egész tipusú mutatót deklarál ami tárolja az a memóriacímét (azaz "b" mutat "a"-ra).

```
int &r = a;
```

Az r egész tipusú változó már létezik és itt referenciát deklarálunk, a referencia más szóval paraméter értékátadás a már meglévő változó értékét jelen esetben megváltoztatjuk és értékül kapja az "a" értékét.

```
int c[5];
```

Egy egész tipusú 5 elemű tömböt deklarál.

```
int (&tr)[5] = c;
```

Ez egy referenciaja a "c" 5 elemű tömbnek (Az összes elemnek).

```
int *d[5];
```

Egy egész tipusú 5 elemű tömböt deklarál melynek az összes eleme egy-egy mutató.

```
int *h();
```

Az egész tipusú változó visszatérési értékét tartalmazó függvény;

```
int *(*l)();
```

Egy egész tipusra mutató visszaadó függvényre mutató mutató.

```
int (*v(int c))(int a, int b)
```

Egy egész tipusú változót visszaadó és két egész tipusú változót kapó függvényre mutató mutatót visszaadó egész tipusú változót kapó függvény.

```
int (**z)(int))(int, int);
```

Egy függvénymutató egy egészet visszaadó és két egészet kapó függvényre mutató mitátót visszaadó egészet kapó függvény.

### 3.9. Vörös Pipacs Pokol/csiga diszkrét mozgási parancsokkal

Megoldás videó: <https://youtu.be/Fc33ByQ6mh8>

Megoldás forrása: <https://github.com/nbatfai/RedFlowerHell>

Ez a feladat az előző Malmő feladathoz hasonló, mivel itt is az a cél hogy Steve csiga vonalban haladjon egyre feljebb az arénában, amíg eléri a lávát. Amiben változik itt a feladat az csak annyi, hogy diszkrét mozgást végezzen, tehát előre meghatározott lépést haladjon és meghatározott időben és mennyiségen ugorjon és fordúljon.

Amit meg kell adnunk az, hogy egy bizonyos gyűrűn való csiga járást négy részre bontjuk és csak egy résszel foglalkozunk mert az arána szimmetrikus, és ezeket az előre haladásokat fordulásokkal és esetleges ugrásokkal kötünk össze.

A diszkrét parancsok a következők:

- egy kockányi ugrás: **move 1**
- ugrás közbeni mozgás: **turn 1**
- egy fordulat jobbra: **jumpmove 1**

A **time.sleep()** parancs a mozgás időtartamát befolyásolja, viszont a kiadott parancs mennyisége nem fog változni.

A program kód elérhető a következő linkek: [csiga\\_diszkreten](#)

DRAFT

## 4. fejezet

# Helló, Caesar!

### 4.1. double \*\* háromszögmátrix

A következő programban egy alsó háromszögmátrixot hozunk létre. Forrás: <https://gitlab.com/nbatfai/bhax/blob/master/homeworks/triangle.c>  
Saját megoldás videó: [https://www.youtube.com/watch?v=vKI3Ri\\_BKtg](https://www.youtube.com/watch?v=vKI3Ri_BKtg)

```
#include <stdio.h>
#include <stdlib.h>

int
main ()
{
    int nr = 5;
    double **tm;

    if ((tm = (double **) malloc (nr * sizeof (double *))) == NULL)
    {
        return -1;
    }

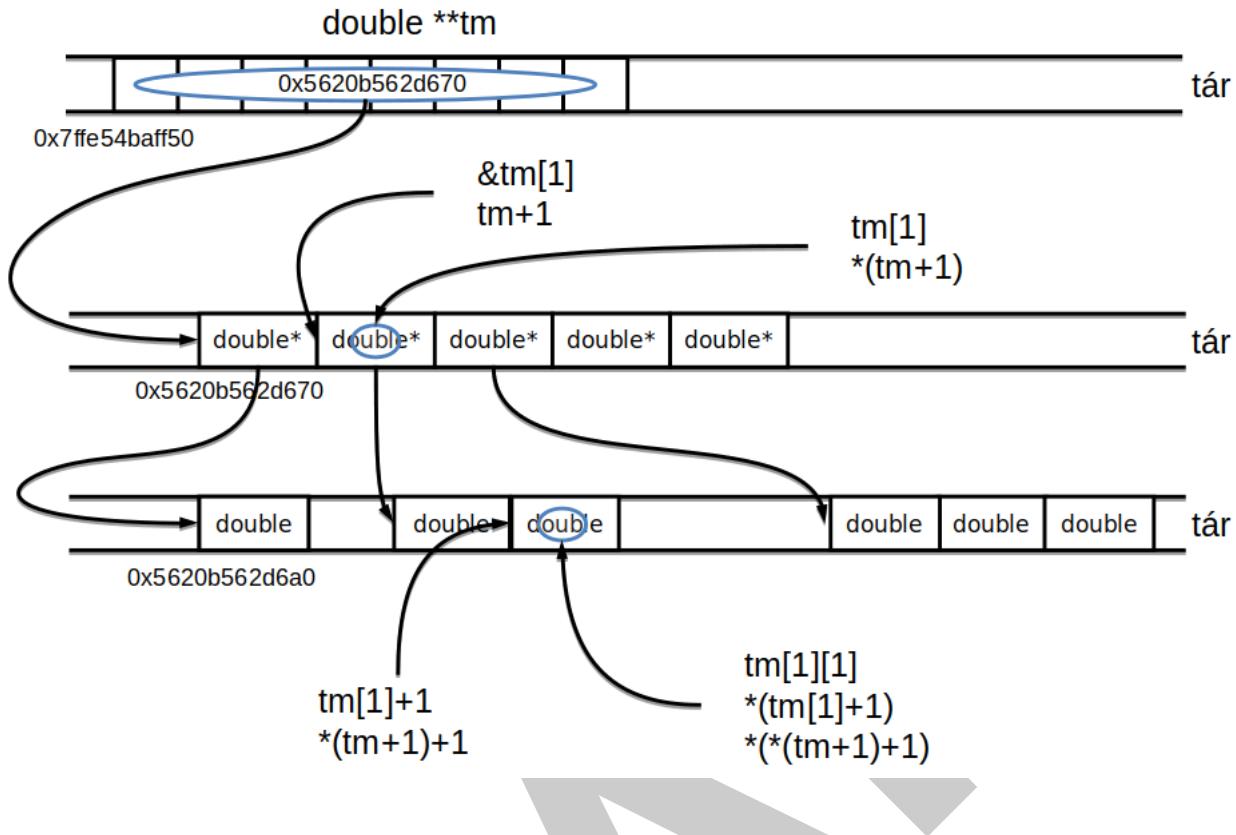
    for (int i = 0; i < nr; ++i)
    {
        if ((tm[i] = (double *) malloc ((i + 1) * sizeof (double))) == NULL)
        {
            return -1;
        }
    }

    for (int i = 0; i < nr; ++i)
        for (int j = 0; j < i + 1; ++j)
            tm[i][j] = i * (i + 1) / 2 + j;

    for (int i = 0; i < nr; ++i)
```

```
{  
    for (int j = 0; j < i + 1; ++j)  
        printf ("%f, ", tm[i][j]);  
    printf ("\n");  
}  
  
tm[3][0] = 42.0;  
(*tm + 3)[1] = 43.0; // mi van, ha itt hiányzik a külső ()  
*(tm[3] + 2) = 44.0;  
*(*(tm + 3) + 3) = 45.0;  
  
for (int i = 0; i < nr; ++i)  
{  
    for (int j = 0; j < i + 1; ++j)  
        printf ("%f, ", tm[i][j]);  
    printf ("\n");  
}  
  
for (int i = 0; i < nr; ++i)  
    free (tm[i]);  
  
free (tm);  
  
return 0;  
}
```

DPRV



4.1. ábra. A double \*\* háromszögmátrix a memóriában

Magyarázat: Includeoljuk a szükséges headereket és a main() főfüggvényben dolgozunk tovább. Az első változó az nr, amely értéke meghatározza hogy hány soros legyen a kimenet. A double \*\*tm sorral foglalunk le tárhelyet a memóriában. Az első if-ben megtaláljuk a malloc függvényt ami dinamikus memória foglaló, ezzel nr számú double \*\* mutatót foglalunk le, ha null értéket ad vissza az azt jelzi, hogy nincs elég hely a foglaláshoz.

A következő if lefoglalja a mátrix sorait, az első sornak egy double \* mutatót foglal le, a másodiknak 2-t, a harmadiknak 3-t, egészen az nr ig. A 3. for ciklussal megadjuk a mátrix elemeit. Az i a matrixnak a sorai, a j pedig a benne lévő mutatók. A  $tm[i][j]=i*(i+1)/2+j;$ -vel érjük el azt, hogy az elemek minden egyel nőjenek. A 4. for ciklus pedig a kiíratás.

Ezek után már csak annyit csinálunk, hogy a 3 sort megváltoztatjuk, mert így is ki lehet íratni. A legvégén pedig a free()-vel felszabadítjuk a lefoglalt memóriát, ezzel megelőzve a memória szivárgást.

## 4.2. C EXOR titkosító

A feladat lényege, hogy egy szöveget titkosítsunk Exor-ral(XOR).

Az XOR (kizáró vagy) művelet biteket vizsgál, tehát ha a bitek azonosak ( 0,0; 1,1 ) akkor 0-t ad vissza értéknek, ha pedig különbözök (1,0;0,1) akkor 1-et.

Forrás: [https://www.tankonyvtar.hu/hu/tartalom/tamop412A/2011-0063\\_01\\_parhuzamos\\_prog\\_linux/ch05s02.htm](https://www.tankonyvtar.hu/hu/tartalom/tamop412A/2011-0063_01_parhuzamos_prog_linux/ch05s02.htm)  
IwAR2X9zgwtSH6GW2\_K67UrjYDAVgljqV0i5KmBHuaZ2DjWIvzyFW4LrCtA

```
#include <stdio.h>
#include <unistd.h>
#include <string.h>

#define MAX_KULCS 100
#define BUFFER_MERET 256

int main (int argc, char **argv)
{
    char kulcs[MAX_KULCS];
    char buffer[BUFFER_MERET];

    int kulcs_index = 0;
    int olvasott_bajtok = 0;

    int kulcs_meret = strlen (argv[1]);
    strncpy (kulcs, argv[1], MAX_KULCS);

    while ((olvasott_bajtok = read (0, (void *) buffer, ←
        BUFFER_MERET)))
    {
        for (int i = 0; i < olvasott_bajtok; ++i)
        {
            buffer[i] = buffer[i] ^ kulcs[kulcs_index];
            kulcs_index = (kulcs_index + 1) % kulcs_meret;
        }
        write (1, buffer, olvasott_bajtok);
    }
}
```

### Magyarázat:

Includeoljuk a szükséges headereket, aztán két állandó (globális) változót definiálunk a #define parancsal, ezek értéke nem változik. Az első állandó a MAX\_KULCS, az értéke 100, a második pedig a BUFFER\_MERET, az értéke pedig 256.

A fő függvényben egy-egy char típusú tömböt hozunk létre amelyek méretei a globális változók( 100 és 256). Aztán 2 egész típusú változót hozunk létre, a kulcs\_index, ami a kulcsunk aktuális elemét tárolja, és az olvasott\_bajtok ami a beolvasott bajtokat tárolja. Egy további egész típusú változó a kulcs\_merete, a változóban a kulcs méretét adjuk meg a strlen() függvény segítségével, amit mi adunk meg egyik argumentumként. Az strncpy függvényt pedig a kulcs kezeléséhez használjuk.

Ezután a while ciklusban beolvassuk a buffer tömbbe a bemenetet, a while ciklus addig fut, ameddig van mit beolvasni. A read() függvényel lépünk ki a ciklusból. A while cikluson belül lévő for ciklusban végig megyünk az összes bajton és így titkosít a program.

A fordítás: gcc fajlnev.c -o fajlnev miután lefut, utánna ./fajlnev 23134012 (ez a kulcs) titkosítando.txt > titkos.szoveg (titkosított fajl). A titkos szöveget, a more titkos.szoveg parancsal nézhetjük meg

## 4.3. Java EXOR titkosító

Turtorált: Talinger Mark Imre

Az előző feladathoz hasonló, a különbség csak a használt programozási nyelv. Itt a Java programozási nyelvet használjuk ami objektum orientált, vagyis objektumokból, osztályokból áll.

Hasonló a C++-hoz, de egyszerűbb mivel csak osztályokat (Classokat) használunk, az osztályokban különböző függvények vannak. Az osztályokat három részre oszthatjuk. A public rész, az ebben lévő függvényeket bárhonnan meghívhatjuk. A private rész, az ebben lévő függvényeket csak az adott osztályon belül hívhatjuk meg (az osztály titkos függvényei). A protected rész hasonló a privatehez de van egy kis különbség, a függvényeket csak az osztályban hívhatjuk meg, kivéve ha barát függvényként definiáljuk őket, ez utóbbi esetben meghívhatóak bárhonnan.

```
public class ExorTitkosito
{
    public ExorTitkosito(String kulcsSzoveg,
                          java.io.InputStream bejovoCsatorna,
                          java.io.OutputStream kimenocsatorna)
                          throws java.io.IOException
    {

        byte [] kulcs = kulcsSzoveg.getBytes();
        byte [] buffer = new byte[256];

        int kulcsIndex = 0, olvasottBajtok = 0;

        while( ( olvasottBajtok = bejovoCsatorna.read(buffer) ) != -1 )
        {
            for(int i=0; i<olvasottBajtok; ++i)
            {
                buffer[i] = (byte)(buffer[i] ^ kulcs[kulcsIndex]);
                kulcsIndex = (kulcsIndex+1) % kulcs.length;
            }

            kimenocsatorna.write(buffer, 0, olvasottBajtok);
        }
    }

    public static void main(String[] args)
    {
        try { new ExorTitkosito(args[0], System.in, System.out); }
        catch(java.io.IOException e) {
            e.printStackTrace();
        }
    }
}
```

Az ExorTitkosito() függvény, kapja meg a bekért argumentumokat. A throw() függvény hiba üzenetet ad vissza ha rosszul adtunk meg valamit.

A titkosítási eljárás ugyan az mint az előző feladatban, XOR-al történik. Ebben a nyelvben van byte típus ami 8 bit. A kulcs és a buffer byte típusú lesz, amik tárolják a kulcsot és a beolvasott szöveget.

Mivel java nyelv ezért a main() függvény az osztály része és egyben az egyik függvénye. A main() függvényt jellemzők, a public(azaz nyilvános), static (azaz része az osztálynak) és void (amely kiíratást végez). A main()-be terminálból is adhatunk értéket. A függvényben pedig van egy try() és egy catch() függvény, a try() hiab üzenetet küld és a catch() ezt elkapja aztán kiírja.

A fordításhoz java fordítót kell használnunk, jelen esetben javac-t ha ez nekünk nincs telepítve akkor jelzi a számítógép hogyan telepíhetjük.

Fordítás: javac ExorTitkosító.java

Futtatás: java ExorTitkosító titkosítandó.szöveg > titkosított.szöveg

## 4.4. C EXOR törő

Írj egy olyan C programot, amely megtöri az első feladatban előállított titkos szövegeket!

Ebben a feladatban írunk egy programot ami feltöri az előző 4.2-es feladatban titkosított szöveget. A program működése azon az elven alapszik, minta 4.2 mivel ugyan így XOR-t használunk és ezzel alakítjuk vissza a szöveget. A kulcsot amivel titkosítottunk azt ismernünk kell, mert ezzel a kulcsal tudjuk feltörni. Úgy működik, hogy a titkosított bajtokat össze exortáljuk a kulcsal, és így újra az eredeti bajtokat kapjuk.

Forrás: [https://www.tankonyvtar.hu/hu/tartalom/tamop412A/2011-0063\\_01\\_parhuzamos\\_prog\\_linux/ch05s02.htm](https://www.tankonyvtar.hu/hu/tartalom/tamop412A/2011-0063_01_parhuzamos_prog_linux/ch05s02.htm)

```
#define MAX_TITKOS 4096
#define OLVASAS_BUFFER 256
#define KULCS_MERET 8
#define _GNU_SOURCE

#include<stdio.h>
#include<unistd.h>
#include<string.h>

int tiszta(const char titkos[], int titkos_meret)
{
    return strcasestr(titkos, "hogy") && strcasestr(titkos, "nem") && !-
        strcasestr(titkos, "az") && strcasestr(titkos, "ha");
}

void exor(const char kulcs[], int kulcs_meret, char titkos[], int ←
titkos_meret)
{
    int kulcs_index=0;

    for(int i=0; i<titkos_meret; ++i)
    {
        titkos[i]=titkos[i]^kulcs[kulcs_index];
```

```
        kulcs_index=(kulcs_index+1)%kulcs_meret;
    }
}

int exor_tores(const char kulcs[], int kulcs_meret, char titkos[], ←
    int titkos_meret)
{
    exor(kulcs, kulcs_meret, titkos, titkos_meret);

    return tiszta_lehet(titkos, titkos_meret);
}

int main(void)
{
    char kulcs[KULCS_MERET];
    char titkos[MAX_TITKOS];
    char *p=titkos;
    int olvasott_bajtok;

    while((olvasott_bajtok = read(0,
                                    (void *) p,
                                    (p-titkos+OLVASAS_BUFFER < ←
                                     MAX_TITKOS) ?
                                    OLVASAS_BUFFER : titkos+←
                                     MAX_TITKOS-p)
                           ) )
    {
        p+=olvasott_bajtok;
    }

    for(int i=0;      i<MAX_TITKOS-(p-titkos);      ++i)
        titkos[p-titkos+i]='\0';

//A kulcs
for(int ii='0';ii<='9';++ii)
    for(int ji='0';ji<='9';++ji)
        for(int ki='0';ki<='9';++ki)
            for(int li='0';li<='9';++li)
                for(int mi='0';mi<='9';++mi)
                    for(int ni='0';ni<='9';++ni)
                        for(int oi='0';oi<='9';++oi)
                            for(int pi='0';pi<='9';++pi)
                            {
                                kulcs[0]=ii;
                                kulcs[1]=ji;
                                kulcs[2]=ki;
                                kulcs[3]=li;
                                kulcs[4]=mi;
                                kulcs[5]=ni;
```

```
        kulcs[6]=oi;
        kulcs[7]=pi;

        if( exor_tores( kulcs,KULCS_MERET, titkos, p-titkos ) )
            printf( "Kulcs: [%c%c%c%c%c%c%c]\n"
                    Tiszta szöveg: [%s]\n",ii,ji,ki,li,mi,ni,oi,pi, ←
                    titkos);

        exor(kulcs,KULCS_MERET,titkos,p-titkos);
    }
    return 0;
}
```

Definiájuk az állandó változókat és includoljuk a headereket. A unistd.h header az strcasest() függvény miatt használjuk. A kulcs mérete ismét 8 mivel a titkosításnál is ennyit használtunk. Az exor() és a tiszta() függvény a törés gyorsaságát segítik elő. Az tiszta() függvényben az új headerből használjuk az strcasestr(miben,mit) függvényt ami megkeresi az első előfordulását egy keresett szövegnek(string) és ignorálja, a 0 (null) biteket nem veszi figyelembe.

A void exor () függvény megkap egy kulcsot, a méretét, a titkos szövegetnek a tömbjét és annak a méretét. És itt a forciklusban a kulcsot össze exortálja a titkos szöveggel.

Az exor\_tores függvény meghívja az exor függvényt és vissza adja a tiszta szöveget. A fő függvényben láthatjuk deklarációk után a titkos szöveg beolvasását. Aztán a program megnézi az összes lehetséges esetet és a megoldást kiíratja a kimenetre, ezzel a kódval a 4.2 programot használva fel tudjuk törni a szöveget.

## 4.5. Neurális OR, AND és EXOR kapu

Tutorált: Talerger Mark Imre

Neurális háló:

Az emberi idegrendszerét neuron idegsejtek építik fel, ezek a sejtek ingerületekben kapnak információkat, majd ezeket feldolgozzák és továbbíják. A mi programunk hasonlóan fog működni, tehát kapunk egy információt, amit logikailag feldolgozunk és aztán továbbítjuk. A program megírásához az R nyelvet használom.

Logikai VAGY (OR):

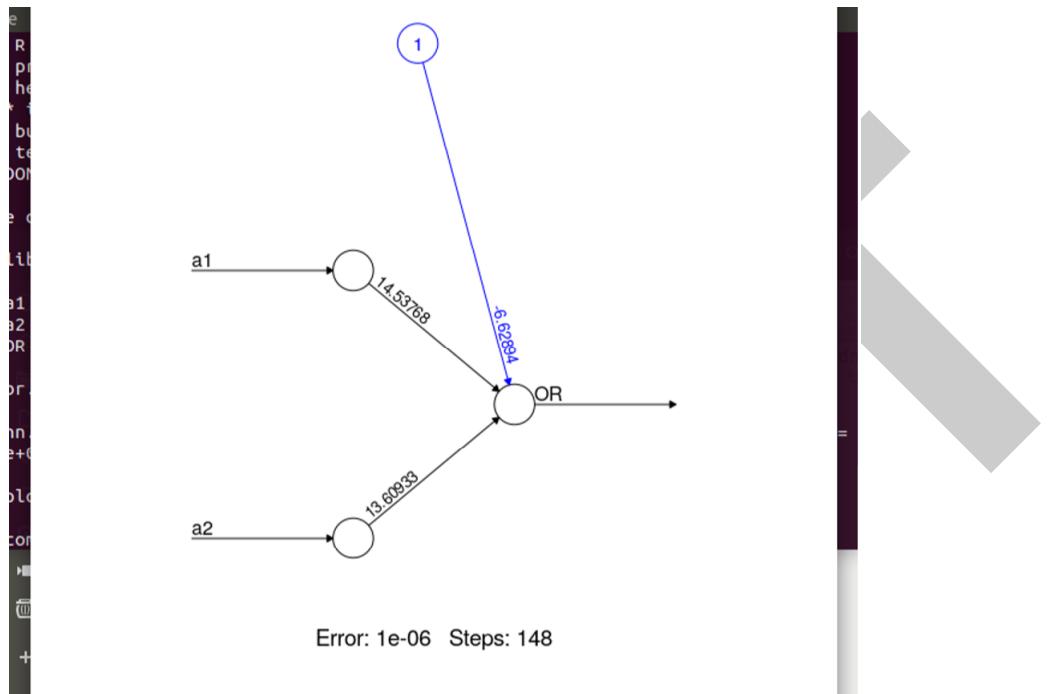
```
library(neuralnet)

a1 <- c(0,1,0,1)
a2 <- c(0,0,1,1)
OR <- c(0,1,1,1)

or.data <- data.frame(a1, a2, OR)

nn.or <- neuralnet(OR~a1+a2, or.data, hidden=0, linear.output=FALSE ←
,
stepmax = 1e+07, threshold = 0.000001)
```

```
plot(nn.or)
compute(nn.or, or.data[,1:2])
```



A program elején meghívjuk a neuralnet könyvtárat ami tartalmazza a szükséges függvényeket. A bemenet az a1 és az a2 lesz, a gép pedig az OR-t (logikai VAGY) fogja megtanulni. Ha a1 és a2 bemenet 0-t ad, akkor az OR értéke is 0 lesz, minden más esetben az OR értéke 1. Ezeket az or.data-ban tárolja el a program. Az nn.or értékét pedig a neuralnet() függvényel határozzuk meg. A függvény:

Az első argumentumában a megtanuladnó érték van, azaz hogy az OR értéke 0 legyen vagy 1.

A második argumentumban adjuk meg az or.data-t, hogy mi alapján tanulja meg a program.

A harmadik argumentumban rejtett neutronok száma van.

A stepmax a lépésszámot adja.

A plot függvényteljesen kirajzolunk (a képen) a tanulás folyamatának egyik esetét.

Logikai ÉS (AND):

```
library(neuralnet)

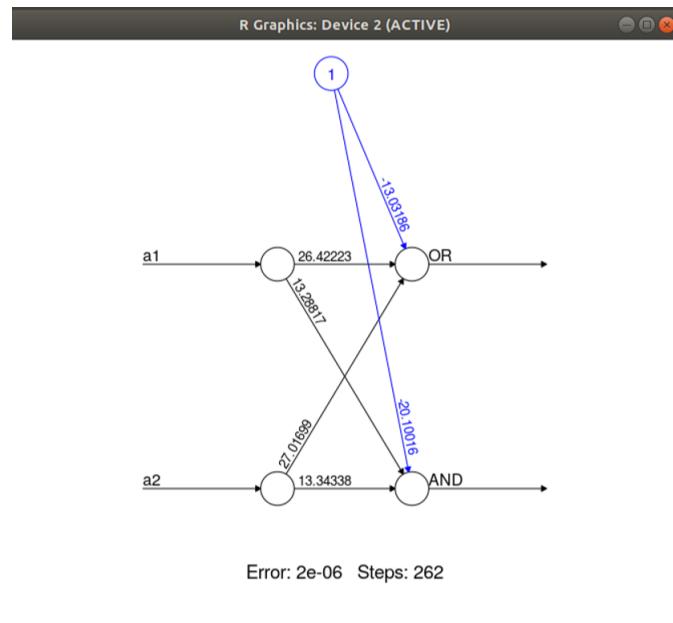
a1 <- c(0,1,0,1)
a2 <- c(0,0,1,1)
OR <- c(0,1,1,1)
AND <- c(0,0,0,1)

orAnd.data <- data.frame(a1, a2, OR, AND)

nn.orAnd <- neuralnet(OR+AND~a1+a2, orAnd.data, hidden=0, linear. -->
output=FALSE,
stepmax = 1e+07, threshold = 0.000001)

plot(nn.orAnd)
```

```
compute(nn.orAnd, orAnd.data[,1:2])
```



A programunk annyival bővül, hogy a program az OR-on kívül az AND-et (logikai ÉS) is meg fogja megtanulni. Az AND csak akkor kap 1 értéket, ha  $a_1$  és  $a_2$  értéke is 1, különben az AND értéke 0. A tanulás folyamata ugyan olyan mint az előző programban. A tanulás módját az `orAnd.data`-ba mentjük.

EXOR:

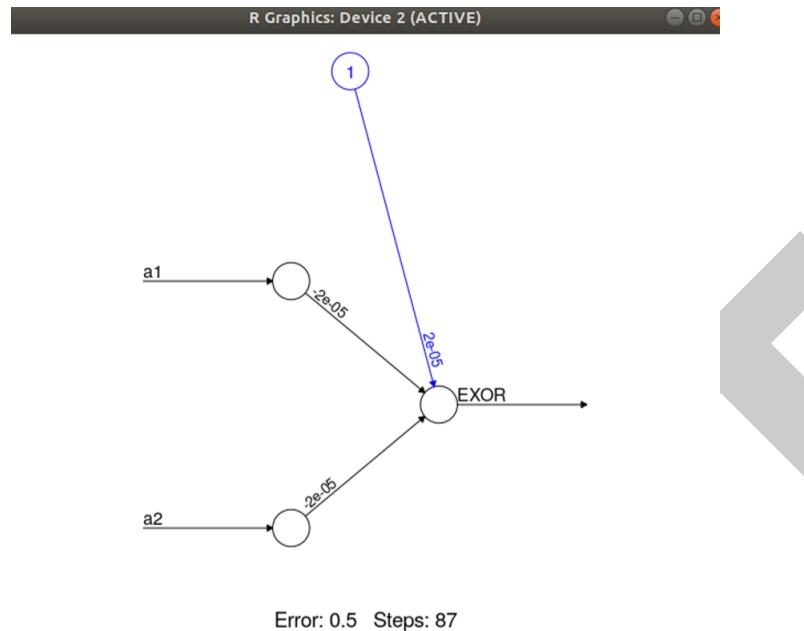
```
library(neuralnet)

a1 <- c(0,1,0,1)
a2 <- c(0,0,1,1)
EXOR <- c(0,1,1,0)

exor.data <- data.frame(a1, a2, EXOR)

nn.exor <- neuralnet(EXOR~a1+a2, exor.data, hidden=0, linear.output <-
  =FALSE,
  stepmax = 1e+07, threshold = 0.000001)

plot(nn.exor)
compute(nn.exor, exor.data[,1:2])
```



Itt pedig az EXOR-t tanítjuk meg a programmal.

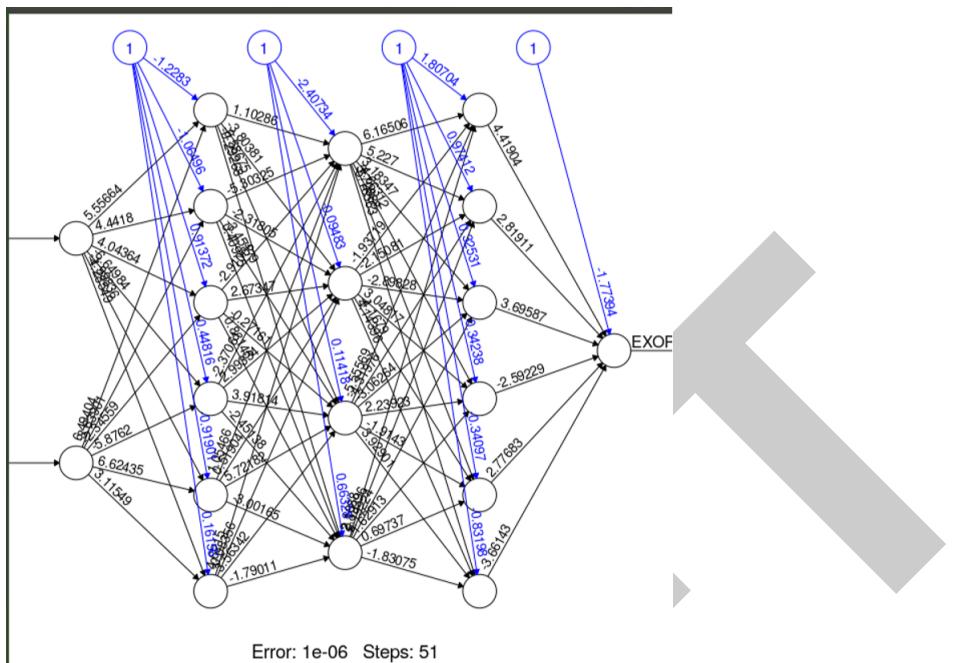
Az EXOR értéke akkor 1 (igaz), ha az a1 és a2 értéke 1,0 (a1= igaz, a2= hamis) vagy 0,1 (a1= hamis,a2= igaz). Ha minden két érték 0,0 (hamis,hamis) vagy 1,1(igaz,igaz) akkor az EXOR értéke 0 lesz. Ezt a tanulási mintát az exor.data-ban mentjük el. És a tanulás megegyezik a fentiekkel. A képen láthatjuk, hogy a program nem tanulta meg amit kell, ugyanis az eredmények hibásak. A kulcs abban van, hogy a rejtett neuronok értéke 0. A következőben nézzük meg a megoldását.

A 3 réteg:

```
a1 <- c(0,1,0,1)
a2 <- c(0,0,1,1)
EXOR <- c(0,1,1,0)

exor.data <- data.frame(a1, a2, EXOR)

nn.exor <- neuralnet(EXOR~a1+a2, exor.data, hidden=c(6, 4, 6), 
                      linear.output=FALSE,
                      stepmax = 1e+07, threshold = 0.000001)
plot(nn.exor)
compute(nn.exor, exor.data[,1:2])
```



Itt annyiban változtattunk a kódon, hogy a rejtett (hidden) neuronoknak 3 réteget hoztunk létre (az értékek: 6,4,6). Ahogy a képen is látszik, így helyes az eredmény.

## 4.6. Hiba-visszaterjesztéses perceptron

A program fel tudja dolgozni és meg tudja tanulni a bemenetet, ami 0-ból és 1-ből áll.

A program c++ nyelven kódolt.

Forrás: <https://youtu.be/XpBnR31BRJY>

```
#include <iostream>
#include "mlp.hpp"
#include "png++/png.hpp"

int main (int argc, char **argv)
{
    png::image<png::rgb_pixel> png_image (argv[1]);

    int size = png_image.get_width() * png_image.get_height();

    Perceptron* p = new Perceptron(3, size, 256, 1);

    double* image = new double[size];

    for(int i {0}; i<png_image.get_width(); ++i)
        for(int j {0}; j<png_image.get_height(); ++j)
            image[i*png_image.get_width() + j] = png_image[i][j].red;

    double value = (*p) (image);

    std::cout << value << std::endl;
```

```
    delete p;  
  
    delete [] image;  
}
```

A programban két új headert includeolunk, az "mlp.hpp"-t és a "png++/png.hpp"-t, ezek a megjelenítéshez kellenek és ebben van a perceptron elve is. A fő függvényünk (main) elején lefoglaljuk a tárhelyet a képnak és megadjuk a méreteit.

Aztán a perceptron létrehozása, a méret (size) a kép méret magasságának és szélességének lesz a szorzata.

A majd létrehozunk egy double tipusú size méretű képet, utána feltöljük a megadott képpel, amelyeket a két for ciklus végel el.

A delete parancsokkal megakadályozzuk a memória szivárgást, azaz töröljük a perceptronról és a képet.

## 4.7. Vörös Pipacs Pokol/írd ki, mit lát Steve

Megoldás videó: <https://youtu.be/-GX8dzGqTdM>

Megoldás forrása: <https://github.com/nbatfai/RedFlowerHell>

Ebben a feladatban Stevenek érzékelnie kell az önmaga körül lévő kockákat. Az alapvető mozgás megmarad viszont kibővíti egy observations használattal. Ezt egy 3x3x3-mas tömb alakban fogjuk használni amiben Steve környezetét vizsgáljuk. A programban a LineOfSight fogja meghatározni, hogy mi van Steve előtt, majd kiíratjuk az információkat a képernyőre.

A kódot két nyelven is futtathatjuk, az első a C++ nyelv, amelyben a program a már leírt módon működik, a második a Python nyelv, amelyben Steve már ha a pipacsokat érzékeli ki is üti azokat.

A program kód Python nyelven elérhető a repómban a következő linken: [mit\\_lat\\_Steve](#)

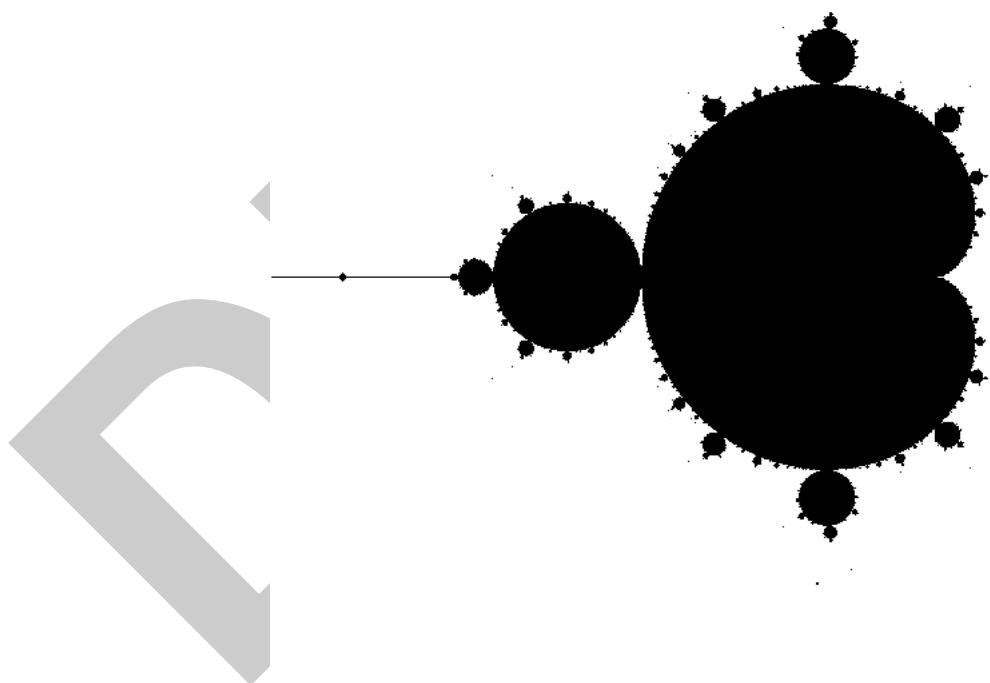
## 5. fejezet

# Helló, Mandelbrot!

### 5.1. A Mandelbrot halmaz

Írj olyan C programot, amely kiszámolja a Mandelbrot halmazt!

Megoldás videó: <https://youtu.be/gvaqijHIRUs>



5.1. ábra. A Mandelbrot halmaz a komplex síkon

A Mandelbrot halmazt 1980-ban találta meg Benoit Mandelbrot a komplex számsíkon. Komplex számok azok a számok, amelyek körében válaszolni lehet az olyan egyébként értelmezhetetlen kérdésekre, hogy melyik az a két szám, amelyet összeszorozva -9-öt kapunk, mert ez a szám például a 3i komplex szám.

A Mandelbrot halmazt úgy láthatjuk meg, hogy a sík origója középpontú 4 oldalhosszúságú négyzetbe lefektetünk egy, mondjuk 800x800-as rácsot és kiszámoljuk, hogy a rács pontjai mely komplex számoknak felelnek meg. A rács minden pontját megvizsgáljuk a  $z_{n+1} = z_n^2 + c$ , ( $0 \leq n$ ) képlet alapján úgy, hogy a  $c$  az éppen vizsgált rácpont. A  $z_0$  az origó. Alkalmazva a képletet a

- $z_0 = 0$
- $z_1 = 0^2 + c = c$
- $z_2 = c^2 + c$
- $z_3 = (c^2 + c)^2 + c$
- $z_4 = ((c^2 + c)^2 + c)^2 + c$
- ... s így tovább.

Azaz kiindulunk az origóból ( $z_0$ ) és elugrunk a rács első pontjába a  $z_1 = c$ -be, aztán a  $c$ -től függően a további  $z$ -kbe. Ha ez az utazás kivezet a 2 sugarú körből, akkor azt mondjuk, hogy az a vizsgált rácpont nem a Mandelbrot halmaz eleme. Nyilván nem tudunk végtelen sok  $z$ -t megvizsgálni, ezért csak véges sok  $z$  elemet nézünk meg minden rácponthoz. Ha eközben nem lép ki a körből, akkor feketére színezzük, hogy az a  $c$  rácpont a halmaz része. (Színes meg úgy lesz a kép, hogy változatosan színezzük, például minél későbbi  $z$ -nél lép ki a körből, annál sötétebbre).

Program kód:

```
include "png++-0.2.9/png.hpp"

#define N 500
#define M 500
#define MAXX 0.7
#define MINX -2.0
#define MAXY 1.35
#define MINY -1.35

void GeneratePNG( int tomb[N][M] )
{
    png::image< png::rgb_pixel > image(N, M);

    for (int x = 0; x < N; x++)
    {
        for (int y = 0; y < M; y++)
        {
            image[x][y] =
                png::rgb_pixel(tomb[x][y], tomb[x][y], tomb[x][y]);
        }
    }

    image.write("kimenet.png");
}
```

```
struct Komplex
{
    double re, im;
};

int main()
{
    int tomb[N][M];
    int i, j;

    double dx = (MAXX - MINX) / N;
    double dy = (MAXY - MINY) / M;

    struct Komplex C, Z, Zuj;
    int iteracio;

    for (i = 0; i < M; i++)
    {
        for (j = 0; j < N; j++)
        {
            C.re = MINX + j * dx;
            C.im = MAXY - i * dy;

            Z.re = 0;
            Z.im = 0;

            iteracio = 0;

            while(Z.re * Z.re + Z.im * Z.im < 4 && iteracio++ < 255)
            {
                Zuj.re = Z.re * Z.re - Z.im * Z.im + C.re;
                Zuj.im = 2 * Z.re * Z.im + C.im;

                Z.re = Zuj.re;
                Z.im = Zuj.im;
            }

            tomb[i][j] = 256 - iteracio;
        }
    }

    GeneratePNG(tomb);
    return 0;
}
```

Elsősorban a png++ headerre van szükségünk ahoz hogy png-t tudunk kezelni. Le kell töltenünk az internetről egy fájlt ami tartalmazza a headert. Mert a géünk nem biztosítja ezt a headert. Ezután még telepíteni kell a libpng könyvtárat az alábbi módon: "sudo apt-get install libpng++-dev".

Továbbá definiálunk globális (állandó) változókat is, ezek a kép magassága és szélessége (x és y koordináta tengelyen).

Aztán létrehozunk egy GeneratePNG() nevű függvényt amely egy képet generál a következő módon. Létrehoz egy üres pngt ami 500x500 pixel (M és N) és itt használtuk először a png headert, majd a for cikluson belül 0-tól kezdve i és j változók (x és y tengely) segítségével "pixelről pixelre" meghatározzuk az rgb színkóddal (a png header használata újra) a színes pixeleket, végül az image.write() függvényel küldjük a kimenetre a képet.

Létrehoztunk egy struktúrát is amelyben két double típusú változót deklaráltunk. A komplex számok létrehozásához szükséges.

A main() fő függvényünkben létrehozunk egy egész típusú 500x500 (NxM) elemű tömböt, azánt két int típusú változót deklaráltunk a lépkedéshez a for ciklusba, továbbá két double típusú a komplex számokat (a pixelek meghatározásához).

Lefoglalunk a memoriában helyet a C, Z és Zuj változóknak, majd deklarálunk egy int típusú iteracio nevű változót amely az RGB színkód meghatározásához lesz szükséges, és a for ciklusban elvégezzük a számításokat amiket a tömbbe tesszük és meghívjuk a GeneratePNG() függvényt amely legenerálja a képet a számítások alapján.

## 5.2. A Mandelbrot halmaz a std::complex osztállyal

Írj olyan C++ programot, amely kiszámolja a Mandelbrot halmazt!

Megoldás videó: <https://youtu.be/gvaqijHIRUs>

Megoldás forrása:

A **Mandelbrot halmaz** pontban vázolt ismert algoritmust valósítja meg a repó [bhax/attention\\_raising/Mandelbrot/3.1.2.cpp](https://github.com/bhaxor/attention_raising/tree/main/Mandelbrot/3.1.2.cpp) nevű állományá.

```
// Verzio: 3.1.2.cpp
// Forditas:
// g++ 3.1.2.cpp -lpng -O3 -o 3.1.2
// Futtatas:
// ./3.1.2 mandel.png 1920 1080 2040 ←
// -0.01947381057309366392260585598705802112818 ←
// -0.0194738105725413418456426484226540196687 ←
// 0.7985057569338268601555341774655971676111 ←
// 0.798505756934379196110285192844457924366
// ./3.1.2 mandel.png 1920 1080 1020 ←
// 0.4127655418209589255340574709407519549131 ←
// 0.4127655418245818053080142817634623497725 ←
// 0.2135387051768746491386963270997512154281 ←
// 0.2135387051804975289126531379224616102874
// Nyomtatas:
// a2ps 3.1.2.cpp -o 3.1.2.cpp.pdf -l --line-numbers=1 --left-footer="←
// BATF41 HAXOR STR34M" --right-footer="https://bhaxor.blog.hu/" --pro= ←
// color
// ps2pdf 3.1.2.cpp.pdf 3.1.2.cpp.pdf.pdf
```

```
//  
//  
// Copyright (C) 2019  
// Norbert Bátfai, batfai.norbert@inf.unideb.hu  
//  
// This program is free software: you can redistribute it and/or modify  
// it under the terms of the GNU General Public License as published by  
// the Free Software Foundation, either version 3 of the License, or  
// (at your option) any later version.  
//  
// This program is distributed in the hope that it will be useful,  
// but WITHOUT ANY WARRANTY; without even the implied warranty of  
// MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the  
// GNU General Public License for more details.  
//  
// You should have received a copy of the GNU General Public License  
// along with this program. If not, see <https://www.gnu.org/licenses/>.  
  
#include <iostream>  
#include "png++/png.hpp"  
#include <complex>  
  
int  
main ( int argc, char *argv[] )  
{  
  
    int szelesseg = 1920;  
    int magassag = 1080;  
    int iteraciosHatar = 255;  
    double a = -1.9;  
    double b = 0.7;  
    double c = -1.3;  
    double d = 1.3;  
  
    if ( argc == 9 )  
    {  
        szelesseg = atoi ( argv[2] );  
        magassag = atoi ( argv[3] );  
        iteraciosHatar = atoi ( argv[4] );  
        a = atof ( argv[5] );  
        b = atof ( argv[6] );  
        c = atof ( argv[7] );  
        d = atof ( argv[8] );  
    }  
    else  
    {  
        std::cout << "Használat: ./3.1.2 fajlnev szelesseg magassag n a b c d ←  
        " << std::endl;  
        return -1;  
}
```

```
}

png::image < png::rgb_pixel > kep ( szelesseg, magassag );

double dx = ( b - a ) / szelesseg;
double dy = ( d - c ) / magassag;
double reC, imC, rez, imZ;
int iteracio = 0;

std::cout << "Szamitas\n";

// j megy a sorokon
for ( int j = 0; j < magassag; ++j )
{
    // k megy az oszlopokon

    for ( int k = 0; k < szelesseg; ++k )

        // c = (reC, imC) a halo racspontjainak
        // megfelelo komplex szam

        reC = a + k * dx;
        imC = d - j * dy;
        std::complex<double> c ( reC, imC );

        std::complex<double> z_n ( 0, 0 );
        iteracio = 0;

        while ( std::abs ( z_n ) < 4 && iteracio < iteraciosHatar )
        {
            z_n = z_n * z_n + c;

            ++iteracio;
        }

        kep.set_pixel ( k, j,
                        png::rgb_pixel ( iteracio%255, (iteracio*iteracio <=
                            )%255, 0 ) );
    }
}

int szazalek = ( double ) j / ( double ) magassag * 100.0;
std::cout << "\r" << szazalek << "%" << std::flush;
}

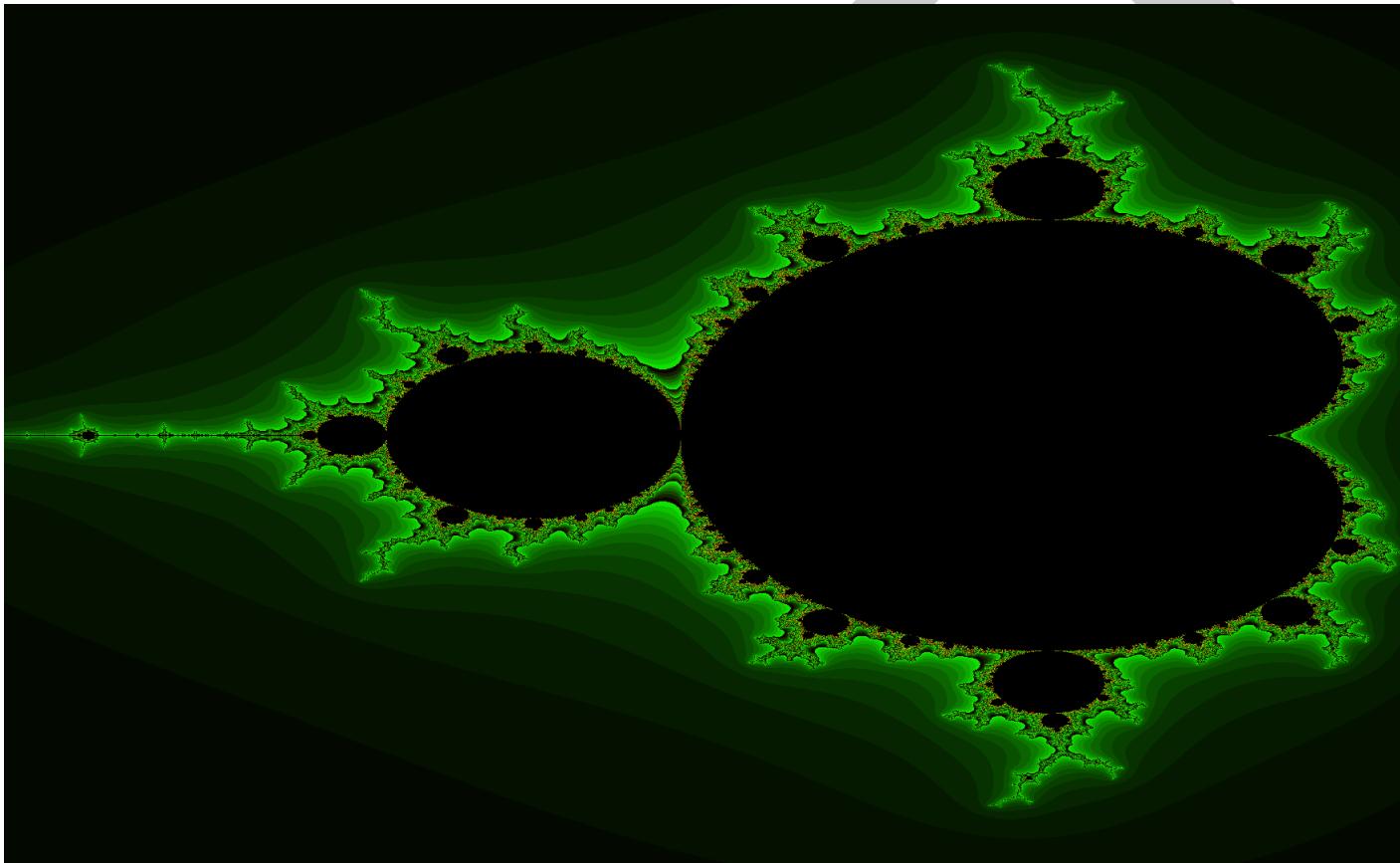
kep.write ( argv[1] );
std::cout << "\r" << argv[1] << " mentve." << std::endl;
}
```

Magyarázat: A külöbség az előző feladathoz képest az, hogy itt nem kell strukturával létrehozni komplex számokat, mert az új headerben a complex header már alapból tartalmaz komplex számokat.

A fő függvényben dekralálunk 2 változót, ha argumentumként jól adjuk meg ezeket, akkor ezeket átadja a változóknak, hanem jól adjuk meg, akkor kiírjuk, hogy kell helyesen használni. Aztán megadjuk a szélességet és a magasságot, ami ebbe az esetbe FullHD(szelesseg = 1920, magassag = 1080) és az iterációs határt.

Továbbá deklarálunk változókat amik a kép elkészítéséhez szükségesek. Ezek után lefoglaljuk a helyet a képnek. A dx, dy-hez hozzá rendeljük a megfelelő változókat. A forciklusban végig megyünk minden elemen (pixelen) és megadjuk a c változó értékét ekkor használjuk a complex headert. A while ciklusban végezzük a számításokat, utánna rgb kóddal a pixeleket kiszinezzük.

A futtatáshoz szükségünk lesz a -lpng kapcsolóra.



### 5.3. Biomorfok

Megoldás videó: <https://youtu.be/IJMbqRzY76E>

Megoldás forrása: [https://gitlab.com/nbatfai/bhax/tree/master/attention\\_raising/Biomorf](https://gitlab.com/nbatfai/bhax/tree/master/attention_raising/Biomorf)

A különbség a **Mandelbrot halmaz** és a Julia halmazok (biomorf) között az, hogy a komplex iterációban az előbbiben a c változó, utóbbiban pedig állandó. A következő Mandelbrot csipet azt mutatja, hogy a c befutja a vizsgált összes rácspontot.

```
// j megy a sorokon
for ( int j = 0; j < magassag; ++j )
```

```
{  
for ( int k = 0; k < szelesseg; ++k )  
{  
  
    // c = (reC, imC) a halo racspontjainak  
    // megfelelo komplex szam  
  
    reC = a + k * dx;  
    imC = d - j * dy;  
    std::complex<double> c ( reC, imC );  
  
    std::complex<double> z_n ( 0, 0 );  
    iteracio = 0;  
  
    while ( std::abs ( z_n ) < 4 && iteracio < ↪  
            iteraciosHatar )  
    {  
        z_n = z_n * z_n + c;  
  
        ++iteracio;  
    }  
}
```

Ezzel szemben a Julia halmazos csipetben a cc nem változik, hanem minden vizsgált z rácpontra ugyanaz.

```
// j megy a sorokon  
for ( int j = 0; j < magassag; ++j )  
{  
    // k megy az oszlopokon  
    for ( int k = 0; k < szelesseg; ++k )  
    {  
        double rez = a + k * dx;  
        double imZ = d - j * dy;  
        std::complex<double> z_n ( rez, imZ );  
  
        int iteracio = 0;  
        for (int i=0; i < iteraciosHatar; ++i)  
        {  
            z_n = std::pow(z_n, 3) + cc;  
            if (std::real ( z_n ) > R || std::imag ( z_n ) > R)  
            {  
                iteracio = i;  
                break;  
            }  
        }  
    }  
}
```

A biomorf programhoz a mandelbrot programkódját vesszük alapul. A mandelbrot halmaz tarttarlmazza az összes ilyen halmazt. A program ugyanúgy bekéri a megfelelő bemeneteket, ha nem jó akkor kiírja. Ha jó, akkor a megfelelő változók megkapják a megfelelő értékeket. Ezután történik a kép létrehozása. Ugyanúgy megkapja a dx és dy az értéket. Aztán pedig a komplex számokat hozzuk létre. Megint végig megy a program minden ponton és ahol kell használjuk az RGB kódos színezést. A legvégén pedig kiküldjük a

képet a kimenetre.

a program kód a köveztekező:

```
#include <iostream>
#include "png++/png.hpp"
#include <complex>

int main ( int argc, char *argv[] )
{
    int szelesseg = 1920;
    int magassag = 1080;
    int iteraciosHatar = 255;
    double xmin = -1.9;
    double xmax = 0.7;
    double ymin = -1.3;
    double ymax = 1.3;
    double reC = .285, imC = 0;
    double R = 10.0;

    if ( argc == 12 )
    {
        szelesseg = atoi ( argv[2] );
        magassag = atoi ( argv[3] );
        iteraciosHatar = atoi ( argv[4] );
        xmin = atof ( argv[5] );
        xmax = atof ( argv[6] );
        ymin = atof ( argv[7] );
        ymax = atof ( argv[8] );
        reC = atof ( argv[9] );
        imC = atof ( argv[10] );
        R = atof ( argv[11] );
    } else
    {
        std::cout << "Hasznalat: ./3.1.2 fajlnev szelesseg ←
                     magassag n a b c ←
                     d reC imC R" << std::endl; return -1;
    }

    png::image< png::rgb_pixel > kep ( szelesseg, magassag );

    double dx = ( xmax - xmin ) / szelesseg; double dy = ( ymax - ←
        ymin ) / magassag;

    std::complex<double> cc ( reC, imC );

    std::cout << "Szamitas\n";
    for ( int y = 0; y < magassag; ++y )
    {
        for ( int x = 0; x < szelesseg; ++x )
        {
```

```
double reZ = xmin + x * dx;
double imZ = ymax - y * dy;
std::complex<double> z_n ( reZ, imZ );
int iteracio = 0;

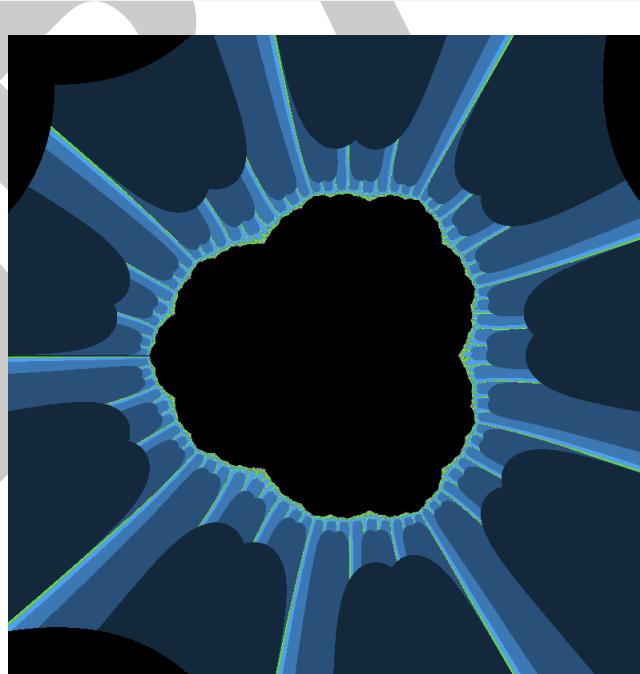
for (int i=0; i < iteracionsHatar; ++i)
{
    z_n = std::pow(z_n, 3) + cc;
    if(std::real ( z_n ) > R || std::imag ( z_n ) > R)
    {
        iteracio = i; break;
    }
}

kep.set_pixel ( x, y, png::rgb_pixel ( (iteracio*20) ←
    %255, (iteracio ← *40)%255, (iteracio*60)%255 ) );

}

int szazalek = ( double ) y / ( double ) magassag * 100.0;
std::cout << "\r" << szazalek << "%" << std::flush;
}

kep.write ( argv[1] );
std::cout << "\r" << argv[1] << " mentve." << std::endl;
```



## 5.4. A Mandelbrot halmaz CUDA megvalósítása

A CUDA az Nvidia videókártyáknak egy párhuzamos számításokat segítő technológia. Ezen technika segítségével fogjuk felgyórsítani a kép létrehozását. Szükségünk lesz egy Nvidida videókártyára ami rendelkezik CUDA-val. Továbbá telepítenünk kell. A kód kiterjesztése ".cu"

Megoldás forrása: [https://progpater.blog.hu/2011/03/27/a\\_parhuzamossag\\_gyonyorkodtet](https://progpater.blog.hu/2011/03/27/a_parhuzamossag_gyonyorkodtet)

A kód és a magyarázat a következő:

```
#include <png++/image.hpp>
#include <png++/rgb_pixel.hpp>
#include <sys/times.h>
#include <iostream>

#define MERET 600
#define ITER_HAT 32000

__device__ int mandel (int k, int j)
{
    float a = -2.0, b = .7, c = -1.35, d = 1.35;

    int szelesseg = MERET, magassag = MERET, iteraciosHatar = ←
        ITER_HAT;

    float dx = (b - a) / szelesseg;
    float dy = (d - c) / magassag;

    float reC, imC, reZ, imZ, ujreZ, ujimZ;

    int iteracio = 0;
    reC = a + k * dx;
    imC = d - j * dy;

    reZ = 0.0;
    imZ = 0.0;
    iteracio = 0;

    while (reZ * reZ + imZ * imZ < 4 && iteracio < ←
        iteraciosHatar)
    {
        // z_{n+1} = z_n * z_n + c

        ujreZ = reZ * reZ - imZ * imZ + reC;
        ujimZ = 2 * reZ * imZ + imC;
        reZ = ujreZ;
        imZ = ujimZ;

        ++iteracio;
    }
}
```

```
    }
    return iteracio;
}
```

Includeoljuk a két állandót, a kép méretét és az iterációs határt. A következő lépés a Mandelbrot halmaz létrehozása, ezt egy függvénytel hozzuk létre. A függvény előtt jelezzük, hogy a számításokat Cudával végezzük majd a fordításnál. A függvényen belül deklarálunk float tipusú változókat a számításokhoz. A matematikai számítás ugyan az mint az 5.1 feladatban.

```
/*
__global__ void mandelkernel (int *kepadat)
{
    int j = blockIdx.x; int k = blockIdx.y;
    kepadat[j + k * MERET] = mandel (j, k);
}

__global__ void mandelkernel (int *kepadat)
{
    int tj = threadIdx.x;
    int tk = threadIdx.y;

    int j = blockIdx.x * 10 + tj;
    int k = blockIdx.y * 10 + tk;

    kepadat[j + k * MERET] = mandel (j, k);
}
```

A következő függvény előtt "`__global__`" jelzés van, ezzel azt jelezzük, hogy a Cuda fogja végezni a számítást. A "threadIdx" jelzi az aktuális szálat és a "blockIdx" pedig, hogy melyik blokkban folyik a számítás. A kép értékeit a j és a k változókban tároljuk el. Ezt a két értéket fogja kapni az előző függvény.

```
void cudamandel (int kepadat [MERET] [MERET])
{
    int *device_kepadat;

    cudaMalloc ((void **) &device_kepadat, MERET * MERET * sizeof (int));

    dim3 grid (MERET / 10, MERET / 10); dim3 tgrid (10, 10);
    mandelkernel <<< grid, tgrid >>> (device_kepadat);

    cudaMemcpy (kepadat, device_kepadat, MERET * MERET * sizeof (int), cudaMemcpyDeviceToHost);

    cudaFree (device_kepadat);
}
```

A következő függvény a cudamandel(), ez egy 600x600-as tömböt kap. Deklarálunk egy mutatót és a Malloc segítségével lefoglaljuk a megfelelő tárhelyet és a mutató ide fog mutatni. Itt hozzuk létre a megfelelő blokkokat és a végén a tárhelyet felszabadítjuk.

```
int main (int argc, char *argv[])
{
    clock_t delta = clock ();

    struct tms tmsbuf1, tmsbuf2;
    times (&tmsbuf1);

    if (argc != 2)
    {
        std::cout << "Használat: ./mandelpngc fajlnev"; return -1;
    }

    int kepadat [MERET] [MERET];

    cudamandel (kepadat);

    png::image < png::rgb_pixel > kep (MERET, MERET);

    for (int j = 0; j < MERET; ++j)
    {
        for (int k = 0; k < MERET; ++k)
        {
            kep.set_pixel (
                k, j, png::rgb_pixel (
                    255 - (255 * kepadat[j][k]) / ITER_HAT,
                    255 - (255 * kepadat[j][k]) / ITER_HAT,
                    255 - (255 * kepadat[j][k]) / ITER_HAT
                )
            );
        }
    }

    kep.write (argv[1]);

    std::cout << argv[1] << " mentve" << std::endl;

    times (&tmsbuf2);
    std::cout << tmsbuf2.tms_utime - tmsbuf1.tms_utime
        + tmsbuf2.tms_stime - tmsbuf1.tms_stime << std::endl;
```

```
    delta = clock () - delta;
    std::cout << (float) delta / CLOCKS_PER_SEC << " sec" << ←
        std::endl;
}
```

A fő függvényünkben egy idő méréssel kezdünk, lemérjük mennyi időbe telik a gépnek, hogy megalkos-sa a képet. Utánna deklaráljuk a tömböt, meghívjuk a cudamandel() függvényt és már az ismert módon létrehozzuk a képet. A kódot az "nvcc" fordítóval fordítjuk, le kell tölteni, ehez a gép ad segítséget.

A fordítás: "nvcc mandelpngc\_60x60\_100.cu -lpng16 -O3 -o mandelpngc".

Fordítás után futtatjuk: "./mandelpngc c.png"-vel.

Ha egymás mellé tesszük a Cudas és a nem Cudas képalkotást, láthatjuk, hogy a kép elkészítési ideje a Cudasnál sokkal gyorsabb.

## 5.5. Mandelbrot nagyító és utazó C++ nyelven

GUI a Mandelbrot algoritmusra, hogy lehessen egérrel nagyítani egy területet, illetve egy pontot egérrel kiválasztva vizualizálja onnan a komplex iteráció bejárta  $z_n$  komplex számokat!

Megoldás forrása: [https://progpater.blog.hu/2011/03/26/kepes\\_egypercesek](https://progpater.blog.hu/2011/03/26/kepes_egypercesek)

A program programozási nyelve a c++, további feladatban más programozási nyelven is magyarázzuk ezt a programot.

A jelenlegi programhoz több forrásra van szükségünk - például "frakablak.h" header - amelyeket egy map-pába kell letölteni, majd telepítenünk kell a "libqt4-dev"-et a következő parancsal:

"sudo apt-get install libqt4-dev"

A qmake -project parancsal létrehozunk egy .pro fájlt, ebbe meg kell adnunk a QT+=Widgets parancsot a megfelelő helyre. Ez létrehoz egy fájlok.o fájlt és egy makefilet, ezek után make parancsal létrehozzuk a nagyítót.

A fraksal.cpp-ben készül el az ábránk amit majd nagyítani fogunk. Az rgb pixel színezést azonban már a frakablak végzi.

A program kód a következő:

```
#include< QApplication>
#include "frakablak.h"

int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    Frakablak w1,
    w2(-.08292191725019529, -.082921917244591272, ←
        -.9662079988595939, -.9662079988551172, 1200, 3000),
    w3(-.08292191724880625, -.0829219172470933, ←
        -.9662079988581493, -.9662079988563615, 1200, 4000),
    w4(.14388310361318304, .14388310362702217, ←
        .6523089200729396, .6523089200854384, 1200, 38655);
```

```
w1.show();  
w2.show();  
w3.show();  
w4.show();  
  
return a.exec();  
}
```

## 5.6. Mandelbrot nagyító és utazó Java nyelven

Tutorált: Talerger Mark Imre

Ez a feladat ugyan az mint az előző az 5.5-ös, azaz belenagyítunk a mandelbrot halmazba, a különbség viszont az, hogy itt a programozási nyelv a Java.

Kód forrása: <https://www.tankonyvtar.hu/hu/tartalom/tkt/javat-tanitok-javat/apbs02.html>

A program elején létrehozzuk a Mandelbrot halmazt. Ehez az extends szóval hozzá kapcsoljuk a Mandelbrothalmazt építő java kódunkat. A mousePressed() függvényel megadjuk a programnak az egér által kijelölt kordinátákat, majd a kijelölt területen újra számoljuk a halmazt. Aztán feldolgozza a létrejött kép szélességét és magasságát.

```
public class MandelbrotHalmazNagyító extends MandelbrotHalmaz  
{  
    private int x, y;  
    private int mx, my;  
  
    public MandelbrotHalmazNagyító(double a, double b, double c, ←  
        double d, int szélesség, int iterációsHatár)  
    {  
        super(a, b, c, d, szélesség, iterációsHatár);  
        setTitle("A Mandelbrot halmaz nagyításai");  
  
        addMouseListener(new java.awt.event.MouseAdapter()  
        {  
            public void mousePressed(java.awt.event.MouseEvent m)  
            {  
                x = m.getX();  
                y = m.getY();  
                mx = 0; my = 0;  
                repaint();  
            }  
  
            public void mouseReleased(java.awt.event.MouseEvent m)  
            {  
                double dx =  
                    (MandelbrotHalmazNagyító.this.b - ←  
                     MandelbrotHalmazNagyító.this.a)
```

```
        /MandelbrothalmazNagyító.this.szélesség ←
        ;
        double dy =
        (MandelbrothalmazNagyító.this.d - ←
        MandelbrothalmazNagyító.this.c)
        /MandelbrothalmazNagyító.this.magasság;

        new MandelbrothalmazNagyító(
            MandelbrothalmazNagyító.this.a+x*dx,
            MandelbrothalmazNagyító.this.a+x*dx+mx*dx,
            MandelbrothalmazNagyító.this.d-y*dy-my*dy,
            MandelbrothalmazNagyító.this.d-y*dy, 600,
            MandelbrothalmazNagyító.this.iterációsHatár
        );
    }
}
addMouseMotionListener(new java.awt.event.MouseMotionAdapter())
{
    public void mouseDragged(java.awt.event.MouseEvent m)
    {
        mx = m.getX() - x; my = m.getY() - y; repaint();
    }
}
}
```

A pillanatfelvétel() függvény egy pillanatfelvételt készítünk. A függvényen belül elnevezzük a tartomány szerint és egy png formátumú képet készítünk a pillanat felvételből. A nagyítás során láthatunk egy segítő négyzetet, ezt a paint() függvényel hozzuk létre.

```
public void pillanatfelvétel()
{
    java.awt.image.BufferedImage mentKép =
        new java.awt.image.BufferedImage(szélesség, magasság, java.awt.image.BufferedImage.TYPE_INT_RGB);

    java.awt.Graphics g = mentKép.getGraphics();

    g.drawImage(kép, 0, 0, this);
    g.setColor(java.awt.Color.BLUE);

    g.drawString("a=" + a, 10, 15);
    g.drawString("b=" + b, 10, 30);
    g.drawString("c=" + c, 10, 45);
    g.drawString("d=" + d, 10, 60);
    g.drawString("n=" + iterációsHatár, 10, 75);

    if (számításFut)
```

```
{  
    g.setColor(java.awt.Color.RED);  
    g.drawLine(0, sor, getWidth(), sor);  
}  
  
g.setColor(java.awt.Color.GREEN);  
g.drawRect(x, y, mx, my);  
g.dispose();  
  
StringBuffer sb = new StringBuffer();  
  
sb = sb.delete(0, sb.length());  
sb.append("MandelbrotHalmazNagyitas_");  
sb.append(++pillanatfelvételszámláló);  
  
sb.append("_");  
sb.append(a);  
sb.append("_");  
sb.append(b);  
sb.append("_");  
sb.append(c);  
sb.append("_");  
sb.append(d);  
sb.append(".png");  
  
try {  
    javax.imageio.ImageIO.write(mentKép, "png", new java.io.File(sb.toString()));  
} catch(java.io.IOException e) {  
    e.printStackTrace();  
}  
}  
  
}  
  
public void paint(java.awt.Graphics g)  
{  
    g.drawImage(kép, 0, 0, this);  
  
    if(számításFut)  
    {  
        g.setColor(java.awt.Color.RED);  
        g.drawLine(0, sor, getWidth(), sor);  
    }  
  
    g.setColor(java.awt.Color.GREEN);  
    g.drawRect(x, y, mx, my);  
}  
  
public static void main(String[] args)  
{
```

```
    new MandelbrotHalmazNagyító(-2.0, .7, -1.35, 1.35, 600, 255);  
}  
}
```

## 5.7. Vörös Pipacs Pokol/fel a láváig és vissza

Megoldás videó: <https://youtu.be/I6n8acZoyoo>

Megoldás forrása: <https://github.com/nbatfai/RedFlowerHell>

Ebben a feladatban már Steve érzékeli a veszélyt és el is menekül előle. Ami itt fontos az a megfelelő időben a veszély (láva) érzékelése. Tehát Steve felszalad egészen addig amíg a 3x3x3-mas önmaga körüli területen érzékeli a lávát. Ahol ugyanis érzékeli gyorsan megfordul és el kezd oldalazva menni lefelé kockánként. Egy **turn 1** utasítással tesszük ezt lehetővé.

A programkód megtalálható a repómban a következő linken: [lava\\_érzékelése](#)

## 6. fejezet

# Helló, Welch!

### 6.1. Első osztályom

Valósítsd meg C++-ban és Java-ban az módosított polártranszformációs algoritmust! A matek háttér teljesen irreleváns, csak annyiban érdekes, hogy az algoritmus egy számítása során két normálist számol ki, az egyiket elspájzolod és egy további logikai taggal az osztályban jelzed, hogy van vagy nincs eltéve kiszámolt szám.

Megoldás videó:

Az Objektum Orientált Programozás (OOP) alapja az osztályozás (class), vagyis minden egyes osztály egy-egy objektum.

Akkor nevezünk egy nyelvet objektum orientálnak, ha egymással kommunikáló és műveleteket végző objektumokból áll egy program. Az OOP egyszerűsít a programon, növeli a hatékonyságot és átláthatóbb. A C++ és a Java egyaránt OOP nyelvek.

AZ első program az OOP bevezetéshez a polargen. Fontos lépés a feladatban, hogy egy számítási lépés két normális eloszlású számot állít elő, és minden második meghíváskor fel fogja használni az előzőleg tároltban elhelyezett számot, tehát nem fog mindig számolni.

Először a C++ nyelven fogunk végignézni a programot. A program 3 részből fog állni.

Megoldás forrása: <https://sourceforge.net/p/udprog/code/ci/master/tree/source/labor/polargen/>

Bátfai Norbert forrását felhasználva.

Az első program C++ nyelven lesz látható, amely képes objektumokkal is dolgozni ezért OOP nyelv.

```
#ifndef POLARGEN__H
#define POLARGEN__H

#include <cstdlib>
#include <cmath>
#include <ctime>

class PolarGen
{
```

```
public:  
PolarGen ()  
{  
    nincsTarolt = true;  
    std::srand (std::time (NULL));  
}  
~PolarGen ()  
{  
}  
double kovetkezo ();  
  
private:  
bool nincsTarolt;  
double tarolt;  
};  
  
#endif
```

Az első kódrészlet a header fájl. Itt definiáljuk a polargen headert és includeoljuk a szükséges headereket. A cmath header a matematikai számításokat tartalmazza, a ctime header a nekünk szükséges srand() időmérő függvényt tartalmazza.

Létrehozzuk a PolarGen osztályt (class), amelynek a publikus részében egy konstruktort majd egy destruktort hozunk létre. A private részben deklarálunk két változót, egy logikai (bool) nincsTarolt ami a tárolásról ad majd információt, és egy double tarolt változót, ezek azért kerültek a privát részbe, hogy az értékükön ne legyen módunk változtatni. Viszzatérve a publikus részhez, a konstruktorban kezdőértéket adunk a logikai változónak, ami jelen esetben igaz, vagyis hogy üres a változó. Az srand() függvény csak az időt méri, felhasználva a számítógép idő bitjét. A destruktur a programunk végén fog lefutni, amely a memóriaszivárgást akadályozza meg.

```
#include "polargen.h"  
  
double  
PolarGen::kovetkezo ()  
{  
if (nincsTarolt)  
{  
    double u1, u2, v1, v2, w;  
    do  
    {  
        u1 = std::rand () / (RAND_MAX + 1.0);  
        u2 = std::rand () / (RAND_MAX + 1.0);  
        v1 = 2 * u1 - 1;  
        v2 = 2 * u2 - 1;  
        w = v1 * v1 + v2 * v2;  
    }  
    while (w > 1);  
  
    double r = std::sqrt ((-2 * std::log (w)) / w);
```

```
tarolt = r * v2;
nincsTarolt = !nincsTarolt;

return r * v1;
}
else
{
nincsTarolt = !nincsTarolt;
return tarolt;
}
}
```

A második kódrészlet a számításokat végzi, itt valósítjuk meg a polártranszformációt. Includeoljuk a már megbeszélt polargen.h headert.

A lényeg, hogy az if fogja vizsgálni a nincsTarolt logikai értékét, azaz hogy van-e számunk a változóban, ha van akkor elvégzi a matematikai számításokat (ezek most nem fontosak). Ha nincs benne szám akkor új számot fogunk készíteni.

```
#include <iostream>
#include "polargen.h"

using namespace std;

int
main (int argc, char **argv)
{
PolarGen pg;

cout<<endl;
for (int i = 0; i < 10; ++i)
    cout <<"\t" << pg.kovetkezo () << endl;

return 0;
}
```

Végül a harmadik kódrészlet maga a fő függvényünk amelyben kidomborodik a program, itt futtatja le a program a for ciklusban a számításokat végző függvéyünket. Jelen esetben 10x fog lefutni.

```
-0.411868
-0.856906
-0.47843
-1.46706
-0.598212
2.21198
-0.656053
0.663344
1.16654
0.0720679
```

6.1. ábra. Polargenteszt.cpp futtatása

A következő program ugyan az mint az előző csak Java nyelven, ez a programozási nyelv amit már az előző fejezetekben tárgyaltunk, egy OOP nyelv amely csak objektumokkal dolgozik, a C++-al szemben.

```
public class PolarGen{

    boolean nincsTarolt = true;
    double tarolt;

    public PolarGen() {

        nincsTarolt = true;
    }

    public double kovetkezo() {

        if(nincsTarolt) {

            double u1, u2, v1, v2, w;
            do {
                u1 = Math.random();
                u2 = Math.random();

                v1 = 2*u1 - 1;
                v2 = 2*u2 - 1;

                w = v1*v1 + v2*v2;

            } while(w > 1);

            double r = Math.sqrt((-2*Math.log(w))/w);
        }
    }
}
```

```
tarolt = r*v2;
nincsTarolt = !nincsTarolt;

return r*v1;watch?v=9_ylSciSjBw&feature=youtu. ←
be

} else {
    nincsTarolt = !nincsTarolt;
    return tarolt;
}
}

public static void main(String[] args) {

    PolarGen g = new PolarGen();

    for(int i=0; i<10; ++i)
        System.out.println(g.kovetkezo());
}

}
```

Különösen nincs mit magyarázni, mert a program ugyan úgy működik mint az előző. A különbség annyi, hogy ebben a nyelvben csak objektumokat/függvényeket tudunk létrehozni, így már a program csipetben látható is, hogy a program kód egyben van és nincs részekre bontva.

## 6.2. LZW

Valósítsd meg C-ben az LZW algoritmus fa-építését!

A program egy bináris fát fog felépíteni bemeneti adatokból. Az LZW binfa minden csomópontjának (elágazás) két gyermeké (további ága) lehet. A csomópontok gyermekei vagy 0-ás vagy 1-es lehet. Az 1-es a jobb oldali, a 0-ás a bal oldali.

A binfa lényege hogy a gyökérből (kitüntetett elem) elérhetünk minden elemet.

Megoldás forrása Bátfai Norbert Tanárúrtól: [https://progpater.blog.hu/2011/03/05/labormeres\\_othon\\_avagy\\_hogp5MQomtcQIdfTeZvPInhgRxu-CCsxGOx453MSrGk](https://progpater.blog.hu/2011/03/05/labormeres_othon_avagy_hogp5MQomtcQIdfTeZvPInhgRxu-CCsxGOx453MSrGk)

```
BINFA_PTR gyoker = uj_elem ();
gyoker->ertek = '/';
gyoker->bal nulla = gyoker->jobb_egy = NULL;
BINFA_PTR fa = gyoker;
long max=0;
while (read (0, (void *) &b, sizeof(unsigned char)))
{
    for (i=0;i<8; ++i)
```

```
{  
    egy_e= b& 0x80;  
    if ((egy_e >>7)==0)  
        c='1';  
    else  
        c='0';  
}  
if (c == '0')  
{  
    if (fa->bal_nulla == NULL)  
    {  
        fa->bal_nulla = uj_elem ();  
        fa->bal_nulla->ertek = 0;  
        fa->bal_nulla->bal_nulla = fa->bal_nulla->jobb_egy = ←  
            NULL;  
        fa = gyoker;  
    }  
    else  
    {  
        fa = fa->bal_nulla;  
    }  
}  
else  
{  
    if (fa->jobb_egy == NULL)  
    {  
        fa->jobb_egy = uj_elem ();  
        fa->jobb_egy->ertek = 1;  
        fa->jobb_egy->bal_nulla = fa->jobb_egy->jobb_egy = ←  
            NULL;  
        fa = gyoker;  
    }  
    else  
    {  
        fa = fa->jobb_egy;  
    }  
}  
}
```

A fa építésében két esetünk van. A 0-ás és az 1-es beépítése a megfelelő helyre.

Ha 0-ás a bemenet akkor a gyökértől kezdve megnézzük, hogy van egy 0-ás (bal oldali) gyermeke, ha van akkor rálépünk, ha viszont nincs akkor létre kell hozni, majd miután létrehoztuk, vissza lépünk a gyökérre és olvassuk tovább a bemenetet.

Az 1-es bemenet esetén hasonlóan járunk el mint a 0-ásnál. A gyökértől kezdve megnézzük, hogy van egy 1-es (jobb oldali) gyermeke, ha van akkor rálépünk, ha viszont nincs akkor létre kell hozni, majd miután létrehoztuk, vissza lépünk a gyökérre és olvassuk tovább a bemenetet.

Végül az LZWBinfa ezen algoritmus alapján fog felépülni, és beépülni minden egyes csomópont (1-es vagy 0-ás) a fába.

Ebben a programrészletben a fát inorder bejárással dolgozzuk fel, ami azt jelenti hogy elsőre a bal oldallal, aztán a gyökérrel és végül a jobb oldallal foglalkozunk.

### 6.3. Fabejárás

Járd be az előző (inorder bejárású) fát pre- és posztorder is!

Az előző feladatban létrehozott fát több féleképpen be lehet járni: inorder, preorder és postorder.

Lent e három fabejárás kódját lehet látni.

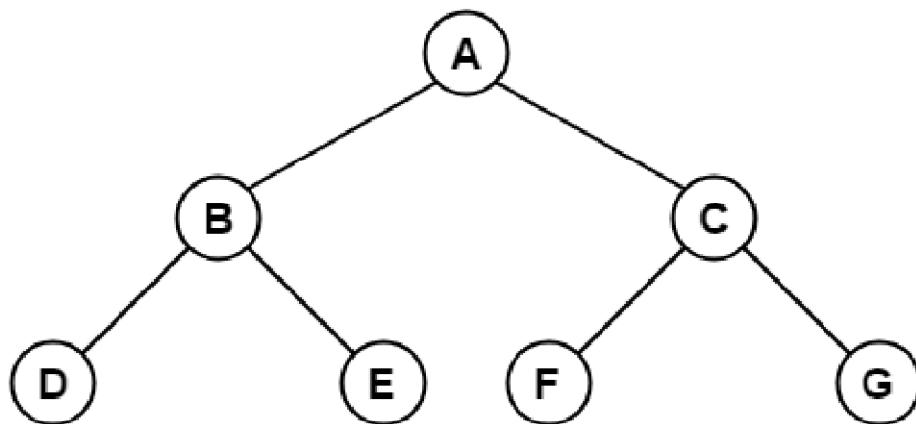
Az első az inorder:

```
void kiir (Csomopont * elem, std::ostream & os)
{
    if (elem != NULL)
    {
        ++melyseg;
        kiir (elem->egyesGyermek (), os); //egyes gyermek ←
                                         feldolgozása

        for (int i = 0; i < melyseg; ++i)
            os << "---";
        os << elem->getBetu () << "(" << melyseg - 1 << ")" << ←
                                         std::endl; //csomópont feldolgozása

        kiir (elem->>nullasGyermek (), os); //nullás gyermek ←
                                         feldolgozása
        --melyseg;
    }
}
```

Ez már az előző feladatban tárgyalva volt. Tehát az inorder bejárással a fát úgy dolgozzuk fel, hogy egy adott csomópontnak minden a bal gyerekét dolgozzuk fel először, azután az aktuális csomópontot, azután pedig a jobb gyermekét. A kitüntetett elem feldolgozása a bejárás közepén kerül sorra.



Inorder Traversal : D , B , E , A , F , C , G

6.2. ábra. Inorder fa bejárás

Kép forrása: <https://medium.com/@andrewmf/iterative-in-order-tree-traversal-using-dynamic-programming-508f189eb494>

Az fenti programrészletben a void tipusú kiir() függvény inorder módon fogja az elemeket kiíratni.

A második a preorder:

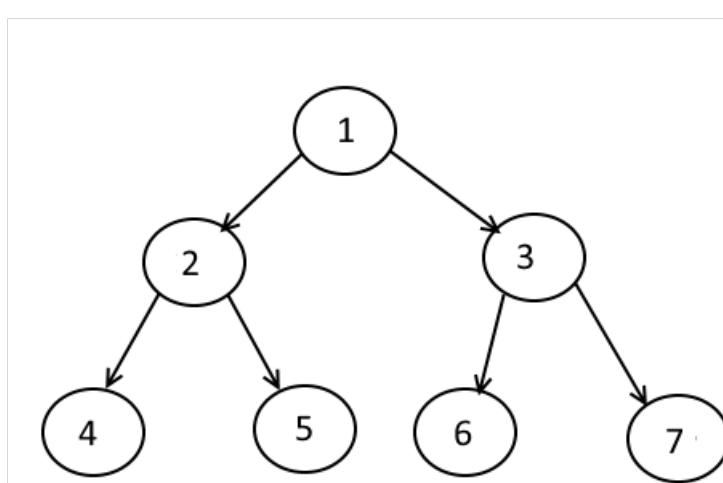
```
void kiir (Csomopont * elem, std::ostream & os)
{
    // Nem létező csomóponttal nem foglalkozunk... azaz ez a ↵
    // rekurzió leállítása
    if (elem != NULL)
    {
        ++melyseg;

        for (int i = 0; i < melyseg; ++i)
            os << "---";
        os << elem->getBetu () << "(" << melyseg - 1 << ")" << ↵
            std::endl;

        kiir (elem->egyesGyermek (), os); //egyes gyermek ↵
                                         feldolgozása

        kiir (elem->>nullasGyermek (), os); //nullás gyermek ↵
                                         feldolgozása
        --melyseg;
    }
}
```

A preorder bejárással elsőre a csomópontot, majd a bal, aztán a jobb gyermeket dolgozzuk fel. Ez azt jelenti hogy a fát a gyökérből indulva, haladva a fa bal oldalán végig feldogozzuk az össze csomópont bal oldali gyemekét, majd a jobb oldalit, amiíg vissza nem érünk a gyökérhez, és akkor a feldolgozás ugyanúgy folytatódik, tovább a fa jobb oldalát is bejárjuk. A kitüntetett elem lesz feldolgozva elsőre.



Preorder Traversal: 1 2 4 5 3 6 7

6.3. ábra. Preorder fa bejárás

Kép forrása: <https://algorithms.tutorialhorizon.com/binary-tree-preorder-traversal-non-recursive-approach/>

Az fenti programrészletben a void tipusú kiir() függvény preorder módon fogja az elemeket kiíratni.

A harmadik a postorder:

```
void kiir (Csomopont * elem, std::ostream & os)
{
    if (elem != NULL)
    {
        ++melyseg;

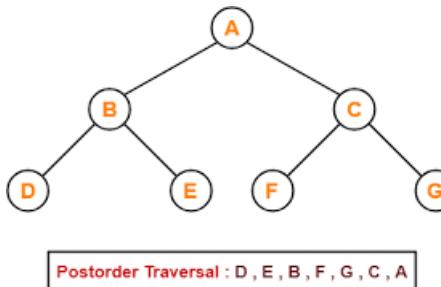
        kiir (elem->egyesGyermek (), os);

        kiir (elem->>nullasGyermek (), os);

        for (int i = 0; i < melyseg; ++i)
            os << "---";
        os << elem->getBetu () << "(" << melyseg - 1 << ")" << endl;
    }
}
```

```
        --melyseg;  
    }  
}
```

A postorder bejárással elsőre a bal oldali gyermeket, majd a jobb oldali gyermeket és végül az adott csomópontot dolgozzuk fel, elérve egészen a gyökérig. Itt a kitüntetett elem kerül utoljára feldolgozásra.



6.4. ábra. Postorder fa bejárás

Kép forrása: <https://www.wikitechy.com/technology/java-algorithm-iterative-postorder-traversal-set-2-using-one-stack/>

Az fenti programrészletben a void tipusú kiir() függvény postorder módon fogja az elemeket kiíratni.

## 6.4. Tag a gyökér

Az LZW algoritmust ültess át egy C++ osztályba, legyen egy Tree és egy beágyazott Node osztálya. A gyökér csomópont legyen kompozícióban a fával!

A Csomopont osztályban létrehozzuk a '/' betűt tartalmazó objektumot, ami része a fának, vagyis tagként veszi a gyökeret is.

Megoldás forrása: <https://github.com/RubiMaistro/Bevprog/blob/master/Binfa/z3a7.cpp>

A forrás Báfai Norbert Tanárúrtól származik, a repó pedig saját.

```
#include <iostream>  
#include <cmath>  
#include <fstream>  
  
class LZWBinFa  
{  
public:  
  
    LZWBinFa () :fa (&gyoker)  
    {  
    }  
}
```

```
~LZWBinFa ()
{
    szabadit (gyoker.egyesGyermek ());
    szabadit (gyoker.nullasGyermek ());
}

void operator<< (char b)
{
    if (b == '0')
    {

        if (!fa->nullasGyermek ())
        {

            Csomopont *uj = new Csomopont ('0');
            fa->ujNullasGyermek (uj);

            fa = &gyoker;
        }
        else
        {
            fa = fa->>nullasGyermek ();
        }
    }
    else
    {
        if (!fa->egyesGyermek ())
        {
            Csomopont *uj = new Csomopont ('1');
            fa->ujEgyesGyermek (uj);
            fa = &gyoker;
        }
        else
        {
            fa = fa->egyesGyermek ();
        }
    }
}

void kiir (void)
{
    melyseg = 0;

    kiir (&gyoker, std::cout);
}

int getMelyseg (void);
```

```
double getAtlag (void);
double getSzoras (void);

friend std::ostream & operator<< (std::ostream & os, LZWBinFa & bf)
{
    bf.kiir (os);
    return os;
}
void kiir (std::ostream & os)
{
    melyseg = 0;
    kiir (&gyoker, os);
}
```

A program magyarázata:

Az LZWBinfa osztályban van privát és publikus rész.

Az osztály konstruktora és destruktora a publikus részében szerepel. A public részben szerepel az algoritmus, azaz a 0-ás és 1-es csomópontok beágyazása. Itt van a még kiir() függvény amely a kimenetre írat.

```
private:
    class Csomopont
    {
public:

    Csomopont (char b = '/') : betu (b), balNulla (0), jobbEgy (0)
    {
    };
    ~Csomopont ()
    {
    };

    Csomopont *nullasGyermekek () const
    {
        return balNulla;
    }

    Csomopont *egyesGyermekek () const
    {
        return jobbEgy;
    }

    void ujNullasGyermekek (Csmopont * gy)
    {
        balNulla = gy;
    }
```

```
void ujEgyesGyermek (Csomopont * gy)
{
    jobbEgy = gy;
}

char getBetu () const
{
    return betu;
}
```

A private részben van az LZWBinfá osztálynak a Csomopont osztálya. Ennek van konstruktora, destruktora (memóriaszivárgás elkerülés végett), nullás és egyes gyermeket lekérdező függvények (a vizsgáló függvények) és új nullás és új egyes gyermek létrehozásáért felelős függvények, valamint a lekérdező függvény, hogy mi található a csomópontban.

```
private:

    char betu;

    Csomopont *balNulla;
    Csomopont *jobbEgy;

    Csomopont (const Csomopont &);
    Csomopont & operator= (const Csomopont &);
};
```

A Csomopont osztály private részben van deklarálva a betű változó, ami megmondja, hogy milyen betű van a csomópontban, majd a jobb és bal gyermeket is deklaráljuk. Ez alatt található a Csomopont osztály másoló konstruktora.

```
Csomopont *fa;

int melyseg, atlagosszeg, atlagdb;
double szorasosszeg;

LZWBInFa (const LZWBInFa &);
LZWBInFa & operator= (const LZWBInFa &);

void kiir (Csmopont * elem, std::ostream & os) //PREORDER
{
    if (elem != NULL)
    {
        ++melyseg;

        for (int i = 0; i < melyseg; ++i)
            os << "---";
        os << elem->getBetu () << "(" << melyseg - 1 << ")" << endl;

        kiir (elem->egyesGyermek (), os);
    }
}
```

```
        kiir (elem->nullasGyermek (), os);

        --melyseg;
    }
}

void szabadit (Csomopont * elem)
{
    if (elem != NULL)
    {
        szabadit (elem->egyesGyermek ());
        szabadit (elem->nullasGyermek ());

        delete elem;
    }
}
```

Tovább haladva, az LZWBina másoló konstruktora, a kiir függvényben kiírjuk a függvényt az os csatornára, itt tudjuk megadni, hogy milyen bejárással irja ki a fát, a fa bejárások feladatban volt róla szó. Ezt követi egy szabadit függvény, mely felszabadítja a szabad táróból az egyes gyermeket, a nullásat rekurzívan és végül az elemet is.

```
protected:

    Csomopont gyoker;
    int maxMelyseg;
    double atlag, szoras;

    void rmelyseg (Csmopont * elem);
    void ratlag (Csmopont * elem);
    void rszoras (Csmopont * elem);

};

int
LZWBInFa::getMelyseg (void)
{
    melyseg = maxMelyseg = 0;
    rmelyseg (&gyoker);
    return maxMelyseg - 1;
}

double
LZWBInFa::getAtlag (void)
{
    melyseg = atlagosszeg = atlagdb = 0;
    ratlag (&gyoker);
    atlag = ((double) atlagosszeg) / atlagdb;
```

```
        return atlag;
    }

    double
LZWBinFa::getSzoras (void)
{
    atlag = getAtlag ();
    szorasosszeg = 0.0;
    melyseg = atlagdb = 0;

    rszoras (&gyoker);

    if (atlagdb - 1 > 0)
        szoras = std::sqrt (szorasosszeg / (atlagdb - 1));
    else
        szoras = std::sqrt (szorasosszeg);

    return szoras;
}

void
LZWBinFa::rmelyseg (Csomopont * elem)
{
    if (elem != NULL)
    {
        ++melyseg;
        if (melyseg > maxMelyseg)
            maxMelyseg = melyseg;
        rmelyseg (elem->egyesGyermek ());

        rmelyseg (elem->>nullasGyermek ());
        --melyseg;
    }
}

void
LZWBinFa::ratlag (Csomopont * elem)
{
    if (elem != NULL)
    {
        ++melyseg;
        ratlag (elem->egyesGyermek ());
        ratlag (elem->>nullasGyermek ());
        --melyseg;
        if (elem->egyesGyermek () == NULL && elem->>nullasGyermek () == NULL)
        {
            ++atlagdb;
            atlagosszeg += melyseg;
        }
    }
}
```

```
        }

    }

    void
LZWBinFa::rszoras (Csomopont * elem)
{
    if (elem != NULL)
    {
        ++melyseg;
        rszoras (elem->egyesGyermek ());
        rszoras (elem->>nullasGyermek ());
        --melyseg;
        if (elem->egyesGyermek () == NULL && elem->>nullasGyermek () == NULL)
        {
            ++atlagdb;
            szorasosszeg += ((melyseg - atlag) * (melyseg - atlag)) /;
        }
    }
}
```

A protected rész arra szolgál, hogy a jövőbeni változtatások látszódjanak majd a gyermek osztályban is. Itt jelenik meg a Csomopont gyoker is, valamint az rmelyseg, ratlag, rszoras függvények is. A protected rész tulajdonképpen öröklődésre, friend függvényekre használatos.

```
    void
usage (void)
{
    std::cout << "Usage: lzwtree in_file -o out_file" << std::endl;
}

int
main (int argc, char *argv[])
{

    if (argc != 4)
    {
        usage ();
        return -1;
    }

    inFile = *++argv;

    if ((*(*++argv) + 1) != 'o')
    {
        usage ();
        return -2;
    }
}
```

```
}

std::fstream beFile (inFile, std::ios_base::in);

if (!beFile)
{
    std::cout << inFile << " nem létezik..." << std::endl;
    usage ();
    return -3;
}

std::fstream kiFile (*++argv, std::ios_base::out);

unsigned char b;
LZWBinFa binFa;

while (beFile.read ((char *) &b, sizeof (unsigned char)))
    if (b == 0x0a)
        break;

bool kommentben = false;

while (beFile.read ((char *) &b, sizeof (unsigned char)))
{

    if (b == 0x3e)
    {
        kommentben = true;
        continue;
    }

    if (b == 0x0a)
    {
        kommentben = false;
        continue;
    }

    if (kommentben)
        continue;

    if (b == 0x4e)
        continue;

    for (int i = 0; i < 8; ++i)
    {
        if (b & 0x80)
            binFa << '1';
        else
            binFa << '0';
        b <= 1;
```

```
    }

}

kiFile << binFa;

kiFile << "depth = " << binFa.getMelyseg () << std::endl;
kiFile << "mean = " << binFa.getAtlag () << std::endl;
kiFile << "var = " << binFa.getSzoras () << std::endl;

kiFile.close ();
beFile.close ();

return 0;
}
```

A main függvényben láthatjuk az egyszerű hibakezeléseket: az argumentumszám, a kapcsoló. Majd olvasunk a bemeneti fájlból a karaktereket az fstream segítségével, ezeket átalakítjuk 0-vá vagy 1-sé, ezt logikai és használatával valósítjuk meg. A kifile-ba irányítjuk a binfát, és kiirunk még róla néhány információt ezt követően (mélység, átlag, szórás). Majd bezárjuk az fstream fájlokat.

## 6.5. Mutató a gyökér

Írd át az előző forrást, hogy a gyökér csomópont ne kompozícióban, csak aggregációban legyen a fával!

A LZWBinfa programot vesszük alapul és a gyökér elemet átalakítjuk mutatóvá. A gyökér elem az osztály protected részében van.

```
protected:
Csomopont *gyoker;
int maxMelyseg;
double atlag, szoras;
```

A Csomopont osztályban a gyoker elemet át is írtuk mutatóvá. De ennyivel még nem vagyunk készen, mert ahogyan most egyszerűen átírtuk a kódban az osztály elemet mutatóra, minden helyen ahol használtuk az elemet, mutatóként kell használnunk.

```
//előtte:
fa=&gyoker;

//utánna:
fa=gyoker;

//előtte:
szabadit (gyoker.egyesGyermek ());
szabadit (gyoker.nullasGyermek ());

//utánna:
szabadit (gyoker->egyesGyermek ());
```

```
    szabadit (gyoker->nullasGyermek ());
```

A fenti kódcsipetben hoztam példákat. Először is nem a mutató memóriacímét akarjuk már átadni hanem az értékét, ezért töröltük a & jeleket. Továbbá nem elemként (.) hivatkozunk a gyokerre hanem mutatóként (->).

A gyökér mutatónak foglalni kell memóriát is, a következőképpen.

```
LZWBinFa ()
{
    gyoker= new Csomopont ('/');
    fa = gyoker;
}
~LZWBinFa ()
{
    szabadit (gyoker->egyesGyermek ());
    szabadit (gyoker->nullasGyermek ());
    delete (gyoker);
}
```

A konstruktorban megy végbe a memóriafoglalás, így a gyökér is egy csomópont lesz. A destruktörben a felszabadítást (a szabadit() függvényünkkel) és a törlést (a delete() függvénnyel) láthatjuk.

## 6.6. Mozgató szemantika

Ír az előző programhoz mozgató konstruktort és értékadást, a mozgató konstruktur legyen a mozgató értékadásra alapozva!

A feladat annyiból áll, hogy mozgató konstruktort adunk hozzá a programunkhoz, hogya fában az elemeket mozgatni/másolni tudjuk.

Ez a feladat a védés második feladata. A program már előre megírt, Bátfai Norbert Tanárúr felhasználásra bocsátása jóvoltából használom fel ehez a feladathoz.

Saját megoldás videó: (videó link) (készül)

Megoldás forrása: [https://gitlab.com/nbatfai/bhax/-/blob/master/distance\\_learning/ziv\\_lempel\\_welch/z3a18qa5\\_f](https://gitlab.com/nbatfai/bhax/-/blob/master/distance_learning/ziv_lempel_welch/z3a18qa5_f)

```
[

    #include <iostream>
    #include <random>
    #include <functional>
    #include <chrono>

    class Unirand {

        private:
            std::function <int ()> random;

        public:
            Unirand (long seed, int min, int max) : random (
```

```
        std::bind(
            std::uniform_int_distribution<>(min, max),
            std::default_random_engine(seed)
        )
    ) {}

int operator() () { return random(); }

};

template <typename ValueType>
class BinRandTree {

protected:
    class Node {

private:
    ValueType value;
    Node *left;
    Node *right;
    int count{0};

    // TODO rule of five
    Node(const Node &);
    Node & operator=(const Node &);
    Node(Node &&);
    Node & operator=(Node &&);

public:
    Node(ValueType value, int count=0) : value(value), count ←
        (count), left(nullptr), right(nullptr) {}
    ValueType getValue() const {return value;}
    Node * leftChild() const {return left;}
    Node * rightChild() const {return right;}
    void leftChild(Node * node){left = node;}
    void rightChild(Node * node){right = node;}
    int getCount() const {return count;}
    void incCount(){++count;}
};

    Node *root;
    Node *treep;
    int depth{0};

private:
    // TODO rule of five

public:
    BinRandTree(Node *root = nullptr, Node *treep = nullptr) : ←
        root(root), treep(treep) {
```

```
        std::cout << "BT ctor" << std::endl;
    }

BinRandTree(const BinRandTree & old) {
    std::cout << "BT copy ctor" << std::endl;

    root = cp(old.root, old.treep);

}

Node * cp(Node *node, Node *treep)
{
    Node * newNode = nullptr;

    if(node)
    {
        newNode = new Node(node->getValue(), 42 /*node-> ←
                                                 getCount () */);

        newNode->leftChild(cp(node->leftChild(), treep));
        newNode->rightChild(cp(node->rightChild(), treep));

        if(node == treep)
            this->treep = newNode;
    }

    return newNode;
}

BinRandTree & operator=(const BinRandTree & old) {
    std::cout << "BT copy assign" << std::endl;

    BinRandTree tmp{old};
    std::swap(*this, tmp);
    return *this;
}

BinRandTree(BinRandTree && old) {
    std::cout << "BT move ctor" << std::endl;

    root = nullptr;
    *this = std::move(old);
}

BinRandTree & operator=(BinRandTree && old) {
    std::cout << "BT move assign" << std::endl;

    std::swap(old.root, root);
    std::swap(old.treep, treep);
```

```
        return *this;
    }

~BinRandTree() {
    std::cout << "BT dtor" << std::endl;
    deltree(root);
}
BinRandTree & operator<<(ValueType value);
void print(){print(root, std::cout);}
void printr(){print(*root, std::cout);}
void print(Node *node, std::ostream & os);
void print(const Node & cnode, std::ostream & os);
void deltree(Node *node);

Unirand ur{std::chrono::system_clock::now().time_since_epoch().count(), 0, 2};

int whereToPut() {
    return ur();
}

};

template <typename ValueType>
class BinSearchTree : public BinRandTree<ValueType> {

public:
    BinSearchTree() {}
    BinSearchTree & operator<<(ValueType value);

};

template <typename ValueType, ValueType vr, ValueType v0>
class ZLWTree : public BinRandTree<ValueType> {

public:
    ZLWTree(): BinRandTree<ValueType>(new typename BinRandTree<ValueType>::Node(vr)) {
        this->treep = this->root;
    }
    ZLWTree & operator<<(ValueType value);

};
```

```
template <typename ValueType>
BinRandTree<ValueType> & BinRandTree<ValueType>::operator<< ( ←
    ValueType value)
{

    int rnd = whereToPut();

    if (!treep) {

        root = treep = new Node(value);

    } else if (treep->getValue() == value) {

        treep->incCount();

    } else if (!rnd) {

        treep = root;
        *this << value;

    } else if (rnd == 1) {

        if (!treep->leftChild()) {

            treep->leftChild(new Node(value));

        } else {

            treep = treep->leftChild();
            *this << value;
        }

    } else if (rnd == 2) {

        if (!treep->rightChild()) {

            treep->rightChild(new Node(value));

        } else {

            treep = treep->rightChild();
            *this << value;
        }

    }

    return *this;
}
```

```
template <typename ValueType>
BinSearchTree<ValueType> & BinSearchTree<ValueType>::operator <-
<<(ValueType value)
{
    if(!this->treep) {

        this->root = this->treep = new typename BinRandTree< <-
ValueType>::Node(value);

    } else if (this->treep->getValue() == value) {

        this->treep->incCount();

    } else if (this->treep->getValue() > value) {

        if(!this->treep->leftChild()) {

            this->treep->leftChild(new typename BinRandTree< <-
ValueType>::Node(value));

        } else {

            this->treep = this->treep->leftChild();
            *this << value;
        }

    } else if (this->treep->getValue() < value) {

        if(!this->treep->rightChild()) {

            this->treep->rightChild(new typename BinRandTree< <-
ValueType>::Node(value));

        } else {

            this->treep = this->treep->rightChild();
            *this << value;
        }

    }

    this->treep = this->root;

    return *this;
}

template <typename ValueType, ValueType vr, ValueType v0>
ZLWTree<ValueType, vr, v0> & ZLWTree<ValueType, vr, v0>:: <-
operator<<(ValueType value)
```

```
{

    if (value == v0) {

        if (!this->treep->leftChild()) {

            typename BinRandTree<ValueType>::Node * node = new ←
                typename BinRandTree<ValueType>::Node(value);
            this->treep->leftChild(node);
            this->treep = this->root;

        } else {

            this->treep = this->treep->leftChild();
        }

    } else {

        if (!this->treep->rightChild()) {

            typename BinRandTree<ValueType>::Node * node = new ←
                typename BinRandTree<ValueType>::Node(value);
            this->treep->rightChild(node);
            this->treep = this->root;

        } else {

            this->treep = this->treep->rightChild();
        }

    }

    return *this;
}

template <typename ValueType>
void BinRandTree<ValueType>::print(Node *node, std::ostream & ←
os)
{
    if (node)
    {
        ++depth;
        print(node->leftChild(), os);

        for (int i{0}; i<depth; ++i)
            os << "---";
        os << node->getValue() << " " << depth << " " << node-> ←
getCount() << std::endl;

        print(node->rightChild(), os);
    }
}
```

```
--depth;
}

}

template <typename ValueType>
void BinRandTree<ValueType>::print(const Node &node, std::←
ostream & os)
{
    ++depth;

    if(node.leftChild())
        print(*node.leftChild(), os);

    for(int i{0}; i<depth; ++i)
        os << "---";
    os << node.getValue() << " " << depth << " " << node.←
        getCount() << std::endl;

    if(node.rightChild())
        print(*node.rightChild(), os);

    --depth;
}

template <typename ValueType>
void BinRandTree<ValueType>::deltree(Node *node)
{
    if(node)
    {
        deltreetree(node->leftChild());
        deltreetree(node->rightChild());

        delete node;
    }
}

BinRandTree<int> bar()
{
    BinRandTree<int> bt;
    BinRandTree<int> bt2;

    Unirand r(0, 0, 1);

    bt << 0 << 0 << 0;
    bt2 << 1 << 1 << 1;
    bt.print();
```

```
        std::cout << " --- " << std::endl;
        bt2.print();

    }

BinRandTree<int> foo()
{
    return BinRandTree<int>();
}

int main(int argc, char** argv, char ** env)
{
    std::cout << " *** " << std::endl;
    BinRandTree<int> bt2{bar()};
    std::cout << " *** " << std::endl;
    bt2.print();

}
```

Az alábbi program fogja használni a mozgató szemantikát.

```
#include "vedes_Binfa.h"

int main()
{
    ZLWTree<char, '/', '0'> tree1;
    ZLWTree<char, '/', '0'> tree2;

    tree1 = tree2;

    return 0;
}
```

Ha lefuttattuk a programot akkor láthatjuk a kimeneten, hogy milyen részek futottak le a programban, mert az egyszerűség kedvéért a feladatban tárgyalt kódrészletek lefutására bizonyítékként kiírattuk, hogy az adott rész lefutott.

Az első két sor a kimeneten a BT ctor, ez annyit jelent hogy a Binfa konstruktora lefutott. Tehát a tree1 és tree2 fákat létrehozta a program.

```
ZLWTree<char, '/', '0'> tree1;
ZLWTree<char, '/', '0'> tree2;
```

Aztán a BT copy assign sor azt jelenti, hogy amikor a tree1-et egyenlővé tesszük a tree2-vel,

```
tree1 = tree2;
```

akkor lefut a másoló értékkopíálás és létrehoz egy átmeneti fát amibe lementi a tree2-t majd később abból rakja a tree1-be. A feladat arra öszponosít hogy a mozgatást az értékkopíálásra alapozzuk, vagyis meghívjuk a függvényben a másoló konstruktort.

```
BinRandTree & operator=(const BinRandTree & old) {
    std::cout << "BT copy assign" << std::endl;

    BinRandTree tmp{old};
    std::swap(*this, tmp);
    return *this;
}
```

Majd a BT copy ctor sor, a másoló konstruktor lefutását mutatja.

```
BinRandTree(const BinRandTree & old) {
    std::cout << "BT copy ctor" << std::endl;

    root = cp(old.root, old.treep);

}
```

Miután lemásolta, haladunk tovább és a mozgató konstruktor is meghívódik, és a mozgató konstruktor a mozgató értékkopíálásra van alapozva, látszik lent hogy két objektum van egyenlővé téve.

```
BinRandTree(BinRandTree && old) {
    std::cout << "BT move ctor" << std::endl;

    root = nullptr;
    *this = std::move(old);
}
```

A BT move assign, azaz a mozgató értékkopíálás meghívódik, mivel van két értékkopíálás és még egy az átmeneti, így fut le háromszor.

```
BinRandTree & operator=(BinRandTree && old) {
    std::cout << "BT move assign" << std::endl;

    std::swap(old.root, root);
    std::swap(old.treep, treep);

    return *this;
}
```

És végül a BT dtor sorok, azt mutatják hogy a destrukturátor is lefutott, vagyis töröltünk a memóriából minden, így megakadályozva a memória folyást vagy memóriaszivárgást.

```
~BinRandTree() {
    std::cout << "BT dtor" << std::endl;
    deltree(root);
}
```

## 6.7. Vörös Pipacs Pokol/5x5x5 ObservationFromGrid

Megoldás forrása: <https://github.com/nbatfai/RedFlowerHell>

Ebben a feladatban kiszélesítjük egy kicsit Steve látóterét, azaz már nem 3x3x3-mas hanem 5x5x5-ös tömbbe helyezzük Stevet amelyben minden érzékelni fog. A programkódban megírt 3x3x3-mas helyére 5x5x5-öst írunk, majd az xml fájlban a szélső értékeket állítsuk át a következőképpen:

- minimumok:  $x = -2, y = -2, z = -2,$
- maximumok:  $x = 2, y = 2, z = 2.$

A programkód megtalálható a repómban a következő linken: [erzekelni\\_5x5x5\\_ben](#)

## 7. fejezet

# Helló, Conway!

### 7.1. Hangyszimulációk

Tutorált: Talinger Mark Imre

Írj Qt C++-ban egy hangyszimulációs programot, a forrásaidról utólag reverse engineering jelleggel készíts UML osztálydiagramot is!

Egy QT programban fogjuk szimulálni a hangyák mozgását.

A hangyák egymás közötti kommunikáció révén tájékozódnak, ezt feromonokkal érik el amit maguk után hagynak. A hangyák a feromon csíkok erősségének megfelelően választnak útvonalat maguknak. A lényege, hogy ez által megjelöljék az útvonalukat és ezt jelezve a többi hangya felé. Ahol erősebb a feromon az azt jelenti, hogy a hangyák nagyon kedvelik azt a helyet ezért egyre többen járnak arra.

Ebben a szimulációban a hangyák közötti kommunikációt fogjuk reprodukálni.

Megoldás videó: <https://bhaxor.blog.hu/2018/10/10/myrmecologist>

Megoldás forrása: [https://gitlab.com/nbatfai/bhax/tree/master/attention\\_raising/Myrmecologist](https://gitlab.com/nbatfai/bhax/tree/master/attention_raising/Myrmecologist)

A kód Bátfa Norbert tanárúrtól származik.

```
// BHAX Myrmecologist
//
// Copyright (C) 2019
// Norbert Bátfa, batfai.norbert@inf.unideb.hu
//
// This program is free software: you can redistribute it and/
// or modify
// it under the terms of the GNU General Public License as
// published by
// the Free Software Foundation, either version 3 of the
// License, or
// (at your option) any later version.
//
// This program is distributed in the hope that it will be
// useful,
// but WITHOUT ANY WARRANTY; without even the implied warranty
// of
```

```
// MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See ↵
the
// GNU General Public License for more details.
//
// You should have received a copy of the GNU General Public ↵
License
// along with this program. If not, see <https://www.gnu.org/ ↵
licenses/>.
//
// https://bhaxor.blog.hu/2018/09/26/hangyaszimulaciok
// https://bhaxor.blog.hu/2018/10/10/myrmecologist
//
```

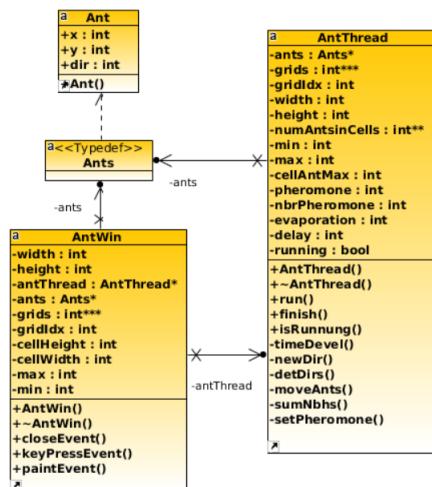
Először is szükségünk van a következő kódrészekre is:

```
ant.h
antwin.h
antthread.h

antwin.cpp
antthread.cpp
main.cpp
```

Ezek a következő linken elérhetők:

Forrás: [https://gitlab.com/nbatfai/bhax/tree/master/attention\\_raising/Myrmecologist](https://gitlab.com/nbatfai/bhax/tree/master/attention_raising/Myrmecologist)



7.1. ábra. Hangya szimuláció

A kód magyarázata pedig a következő:

1. antthread.h:

```
#ifndef ANTTHREAD_H
```

```
#define ANTTHREAD_H

#include <QThread>
#include "ant.h"

class AntThread : public QThread
{
    Q_OBJECT

public:
    AntThread(Ants * ants, int ***grids, int width, int height,
              int delay, int numAnts, int pheromone, int nbrPheromone,
              int evaporation, int min, int max, int cellAntMax);

    ~AntThread();

    void run();
    void finish()
    {
        running = false;
    }

    void pause()
    {
        paused = !paused;
    }

    bool isRunning()
    {
        return running;
    }

private:
    bool running {true};
    bool paused {false};
    Ants* ants;
    int** numAntsinCells;
    int min, max;
    int cellAntMax;
    int pheromone;
    int evaporation;
    int nbrPheromone;
    int ***grids;
    int width;
    int height;
    int gridIdx;
    int delay;

    void timeDevel();
}
```

```
int newDir(int sor, int oszlop, int vsor, int voszlop);
void detDirs(int irany, int& ifrom, int& ito, int& jfrom, ←
    int& jto );
int moveAnts(int **grid, int row, int col, int& retrow, int ←
    & retcol, int);
double sumNbhs(int **grid, int row, int col, int);
void setPheromone(int **grid, int row, int col);

signals:
    void step ( const int &);

};

#endif
```

Ebben a headerben létrehozunk egy osztályt amiben a public részben a QT-s program ablakot fogjuk beállítani, méretezni és szabályozni. A private részben pedig az ablak adatai fogjuk tárolni (feromon, minimum, maximum, szélesség, magasság, fut vagy sem stb.). Aztán alatta a hangya vezérlő, útvonal meghatározó, feromon szint beállító függvények is itt lesznek, valamint a step konstans is.

2. ant.h header:

```
#ifndef ANT_H
#define ANT_H

class Ant
{

public:
    int x;
    int y;
    int dir;

    Ant(int x, int y): x(x), y(y) {

        dir = qrand() % 8;

    }

};

typedef std::vector<Ant> Ants;

#endif
```

Az ant.h headerben létrehozunk egy osztályt amely a hangyák jellem osztálya, itt definiáljuk a hangyák tulajdonságait, a koordinátáit az ablakon belüli elhelyezkedésüket (x,y), valamint az útvonalukat a konstruktorral, ami randomizált a qrand() függvénnyel.

## 3. antwin.h:

```
#ifndef ANTWIN_H
#define ANTWIN_H

#include <QMainWindow>
#include <QPainter>
#include <QString>
#include <QCcloseEvent>
#include "antthread.h"
#include "ant.h"

class AntWin : public QMainWindow
{
    Q_OBJECT

public:
    AntWin(int width = 100, int height = 75,
           int delay = 120, int numAnts = 100,
           int pheromone = 10, int nbhPheromon = 3,
           int evaporation = 2, int cellDef = 1,
           int min = 2, int max = 50,
           int cellAntMax = 4, QWidget *parent = 0);

    AntThread* antThread;

    void closeEvent ( QCcloseEvent *event ) {

        antThread->finish();
        antThread->wait();
        event->accept();
    }

    void keyPressEvent ( QKeyEvent *event )
    {

        if ( event->key() == Qt::Key_P ) {
            antThread->pause();
        } else if ( event->key() == Qt::Key_Q
                    || event->key() == Qt::Key_Escape ) {
            close();
        }
    }

    virtual ~AntWin();
    void paintEvent(QPaintEvent*);
```

private:

```
int ***grids;
```

```
int **grid;
int gridIdx;
int cellWidth;
int cellHeight;
int width;
int height;
int max;
int min;
Ants* ants;

public slots :
void step ( const int &);

};

#endif
```

Ez az utolsó header amelyben használni fogjuk a másik két headert. Ebben a headerben a QT-s ablakot bővítjük ki a kezelhetőséggel, vagyis itt találhatók azok a függvények amelyek leállíthatják az ablakot (closeEvent()), szüneteltethetik (keyPressEvent). Valamint megtaláljuk a private részben ismét az ablak tulajdonságait (szélesség, magasság, min, max stb.). De az Ants vektor példányosítása is itt van.

#### 4. antthread.cpp

```
#include "antthread.h"
#include <QDebug>
#include <cmath>
#include <QDateTime>

AntThread::AntThread ( Ants* ants, int*** grids,
                      int width, int height,
                      int delay, int numAnts,
                      int pheromone, int nbrPheromone,
                      int evaporation,
                      int min, int max, int cellAntMax)
{
    this->ants = ants;
    this->grids = grids;
    this->width = width;
    this->height = height;
    this->delay = delay;
    this->pheromone = pheromone;
    this->evaporation = evaporation;
    this->min = min;
    this->max = max;
    this->cellAntMax = cellAntMax;
    this->nbrPheromone = nbrPheromone;

    numAntsinCells = new int*[height];
    for ( int i=0; i<height; ++i ) {
```

```
    numAntsinCells[i] = new int [width];
}

for ( int i=0; i<height; ++i )
    for ( int j=0; j<width; ++j ) {
        numAntsinCells[i][j] = 0;
    }

qsrand ( QDateTime::currentMSecsSinceEpoch() );

Ant h {0, 0};
for ( int i {0}; i<numAnts; ++i ) {

    h.y = height/2 + qrand() % 40-20;
    h.x = width/2 + qrand() % 40-20;

    ++numAntsinCells[h.y][h.x];

    ants->push_back ( h );
}

gridIdx = 0;
}

double AntThread::sumNbhs ( int **grid, int row, int col, int ←
dir )
{
    double sum = 0.0;

    int ifrom, ito;
    int jfrom, jto;

    detDirs ( dir, ifrom, ito, jfrom, jto );

    for ( int i=ifrom; i<ito; ++i )
        for ( int j=jfrom; j<jto; ++j )

            if ( ! ( ( i==0 ) && ( j==0 ) ) ) {
                int o = col + j;
                if ( o < 0 ) {
                    o = width-1;
                } else if ( o >= width ) {
                    o = 0;
                }

                int s = row + i;
                if ( s < 0 ) {
                    s = height-1;
                } else if ( s >= height ) {
```

```
        s = 0;
    }

    sum += (grid[s][o]+1)*(grid[s][o]+1)*(grid[s][o ↔
        ]+1);

}

return sum;
}

int AntThread::newDir ( int sor, int oszlop, int vsor, int ←
    voszlop )
{
    if ( vsor == 0 && sor == height -1 ) {
        if ( voszlop < oszlop ) {
            return 5;
        } else if ( voszlop > oszlop ) {
            return 3;
        } else {
            return 4;
        }
    } else if ( vsor == height - 1 && sor == 0 ) {
        if ( voszlop < oszlop ) {
            return 7;
        } else if ( voszlop > oszlop ) {
            return 1;
        } else {
            return 0;
        }
    } else if ( voszlop == 0 && oszlop == width - 1 ) {
        if ( vsor < sor ) {
            return 1;
        } else if ( vsor > sor ) {
            return 3;
        } else {
            return 2;
        }
    } else if ( voszlop == width && oszlop == 0 ) {
        if ( vsor < sor ) {
            return 7;
        } else if ( vsor > sor ) {
            return 5;
        } else {
            return 6;
        }
    } else if ( vsor < sor && voszlop < oszlop ) {
        return 7;
    } else if ( vsor < sor && voszlop == oszlop ) {
```

```
        return 0;
    } else if ( vsor < sor && voszlop > oszlop ) {
        return 1;
    }

    else if ( vsor > sor && voszlop < oszlop ) {
        return 5;
    } else if ( vsor > sor && voszlop == oszlop ) {
        return 4;
    } else if ( vsor > sor && voszlop > oszlop ) {
        return 3;
    }

    else if ( vsor == sor && voszlop < oszlop ) {
        return 6;
    } else if ( vsor == sor && voszlop > oszlop ) {
        return 2;
    }

    else { // (vsor == sor && voszlop == oszlop)
        qDebug() << "ZAVAR AZ EROBEN az iranynal";
        return -1;
    }

}

void AntThread::detDirs ( int dir, int& ifrom, int& ito, int& ←
    jfrom, int& jto )
{
    switch ( dir ) {
    case 0:
        ifrom = -1;
        ito = 0;
        jfrom = -1;
        jto = 2;
        break;
    case 1:
        ifrom = -1;
        ito = 1;
        jfrom = 0;
        jto = 2;
        break;
    case 2:
        ifrom = -1;
        ito = 2;
        jfrom = 1;
        jto = 2;
        break;
    }
```

```
        case 3:
            ifrom =0;
            ito = 2;
            jfrom = 0;
            jto = 2;
            break;
        case 4:
            ifrom = 1;
            ito = 2;
            jfrom = -1;
            jto = 2;
            break;
        case 5:
            ifrom = 0;
            ito = 2;
            jfrom = -1;
            jto = 1;
            break;
        case 6:
            ifrom = -1;
            ito = 2;
            jfrom = -1;
            jto = 0;
            break;
        case 7:
            ifrom = -1;
            ito = 1;
            jfrom = -1;
            jto = 1;
            break;
    }

}

int AntThread::moveAnts ( int **racs,
                         int sor, int oszlop,
                         int& vsor, int& voszlop, int dir )
{
    int y = sor;
    int x = oszlop;

    int ifrom, ito;
    int jfrom, jto;

    detDirs ( dir, ifrom, ito, jfrom, jto );

    double osszes = sumNbhs ( racs, sor, oszlop, dir );
    double random = ( double ) ( qrand() %1000000 ) / ( double )
```

```
    ) 1000000.0;
double gvalseg = 0.0;

for ( int i=ifrom; i<ipto; ++i )
    for ( int j=jfrom; j<jto; ++j )
        if ( ! ( ( i==0 ) && ( j==0 ) ) )
    {
        int o = oszlop + j;
        if ( o < 0 ) {
            o = width-1;
        } else if ( o >= width ) {
            o = 0;
        }

        int s = sor + i;
        if ( s < 0 ) {
            s = height-1;
        } else if ( s >= height ) {
            s = 0;
        }

        //double kedvezo = std::sqrt((double)(racs[s][o] ←
        //] + 2)); // (racs[s][o] + 2) * (racs[s][o] + 2);
        //double kedvezo = (racs[s][o] + b) * (racs[s][o] + b ←
        //);
        //double kedvezo = ( racs[s][o] + 1 );
        double kedvezo = (racs[s][o] + 1) * (racs[s][o] + 1) ←
            * (racs[s][o] + 1);

        double valseg = kedvezo/osszes;
        gvalseg += valseg;

        if ( gvalseg >= random ) {

            vsor = s;
            voszlop = o;

            return newDir ( sor, oszlop, vsor, voszlop ←
                );
        }
    }

qDebug() << "ZAVAR AZ EROBEN a lepesnel";
vsor = y;
voszlop = x;

return dir;
```



```
for ( int i=-1; i<2; ++i )
    for ( int j=-1; j<2; ++j )
        if ( ! ( ( i==0 ) && ( j==0 ) ) )
    {
        int o = oszlop + j;
        {
            if ( o < 0 ) {
                o = width-1;
            } else if ( o >= width ) {
                o = 0;
            }
        }
        int s = sor + i;
        {
            if ( s < 0 ) {
                s = height-1;
            } else if ( s >= height ) {
                s = 0;
            }
        }

        if ( racs[s][o] + nbrPheromone <= max ) {
            racs[s][o] += nbrPheromone;
        } else {
            racs[s][o] = max;
        }
    }

    if ( racs[sor][oszlop] + pheromone <= max ) {
        racs[sor][oszlop] += pheromone;
    } else {
        racs[sor][oszlop] = max;
    }
}

void AntThread::run()
{
    running = true;
    while ( running ) {

        QThread::msleep ( delay );

        if ( !paused ) {
            timeDevel();
        }
    }
}
```

```
        emit step ( gridIdx );  
  
    }  
  
}  
  
AntThread::~AntThread()  
{  
    for ( int i=0; i<height; ; ++i ) {  
        delete [] numAntsinCells[i];  
    }  
  
    delete [] numAntsinCells;  
}
```

Az antthread.cpp nevű fájlban az antthread.h header fileban létrehozott függvényeket, osztályt stb. használjuk, ezeket dolgozzuk ki.

A függvények amelyek nagyon fontosak:

sumNbhs() függvény: Statisztikát végző rész, és összegzés.

newDir() függvény: Létrehoz egy új pontot az ablakban. Azaz újabb hangya felbukkanása.

detDirs() függvény: Amelyben egy switch segítségevel döntjük el a hangya tájékozódási pontjának helyzetét.

moveAnts() függvény: A hangyák hogyan fognak mozogni.

timeDevel() függvény: Az idő műlásának megfelelően minden változások mennek végebe az ablakban.

setPheromone() függvény: A feromon szint nyilvántartása, változtatása.

run() függvény: A program futtatásért felelős.

destruktor: Ami törli a cellákból a hangyákat, amiket abba tárolunk, hogy meg tudjuk jeleníteni őket.

5. antwin.cpp:

```
#include "antwin.h"  
#include <QDebug>  
  
AntWin::AntWin ( int width, int height, int delay, int numAnts,  
                 int pheromone, int nbhPheromon, int evaporation ←  
                 , int cellDef,  
                 int min, int max, int cellAntMax, QWidget * ←  
                           parent ) : QMainWindow ( parent )  
{  
    setWindowTitle ( "Ant Simulation" );  
  
    this->width = width;  
    this->height = height;  
    this->max = max;  
    this->min = min;
```

```
cellWidth = 6;
cellHeight = 6;

setFixedSize ( QSize ( width*cellWidth, height*cellHeight ) ←
);

grids = new int**[2];
grids[0] = new int*[height];
for ( int i=0; i<height; ++i ) {
    grids[0][i] = new int [width];
}
grids[1] = new int*[height];
for ( int i=0; i<height; ++i ) {
    grids[1][i] = new int [width];
}

gridIdx = 0;
grid = grids[gridIdx];

for ( int i=0; i<height; ++i )
    for ( int j=0; j<width; ++j ) {
        grid[i][j] = cellDef;
    }

ants = new Ants();

antThread = new AntThread ( ants, grids, width, height, ←
    delay, numAnts, pheromone,
    nbhPheromon, evaporation, min, ←
    max, cellAntMax);

connect ( antThread, SIGNAL ( step ( int ) ),
    this, SLOT ( step ( int ) ) );

antThread->start();

}

void AntWin::paintEvent ( QPaintEvent* )
{
    QPainter qpainter ( this );

    grid = grids[gridIdx];

    for ( int i=0; i<height; ++i ) {
        for ( int j=0; j<width; ++j ) {

            double rel = 255.0/max;
```

```
qpainter.fillRect ( j*cellWidth, i*cellHeight,
                    cellWidth, cellHeight,
                    QColor ( 255 - grid[i][j]*rel,
                            255,
                            255 - grid[i][j]*rel ) ) ←
                    ;

if ( grid[i][j] != min )
{
    qpainter.setPen (
        QPen (
            QColor ( 255 - grid[i][j]*rel,
                      255 - grid[i][j]*rel, 255 ),
            1 )
    );

    qpainter.drawRect ( j*cellWidth, i*cellHeight,
                        cellWidth, cellHeight );
}

qpainter.setPen (
    QPen (
        QColor ( 0,0,0 ),
        1 )
);

qpainter.drawRect ( j*cellWidth, i*cellHeight,
                    cellWidth, cellHeight );

}

}

for ( auto h: *ants) {
    qpainter.setPen ( QPen ( Qt::black, 1 ) );

    qpainter.drawRect ( h.x*cellWidth+1, h.y*cellHeight+1,
                        cellWidth-2, cellHeight-2 );

}

qpainter.end();
}

AntWin::~AntWin()
{
    delete antThread;

    for ( int i=0; i<height; ++i ) {
```

```
        delete[] grids[0][i];
        delete[] grids[1][i];
    }

    delete[] grids[0];
    delete[] grids[1];
    delete[] grids;

    delete ants;
}

void AntWin::step ( const int &gridIdx )
{
    this->gridIdx = gridIdx;
    update();
}
```

Ez a file a megjelenítésért lesz felelős. Itt láthatjuk azt a függvényt amely a hangyákat jeleníti meg (antWin), és a paintEvent() függvény amely a feromon csíkokat jeleníti meg a cellákban feromon erőssége szerint. A destruktur pedig elvégzi a piszkos munkát, vagyis törli a cellákat és a step() függvény frissíti a lépéseket.

## 6. main.cpp:

```
#include <QApplication>
#include <QDesktopWidget>
#include <QDebug>
#include <QDateTime>
#include <QCommandLineOption>
#include <QCommandLineParser>

#include "antwin.h"

/*
*
* ./myrmecologist -w 250 -m 150 -n 400 -t 10 -p 5 -f 80 -d 0 -a ←
*   255 -i 3 -s 3 -c 22
*
*/
int main ( int argc, char *argv[] )
{
    QApplication a ( argc, argv );

    QCommandLineOption szeles_opt ( { "w", "szelesseg" }, " ←
        Oszlopok (cellakban) szama.", "szelesseg", "200" );
    QCommandLineOption magas_opt ( { "m", "magassag" }, "Sorok ( ←
        cellakban) szama.", "magassag", "150" );
    QCommandLineOption hangyaszam_opt ( { "n", "hangyaszam" }, " ←
```

```
Hangyak szama.", "hangyaszam", "100" );
QCommandLineOption sebesseg_opt ( {"t","sebesseg"}, "2 ←
    lepes kozotti ido (millisec-ben).", "sebesseg", "100" );
QCommandLineOption parolgas_opt ( {"p","parolgas"}, "A ←
    parolgas erteke.", "parolgas", "8" );
QCommandLineOption feromon_opt ( {"f","feromon"}, "A ←
    hagyott nyom erteke.", "feromon", "11" );
QCommandLineOption szomszed_opt ( {"s","szomszed"}, "A ←
    hagyott nyom erteke a szomszedokban.", "szomszed", "3" ) ←
;
QCommandLineOption alapertek_opt ( {"d","alapertek"}, " ←
    Indulo ertekek a cellakban.", "alapertek", "1" );
QCommandLineOption maxcella_opt ( {"a","maxcella"}, "Cella ←
    max erteke.", "maxcella", "50" );
QCommandLineOption mincella_opt ( {"i","mincella"}, "Cella ←
    min erteke.", "mincella", "2" );
QCommandLineOption cellamerete_opt ( {"c","cellameret"}, " ←
    Hany hangya fer egy cellaba.", "cellameret", "4" );
QCommandLineParser parser;

parser.addHelpOption();
parser.addVersionOption();
parser.addOption ( szeles_opt );
parser.addOption ( magas_opt );
parser.addOption ( hangyaszam_opt );
parser.addOption ( sebesseg_opt );
parser.addOption ( parolgas_opt );
parser.addOption ( feromon_opt );
parser.addOption ( szomszed_opt );
parser.addOption ( alapertek_opt );
parser.addOption ( maxcella_opt );
parser.addOption ( mincella_opt );
parser.addOption ( cellamerete_opt );

parser.process ( a );

QString szeles = parser.value ( szeles_opt );
QString magas = parser.value ( magas_opt );
QString n = parser.value ( hangyaszam_opt );
QString t = parser.value ( sebesseg_opt );
QString parolgas = parser.value ( parolgas_opt );
QString feromon = parser.value ( feromon_opt );
QString szomszed = parser.value ( szomszed_opt );
QString alapertek = parser.value ( alapertek_opt );
QString maxcella = parser.value ( maxcella_opt );
QString mincella = parser.value ( mincella_opt );
QString cellameret = parser.value ( cellamerete_opt );

qsrand ( QDateTime::currentMSecsSinceEpoch() );
```

```
    AntWin w ( szeles.toInt(), magas.toInt(), t.toInt(), n. ←
       ToInt(), feromon.toInt(), szomszed.toInt(), parolgas. ←
       ToInt(),
        alapertek.toInt(), mincella.toInt(), maxcella. ←
       ToInt(),
        cellameret.toInt() ) ;

    w.show();

    return a.exec();
}
```

A main (fő) függvényben QT-s parancsokat használjuk és definiáljuk, valamint a korábban beszélt antwin.h headert is megadjuk. Itt tulajdonképpen a futtatáskor megjelenő beállítható paraméterezést láthatjuk.

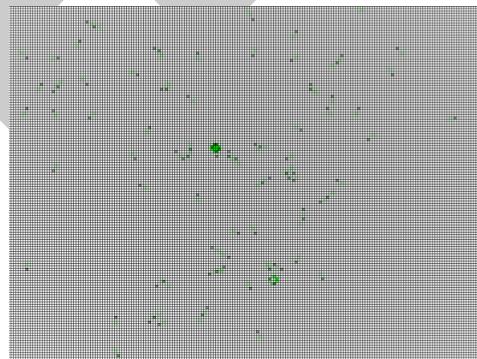
```
QT += widgets

TEMPLATE = app
TARGET = myrmecologist
INCLUDEPATH += .

HEADERS += ant.h antwin.h antthread.h
SOURCES += main.cpp antwin.cpp antthread.cpp
```

Ez a rész gyűjti össze a header és cpp fájlokat ami a szimuláció működéséhez szükségesek.

A programunkat fordítani a **qmake myrmecologist.pro** és a **make** parancsokkal tudjuk, futtatni pedig a **./myrmecologist** parancsal. Ez egy default értékekkel történő futtatás, de meg lehet adni paramétereket is: **./myrmecologist -w 250 -m 150 -n 400 -t 10 -p 5 -f 80 -d 0 -a 255 -i 3 -s 3 -c 22** parancsal, ami a sima futtatásnál látványosabb.



7.2. ábra. Hangya szimuláció

## 7.2. Java életjáték

Írd meg Java-ban a John Horton Conway-féle életjátékot, valósítsa meg a sikló-kilövőt!

Az életjátékot azaz sejtautomatákat először Naumen János vetette fel, a gép önreprodukciójának matematikai modellalkotást tartalmazta. A legismertebb modell a John Horton Conway-féle életjáték.

A "játék" egy négyzetrácsos mezőn zajlik amin mozognak a sejtek. A sejtek "élete" szabályokhoz van kötve. Megvan adva hogy mi a feltétele egy sejt kialakulásáak, életbenmadásának vagy elpusztulásának. Conway erre 3 feltételt szabott meg:

1.szabály (túlélés): Egy sejt csak úgy éli túl, ha kettő vagy három szomszédja van.

2.szabály (elpusztulás): Egy sejt elpusztul, ha kettőnél kevesebb szomszédja van, ezt az elszigetelődés, vagy ha háromnál több szomszédja van, ez a túlnépesedés.

3.szabály (születés): Egy sejt születik, ha egy cellának a körzetében 3 sejt található.

Ezen a 3 szabály meghatározásával kapunk egy önműködő sejtautomatát. Beleszólásunk csak kezdetben van, utánna a szabyályok szerint önállóan működik a program. Mi most külön a sikló-kilövőt fogjuk vizsgálni. Hogy ezt elérjük, rögzítenünk kell adott cellákban sejteket, így létre jön egy "sikló ágyú", ez időközönként "siklókat" fog lőni.

Megoldás forrása: <https://www.tankonyvtar.hu/hu/tartalom/tkt/javat-tanitok-javat/apb.html?fbclid=IwAR0GcQ7v>

```
public class Sejtautomata extends java.awt.Frame implements Runnable
{
    public static final boolean ÉLŐ = true;
    public static final boolean HALOTT = false;
    protected boolean[][][] rácsok = new boolean [2][][][];
    protected boolean [][] rács;
    protected int rácsIndex = 0;
    protected int cellaSzélesség = 20;
    protected int cellaMagasság = 20;
    protected int szélesség = 20;
    protected int magasság = 10;
    protected int várakozás = 1000;
    private java.awt.Robot robot;
    private boolean pillanatfelvétel = false;
    private static int pillanatfelvételszámláló = 0;

    public Sejtautomata(int szélesség, int magasság)
    {
        this.szélesség = szélesség;
        this.magasság = magasság;
        rácsok[0] = new boolean[magasság][szélesség];
        rácsok[1] = new boolean[magasság][szélesség];
        rácsIndex = 0;
        rács = rácsok[rácsIndex];

        for(int i=0; i<rács.length; ++i)
            for(int j=0; j<rács[0].length; ++j)
                rács[i][j] = HALOTT;

        siklóKilövő(rács, 5, 60);
    }
}
```

```
addWindowListener(new java.awt.event.WindowAdapter()
{
    public void windowClosing(java.awt.event.WindowEvent e)
    {
        setVisible(false); System.exit(0);
    }
});

addKeyListener(new java.awt.event.KeyAdapter()
{
    public void keyPressed(java.awt.event.KeyEvent e)
    {
        if(e.getKeyCode() == java.awt.event.KeyEvent.VK_K)
        {
            cellaSzélesség /= 2;
            cellaMagasság /= 2;
            setSize(Sejtautomata.this.szélesség* ←
                    cellaSzélesség,
                    Sejtautomata.this.magasság* ←
                    cellaMagasság);
            validate();
        }
        else if(e.getKeyCode() == java.awt.event.KeyEvent.VK_N)
        {
            cellaSzélesség *= 2;
            cellaMagasság *= 2;
            setSize(Sejtautomata.this.szélesség* ←
                    cellaSzélesség,
                    Sejtautomata.this.magasság* ←
                    cellaMagasság);
            validate();
        }
        else if(e.getKeyCode() == java.awt.event.KeyEvent.VK_S)
            pillanatfelvétel = !pillanatfelvétel;

        else if(e.getKeyCode() == java.awt.event.KeyEvent.VK_G)
            várakozás /= 2;

        else if(e.getKeyCode() == java.awt.event.KeyEvent.VK_L)
            várakozás *= 2;

        repaint();
    }
});
});
```

```
addMouseListener(new java.awt.event.MouseAdapter()
{
    int x = m.getX()/cellaSzélesség;
    int y = m.getY()/cellaMagasság;
    rácsok[rácsIndex][y][x] = !rácsok[rácsIndex][y][x];
    repaint();
}
));

addMouseMotionListener(new java.awt.event.MouseMotionAdapter()
{
    // Vonszolással jelöljük ki a négyzetet:
    public void mouseDragged(java.awt.event.MouseEvent m)
    {
        int x = m.getX()/cellaSzélesség;
        int y = m.getY()/cellaMagasság;
        rácsok[rácsIndex][y][x] = ÉLŐ;
        repaint();
    }
});

cellaSzélesség = 10;
cellaMagasság = 10;

try
{
    robot = new java.awt.Robot( java.awt.GraphicsEnvironment. ←
        getLocalGraphicsEnvironment(). getDefaultScreenDevice() ) ←
        ;
}
catch(java.awt.AWTException e)
{
    e.printStackTrace();
}

setTitle("Sejtautomata");
setResizable(false);
setSize(szélesség*cellaSzélesség, magasság*cellaMagasság);
setVisible(true);
new Thread(this).start();
}

public void paint(java.awt.Graphics g)
{
    boolean [][] rács = rácsok[rácsIndex];
    for(int i=0; i<rács.length; ++i)
    {
        // végig lépked a sorokon
        for(int j=0; j<rács[0].length; ++j)
```

```
{  
    // s az oszlopok  
    if(rács[i][j] == ÉLŐ)  
        g.setColor(java.awt.Color.BLACK);  
    else  
        g.setColor(java.awt.Color.WHITE);  
  
    g.fillRect(j*cellaSzélesség, i*cellaMagasság, ←  
              cellaSzélesség, cellaMagasság);  
    g.setColor(java.awt.Color.LIGHT_GRAY);  
    g.drawRect(j*cellaSzélesség, i*cellaMagasság, ←  
              cellaSzélesség, cellaMagasság);  
}  
}  
if(pillanatfelvétel)  
{  
    pillanatfelvétel = false;  
    pillanatfelvétel(robot.createScreenCapture  
                      (new java.awt.Rectangle  
                      (getLocation().x, getLocation().y,  
                      szélesség*cellaSzélesség,  
                      magasság*cellaMagasság)  
                      ))  
};  
}  
}  
public int szomszédokSzáma(boolean [][] rács, int sor, int ←  
                           oszlop, boolean állapot)  
{  
    int állapotúSzomszéd = 0;  
  
    for(int i=-1; i<2; ++i)  
        for(int j=-1; j<2; ++j)  
            if(!((i==0) && (j==0)))  
            {  
                int o = oszlop + j;  
                if(o < 0)  
                    o = szélesség-1;  
                else if(o >= szélesség)  
                    o = 0;  
  
                int s = sor + i;  
  
                if(s < 0)  
                    s = magasság-1;  
                else if(s >= magasság)  
                    s = 0;  
  
                if(rács[s][o] == állapot)  
                    ++állapotúSzomszéd;  
            }  
    }  
}
```

```
        }
        return állapotúSzomszéd;
    }

    public void idő_Fejlődés() {
boolean [][] rácsElőtte = rácsok[rácsIndex];
boolean [][] rácsUtána = rácsok[(rácsIndex+1)%2];
for(int i=0; i<rácsElőtte.length; ++i)
{
    // sorok for(int j=0; j<rácsElőtte[0].length; ++j)
    {
        // oszlopok
        int élők = szomszédomszáma(rácsElőtte, i, j, ÉLŐ);
        if(rácsElőtte[i][j] == ÉLŐ)
        {
            if(élők==2 || élők==3)
                rácsUtána[i][j] = ÉLŐ;
            else
                rácsUtána[i][j] = HALOTT;
        } else {
            if(élők==3)
                rácsUtána[i][j] = ÉLŐ;
            else
                rácsUtána[i][j] = HALOTT;
        }
    }
    rácsIndex = (rácsIndex+1)%2;
}
public void run() {
while(true) {
    try {
        Thread.sleep(várakozás);
    } catch (InterruptedException e) {}

    időFejlődés();
    repaint();
}
}

public void sikló(boolean [][] rács, int x, int y)
{
    rács[y+ 0][x+ 2] = ÉLŐ;
    rács[y+ 1][x+ 1] = ÉLŐ;
    rács[y+ 2][x+ 1] = ÉLŐ;
    rács[y+ 2][x+ 2] = ÉLŐ;
    rács[y+ 2][x+ 3] = ÉLŐ;
}
public void siklóKilövő(boolean [][] rács, int x, int y)
{
    rács[y+ 6][x+ 0] = ÉLŐ;
```

rács[y+ 6][x+ 1] = ÉLŐ;

rács[y+ 7][x+ 0] = ÉLŐ;  
rács[y+ 7][x+ 1] = ÉLŐ;

rács[y+ 3][x+ 13] = ÉLŐ;

rács[y+ 4][x+ 12] = ÉLŐ;  
rács[y+ 4][x+ 14] = ÉLŐ;

rács[y+ 5][x+ 11] = ÉLŐ;  
rács[y+ 5][x+ 15] = ÉLŐ;  
rács[y+ 5][x+ 16] = ÉLŐ;  
rács[y+ 5][x+ 25] = ÉLŐ;

rács[y+ 6][x+ 11] = ÉLŐ;  
rács[y+ 6][x+ 15] = ÉLŐ;  
rács[y+ 6][x+ 16] = ÉLŐ;  
rács[y+ 6][x+ 22] = ÉLŐ;  
rács[y+ 6][x+ 23] = ÉLŐ;  
rács[y+ 6][x+ 24] = ÉLŐ;  
rács[y+ 6][x+ 25] = ÉLŐ;

rács[y+ 7][x+ 11] = ÉLŐ;  
rács[y+ 7][x+ 15] = ÉLŐ;  
rács[y+ 7][x+ 16] = ÉLŐ;  
rács[y+ 7][x+ 21] = ÉLŐ;  
rács[y+ 7][x+ 22] = ÉLŐ;  
rács[y+ 7][x+ 23] = ÉLŐ;  
rács[y+ 7][x+ 24] = ÉLŐ;

rács[y+ 8][x+ 12] = ÉLŐ;  
rács[y+ 8][x+ 14] = ÉLŐ;  
rács[y+ 8][x+ 21] = ÉLŐ;  
rács[y+ 8][x+ 24] = ÉLŐ;  
rács[y+ 8][x+ 34] = ÉLŐ;  
rács[y+ 8][x+ 35] = ÉLŐ;

rács[y+ 9][x+ 13] = ÉLŐ;  
rács[y+ 9][x+ 21] = ÉLŐ;  
rács[y+ 9][x+ 22] = ÉLŐ;  
rács[y+ 9][x+ 23] = ÉLŐ;  
rács[y+ 9][x+ 24] = ÉLŐ;  
rács[y+ 9][x+ 34] = ÉLŐ;  
rács[y+ 9][x+ 35] = ÉLŐ;

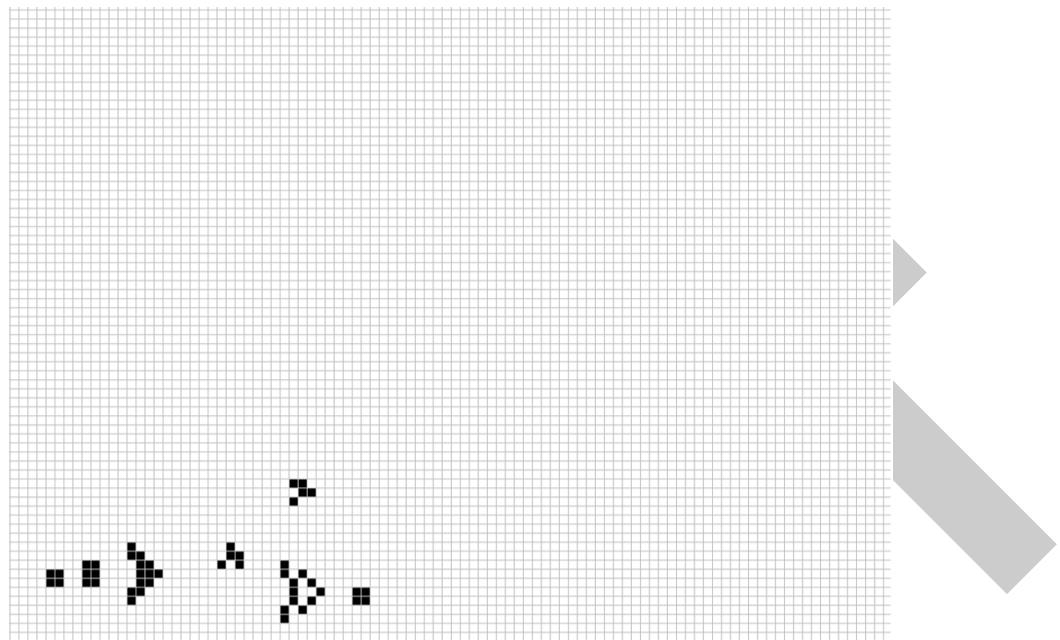
rács[y+ 10][x+ 22] = ÉLŐ;  
rács[y+ 10][x+ 23] = ÉLŐ;  
rács[y+ 10][x+ 24] = ÉLŐ;  
rács[y+ 10][x+ 25] = ÉLŐ;

```
rács[y+ 11][x+ 25] = ÉLŐ;
}

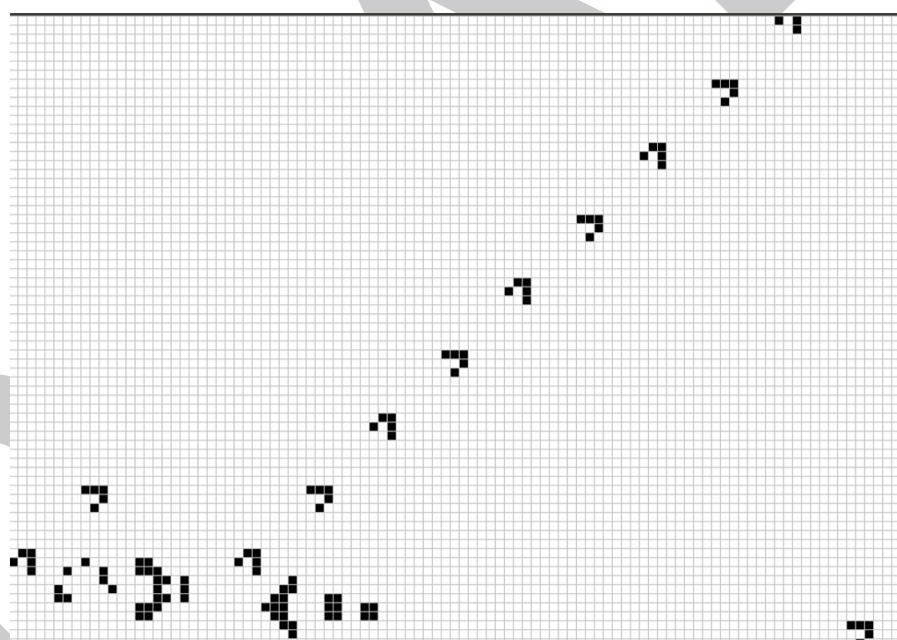
public void pillanatfelvétel(java.awt.image.BufferedImage felvetel)
{
    // A pillanatfelvétel kép fájlneve
    StringBuffer sb = new StringBuffer();
    sb = sb.delete(0, sb.length());
    sb.append("sejtautomata");
    sb.append(++pillanatfelvételSzámláló);
    sb.append(".png"); // png formátumú képet mentünk

    try
    {
        javax.imageio.ImageIO.write(felvetel, "png", new java.io.File(sb.toString()));
    } catch(java.io.IOException e) {
        e.printStackTrace();
    }
}
public void update(java.awt.Graphics g)
{
    paint(g);
}
public static void main(String[] args)
{
    new Sejtautomata(100, 75);
}
}
```





7.3. ábra. Életjáték



7.4. ábra. Életjáték

A program elején megadjuk, hogy egy sejt lehet élő vagy halott. A feladatban 2 rácsfélét használunk, az egyik rács a sejt állapotát fogja tárolni míg a második az egy másdoperel későbbi tulajdonságait.

Meghatározzuk az aktuális rácsot a rácsIndex-el, utánna pedig egy cella magasságát és szélességét, ezt követően hány cellából álljon a "játék". A következő hogy a az állapotok között mennyi idő teljen el.

A függvények közül az első függvény megkapja a méreteket és létrehozza az ablakot. Itt készíti el a 2 rácsot is és az indexet is elindítja. Kezdetben minden rács HALOTT. Ezen belül lesz meghívva a siklólövő aminek a kód végén minden kordinátája megvan adva. Vannak billentyűről beérkező parancsaink is, különböző feladatokkal ellátva pl a "g" betűvel, a ké állapot közötti időt csökkentjük. Ugyan így vannak az egérrel történő információk feldolgozására szolgáló függvények, külön kattintásra és mozgatásra. Külön tudunk készíteni pillanatfelvételt az aktuális állapotról az "s" gomb segítségével.

A programban a sejttér rajzolását a paint() függvénytel végezzük. A szomszédokSzáma() függvényben vizsgáljuk a szabályokat és a szerint történik a sejtek viselkedése.

### 7.3. Qt C++ életjáték

Most Qt C++-ban!

Ez a feladat ugyan az mint az előző, a különbség itt a program nyelvben van, ez a kód QT C++-ban van írva.

A programhoz szükségesek az alábbi forráskódok.

Megoldás forrása: <https://sourceforge.net/p/udprog/code/ci/master/tree/source/labor/Qt/Sejtauto/>

```
#include <QApplication>
#include "sejtablak.h"
#include <QDesktopWidget>

int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    SejtAblak w(100, 75);
    w.show();

    return a.exec();
}
```

A forrás fájljainak a Sejtablak.cpp, sejtszal.h, sejtszal.cpp és a sejtablak.h.

A sejtablak.h és sejtablak.cpp tartalmazza a függvényeket amivel majd a kirajzolás fog történni és ebben van a sikló lövés is, mint a java kódban megírtanál, külön minden egyes cellát megadunk amiben sejt van.

A szejtszal.h és sejtszal.c pedig az életjátékhoz szükséges szabályokat. Ezen belül vannak a függvények melyek az adott állapotokat vizsgálják és a szabályok szerint alakítják a programot.

### 7.4. BrainB Benchmark

A benchmark egy elemzés, tesztfeladat. Egy bizonyos tesztet végez el és azt az elért pontszám alapján összehasonlítja a tesztet elvégzők között és megtudhatjuk, hogy ki teljesített a legjobban és egymáshoz is tudjuk vizsgálni őket.

A BrainB egy kutatás céljával elkészült program, amely felméri az esport játékosok koncentrációs képességét. Nem csak esport játékosokra van kifejlesztve a program, hanem akik szeretik a videójátékokat, és egy rövid felmérést szeretnének kapni a saját koncentrációs képességükről.

Ebben a programban egy Samu Entropy nevű köröcskén kell lenyomva tartani a cursort 10 percig. A koncentráció méréséhez azt várja el a program, hogy a lenyomott cursorralkövessük az egyébként mozgolódó Samu Entropyt. Minél több ideig sikerül a köröcskében maradni annál gyorsabban nő a pontszámunk, és kezdenek megjelenni más Entropy köröcskék is, továbbá Samu is fürgébben fog mozogni. Amikor elveszítjük a Samut a sok Entropy között, akkor lelassul és a pontunk is csökkenni kezd. Az eredményünkről a 10 perc lejárta után kapunk információt.

Forrás: <https://github.com/nbatfai/esport-talent-search>

A programhoz minden forrás megtalálható a fenti linken amely Bátfai Norbert tanárúrtól származik.

A következő programcsipet a BraintBTheard.h headerből való:

```
class Hero
{
public:
    int x;
    int y;
    int color;
    int agility;
    intconds {0};
    std::string name;

    Hero ( int x=0, int y=0, int color=0, int agility=1, std::string name ="Samu Entropy" ) :
        x ( x ), y ( y ), color ( color ), agility ( agility ), name ( name )
    {}
    ~Hero () {}

    void move ( int maxx, int maxy, int env ) {

        int newx = x+ ( ( ( double ) agility*1.0 ) * ( double ) ( std::rand() / ( RAND_MAX+1.0 ) )-agility/2 ) ;
        if ( newx-env > 0 && newx+env < maxx ) {
            x = newx;
        }
        int newy = y+ ( ( ( double ) agility*1.0 ) * ( double ) ( std::rand() / ( RAND_MAX+1.0 ) )-agility/2 );
        if ( newy-env > 0 && newy+env < maxy ) {
            y = newy;
        }

    }

};
```

Itt láthatunk egy Hero osztályt amiben tulajdonképpen létrehozzuk az Entropynkat. Továbbá létrehozzuk a tulajdonságait is: az elhelyezkedését, a nevét, a színét, a nagyságát stb.. Majd látunk még egy move() függvényt amely a mozgásért felelős.

A következő a BraintBTheard.cpp:

```
#include "BrainBThread.h"

BrainBThread::BrainBThread ( int w, int h )
{
    dispShift = heroRectSize+heroRectSize/2;
    this->w = w - 3 * heroRectSize;
    this->h = h - 3 * heroRectSize;

    std::srand ( std::time ( 0 ) );

    Hero me ( this->w / 2 + 200.0 * std::rand() / ( ←
        RAND_MAX + 1.0 ) - 100,
        this->h / 2 + 200.0 * std::rand() / ( ←
        RAND_MAX + 1.0 ) - 100, 255.0 * std::←
        rand() / ( RAND_MAX + 1.0 ), 9 );

    Hero other1 ( this->w / 2 + 200.0 * std::rand() / ( ←
        RAND_MAX + 1.0 ) - 100,
        this->h / 2 + 200.0 * std::rand() / ( ←
        RAND_MAX + 1.0 ) - 100, 255.0 * std::←
        rand() / ( RAND_MAX + 1.0 ), 5, "Norbi ←
        Entropy" );

    Hero other2 ( this->w / 2 + 200.0 * std::rand() / ( ←
        RAND_MAX + 1.0 ) - 100,
        this->h / 2 + 200.0 * std::rand() / ( ←
        RAND_MAX + 1.0 ) - 100, 255.0 * std::←
        rand() / ( RAND_MAX + 1.0 ), 3, "Greta ←
        Entropy" );

    Hero other4 ( this->w / 2 + 200.0 * std::rand() / ( ←
        RAND_MAX + 1.0 ) - 100,
        this->h / 2 + 200.0 * std::rand() / ( ←
        RAND_MAX + 1.0 ) - 100, 255.0 * std::←
        rand() / ( RAND_MAX + 1.0 ), 5, "Nandi ←
        Entropy" );

    Hero other5 ( this->w / 2 + 200.0 * std::rand() / ( ←
        RAND_MAX + 1.0 ) - 100,
        this->h / 2 + 200.0 * std::rand() / ( ←
        RAND_MAX + 1.0 ) - 100, 255.0 * std::←
        rand() / ( RAND_MAX + 1.0 ), 7, "Matyi ←
        Entropy" );

    heroes.push_back ( me );
    heroes.push_back ( other1 );
    heroes.push_back ( other2 );
    heroes.push_back ( other4 );
    heroes.push_back ( other5 );
```

```
}

BrainBThread::~BrainBThread() {} //Destruktor

void BrainBThread::run()
{
    while ( time < endTime ) {
        QThread::msleep ( delay );
        if ( !paused ) {
            ++time;
            devel();
        }
        draw();
    }
    emit endAndStats ( endTime );
}

void BrainBThread::pause()
{
    paused = !paused;
    if ( paused )
    {
        ++nofPaused;
    }
}

void BrainBThread::set_paused ( bool p )
{
    if ( !paused && p )
    {
        ++nofPaused;
    }
    paused = p;
}
```

Ebben a header fájlban lévő függvények kidolgozását láthatjuk. Itt hozunk létre a Samu Entropy mellé még másik négy Entropyt, majd egy destruktort. Alatta egy run() függvényt ami a program futtatásáért felelős, továbbá egy pause() illetve egy set\_pause() függvény amely a szüneteltetésért illetve a leállásért felelős.

A következő a BrainBWin.h:

```
#include <QKeyEvent>
#include <QMainWindow>
#include <QPixmap>
#include <QPainter>
#include <QFont>
#include <QFile>
#include <QString>
#include <QCLOSEEvent>
#include <QDate>
```

```
#include <QDir>
#include <QDateTime>
#include "BrainBThread.h"

enum playerstate {
    lost,
    found
};

class BrainBWin : public QMainWindow
{
    Q_OBJECT

    BrainBThread *brainBThread;
    QPixmap pixmap;
    Heroes *heroes;

    int mouse_x;
    int mouse_y;
    int yshift {50};
    int nofLost {0};
    int noffound {0};

    int xs, ys;

    bool firstLost {false};
    bool start {false};
    playerstate state = lost;
    std::vector<int> lost2found;
    std::vector<int> found2lost;

    QString statDir;

public:
    static const QString appName;
    static const QString appVersion;
    BrainBWin ( int w = 256, int h = 256, QWidget *parent = 0 ) ←
    ;

    void closeEvent ( QCloseEvent *e ) {

        if ( save ( brainBThread->getT() ) ) {
            brainBThread->finish();
            e->accept();
        } else {
            e->ignore();
        }

    }
}
```

```
virtual ~BrainBWin();
void paintEvent ( QPaintEvent * );
void keyPressEvent ( QKeyEvent *event );
void mouseMoveEvent ( QMouseEvent *event );
void mousePressEvent ( QMouseEvent *event );
void mouseReleaseEvent ( QMouseEvent *event );
```

Mivel a programhoz ismét használunk QT-t itt fent látható az ablakkezelés. Amely jelen esetben is az ablak méretezéssel, paraméterezéssel foglalkozik, valamit az eventekkel.

A főfüggvény a main a következő:

```
#include <QApplication>
#include <QTextStream>
#include <QtWidgets>
#include "BrainBWin.h"

int main ( int argc, char **argv )
{
    QApplication app ( argc, argv );
    QTextStream qout ( stdout );
    qout.setCodec ( "UTF-8" );

    qout << "\n" << BrainBWin::appName << QString::fromUtf8 ( " Copyright (C) 2017, 2018 Norbert Bátfa" ) << endl;
    qout << "This program is free software: you can redistribute it and/or modify it under" << endl;
    qout << "the terms of the GNU General Public License as published by the Free Software" << endl;
    qout << "Foundation, either version 3 of the License, or (at your option) any later" << endl;
    qout << "version.\n" << endl;

    qout << "This program is distributed in the hope that it will be useful, but WITHOUT" << endl;
    qout << "ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS" << endl;
    qout << "FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.\n" << endl;

    qout << QString::fromUtf8 ( "Ez a program szabad szoftver; terjeszthető illetve módosítható a Free Software" ) << endl;
    qout << QString::fromUtf8 ( "Foundation által kiadott GNU General Public License dokumentumában leírtak;" ) << endl;
    qout << QString::fromUtf8 ( "akár a licenc 3-as," ) << endl;
```

```
akár (tetszőleges) későbbi változata szerint.\n" ←
) << endl;

qout << QString::fromUtf8 ( "Ez a program abban a ←
reményben kerül közreadásra, hogy hasznos lesz, ←
de minden" ) << endl;
qout << QString::fromUtf8 ( "egyéb GARANCIA NÉLKÜL, ←
az ELADHATÓSÁGRA vagy VALAMELY CÉLRA VALÓ" ) << ←
endl;
qout << QString::fromUtf8 ( "ALKALMAZHATÓSÁGRA való ←
származtatott garanciát is beleértve. További" ←
) << endl;
qout << QString::fromUtf8 ( "részleteket a GNU ←
General Public License tartalmaz.\n" ) << endl;

qout << "http://gnu.hu/gplv3.html" << endl;

QRect rect = QApplication::desktop() -> ←
availableGeometry();
BrainBWin brainBWin ( rect.width(), rect.height() ) ←
;
brainBWin.setWindowState ( brainBWin.windowState() ←
^ Qt::WindowFullScreen );
brainBWin.show();
return app.exec();
}
```

Itt tulajdonképpen csak leírást kapunk a program céljáról és a fontosabb infomációkról. Majd a kódcsipet végén meghívjuk a futáshoz szükséges függvényeket.

Végül láthatjuk azt a programrészét is amely összegyűjt a header és cpp fájlokat amelyek a működéshez szükségesek.

```
QT += widgets core
CONFIG += c++11 c++14 c++17
QMAKE_CXXFLAGS += -fopenmp
LIBS += -fopenmp
LIBS += `pkg-config --libs opencv` 

TEMPLATE = app
TARGET = BrainB
INCLUDEPATH += .

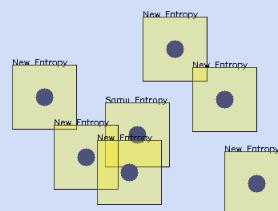
HEADERS += BrainBThread.h BrainBWin.h
SOURCES += BrainBThread.cpp BrainBWin.cpp main.cpp
```

A fordítása a programnak a **qmake** parancsal, valamint a **make** parancsal, a fordítása pedig a **./BrainB** parancsal történik.

```
NEMESPOR BrainB Test 6.0.3
time      : 241
```

```
bps      : 11160
noc     : 7
nop     : 0
lost    :
17430 14380 3640 5250 0 0 0 12750
mean    : 6681
var     : 7139.94
found   : 0 2940 11710 10180 11340 16150 36520
mean    : 12691
var     : 11867.5
lost2found: 0
mean    : 0
var     : 0
found2lost: 12750
mean    : 12750
var     : 0
mean(lost2found) < mean(found2lost)
time    : 0:22
U R about 0.778198 Kilobytes
```

Egy tesztként lefuttatott próba eredménye látható fent és vizuálisan a program ablakban az Entropyk lent.



7.5. ábra. BrainB Benchmark

## 7.5. Vörös Pipacs Pokol/19 RF

Megoldás videó: <https://youtu.be/VP0kfvRYD1Y>

Megoldás forrása: <https://github.com/nbatfai/RedFlowerHell>

Első passz.

DRAFT

## 8. fejezet

# Helló, Schwarzenegger!

### 8.1. Szoftmax Py MNIST

Python

Megoldás videó: <https://youtu.be/j7f9SkJR3oc> linken elérhető, Bátfai Norbert megoldása.

Megoldás forrása: <https://github.com/tensorflow/tensorflow/releases/tag/v0.9.0>

A Python nyelv egy magas szintű programozási nyelv melyet 1989-ben Guido van Rossum holland származású programozó kezdett kifejleszteni, majd 1991-ben kiadta művét. A python nyelv dinamikus típusokat használ, a típusoknak két fajtája létezik, úgymond a mutálható és a mutálhatatlan, ebből érthetjük hogy tömören csak a megváltoztathatóságról van szó. Ez egy kicsit hasonlíthat már az előbbi feladatokban a C++ nyelvben -ami a python után alakult ki- a globális és nem globális OPP részekhez. Valmint a már számunkra ismert OPP (objektumorientált) programozást is támogatja.

Az eredeti forráskód:

```
# Copyright 2015 The TensorFlow Authors. All Rights Reserved.
#
# Licensed under the Apache License, Version 2.0 (the "License"
# );
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#     http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
```

```
#  ↵
=====
"""A very simple MNIST classifier.

See extensive documentation at
http://tensorflow.org/tutorials/mnist/beginners/index.md
"""

from __future__ import absolute_import
from __future__ import division
from __future__ import print_function

# Import data
from tensorflow.examples.tutorials.mnist import input_data

import tensorflow as tf

flags = tf.app.flags
FLAGS = flags.FLAGS
flags.DEFINE_string('data_dir', '/tmp/data//', 'Directory for ←
    storing data')

mnist = input_data.read_data_sets(FLAGS.data_dir, one_hot=True)

sess = tf.InteractiveSession()

# Create the model
x = tf.placeholder(tf.float32, [None, 784])
W = tf.Variable(tf.zeros([784, 10]))
b = tf.Variable(tf.zeros([10]))
y = tf.nn.softmax(tf.matmul(x, W) + b)

# Define loss and optimizer
y_ = tf.placeholder(tf.float32, [None, 10])
cross_entropy = tf.reduce_mean(-tf.reduce_sum(y_ * tf.log(y), ←
    reduction_indices=[1]))
train_step = tf.train.GradientDescentOptimizer(0.5).minimize(←
    cross_entropy)

# Train
tf.initialize_all_variables().run()
for i in range(1000):
    batch_xs, batch_ys = mnist.train.next_batch(100)
    train_step.run({x: batch_xs, y_: batch_ys})

# Test trained model
correct_prediction = tf.equal(tf.argmax(y, 1), tf.argmax(y_, 1) ←
    )
accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.←
```

```
    float32))
print(accuracy.eval({x: mnist.test.images, y_: mnist.test.
    labels}))
```

A fenti forráskód szabadon terjeszthető és felhasználható a TensorFlow Authors fejlesztőinek feltüntetésével. így ebben a kódban kedvünkre dolgozhatunk, az alábbiakban egy átdolgozott megvalósítását láthatjuk a TensorFlow programnak.

Az átdolgozott progi és magyarázata:

A Minst kézzel írott számok adatbázisa, amely 6000 képet tartalmaz. Ez az alapja azoknak a programoknak ami képről ismeri fel a tárgyakat. A Minst program a kézel írt számokról el fogja döntenи hogy milyen szám, de a készírás mindenkinél más, viszont a programnak minden tudnia kell, hogy melyik számot kell felismernie, ez az igazán izgalmas dolog ebben a programban.

A programhoz a TensorFlowet használjuk. A TensorFlow egy a Google által alkotott gépi tanulási rendszer, melyet sok helyen használnak, az egyik az a Google Mapsben található utcakép. A TensorFlow nyílt forráskódú, ezért bárki letöltheti, felhasználhatja, bővítheti és új 5leteit valósíthatja meg egy már megírt kód segítségével.

A forráskódot részekre bontva magyarázom:

```
from __future__ import absolute_import
from __future__ import division
from __future__ import print_function

import argparse

# Import data
from tensorflow.examples.tutorials.mnist import input_data

import tensorflow as tf
old_v = tf.logging.get_verbosity()
tf.logging.set_verbosity(tf.logging.ERROR)

import matplotlib.pyplot

FLAGS = None
```

Azokat a könyvtárakat amelyek szükségesek a **from** kulcsszóval adjuk a programunkhoz, hogy dolgozni tudjunk velük. Majd importáljuk a "TensorFlow" könyvtárt az **import** kulcsszóval és elnevezzük tf-nek, hogy később könnyebben hivatkozhassunk a rövid névre. Majd importáljuk a "matplotlib.pyplot" könyvtárat ami majd a kép kirajzolásához lesz szükséges.

```
def readimg():
    file = tf.read_file("sajat.png")
    img = tf.image.decode_png(file, 1)
    return img
```

Ezután következik readimg() függvény, ami a képeinket olvassa be és dekódolja, majd lent a main() függvény, amiben a kiíratás felépítése van, hogy hogyan küldjük ki az eredményeket, a függvények definiálásához **def** kulcsszót használjuk.

```
def main(_):
    mnist = input_data.read_data_sets(FLAGS.data_dir, one_hot=True)

    # Create the model
    x = tf.placeholder(tf.float32, [None, 784])
    W = tf.Variable(tf.zeros([784, 10]))
    b = tf.Variable(tf.zeros([10]))
    y = tf.matmul(x, W) + b
    mylist=[]
    ilist=[]
    # Define loss and optimizer
    y_ = tf.placeholder(tf.float32, [None, 10])

    # The raw formulation of cross-entropy,
    #
    # tf.reduce_mean(-tf.reduce_sum(y_ * tf.log(tf.nn.softmax(y))) ←
    ,
    #                                     reduction_indices=[1]))
    #
    # can be numerically unstable.
    #
    # So here we use tf.nn.softmax_cross_entropy_with_logits on the ←
    raw
    # outputs of 'y', and then average across the batch.
    cross_entropy = tf.reduce_mean(tf.nn.←
        softmax_cross_entropy_with_logits(labels = y_, logits = y))
    train_step = tf.train.GradientDescentOptimizer(0.5).minimize(←
        cross_entropy)

    sess = tf.InteractiveSession()
    # Train
    tf.initialize_all_variables().run(session=sess)
    print("-- A halozat tanitása")
    for i in range(1000):
        batch = mnist.train.next_batch(50)
        batch_xs, batch_ys = mnist.train.next_batch(100)
        correct_prediction = tf.equal(tf.argmax(y, 1), tf.argmax(y_ ←
            , 1))
        accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.←
            float32))
        train_accuracy = accuracy.eval(feed_dict={
            x: batch[0], y_: batch[1]})
        sess.run(train_step, feed_dict={x: batch_xs, y_: batch_ys})
        mylist.append(train_accuracy)
        ilist.append(i)
```

```
if i % 100 == 0:  
    print(i/10, "%")
```

A sess azaz egy session segítségével fogjuk a tanítást végezni, hasonlóan mint a neurális hálózatnál. A session objektum lehetővé teszi, hogy egy kérés folytatódjon egy bizonyos paraméteren keresztül. Ez a Train program rész azaz a vonat rész, a session tulajdonságából kiindúlva, hogy hosszú időn keresztül folytatódik a folyamat. A pontosság miatt a ciklust ezerszer fogjuk futtatni egy for ciklusban, hogy az eredmény tökéletest megközelítse. Aztán kírjuk mennyire lett pontos az eredmény százalékban, a print() függvény segitségével.

```
# Test trained model  
print("-- A halozat tesztelese")  
  
print("-- A MNIST 42. tesztkepenek felismerese, mutatom a ←  
      szamot, a továbblepeshez csukd be az ablakat")  
  
img = mnist.test.images[42]  
image = img  
  
matplotlib.pyplot.imshow(image.reshape(28, 28), cmap=matplotlib ←  
                        .pyplot.cm.binary)  
matplotlib.pyplot.savefig("4.png")  
matplotlib.pyplot.show()  
  
classification = sess.run(tf.argmax(y, 1), feed_dict={x: [image ←  
] })  
  
print("-- Ezt a halozat ennek ismeri fel: ", classification[0])  
print("←  
-----")  
  
print("-- A sajat kezi -asom felismerese, mutatom a szamot, a ←  
      továbblepeshez csukd be az ablakat")  
img = readimg()  
image = img.eval()  
image = image.reshape(28*28)  
matplotlib.pyplot.imshow(image.reshape(28,28), cmap=matplotlib. ←  
                        pyplot.cm.binary)  
matplotlib.pyplot.savefig("8.png")  
matplotlib.pyplot.show()  
  
classification = sess.run(tf.argmax(y, 1), feed_dict={x: [image ←  
] })  
  
print("-- Ezt a halozat ennek ismeri fel: ", classification[0])  
print("←  
-----")  
matplotlib.pyplot.plot(ilist,mylist, color='red', linestyle='←  
solid', linewidth = 1,marker='o',
```

```
markerfacecolor='blue', markersize=3)

matplotlib.pyplot.show()
```

Elérkeztünk a képzett modell tesztelés (Test trained model) kódrészhez. A programcsipetben először megjelenik egy a mnist 42. tesztkép, a rajzoláshoz a **matplotlib.pyplot**-t használjuk. Majd megjelenik a saját képünk és a session tovább futtat. Végül megjelenik a saját képünk értékelése, hogy mennyire volt felismerhető a kézírásunk.

```
if __name__ == '__main__':
    parser = argparse.ArgumentParser()
    parser.add_argument('--data_dir', type=str, default='/tmp/ ←
        tensorflow/mnist/input_data',
                        help='Directory for storing input data')
    FLAGS = parser.parse_args()
    tf.app.run()
```

Ha egy képet bezárunk akkor jön a következő kép és így tovább. Legvégül program a kapott eredményeket egy tömben tárolja el.

A program viszonylag nagy pontossággal fogja felismerni a képeken lévő számokat, ezt a többszöri futtatásnak, jelen esetben már említett 1000 futtatásnak köszönhetünk.

## 8.2. Mély MNIST

Python

Passz.

## 8.3. Minecraft-MALMÖ

Információkat a Malmö projekttel kapcsolatban bevezetésképpen a Turing fejezetben még az első ilyen feladat előtt írtam tájékoztatást, hogy érthető legyen ezen projekt célja. Itt már egy picit komolyabb kifejtésre kerül sor, de a számos feladatnak a külön-külön magyarázatának az olvasásával mélyebb betekintést nyerhet az olvasó.

Megoldás videó: initial hack: <https://youtu.be/bAPSu3Rndi8>. Red Flower Hell: <https://github.com/nbatfai/RedFlowerHell>.

A projektben számtalan kreatív lehetőséggel rendelkezünk ahogyan ezt a projekt vezetői is említtik. A lehetőségünk az ágens irányításával kezdve a különböző blockok azonosításával és a környezet felismerésével a blockok mozagtásán át, a különböző sziutációkban, az npc és mob közelség reakcióig és még tovább egy kreatív programozó és fan számára kimeríthatetlen lehet.

Ezen RFH vagy Vörös Pipacs Pokol projektnek lényege, hogy egy aréna jellegű map-al rendelkezünk, mely egy tölcserhez hasonló. A láva folyik az aréna peremétől egészen az aréna közepéig, pontosan 300

másodperc alatt ér le a közepére, és feladatunk, hogy a lehető legtöbb Pipacsot szedjünk össze ez idő alatt, melyek minden szinten el vannak helyezve, random koordinátákkal.

Van lehetőség előre és hátra menni a **move**, fordulni a **turn**, a nézés irányát változtatni a **look**, ütni a **attack** parancsok segítségével és még több másra is képesek vagyunk. A 2020-as tavaszi félév során a kezdetben még csak a csigavonalban haladó Steve intelligenciája a program megírása alapján már eljutott addig, hogy a piros virágokat érzékelje és kiüsse majd felszedje és tovább menjen, természetesen ezt mind időre a láva leérkezése előtt.

Az ágensprogramozás nem olyan egyszerű mint ahogyan azt gondolnánk, de a végeredmény mosolyt tud csalni az racunkra. Ellentétben azzal, hogy számtalan szor lehet nemvárt hibákba futni, amelyeket nem a legkönnyebb kijavítani és tökéletessé csiszolni. Viszont a cél az ágens megismerése és minden paramétert megfelelően beállítani annak érdekében, hogy a megfelelő eredményt érjük el.

A [Red Flower Hell](#) repóban különböző érdekes programkódok elérhetők.

A mesterséges intelligencia fejlesztői is azon dolgoznak, hogy ezt ne csak egy ilyen MALMÖ projektben ahol különböző blockok felismerésével érünk el eredményt hanem már pixelek alapján is értelmezhetővé váljon a vizuális térben való elhelyzkedés és különböző feladatak elvégzése.

## 8.4. Vörös Pipacs Pokol/javíts a 19 RF-en

Megoldás forrása: <https://github.com/nbatfai/RedFlowerHell>

Második passz.

## 9. fejezet

# Helló, Chaitin!

### 9.1. Iteratív és rekurzív faktoriális Lisp-ben

Tutorált: Talinger Mark Imre

Megoldás videó: <https://youtu.be/z6NJE2a1zIA>

A Lisp egy olyan programozási nyelv amely felkapott lett a MI (mesterséges intelligencia) kutató, fejlesztők körében, habár a nyelv nem erre a célra jött létre. A nyelv szintaktikája egyedi, listákkat lácol egybe és ezeket dolgozza fel. A listákat és némely esetben segítségül hívva ezt a program nyelvet matematikai órákon és feladatok számításában használják.

Ez a nyelv egy kifejezés orientált nyelv, mely segítségével ebben a feladatban a faktoriálisliságot szemlél-tetjük.

Iteratívan:

```
(defun faktorialisi (n)
  (do
    ((i 1 (+ 1 i))
     (prod 1 (* i prod)))
    ((equal i (+ n 1)) prod)))
```

A **defun** szóval adjuk meg a függvény és a benne lévő változó nevét. Majd a **do** konstrukcióval megadjuk azokat az utasításokat amelyeket el kell végezni. A következő sorban megnöveli az *i* értékét, érdekes lehet hogy a műveleti jelet listában a két érték előtt tesszük. A következő sorban szorozzuk az *i*-t a proddal, majd a következőben addig növelgetjük az *i*-t amíg egyenlő nem lesz a proddal.

Rekurzívan:

```
(defun faktorialisr(n)
  (if (= n 1)
      1
      (* n (faktorialisr (- n 1)))))
```

Itt is a **defun** szóval megadjuk a függvény és benne lévő változó nevét. Majd egy **if** feltétellel megnézzük hogy az *n* egyenlő-e eggyel, ha igen akkor szorozzuk össze az (*n*-1)-szeresével.

## 9.2. Gimp Scheme Script-fu: króm effekt

Írj olyan script-fu kiterjesztést a GIMP programhoz, amely megvalósítja a króm effektet egy bemenő szövegre!

Megoldás videó: [https://youtu.be/OKdAkI\\_c7Sc](https://youtu.be/OKdAkI_c7Sc)

Megoldás forrása: [https://gitlab.com/nbatfai/bhax/tree/master/attention\\_raising/GIMP\\_Lisp/Chrome](https://gitlab.com/nbatfai/bhax/tree/master/attention_raising/GIMP_Lisp/Chrome)

A **GNU Image Manipulation Program** (Gimp) egy képszerkesztő, vagy képmánipuláló program melyben rétegenként is felépíthető egy kép.

A GIMP-en belüli utasítások helyettesíthetők szkriptnyelvekkel is. Az a szkriptnyelv amit mi fogunk használni az a **Script-Fu**, amely egy **Scheme** alapú szkriptnyelv amely egy beépített szkriptnyelv. A Scheme nyelv a **Lisp** nyelvhez hasonlít mivel egy családba tartoznak és így a Scheme szintaxisa a Lispével megegyezik.

Ebben a feladatban egy olyan szkriptet használunk amely a bemeneti szöveget krómozott efektel látja el.

A forráskód a következő:

```
(define (color-curve)
  (let* (
    (tomb (cons-array 8 'byte))
  )
    (aset tomb 0 0)
    (aset tomb 1 0)
    (aset tomb 2 50)
    (aset tomb 3 190)
    (aset tomb 4 110)
    (aset tomb 5 20)
    (aset tomb 6 200)
    (aset tomb 7 190)
  tomb)
)
```

Ebben a kódrészletben a **color-curve** függvényünket definiáljuk, létrehozunk egy tömböt amely majd a szövegünk krómozását fogja végezni, és ehhez kapcsolódó értékeket tartalmaz. A szintaktikán itt láthatjuk is, hogy nagyon hasonló a Lispéhez.

```
(define (elem x lista)
  (if (= x 1) (car lista) (elem (- x 1) (cdr lista) ) )
)

(define (text-wh text font fontsize)
(let*
(
  (text-width 1)
```

```
(text-height 1)
)

(set! text-width (car (gimp-text-get-extents-fontname text ←
    fontsize PIXELS font)))
(set! text-height (elem 2 (gimp-text-get-extents-fontname text ←
    fontsize PIXELS font)))

(list text-width text-height)
)
)
```

Ebben a forrásrészletben az **elem** majd a **text-wh** függvények kerülnek létrehozásra. Az elem függvény paraméterként kap egy listát és egy indexet, ezt fogja visszaadni. Ha ez az index 1, akkor a lista első eleméről van szó, tehát a **car** adja vissza, ha viszont nem erről van szó, akkor a **cdr** fogja visszaadni. A második függvény a **text-wh>/** az a szövegünknek fogja a szélességét és a magasságát beállítani, ez egy kezdőérték lesz, majd ezt követően beállít egy bizonyos értéket nekik. A végén visszaadja a szöveg szélességét és magasságát.

```
(define (script-fu-bhax-chrome text font fontsize width height ←
    color gradient)
(let*
(
    (image (car (gimp-image-new width height 0)))
    (layer (car (gimp-layer-new image width height RGB-IMAGE "←
        bg" 100 LAYER-MODE-NORMAL-LEGACY)))
    (textfs)
    (text-width (car (text-wh text font fontsize)))
    (text-height (elem 2 (text-wh text font fontsize)))
    (layer2)
)
)
```

Ebben a kódrészletben pedig a szkriptünket fogjuk létrehozni, valamint néhány paramétert is megadunk. Ezt követően a **let** parancs végzi a deklarálásokat. Létrehozunk egy képet, egy réteget, egy szöveg magasságot és szélességet, thehát ami a szkriptünknek szükséges a működéshez.

Ezután lépésekre van bontva a programunk:

```
(gimp-image-insert-layer image layer 0 0)
(gimp-context-set-foreground '(0 0 0))
(gimp-drawable-fill layer FILL-FOREGROUND )
(gimp-context-set-foreground '(255 255 255))

(set! textfs (car (gimp-text-layer-new image text font fontsize ←
    PIXELS)))
(gimp-image-insert-layer image textfs 0 0)
```

```
(gimp-layer-set-offsets textfs (- (/ width 2) (/ text-width 2)) (- ←
  (/ height 2) (/ text-height 2)))

(set! layer (car(gimp-image-merge-down image textfs ←
  CLIP-TO-BOTTOM-LAYER)))
```

Az első lépésben hozzáadunk egy réteget adunk a képünkhez, egy fekete hátteret amellyel filleljük is a foregroundot, majd a szövegnek fehér szint adunk. A fekete szint a 0 0 0-ás RGB kód adja meg, míg a fehérét a 255 255 255. A további részletek további réteg beillesztését tartalmazza, offset-beállítást, fontsize pontositást.

```
(plug-in-gauss-iir RUN-INTERACTIVE image layer 15 TRUE TRUE)
```

A második lépésben plug-in-t használunk, ahhoz, hogy Gauss elmosást alkalmazzunk a szövegen, melyet a számérték szabályoz.

```
(gimp-drawable-levels layer HISTOGRAM-VALUE .11 .42 TRUE 1 0 1 TRUE ←
  )
```

A harmadik lépésben is különböző értékeket láthatunk, mely a layer-hez kötődik, és a HISTOGRAM-VALUE-t szabályozzuk vele, mely a képünk frekvenciáját vagyis részletességét adja meg.

```
(plug-in-gauss-iir RUN-INTERACTIVE image layer 2 TRUE TRUE)
```

A negyedik lépésben szintén Gauss elmosást alkalmazunk, itt már kisebb mértékben, tehát kisebb lesz az elmosás.

```
(gimp-image-select-color image CHANNEL-OP-REPLACE layer '(0 0 0))
(gimp-selection-invert image)
```

A ötödik lépésben a fekete színünket fogjuk kiválasztani és elkülöníteni. Ezt követően invertáljuk a szelktálást.

```
(set! layer2 (car (gimp-layer-new image width height RGB-IMAGE "2" ←
  100 LAYER-MODE-NORMAL-LEGACY)))
(gimp-image-insert-layer image layer2 0 0)
```

A hatodik lépésben **layer2** nevezetű réteget állítunk be, mely a 100as érték miatt átlátszó réteg lesz. Ezt be is illesszük az **insert-layer**-el.

```
(gimp-context-set-gradient gradient)
(gimp-edit-blend layer2 BLEND-CUSTOM LAYER-MODE-NORMAL-LEGACY ←
  GRADIENT-LINEAR 100 0 REPEAT-NONE
  FALSE TRUE 5 .1 TRUE width (/ height 3) width (- height (/ ←
    height 3)))
```

A hetedik lépésben a kontexus gradiensét állítjuk be, valamint a layer2-es rétegünk átmenetét állítjuk be és a végeredmény egy szürke szint fog adni a **layer2**-nek.

```
(plug-in-bump-map RUN-NONINTERACTIVE image layer2 layer 120 25 7 5 ←
 5 0 0 TRUE FALSE 2)
```

A nyolcadik lépésben egy plug-in-t használunk, amely egy részletességet és egy térbeli hatást ad hozzá a képünkhez.

```
(gimp-curves-spline layer2 HISTOGRAM-VALUE 8 (color-curve))

(gimp-display-new image)
(gimp-image-clean-all image)
)
```

A kilencedik lépésben a **curves-spline** függvényt fogjuk használni, melyet követően a **layer2** HISTOGRAM-értékét állítjuk be, a végeredmény pedig az, hogy egy krómos hatást érünk el a képen.

```
(script-fu-register "script-fu-bhax-chrome"
  "Chrome3"
  "Creates a chrome effect on a given text."
  "Norbert Bátfai"
  "Copyright 2019, Norbert Bátfai"
  "January 19, 2019"
  ""
  SF-STRING      "Text"        "Bátf4i Haxor"
  SF-FONT        "Font"        "Sans"
  SF-ADJUSTMENT  "Font size"   '(100 1 1000 1 10 0 1)
  SF-VALUE       "Width"       "1000"
  SF-VALUE       "Height"      "1000"
  SF-COLOR       "Color"       '(255 0 0)
  SF-GRADIENT    "Gradient"    "Crown molding"
)
(script-fu-menu-register "script-fu-bhax-chrome"
  "<Image>/File/Create/BHAX"
)
```

Végül a szkriptünk regisztrálására kerül sor, hogy használni tudjuk. Ezen belül pedig kezdőértékeket is látunk amelyeket beállíthatunk.

A programunk vagyis szkriptünk futtatásához le kell töltenünk a Gimp nevű alkalmazást. Ezt követően az alkalmazáson belül a **fu-script**-eket tartalmazó mappába mentjük a saját szkriptünket (fájlnév.scm). A Gimp alkalmazásban a **Filters** azon belül a **Script-Fu** nevű lehetőségben a **Console**-ba behelyezzük a **(script-fu-bhax-chrome "Bátf4i Haxor" "Sans" 120 1000 1000 '(255 0 0) "Crown molding")** sort futtatjuk. Ezt követően megjelenik a krómozott képünk.

### 9.3. Gimp Scheme Script-fu: név mandala

Írj olyan script-fu kiterjesztést a GIMP programhoz, amely név-mandalát készít a bemenő szövegből!

Megoldás videó: [https://bhaxor.blog.hu/2019/01/10/a\\_gimp\\_lisp\\_hackelete\\_a\\_scheme\\_programozasi\\_nyelv](https://bhaxor.blog.hu/2019/01/10/a_gimp_lisp_hackelete_a_scheme_programozasi_nyelv)

Megoldás forrása: [https://gitlab.com/nbatfai/bhax/tree/master/attention\\_raising/GIMP\\_Lisp/Mandala](https://gitlab.com/nbatfai/bhax/tree/master/attention_raising/GIMP_Lisp/Mandala)

Harmadik passz.

### 9.4. Vörös Pipacs Pokol/javíts tovább a javított 19 RF-eden

Megoldás forrása: <https://github.com/nbatfai/RedFlowerHell>

Negyedik passz.

## 10. fejezet

# Helló, Gutenberg!

### 10.1. Programozási alapfogalmak

A programozási nyelveknek három szintje van: gépi nyelv, assembly szintű nyelv, magas szintű nyelv. A magas szintű nyelven írt programot forrásszövegnek nevezünk. Egy magas szintű programozási nyelv: szemantikai és szintaktikai szabályok együttese. A szintaktikai szabályok: forrásszöveg összeállítására vonatkozó formai és nyelvtani szabályok összessége. A szemantikai szabályok: tartalmi, értelmezési és jelentésbeli szabályok.

A gépi nyelvet a processor ismeri, tehát a magas szintű forrásszöveget fordítóprogram segitségével vagy interpreteléssel kell a processorhoz juttatni.

Fordítóprogram: gépi kódú tárgyprogramot állít elő. Lépései:

lexikális elemzés

szintaktikai elemzés

szemantikai elmezés

kódgenerálás (csak szintaktikailag helyes forrásprogramból lehet elíállítani tárgyprogramot.)

Az interpreteres technika esetén is megvan az első három lépés, de az interpreter nem készít tárgyprogramot, hanem utasításonként értelmezi a forrásprogramot és végrehajtja azt. Az egyes programnyelvek együttesen alkalmazzák minden két technikát.

Hivatkozási nyelv: a programnyelv szabálya.

Implementációk: fordítóprogramok vagy interpretek. A hordozhatóság problémája az implementációk inkompatibilitását jelenti, melyek adott platformon realizált fordítóprogramok vagy interpretek. A problémára már 50 éve nincs teljesen korrekt megoldás.

Programozási nyelvek osztályai: Imperatív: algoritmus nyelvek és utasítások sorozata, az algoritmus működteti a processzort, a változó jelenléte és közvetlen elérése (Alcsoportjai: Eljárásorientált, Objektumorientált nyelvek).

Deklaratív nyelvek: nem algoritmikus nyelvek, nincs lehetőség memóriaműveletekre, csak a problémát adja meg a programozó (Alcsoportjai: Funkcionális, Logikai nyelvek).

Más nyelvek: nincs egységes jellemzőjük, tagadják valamelyik imperatív jellemzőt.

A forrásszöveg legkisebb alkotórészei a karakterek. Alapvető a karakterkészlet, ezekből állíthatók össze a bonyolult nyelvi elemek. Eljárásorientált nyelvek esetén ezek: lexikális egységek, szintaktikai egységek és utasítások, programegységek, program.

A karakterek kategorizálása: betűk, számjegyek, egyéb karakterek. minden programnyelvben betű az angol ABC 26 nagybetűje, és ezek közül a kis és nagy betűket is egyes nyelvek elfogadnak (C), más nyelvek viszont nem (Pascal).

Még előforduló karakterek: \_, \$, #, @, +, -, \*, /, [, ], ., :, {, }, ;, :, ?, !, ~.

A lexikális egységek a program szövegének azon elemei, melyeket a fordító a lexikális elemzés során felismer és tokenizál.

Fajtái: Többkarakteres szimbólumok: ++, --, /\*, \*/ stb.

Szimbolikus nevek:

Azonosító (karaktersorozat ami betűvel kezdődik és betűvel vagy számjeggyel folytatódik) pl: x, ab, hall-gato\_azonosito, SzemelyNev

Nem azonosító pl: x+y, 123abc.

Kulcsszavakb(alapszó) pl: if,for,case,break.

Cimke: speciális karaktersorozat, amely lehet előjel nélküli egész szám vagy azonosító. Általános, hogy utasítás előtt áll és : -al van elválasztva.

Cimke felépítése példa: azonosító (C), 4 számjegyből álló egész szám (Pascal)

Megjegyzés (Komment vagy magyarázat): A program szövegét olvasó embernek szól, nem a fordítónak. // vagy /\* itt magyarázhatjuk a programrész működését \*/

Literálok (Konstansok): Fix, explicit értékek a program szövegében. pl: egész literálok, valós literálok, karakter literálok, sztring literálok.

Forrásszöveg összeállításának általános szabályai:

A kötött formátumú nyelvek esetén egy sorban egy utasítás volt elhelyezhető, a szabad formátumú nyelvek esetén akárhány utasítás egy sorban, két pontosvessző között áll egy utasítás. A lexikális egységeket alapszóval vagy szóközzel kell elválasztani.

Adattípusok:

Az adattípus egy absztrakt programozási eszköz, amely minden más. Konkrét programozási eszköz egy komponens. Az adattípusnak neve van, egy azonosító.

Egy adattípust meghatároz:

a tartománya: ahol felvehető értékként,

a műveletei: a tartomány elemein,

és reprezentációja: egyes típusok tartományába tartozó értékek tárban való megjelenése.

Egyszerű típusok: egész (fixpontos), valós (lebegőpontos), karakteres (karakteres ábrázolás), logikai (igaz vagy hamis), felsorolás, sorszámozott.

Összetett típusok: tömb (dimenzió száma, indexkészlet típusa és tartománya, elemeinek típusa) A C nem ismeri a többdimenziós tömböt (egydimenziós tömb egydimenziós tömb-elemekkel képzeli el). A mutató: egyszerű típus, tartományának elemei tárcímek, azaz egy adott tárbeli területre mutat. Speciális tartománybeli eleme a NULL.

A nevesített konstans három komponensből áll: név, típus és érték. Mindig deklarálni kell, akkor használjuk amikor egy érték sokszor előfordul és ezt elnevezzük egy nevesített konstansra, amire egyszerűen hivatkozhatunk. A C ben #define név a literál.

A változó négy komponense: név, attribútumok, cím, érték. Ezek minden változónál szerepelnek. A név egy azonosító.

Az attribútumok a futásközbeni viselkedést határozzák meg (ez lehet akár a tipusa). Változó attributumok esetén deklarációt alkalmazunk: explicit deklaráció: programozó végzi, teljes nevéhez kell az attribútumokat megadni; implicit deklaráció: betűhöz rendel attribútumokat, azonos kezdőbetűjű változók u.olyan attribútumúak lesznek; automatikus deklaráció: a fordítóprogram rendel attribútumokat a változóhoz.

A változó címe: ahol a tárnak azt a részét határozza meg ahol a változó értéke elhelyezkedik. A cím rendelhető: statikusan, dinamikusan, programozó által. De minden hárromra kell olyan eszköz ami megszűnteti a cimkomponenst.

A változó értéke: értékkedő utasítás által (változó = kifejezés), input (egy perifériáról), kezdőértékkedés (explicit, implicit).

Az alapelemek a C nyelvben: aritmetikai típusok (egyszerű); integrális típusok; valós típusok; származtatott (összetett) pl tömb (egydimenziós), függvény, mutató, struktúra; void típus.

Kifejezések: Szintaktikai eszközök.

Két komponensük az érték és a típus.

Formális összetevői az operandusok (érték), az operátorok (műveleti jelek) és a kerek zárójelek (sorrend szabályozásra).

Vannak egyoperandusú (unáris), kétoperandusú (bináris) és háromoperandusú (ternáris) operátorok.

Három alakja van a kifejezéseknek:

prefix (operátor operandusok előtt): \* 3 5,

infix (között): 3 \* 5,

postfix (után): 3 5 \*.

Műveletek végrehajtási sorrendje:

- balról jobbra (standard)

- jobbról balra (fordított)

- balról jobbra (precedencia táblával, vagyis prioritás(zárójelekkel) segítségével)

Infix alakban balról-jobbra történő művelet végrehajtási szabály van. Infix alak esetén kell használni zárójeleket (ezek lesznek az elsődlegesek). Vannak logikai operátorok is (és, vagy...).

A kiérékelések típusai:

- teljes (pl.FORTRAN)

- rövidzár (pl.PL/I)

- rövidzár operátorok: and then, or else

- nem rövidzár op.-ok: and, or

A kifejezés típusa lehet:

tipusegyenértékű (kétoperandusú operátornak csak azonos típusú operandusai lehetnek) vagy tipuskényszerítő (különböző típusú operandusok is lehetnek).

A konstans kifejezés kiértékelését a fordító végzi. Operandusai lehetnek literálok és nevesített konstansok.

A C egy alapvetően kifejezésorientált nyelv. A mutató tipussal összeadás és kivonás végezhető. A tömb típusú eszköz neve mutató típusú, tehát  $a[i] = *(a+i)$ .

Példák C beli operátorokra (precedencia táblázat alapján):

1. balról jobbra:

$()$ (függvényoperátor,precedencia felülirás),

$[]$ (tömboperátor),

$&&$  (és operátor, kétoperandusú)

$?:($ (háromoperandusú).

2. jobbról balra:

$=$  (értékkadás)

$*=$  (szorzás és értékkadás a bal oldalra),

$+=, ^=,$  stb.

Utasítások:

Az utasítások az eljárásorientált nyelvek egyes lépéseit adják meg, és ezáltal generálja a fordítóprogram a tárgyprogramotm, melynek két csoportja van, deklarációs és végrehajtható utasítások. A deklarációs utasítások mögött nem áll tárgykód, ezek a fordítóprogramnak szólnak, tehát befolyásolják a tárgykódot, de ők nem kerülnek fordításra. A végrehajtható utasításokból generálódik a tárgykód a fordítóprogram által. Besorolhatjuk őket több alosztályba:

Értékkadó utasítás: beállít vagy módosít egy változó értékén a program futása közben egy bizonyos időpontban.

Üres utasítás: gyártelmű programszerkezet alakítható ki velük.

Ugró utasítás: Korai nyelvekben használatos, mely egy feltétel teljesülésének következtében egy meghatározott részére ugrik a programnak, és egy adott címkelvel ( felétel ) ellátott utasítást fog végrehajtani.

Elágaztató utasítás: Két választási lehetőség van a program adott részén feltételek alapján (pl. if else), vagy több lehetőség közül (pl. switch case 1, case 2, ...stb. )

Ciklusszervező utasítás: A program adott pontján egy tevékenységet akárhányszor elvégezhetünk.

A ciklusszervező utasítások, a C nyelvben:

Kezdőfeltételes: While(feltétel){utasítás},

Végfeltételes: Do{utasítás} while(feltétel),

For-ciklus: For(kif1; kif2; kif3){utasítás}

Vezérlő utasítások a C-ben:

Continue; Újrakezdi a feltételvizsgálatot, ami pedig utána van az nem hajtódiék végre,

Break; Megtörí vagy leállítja a ciklust, és kilép az utasításból,

Return (kifejezés); (Befejezteti a függvényt és visszaadja a vezérlést a hívónak).

Programok szerkezete:

Az eljárásorientált nyelvekben a program szövege programegységekre tagolható

Az alábbi programegységek léteznek:

Alprogramok: Akkor használjuk, ha több helyen is felhasználunk egy programrészét, és ezt külön egy helyre leírjuk, amit majd később ismertetek, hogy hogyan használhatjuk fel többször is. Négy komponensből áll: név, paraméter lista, törzs, környezet.

A neve egy azonosító, a paraméter lista lehet üres is vagy azonosítók szerepelnek benne, amelyeknek szerepe lesz az alprogramban. A törzsben deklarációs, végrehajtandó utasítások vannak, itt van leírva, hogy mit csináljon az alprogramunk. Az alprogram környezete alatt a globális változók együttesét értjük.

Az alprogramok két kategóriába tartoznak: eljárás és függvény.

Az eljárás egy olyan alprogram amely több utasítást hajt végre, a hívás helyén az eredményét használjuk fel. A függvény egy olyan alprogram, mely egyetlen értéket határoz meg, és ezzel tér vissza. A függvény visszatérési érték a hívás helyére tér vissza.

Függvény hívás: függvénynév (paraméter lista)

Eljárás hívása: [alapszó] eljárásnév(paraméter lista).

A hívási lánc, vagy rekurzió az bizonyos programegységek egymásba ágyazott hívásán alapszik. Egy programegység bármikor meghivhat egy másik programegységet, és a vezérlés oda ugrik.

Amikor egy aktiv alprogramot hivunk meg, azt nevezzük rekurzióknak.

Rekurzió lehet:

- közvetlen: egy alprogram önmagát hívja, azaz magára hivatkozik.
- közvetett: a hívási láncban korábban szereplő alprogramot hivunk meg.

Másodlagos belépési pontok: Vannak nyelvek, melyek megengedik, hogy egy alprogramot ne csak fejen keresztül lehessen meghivni, hanem a törzsben ki lehessen alakítani ún. másodlagos belépési pontokat, tehát ezzel is lehet hivatkozni az alprogramra.

A paraméterkiértékelés az a folyamat, amikor egymáshoz rendelődnek a formális és aktuális paraméterek egy alprogram hívásánál. Mindig a paraméter lista az elsődleges, az aktuális paraméterlistából akárhány lehet, attól függ hogy hányszor hívjuk az alprogramot. A paraméterszám lehet fix, de tetszőleges is.

A paraméter átadás egy kommunikációs forma az alprogramok és más programegységek között. Mindig van egy hívó, és egy hívott.

A nyelvek által ismert paraméterátadási módok:

- érték szerinti,
- cím szerinti,
- eredmény szerinti,
- érték-eredmény szerinti,
- név szerinti,
- szöveg szerinti.

A blokk egy programegység. Más programegység belsejében helyezkedik el kizárolag. Van kezdete, törzse és vége.

A hatáskör nevekhez kapcsolódik. Hatáskör alatt értjük a program szövegének egy olyan részét, ahol jelentése felhasználási módja és jellemzői azonosak. A hatáskör lehet lokális, vagyis egy programegységen belül van deklarálva, és lehet globális, amely mindenhol elérhető a program területén.

## 10.2. C programozás bevezetés

Rövid olvasónapló a [KERNIGHANRITCHIE] könyvről.

Megoldás videó: <https://youtu.be/zmfT9miB-jY>

### Alapismeretek

A könyv első fejezete a C nyelv alapismereteinek elsajátításairól szól, főképpen azok irányába ajánlott aki már tanultak programozni. A képernyőre való kiiratás megvalósítását ismerjük meg a printf() függvény használatával.

Megismerjük a Fahrenheit-hőmérséklet Celsius-értékké alakításának megoldását is, valamint a változó tipusokat. Az ismétlő ciklusokat is bemutatja, azaz a while és a for.

Bevezetést kapunk a szimbolikus állandók fogalmába is, néhány alapvető függvényt (getchar(), putchar()) ismertet. Napi rendszerességgel használt eljárás: karakterek számlálása, szavak számlálása, sorok számlálása, valamint tömbök, függvények, argumentumok és érték szerinti hívások.

### Tipusok, operátorok, kifejezések

Ebben a fejezetben változónevek szabályairól, használati utasításait olvashatjuk, valamint tovább boncoljuk az adattipusokat és a hozzá kapcsolódó méreteket. A matematikai állandók is szóbajönnek, mint pl. az e vagy pi, de állandó létrehozást is kifejtik a fejezetben.

A deklarációra is kitér a tipusok, operátorok, kifejezések, változók tekintetében. Megismerkedhetünk az aritmetikai operátorokkal, relációs és logikai operátorokkal. A tipuskonverziók témaköre rendkívül hasznos és megtudhatjuk, hogy csak az értelmes konverziók történnek meg.

Az inkrementáló, dekrementáló operátorok is szóbajönnek, amik szintén alap szintű felfogást igényelnek a megértéshez, valamint a bitenkénti logikai operátorok. Értékadó operátorok, feltételes kifejezések, precdencia (kiértékelés sorrendje).

### Vezérlési szerkezetek

Itt ismerjük meg az utasításokat és blokkokat, valamint az if-else utasítást, else-if utasítást, switch utasítást, while, for utasítást, do-while utasítást, break utasítást, continue utasítást, goto utasítást, cimkéket.

Fontos ezeket megjegyezni, mivel sok példával van szemlélteve, mert tulajdonképpen alapkövek a C nyelvben, és más nyelvekben is előkerülnek, lehet hogy más formában, de előkerülnek. Egyszerűek, meg lehet őket jegyezni, sok gyakorlással és alkalmazással meg a kisujjunkba kerülhetnek.

### Függvények és programstruktúra

Itt a függvényeket és a program felépítési szabályait vesézzük ki. A függvények előnyeiről, hasznosságáról találhatunk példákat, valamint konkrét programrészleteket, helyes és működő programstruktúrákról. Külső változók, regiszter változók, érvényességi tartomány szabályai sem marad ki, ezeket is nagyon érthetően elemzi a könyv, és a kiváló példákkal elülteti az ember agyában. Statikus változók és a blokkstruktúra is

megjelenik, valamint további nyelvekkel kerül összehasonlításba a C nyelv. Az inicializálás, rekurzió, C előfeldolgozó, állomány beiktatás és makrohelyettesítés témakörei is taglalva vannak.

### Utasítások

Az utasítások egymást követően sorban hajtódnak végre. Több fajtáját is megkülönböztetjük az utasításoknak:

Kifejezés utasítás: értékkedások, függvényhívások.

Összetett utasítás/blokk: ahol elvileg egy utasítás helyezhető el, ott a blokk használatával többet is használhatunk.

Feltételes utasítás: Akkor használjuk, ha két lehetőségből kell választani. if-else a példa rá, ha az if igaz akkor azt végzi el, különben az else ágat.

While utasítás: Végrehajtódik az utasítás mindaddig amíg amíg a kifejezés értéke nem nulla marad.

Do-while utasítás: Mindaddig ismétlődik az utasítás amíg a kifejezés értéke nullává nem válik.

For utasítás: Képesek vagyunk megadni, hogy hányszor hajtódjón végre az utasítás, és hogy mi legyen a feltétel.

Switch utasítás: ez a többágú feltételes utasítás: egy kifejezést több esetre bonthatunk, case:-kre valamint van egy default: eset is.

Break utasítás: Befejeződik az őt körülvevő while, do-while, for vagy switch utasítás.

Continue: átugorja a többi utasítást és az őt körülvevő while,do-while,for utasítás ciklusfolytató részére ugrik a vezérlés.

Return utasítás: A függvény a hívójához a return utasítással tér vissza.

Goto utasítás: a vezérlés feltétel nélkül adható át az adott helyre.

Cimkézett utasítás: Az utasítások címkelvel láthatók el, amely a goto célpontjaként szolgál.

Nulla utasítás: hordozhat címkét, vagy képezhet üres ciklustörzset.

## 10.3. C++ programozás

Rövid olvasónapló a BMECPP könyvről.

Az első fejezetben megismerhetjük a C++ nyelv rövid történetét, objektum-orientált tulajdonságait és a generikus programozás fogalmait. A C++ C nyelvre való épülése fontos dolog, amit mindenki tudni kell, és ezen nyelvek szoros, testvéri kapcsolatáról olvashatunk.

A második fejezetben néhány nem objektum orientált újdonságait ismerhetjük meg a C++ nyelvnek. Összehasonlításra kerül a C nyelvvel szintén, a tetszőleges számú paraméterrel való hívás példája van kiemelve (C ben void f() mig C++ban void f(void)). Megismerjük a main függvény használati módját is, valamint a return 0 nem kötelező használatát ebben a függvényben.

A bool mint logikai tipus bevezetése is meg van emlitve, mely a C nyelvben még nem szerepelt. Több bájtos stringek fogalma is ismertetésre kerül, és ezek használásához szükséges includeolandó könyvtárak is fel vannak sorolva. A változó deklaráció mint utasítás is szóba jön, azaz minden olyan helyen állhat változódeklaráció, ahol utasítás is állhat. Megismerjük a függvények túlterheltségét is: C ben a függvény neve

azonosítja egyértelműen a függvényt, míg C++ ban a függvény neve és az argumentumlistájuk együttesen azonosítja. Ezért történhet meg az, hogy C++ ban azonos néven létezhet két függvény ha az argumentumlistájuk különböző és egyedi.

Lehetőségünk van arra is a C++ nyelvben, hogy a függvény argumentumainak alapértelmezett értéket adjunk meg.

A C++ nyelven továbbá lehetőségünk van paraméterátadásra referencia tipussal. Ez azt jelenti, hogy a változó címét adjuk át, nem pedig az értékét, és ez nagyon hatékony tud lenni egyes esetekben.

A C++ a cím szerinti paraméter átadást referenciákkal valósítjuk meg, ezt a mechanizmust referencia szerinti paraméterátadásnak nevezzük.

A függvénynek átadott argumentum könnyű megváltoztathatósága a referenciajának csak az egyik alkalmazási területe. ezért például nagy méretű argumentumok:pl struktúrák esetén teljesítménynövekedést érhetünk el.

### Objektumok és osztályok

Az objektum orientáltság bevezetése több alapelvet követett: legyen átláthatóbb a program, a program bonyolultsága ne növekedjen drasztikusan stb.

Az osztályoknak lehetnek példányai, önálló egyedei melyeket objektumoknak nevezünk, és ezek az objektumok tudnak egymást közt kommunikálni. Egy adott témakörhöz, pl a Számlához létrehozhatunk egy osztályt, abban egyedeket, és műveleteket. Ezek egy egységbe záródnak, és értelem szerűen együtt működnek.

Az objektum orientáltság egy szemléletmód, ami a modern felfogást és gondolkodást szemlélteti, valamint az évek műlásával és a programozás fejlődésével alakult ki.

Egységbe zárás a C++-ban: Egy c++ programban megvalósíthatók olyan programok is, melyek tartalmaznak tagváltozókat(struktúra adattagjai), és tagfüggvényeket(osztály részeként, lehetséges osztályon belül vagy osztályon kívül).

Adatrejtés: Az egységbe záráshoz kapcsolódik, mely átláthatóbbá tette a programunkat. Lehetőség van rejteni az adatainkat,

private: részként megadni a struktúrában,

public: részben pedig amit nem védünk.

Osztályon belül alapból elendő csak a public: kihangsúlyozása

Konstruktorok, destrukturok: Az osztályunkba a ha nem irunk konstruktort, akkor alapból létezik egy olyan konstruktor ami nem csinál semmit. Ha irunk paraméteres konstruktort, akkor példányositani tudjuk az osztályunkat mikor objektumot hozunk létre. Tehát a konstruktor szerepe az inicializálás, a destruktur ~ jellel kezdődik és akkor hívódik, ha az objektum megszűnik: felszabadul.

Dinamikus memóriakezelésnek nevezzük azt amikor new utasítással foglalunk helyet a free storeban, a delete szóval pedig töröljük azt.

Dinamikus adattagokat a dinamikus memóriakezelés során hozzuk létre, szóval nem gyártjuk le előre az adattagokat mint a gyár, hanem csak mikor kell, akkor hozunk létre újakat, és foglalunk neki annyi memóriát amennyit kell.

A másoló konstruktor is egy konstruktor, mellyel az a célunk, hogy már meglévő objektum alapján az újonnan létrehozott objektumot inicializáljuk, tehát egy másolatot szeretnénk létrehozni. A másolókonstruktorok átadott argumentumból kell egy másolatot létrehozni, a függvényparamétert inicializáljuk az átadott értékkel.

A friend függvények és osztályok azzal a jellemzővel birnak, hogy feljogosítanak bizonyos más osztálybeli tagfüggvényeket vagy globális függvényeket a saját védett tagváltozói és tagfüggvényei elérésére. Ezt a feljogosítást a friend kulcsszóval tehetjük meg. Az osztály tervezője mondja meg, hogy milyen függvények és vagy osztályok férhetnek hozzá a saját osztályához.

A tagváltozókat inicializálhatjuk konstruktorainkban, a : karakter után felsoroljuk az inicializálni kívánt tagváltozókat. Fontos, hogy az inicializálás kezdő értéket állít be, azaz konstruktorhívás, mikor az értékadás egy meglévő objektumnak ad értéket.

Az osztályon belül létrehozhatók statikus tagok is, melyek tulajdonsága, hogy az adott osztályhoz tartoznak, nem azok objektumaihoz. A statikus tagok lehetnek statikus tagváltozók és tagfüggvények. A statikus tagok objektum nélkül is használhatók. A memóriában egy helyen vannak. A statikus tagfüggvények statikus változókkal dolgoznak. Ezeket is a static kulcsszóval lehet jelezni. Akkor kell statikus tagváltozókat használni, ha az osztály minden objektumára közös változóra van szükségünk. Ez a helyzet a statikus tagfüggvényekkel is. Fontos tudni, hogy a statikus tagváltozók az alkalmazás indításakor inicializálódnak, a main függvénybe való lépés előtt, a globális változókkal egyidejűleg.

## 10.4. Python nyelvi bevezetés

Rövid olvasónapló a BMEPY könyvről.

Python nyelv bemutatása

A python nyelv átlagos számú elérhető funkciókkal a többi nyelvhez képest, nagyon gyors fejlesztési gyorsasággal rendelkező és sok támogatott eszköz számmal rendelkező magas szintű, általános célú programozási nyelv.

Guido van Rossum 1990-ben alkotta meg, a célja a rengeteg pozitív tulajdonsággal rendelkező, magas szintű, dinamikus, objektumorientált és platform független nyelv megalkotása volt.

Fontos jellemzője a Python nyelvnek, hogy fordítóra nincs szükség. A Python interpreter számos platformon elérhető : Windows, MacOS X, Unix...

A Python nyelv szintaxisában nem találhatók meg a jól ismert begin, end, pontosvessző használatra, mivel behúzásalapú a szintaxisa. A sor végéig tart az utasítás. Az értelmező minden sort tokenekre bont amelyek között tetszőleges whitespace karakter lehet. Lefoglalt kulcsszavak közül néhány: and, del, for, if, is, elif, while, print, import, class, break, return..

Megismerkedünk a Tipusok és változók fogalmával a Python nyelvben. Nincs szükség a változók tipusainak explicit megadására, mivel a rendszer futási időben kitalálja azt. Néhány adattípus a Python nyelvben: számok, stringek, listák, szótárak..

A felsoroltak közül az ennek lehet ismeretlen számunkra, ezek az objektumok gyűjteményei vesszővel elválasztva. Ezeket általában zárójelek közé írjuk, vesszővel elválasztva, pl: ('a','b','c'), (), (1,"szia",3)..

A lista elemeit szögletes zárójelek közé írjuk pl: ['a','b','c'], list('abc')..

A szótár kulcsokkal azonosított elemek rendezetlen halmaza. Pl: {'a':1, 'b':5, 'e':1982}..

A pythonban a NULL érték neve a None.

A nyelvben a változóknak nincsenek tipusai, akár több tipusú objektumra is hivatkozhatnak. Pl: a=b=c=1; x,y=y,x (felcseréli a két változót).

A del kulcsszóval törlünk változó hozzárendelést. Itt is léteznek globális illetve lokális változók, alapból lokális.

Néhány listán végezhető műveletre példa: count(e) visszaadja az e előfordulásainak számát,

insert(i,e) beszúrja az e elemet az i-edik helyre,

sort([f]) sorbarendezí(helyben) a lista elemeit az f függvény felhasználásával.

copy() visszatér a szótár egy másolatával,

keys(), iterkeys() a szótár kulcsait tartalmazó listával, illetve iterátorral tér vissza.

clear() kitörli az összes elemet a szótárból.

A könyv további részében megismerkedünk a nyelv eszközeivel. Ide tartozik a print, az elágazás(if elif else), ciklusok(for), while(i kisebb mint 3): print i stb.

Találhatók címek, ugrások is a nyelvben, valamint függvények, melyeket def kulcsszóval definiálhatunk, pl:

```
def hello(): print "Hello" return
```

A python nyelvben is találhozhatunk az osztályok és az objektumok témakörével, melyek klasszikus objektumorientált fejlesztési eljárások. pl: class Koszonto: def MonddSzia(self, ember): print 'Kedves', ember, ', udvozollek.'

A python nyelvben még megismerkedhetünk a modulokkal (mobilkészülékeken való fejlesztés megkönnyítésére), kivételkezeléssel, mely hasonló alapon működik mint a többi objektum orientált nyelv esetén.

És még a végén találhatunk részletes példákat forráskódokkal az eddig tárgyalt témakörök megalapozásának érdekében.

**III. rész**

**Második felvonás**

DRAFT

**Bátf41 Haxor Stream**

A feladatokkal kapcsolatos élő adásokat sugároz a <https://www.twitch.tv/nbatfai> csatorna, melynek permanens archívuma a <https://www.youtube.com/c/nbatfai> csatornán található.

DRAFT

# 11. fejezet

## Helló, Berners Lee!

### 11.1. Java és C++ OOP programozási nyelvek:

Az első felvonásban a C++ program nyelvel foglalkoztunk. Ebben a fejezetben betekintést nyerhetünk a **Java** program nyelvbe, amely az objektum orientált programozási nyelvek családjába tartozik.

Bevezetőként ismerjük meg a jelölések és rövidítések jelentését:

**OOP** = Objektum Orientált Programozás

**OO** = Objektum Orientált

**Példány** = Egy osztály objektum, melyből egyetlen és egyedi létezik. Viszont egy osztálynak több objektuma is létezhet.

Mi az az objektum oriáltság? Erre a kérdésre egy egyszerű válasz az **osztályozás** használata. Tehát az osztályozás azt jelenti, hogy egy programban több kategorizálást hozhatunk létre és ezekkel jól elkülönítve tudunk dolgozni. Egy egyszerűbb példán keresztül demonstrálva az osztályozás és az objektum orientáltság azt jelenti, hogy választunk például egy bizonyos tárgyat, amelynek több tulajdonsága megegyezik egy másik hasonló tárgyéval, jelent esetben legyen egy **autó**. Ha van egy autónk akkor biztosan vannak kerekei, ablakai, súlya, színe, és még sorolhatnánk, tehát bizonyos tulajdonságokkal rendelkezik. De ha nekünk van autónk akkor másnak miért ne lenne, és ha több autó is létezik akkor kell legyen egy gyár ahol elkészülnek az autók. Ha az autó gyárra gondolunk képzeliük el, hogy a gyárban több autót gyártanak naponta, tehát minden egyes autó egy példány, hiszen minden autó kap gyártáskor egy alvázszámot, ezért még ha nagyon is hasonlít két autó akkor sem ugyan arról az autóról beszélünk. A **java**-ban az OO-ságot képzeljük el gyáraknak ahol minden gyárban, azaz oztályban, más-más termék gyártása történik. Az osztályokat, mint a gyárakat, azért hozták létre, hogy megkönyísenek valamit, hiszen egyszerűbb általánosságban foglalkozni egy autó megtervezésével mintha minden egyes autó legyártása után el kellene gondolkozni azon, hogy hogyan is készült az előbb elkészült autó? Hanem egy osztályt, azaz egy tervet hozva létre, így minden autót márktól, modeltől függően gyártanak egymás után.

Ezen egyszerű példán keresztül a **java** nyelv is felfogható nagyon egyszerűen, hiszen egy OO nyelv. A C++ viszont többelvű programozási nyelvnek mondható, mert abban nem feltétlenül kell OO szemléletben programozni. A java nyelv a C nyelvhez nagyon hasonlít szintaksisban hiszen a minden két nyelv egymáshoz időben közel kerül kiadásra. A Java még a régebbi C nyelvhez hasonlít kulcsszavaiban, de a C++ mára már sok fejlesztésen esett keresztül, ezért kulcsszavakban már rövidebb szintaktikát használ. Mindkét nyelvről

elmondható, hogy magas szintű programozási nyelv, azaz felhasználó közeli nyelv, ellentétben az alacsony szintűekkel amelyek gépközeli, a gép által könnyen megértett nyelvet képviselik.

A Java nyelv hírnevét eredetileg a **WWW** oldalakon betölött szerepének köszönheti, ahol appleteket hoztak létre melyeket **HTML** oldalak használtak. Egyszerű példa erre egy bejelentkező ablak felület melynek a működését java nyelven írtak.

Mivel a Java nyelv nagyban hasonlít a C++ nyelvhez ezért érdemes szinkronban beszélni róluk.

Változó:

Olyan tárterület amely értékeket tárol, mely rendelkezik névvel, értékkel, és azonosítóval.

A változóknak több típusa lehet:

- **boolean** : igaz vagy hamis logiaki érték.
- **char** : 16 bites Unicode karakter.
- **int** : 32 bites előjeles egész szám.
- **short** : 16 bites előjeles egész szám.
- **long** : 64 bites előjeles egész szám.
- **float** : 32 bites lebegőpontos racionális szám.
- **double** : 64 bites lebegőpontos racionális szám.

A operátorral a bal oldalán lévő változóhoz a jobb oldalán lévő értéket rendeljük.

Kommenteket is lehet írni amiben magyarázni tudjuk a kódunk mellett működését. Egy soros kommenthez "://" jelet, több soros kommenthez "/\* commnet \*/"-et használunk.

Literálok:

Bármilyen konstans értéket, amit egy változónak értékül adhatunk literálnak nevezünk. Például:

- "Peti" - Sztring literál
- 45 - Egész szám (int) literál
- 21.043 - Tört szám (double) literál
- 53.023f - Tört szám (float) literál

Operátorok:

A java nyelvben is léteznek értékkadó operátorok mint a C++-ban, ezek az értékkadás (=), értékkadás összeadással (+=), értékkadás kivonással (-=), értékkadás szorzással (\*=), értékkadás egész osztással (/=), értékkadás maradékos osztással (%=).

Az aritmetikai operátorok is megtalálhatók, ezek az összeadás (+), kivonás (-), osztás (/), szorzás (\*) és maradékos osztás (%). Érdekességképpen a java nyelvben van előre definiált függvény operátor is ami az előbb említett operátoroknak a tulajdonságával bír, mint például a minus() vagy a plus().

A relációs operátorok a következők, egyenlő (==), nem egyenlő (!=), kisebb (<), nagyobb (>), kisebb egyenlő (<=) és nagyobb egyenlő (>=).

A logikai operátorok a következők, vagy (||), és (&&), negáció (!);

A tömbök vagy listák:

Javaban is mint a C++-ban léteznek tömbök amelyket itt inkább listaként definiálnak. A tömbök lehetnek szöveg és szám tipusúak, a lényegük, hogy több értéket képesek eltárolni. Az **enum** kulcsszóval speciális listát hozhatunk létre, melyre hivatkozhatunk később a kódunkban, amely egy nem bővíthető listaként funkcionál.

Structúra:

A **Struct** kulcsszóval hozhatjuk létre és megadhatunk több típusú változót egy structurában amelyet kézőbb mint egy definiált új tipust használhatunk összesített típusként. Egy struktura elemeire külön-külön hivatkozhatunk a **strukturaNeve.elem** kombinációval.

A Math osztályban előre definiált matematikai műveletek vannak, mint például a sum() azaz az összegzés, az sqrt() azaz a négyzetgyökvonás, az avg() azaz az átlag számítás vagy a rand() a véletlenszerű szám létrehozás.

A vektor már jól ismert a C++-ból. Egy vektor létrehozása a javaban a következőképpen néz ki:

```
Vector <int> num = new Vector <int> ();
Vector <String> str = new Vector <String> ();
```

Ha értékeket akarunk adni egy vektorba azt az **add()** függvénytelhetjük meg.

```
num.add(1);
num.add(2);
num.add(3);

str.add("Név: Pisti");
str.add("Kor: 21");
str.add("Lakhely: Debrecen");
```

A stringes vagy szöveges vektor helyet érdemees egy structurát létrehozni, ha a fent látott tulajdonságokat akarjuk eltárolni és így több ilyen személynek a tulajdonságait is tárolhatjuk, ha tömbként hozzuk létre.

### Struct

Javaban a legkisebb önálló elem az osztály, míg C++-ban nem feltétenül kell osztályokat használni. Tehát egy osztályt a **class** kulcsszóval hozunk létre. Az osztály fogalmát feljebb megérthettük, röviden az azonos típusú dolgok modeljét írja le. Az osztályban létrehozhatunk változókat és függvényeket, metódusokat. Változókat a fent ismertetett típusnév kulcsszót a neve elő írva hozhatunk létre. Egy függvényt pedig úgyan úgy mint egy változót, tehát a típusát megadva, de a név után egy "(" zárójeleket írva majd ezt követve a ")" zárójelek közé írva megadjuk a működését.

Osztályok:

A java egy objektum orientált nyelv. A javaban minden példányokhoz, vagy azaz objektumokhoz és osztályokhoz van társítva. Az osztály egy objektumkészítő, amellyel létrehozhatunk új objektumokat. Olyan dolgokat, tárgyakat, élőlényeket érdemes osztályban definiálni amelyek közös tulajdonságokkal rendelkeznek, mint például az autó, aminek van színe, súlya, kereke stb..

Az osztályok számunkra megkönnyítik a programozást mivel egy átláthatóbb osztályozott környezeten keresztül biztosítva bármely más programozó egy kis tanulmányozással megérte a programíró célját a programmal kapcsolatban. Egy osztályban érdemes olyan függvényeket létrehozni amelyeket sűrűn használ a programunk, ezzel egy helyre csoportosítva őket, programozói nézetben könnyebb tájékozódni a program kódján.

Egy osztály létrehozása a class kulcsszóval majd egy név megadásával történik, példa egy osztályra benne egy függvénnnyel:

```
//java
public class Pelda{

    String szoveg;
    int szam;

    void peldametodus(int parameter) {
        szam+= parameter
    }
}

//C++
Class Pelda{

public:

    void Hello(){
        cout<<"Hello";
    }
}
```

Az osztály szintjei, vagy a jogosultságok:

Ezeket a C++-ban teljes kód részek amelyeket elég egyszer kiírni aztán, minden ami utána van az ahoz a részhez tarozik, egészen addig amíg egy másik rész kulcsszava nem következik. C++-ban minden osztályban egy-egy ilyen rész hozható létre, míg Javaban a függvény vagy változó elé írva lesz definiálva, másszóval Javaban egyfajta tipusként van definiálva.

**Public** rész, amely egy kulcsszó is egyben, tehát Javaban ezt elé írva egy változónak vagy függvénynek elérjük hogy más osztályban is látható legyen, ne csak abban ahol létrehoztuk.

**Private** rész, amely lehetővé teszi, hogy csak az az osztály lássa amelyben létrejött.

**Protected** rész, amely lehetővé teszi a megosztását más osztályokban, vagyis a leszármazott osztályok, vagy örökölt osztályok, C++-ban általában azokat a függvényeket amiket itt definiálunk, friend, azaz barát függvényeknek nevezzük.

A Java támogat úgynevezett félnyilvános tagokat is, ami azt jelenti hogy nincs jelezve milyen jogosultsága van, viszont ezekre nem hivatkozhatunk bármely osztállyal.

Objektumokat, vagy példányokat a következő módon hozhatunk létre:

```
//java
Pelda objektumNeve = new Pelda();
```

```
//C++  
  
Pelda objektumNeve;
```

Az osztálynak létezik egy inicializáló függvénye is amely neve hivatkozik maga az osztály nevére, ezt konstruktornak nevezzük.

```
public class Pelda {  
  
    Pelda () {  
        //Bármilyen inicializálást elvégezhetünk itt  
        System.out.println("A Pelda osztály konstrukrota");  
    }  
}
```

A fenti konstruktornak nincs paramétere és csak egy kiiratás van benne, de több konstruktort is létrehozhatunk, mint a C++-ban több fajta paraméterrel, amelyek alapján már több bejövő adat alapján is inicializálhatunk.

Minden programnak van egy fő része, egy fő függvénye, ezt nevezik a **main**-nek. A létrehozása eltér a két nyelvben. Míg C++-ban csak egyszerű függvényként hozzuk létre, addig Javában egy fő osztály lesz amely a main-t tartalmazza. A main() függvény:

```
//java  
  
public class Main {  
  
    public static void main(String[] argv) {  
  
    }  
}  
  
//C++  
  
int main(String[] argv) {  
  
    return 0;  
}
```

Létrehozhatunk más függvényeket, azaz metódusokat is amelyek majd a main() metódusban hívhatóak meg, ahol öszpontosul a programunk.

Metódusok:

Egy metódust a következőképpen hozunk létre:

**metódus\_Típusa metódus\_Neve () { utasítás1; utasítás2; stb... }**

Egy osztályban egy metódus lehet public, private és protected, ezt a típus előre írni.

A metódusoknak vagy függvényeknek létezik visszatérési értéke. A visszatérési értéke a metódus típusától függ. A void az egyetlen függvény típus amelynek nincs visszatérési értéke.

A metódusokat a {...} között írt utasítások alkotnak, úgynevezett blokkok. minden utasítást pontosveszővel zárunk le. Léteznek elágazások amelyek mindenkor nyelvben azonosan működnek. Elágazást kétféleképpen hozhatunk létre, egyszerűt az **if** kulcsszóval, összetettet a **switch** kulcsszóval, ez utóbbi esetben eseteket hozunk létre amelyeket **case** kulcsszavakkal hozunk létre.

Egy metódusnak létezhet ciklusa, az a rész ami az utasításokat tartalmazza. Több ciklusfjata létezik, a java nyelvben a ciklus fajták megegyeznek a C++ nyelvben is létező ciklusokkal.

A ciklusok:

A **for()** ciklus ami lépésről lépésre végighalad egy megadott struktúrán. Ennek a ciklusnak 3 része van, amelyeket pontosveszővel választunk el egymástól és a ()-zárójelek közé írunk. AZ első rész, a kezdőpont (pl. i = 0) ahonnan kezdjük az elemeket számolni. A második rész a feltétel (pl. i < array.length()), avagy végállapot, azaz meddig olvassuk az elemeket és haladjunk a strukturában. A harmadik rész a lépésköz (pl. i++), tehát az idndexelést milyen nagyságban léptetjük, ha minden elemen végig akarunk haladni, akkor egyesével, ha pedig bizonyos elemekre van szükség akkor több elemet is kihagyva.

A **while()** ciklus, másnéven előtesztelő vagy kezdőfeltételes ciklus, a ciklus ()-zárójelek között lévő feltétel alapján eldönti hogy lefuttatja-e a benne lévő utasításokat vagy sem, ha igaz a feltétel akkor lefut, egyébként hamis érték esetén nem.

A **do {} while();** ciklus, másnéven a háltaltesztelő vagy végfeltételes ciklus, a ciklus hasonlít az előzőre, viszont itt a **do** kulcsszónak köszönhetően minden utasítás lefut a ciklusban, aztán következik a while()-ban lévő feltétel kiértékelése, ha ez hamis akkor nem fut le újra a ciklus, egyébként igaz érték esetén újra lefuttatja a benne lévő utasításokat.

A **foreach** ciklus, amely olyan mint a for() ciklus, viszont ebben nem lehet megadni a lépésközt, és a feltételt, tehát a ciklus a megadott struktura minden elemén végighalad.

Az öröklődés:

Az osztályokat örököltethetjük más osztályokba. Egy osztály létrehozása a class kulcsszóval majd egy név megadásával történik amelyet feljebb olvashattunk. minden függvényt ki kell fejteni amit létrehoztunk egy egyszerű osztályban.

Egy interfész létrehozása az interface kulcsszóval majd egy névvel történik. Majd mint egy osztályban függvényeket hozunk létre amelyeknek csak a deklarációja van. A kifejtést egy öröklített osztályban fogjuk végrehajtani.

```
interface Worker_interface {  
    public void getPayment();  
    public void getWorktime();  
}
```

Egy interfész osztálynál az öröklést a következő formula alapján hajtjuk végre az implements kulcsszóval:

**class az\_Osztályunk\_Neve implements az\_Osztály\_Neve\_Amelyból\_Örökíteni\_Akarunk**

Több interfészt is örökölhet egy osztály, így elkülönítve különböző tulajdonságú osztályokat.

Az absztrakt osztály az előbb említett két osztály ötvözetén alapszik, hiszen ebben az osztály típusban van olyan függvény amely csak definiálva van, és van kifejtett is.

Az a függvényt amelyik csak definiált, a létrehozásnál abstract kulcsszóval kell jelezni, amelyik pedig nincs jelezve azt kötelesek vagyunk kifejteni.

```
class Programmer_abstract extends Worker_abstract {  
  
    public long payment = 210000;  
    public int worktime = 8;  
  
    public void getPayment() {  
        System.out.println("The payment of prog: " + this.payment + ←  
            " Ft.");  
    }  
}
```

Egy abstract osztálynál az öröklést a következő formula alapján hajtjuk végre az extends kulcsszóval:

**class az\_Osztályunk\_Neve extends az\_Osztály\_Neve\_Amelyből\_Örökíteni\_Akarunk**

## 11.2. Bevezetés a mobilprogramozásba

Az alábbi szövegben a Python programozási nyelv foglalkozni a megadott könyv alapján. Kezdetben a nyelv jellemzőiről olvashatunk. A Python nyelv, más programozási nyelvekkel (C++, Java, C) ellentétben elég csak a forrást megadni, ugyanis a fordítási fázisra itt nincs szükség. A Python használható a neves platformokon, mint például Unix, Windows, MacOS ...stb. A nyelv alkalmas prototípus alkalmazások elkészítésére, hiszen sokkal kevesebb erőfelhasználással lehet benne dolgozni mint például a C++-ban vagy a Javaban.

A Python nyelv egy magas szintű programozási nyelv, mégis egyszerűsége hasonlítható az awk vagy Perl nyelkekhez. A nyelvben a Python kódkönyvtárat használjuk. Az ebben lévo modulok, gyorsabbá teszik a programok fejlesztését. A modulok használhatóak rendszerhívásokra, hálózatkeresésre és fájkezelésre is. A nyelv könnyen olvasható alkalmazást készíthetünk, ennek oka az, hogy az adattípusok engedik, hogy összetett kifejezéseket röviden tudunk leírni. Továbbá a más nyelvekkel ellentétbe a kód csoportosításának tagolása tabulátorral vagy új sorral történik és nem kell definiálnunk a változókat vagy az argumentumokat.

A Python nyelv szintaxisa behúzás alapú. Nem szükséges kapcsos zárójel vagy kulcsszavak használata. Egy blokk végét egy behúzással végezzük, ez lehet akár üres sor. Behúzással nem lehet kezdeni egy szkriptet. minden utasítás a sor végéig tart, így nem szükséges a C,C++ vagy a Java-ban ismert ";" használatára. Ha túl hosszú lenne egy sor, akkor "\n" jellel lehet ezt jelölni. Egy behúzás nem érvényes a folytatósorokra.

A sorokat tokenekre bontja a nyelv, ezek között lehet tetszőleges üres karakter. A token fajták: azonosító operátor, kulcsszó ...stb.

A nyelvben megkülönböztetjük a kis és nagy betűket. Pythonban objektumokkal reprezentálunk minden adatot és az ezzel kapcsolatos műveleteket az objektumhatároz meg. Adattípusok hasonlóan a többi nyelvhez itt is lehetnek, sztringek, számok, ennesek, szótárak és listák. A szám típuson belül lehet egész szám, ami lehet lebegő pontos és komplex szám, ezen belül is decimális, oktális vagy hexadecimális. A lebegőpontos szám a C++ ban ismert double-nek felel meg.

A pythonba megtalálható a szekvencia is, ez egy nem negatív egész számokkal indexelt gyűjtő. A sztringet kétféleképpen lehet megadni: idéziskelek között "példa" vagy aposztrófok között 'példa'. Az ennes típusok vesszővel elválasztott gyűjteményei az objektumoknak. A lista lehet több különböző típusú elemekből.

A elemeket szögletes zárójelek között kell felsorolni. A szótár pedig kulcsokkal azonosított rendezetlen halmaza az elemeknek.

A változók Pythonba objektumokra mutató referenciák. A változóknak itt nincs típusa, így különböző típusú objektumokra lehet hívatkozni. A változóknak a "=" egyenlőség jellel lehet értéket adni. A "del" kulcsszóval törölhetünk egy változó hozzárendelést. A nyelvben két féle változót tudunk megadni, a globálist és a lokálist. A globális változót úgy tudjuk létrehozni, hogy "global", ezzel jelezve, hogy egy globális válzotó lesz. Továbbá fontos, hogy a globális változókat a függvény elején kell felvenni. minden más, a függvényben létrehozott változó alapmérézetben lokális lesz.

A Python rövidrezzárt kiértékelést hajt végre, ezt magyarázzuk egy példával: `a < b and b <= c and c == d` kifejezést a Pythonba így tudjuk írni: `a < b <= c == d`. A beépített típusaink között lehetőség van a típus konverzióra (ilyen az int,float,long is). Képesek vagyunk sztringbol is számot képezni, ezt a használt szárendszer megadásával tudjuk. Pl `int('11',8)`. Ennek az értéke a 9 lesz.

Sokféle műveletet tudunk a szekvenciákon végrehajtani, továbbá beépített függvényeket is alkalmazhatunk rajtuk. Ilyen függvényekkel már találkozhattunk számtalanszor más programozási nyelvekben. Ilyenek a max és min függvények, amivel szélsoérteket tudunk meghatározni, vagy a len() függvény amivel mekapjuk egy szekvencia hosszát. A szekvenciákat '+'-al tudjuk össze fűzni. Tudunk kifejezéseket is alkalmazni rajtuk, ezek az 'in' és a 'not in', ezekkel tudjuk megnézni, hogy eleme-e vagy nem eleme a szekvenciának. Az elemeket indexekkel látjuk el, ezek alapján fogjuk majd elérni, ezt 0-tól szoktuk indítani. Van a negatív indexelés, ez a szekvencia végéről kezdve vissza felé halad. Intervallumot a ':' jellel tudunk megadni. Például ha `a=[5,6,7,8,9]` akkor ha `b=a[1:2]` az eredmény az 6,7 értékek lesznek.

A könyven láthatjuk a további műveleteket mint az ismert `pop([j])` függvényt, ezzel eltávolítjuk a 'j'-ik elemet, ha nem adunk meg elemet, akkor az utolsó elem kerül törlésre. A `reverse()` függvény megfodítja a sorrendjét az elemeknek. Az `append()` függvénytel tudjuk bovíteni egy lista végét, míg az `extend()` függvényel egy másik lista elemeit fűzzük egy lista végéhez. A `clear()` függvénytel pedig töröljük az egész listát.

Most pedig következzenek a nyelv eszközei, ezek közül nézzük meg a `print` metódust, ezzel a metódussal változókat vagy egy sztringet írhatunk ki konzolra.

A python nyelvben is van lehetőségünk az elágazásokra. Ezeket csak úgy mint más programozási nyelvekben if, elif és else kulcsszavakkal hívjuk meg. A szintaktika is hasonló: Első a kulcszó, azután a feltétel és végül pedig hogy mi történjen ha a feltétel teljesül.

A másik nagyon fontos nyelvi eszközök a ciklusok. A programozásban az egyik legfontosabb és leghasznosabb metódus a ciklus, ezzel képesek vagyunk adatokat bejárni, feltölteni, megvizsgálni, keresni és még számtalan hasznos dolog.

Elsőnek nézzük a `for` ciklust. A `for` ciklussal akár szótárakban, tömbökben, vektorokon is véig lehetünk, ha a cikust az előbbiek elemire definiáljuk. Fontos még szót ejteni a `range` függvényről. Ez a ciklus futása közben listát generál egész típusú értékekből.

A következő ciklusunk a `while` ciklus, ez a feltételes ciklus, ugyanis a ciklus addig fut, amíg a megadott feltétel teljesül. Bónusz még, hogy a Python a `break` és `continue` kulcsszavakat is ismeri és támogatja. Használhatunk címkéket és ugrásokat is. A címkék kulcsszava a `label`, ezzel tudunk a programban címkéket elhelyezni, amikhez a `goto` parancsal ugorkhatunk.

A Python programozási nyelvben is léteznek a függvények. Ezeket pedig a `def` kulcsszóval definiálhatjuk. A függvényt hasonlíthatjuk egy értékhez, hiszen tovább lehet adni és másfüggvény is megkaphatja. A függvények vannak paraméterei, ezeket számunkra megfelelo megkötésekkel szintaxissal adhatunk meg.

Fontos dolog, hogy a paraméterek az érték alapján adódnak át, kivételt képez a mutable típusú érték. Az argumentumokat a függvény hívásánál tudjuk megadni. A függvények rendelkeznek egy visszatérési értékkel, de ennesekkel is visszatérhet.

Az Osztályokat és objektumokat is támogatja a nyelv. Ez azt jelenti, hogy tudunk osztályokat definiálni, ezeknek az objektumok lesznek a példányai. Az osztály attributumai: függvények és objektumok. Az osztály lépes örökölni és örökölhetni más osztályokból/nak. Az osztályt a "class" kulcsszóval definiáljuk. A osztályok definíciója az, hogy: már definiált osztályok, opcionális listái vesszővel elválasztva. Attributumokat tudjuk bővíteni az osztályban és a példányokban is. Az attribútumokat az osztály törzsén belül határozzuk meg. Az osztály metódusai hasonlóak mint a globális függvények, annyi különbséggel, hogy itt az első paraméternek kötelezően a selfnek kell lennie, ennek az értéke minden az adott objektumpéldány lesz, melyen a függvényt meghívták.

Az `_intin_` egy speciális konstruktor metódus. A nyelv tartalmaz modulokat a könyebb fejlesztés érdekében. Ilyen például az appuifw, a messaging (SMS, MMS), camera, audio vagy a sysinfo modul. A váratlan helyzetekre itt is tudunk alkalmazni kivitelezést. Az adott kódot a try blokkban megadjuk, ezután egy expect jön, ami hiba esetén fog életbelépni.

Végezetül pedig essen pár szó a PYS60 grafikus felületetől, hiszen minden komolyabb program rendelkezik grafikus felülettel. A GUI szolgál az alkalmazáson belül az információk elrejtéséről, megjelenítéséről. De ezen felül még kezeli is a felhasználói interakciókat.

DRAFT

## 12. fejezet

# Helló, Arroway!

### 12.1. OO szemlélet

A következő program objectum orientált ahogyan ezt a címben rövidítve (OO) is láthatjuk. Az OO programozás osztályokkal és objektumokkal fog dolgozni, jelen esetben létrehoz 10 véletlenszerű számot.

Megoldás forrása: [PolarGenerator](#)

A programot három fő részre lehet osztani, a főosztály azaz a main, a kisegítő mellék osztály és ebben a generálást végző függvény.

A programkód részekre bontva a következő:

```
class PolarGen {  
  
    boolean nincsTarolt = true;  
    double tarolt;  
  
    // Constructor .  
    public PolarGen() {  
        nincsTarolt = true;  
    }  
}
```

A PolarGen mellékosztály amelyben egy Constructort láthatunk, ez a függvény minden osztályban létrehozható és inicializálásokat lehet végrehajtani benne. Mivel ez a függvény akkor fut le ha példányosítunk, azaz létrehozunk egy osztály objectumot, ezt majd a main részben, tesszük meg.

A Constructor egy logikai változónak alapértéket definiál, jelen esetben igaz értéket. Az osztályban még létrehozunk egy double típusú változót ami majd a generált számot fogja tartalmazni.

```
/// Generator .  
public double kovetkezo() {  
  
    if(nincsTarolt) {  
        double n1, n2, v1, v2, s;  
  
        do {  
            n1 = Math.random();           // nextDouble() /// .  
        } while (n1 == 0);  
        nincsTarolt = false;  
    }  
    else {  
        double n1, n2, v1, v2, s;  
        do {  
            n1 = Math.random();           // nextDouble() /// .  
        } while (n1 == 0);  
        nincsTarolt = true;  
    }  
    return s;  
}
```

```
n2 = Math.random();           // nextDouble()

    /// . from nextGaussian

    v1 = 2 * n1 - 1;          // v1 = 2 * nextDouble() ←
    - 1;
    v2 = 2 * n2 - 1;          // v2 = 2 * nextDouble() ←
    - 1;

    s = n1 * n2 + v1 * v2;

} while ( s > 1 );

double r = Math.sqrt((-2 * Math.log(s)) / s);
tarolt = r * v2;
nincsTarolt = !nincsTarolt;
return r * v1;
} else {

    nincsTarolt = !nincsTarolt;
    return tarolt;
}
}

}
```

A kovetkezo() nevű metódus matematikai számítás alapján hoz létre számokat. Az if elágazás eldönti, hogy van-e már legenerált szám, ha nincs akkor belép a ciklusba és létrehoz 2 változót aminek a random() függvényel generál számot, majd jön a matematikai számítás a while() ciklusban hogy ha a szám nagyobb mint 1 akkor elvégez egy műveletet, majd értéket ad a tarolt nevű változónak. Majd a nincsTarolt változó logikai értékét megváltoztatja, aztán vissza ad egy számot a return kulcsszóval.

Ha már volt generált szám akkor az else ágra lép, ahol a nincsTarolt változó logikai értékét megváltoztatja és vissza adja a tarolt valtozo értékét a return kulcsszóval.

```
public class PolarGenerator {

    public static void main(String[] args) {

        /// Instance .
        PolarGen g = new PolarGen();

        for(int i=0; i<10; ++i)
            System.out.println(g.kovetkezo());
    }
}
```

A PolarGenerator osztály, a főosztály, ahol a main() függvényben létrehoz egy a PolarGen osztályhoz tartozó példányt, a g nevű objektumot, majd egy for() ciklust indít, amely addig fut amíg 10 számot nem generált a kovetkezo() nevű metódussal, s ezeket a számokat kiírja.

A következő programcsipet a JDK beéített metódusa amely megtalálható a java.util.Random forrásban a nextGaussian néven.

```
public class NextGuassian {

    private double nextNextGaussian;
    private boolean haveNextNextGaussian = false;

    public double nextGaussian() {
        if (haveNextNextGaussian) {
            haveNextNextGaussian = false;
            return nextNextGaussian;
        } else {
            double n1, n2, v1, v2, s;
            do {
                n1 = Math.random(); // nextDouble()
                n2 = Math.random();

                v1 = 2 * n1 - 1; // between -1.0 and 1.0
                v2 = 2 * n2 - 1; // between -1.0 and 1.0
                s = v1 * v1 + v2 * v2;
            } while (s >= 1 || s == 0);
            double multiplier = StrictMath.sqrt(-2 * StrictMath.log(1 - s));
            nextNextGaussian = v2 * multiplier;
            haveNextNextGaussian = true;
            return v1 * multiplier;
        }
    }
}
```

Ebben a forrásban egy hasonló generálást láthatunk mint a kovetkezo() metódusban amit fent láthattunk, de ez valamennyivel pontosabb számítást végez.



```
50 public class PolarGenerator {
51
52    public static void main(String[] args) {
53
54        // Instance .
55        PolarGen g = new PolarGen();
56
57        for(int i=0; i<10; ++i)
58            System.out.println(g.kovetkezo());
59    }
60
61 }
62 <

```

Console X  
<terminated> PolarGenerator [Java Application] C:\Program Files\Java\...  
0.24828676011681017  
0.7350252634398798  
-0.6136510428602789  
-0.5845450852560363  
-1.2829439936544889  
0.6688946117205378  
0.5728380295855658  
0.29464624366891806  
-1.2996984683782467  
0.47681762723548066

A következő program pedig ugyan az mint amit láthattunk feljebb csak C++ nyelven:

Link: <https://github.com/RubiMaistro/codes-prog2/blob/master/Arroway/PolarGen.cpp>

## 12.2. Gagyi

Megoldás forrása: [Gagyi](#)

Érdekesség a javáról:

A java feltételezi hogy kis számokat kell használnia ezért futáskor létrehoz a Thread pooljában egy rész tárhelyet, amelybe -128-tól 127-ig fog eltárolni összesen 255 egész számot és mindegyikhez rendel egy memóriacímet.

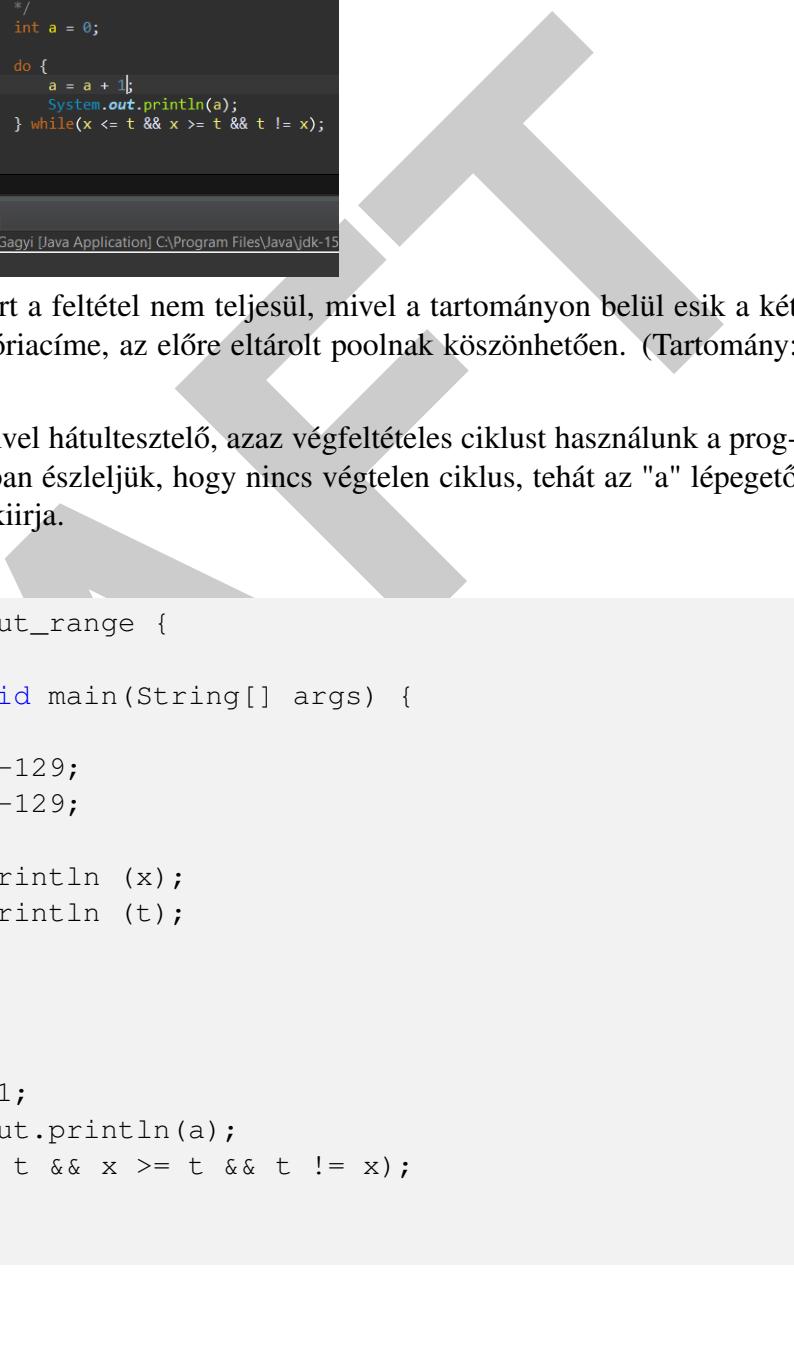
Ha a -128 és 127 közötti intervallumból szeretnénk értéket adni egy változónak, akkor az előre eltárolt poolból fog hozzá memória címet rendelni. Amikor megtörténik a kérés, a java meghívja az Integer.value.of függvényt amely kiosztja a memóriacímet.

Ha ezen tartományon kívül rendelünk értéket a változóhoz, például -129-et vagy akár 128-at akkor ugyan ez a függvény létrehoz egy új objektumot.

Ezek alapján ha x és t változó értéke egyenlő és ezen értéke az adott tartományon belül vannak, akkor azonos lesz a két változó címe. Viszont ha x és t értéke azonos, de nem a -128 és 127 tartományból lesz, akkor a két változó címe nem fog megegyezni, mivel két külön objektum fog létrejönni ezeknek a számoknak.

Első esetben:

```
public class Gagyi {  
  
    public static void main(String[] args) {  
  
        Integer x = -128;  
        Integer t = -128;  
  
        System.out.println (x);  
        System.out.println (t);  
  
        int a = 0;  
  
        do {  
            a = a + 1;  
            System.out.println(a);  
        } while(x <= t && x >= t && t != x);  
    }  
}
```



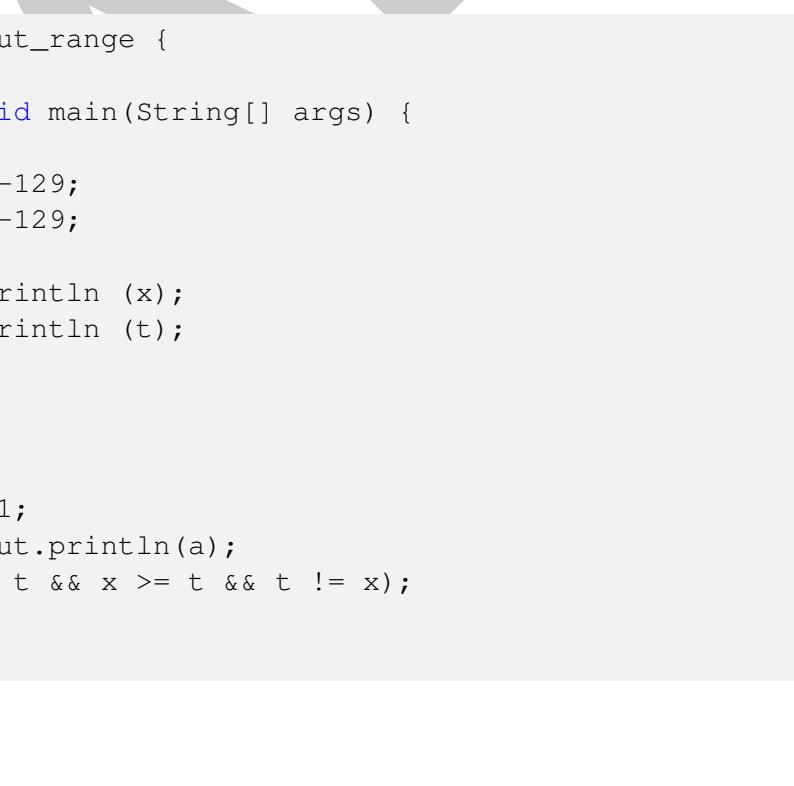
```
1 package Feladatok;
2
3 public class Gagyi {
4
5     public static void main(String[] args) {
6
7         Integer x = -128;
8         Integer t = -128;
9
10        /*System.out.println (x);
11        System.out.println (t);
12        */
13        int a = 0;
14
15        do {
16            a = a + 1;
17            System.out.println(a);
18        } while(x <= t && x >= t && t != x);
19    }
20 }
```

Console x  
<terminated> Gagyi [Java Application] C:\Program Files\Java\jdk-15  
1

Nem kerül a program végtelen ciklusba mert a feltétel nem teljesül, mivel a tartományon belül esik a két szám értéke, ezért azonos a két szám memóriacíme, az előre eltárolt poolnak köszönhetően. (Tartomány: -128 - 127)

Viszont láthatjuk a Consoleban egy 1-est mivel hárultesztelő, azaz végfeltételes ciklust használunk a programban, ez elősegíti azt hogy már Consoleban észleljük, hogy nincs végtelen ciklus, tehát az "a" lépegető változó csak egy futást fog számolni, majd kiirja.

Második esetben:

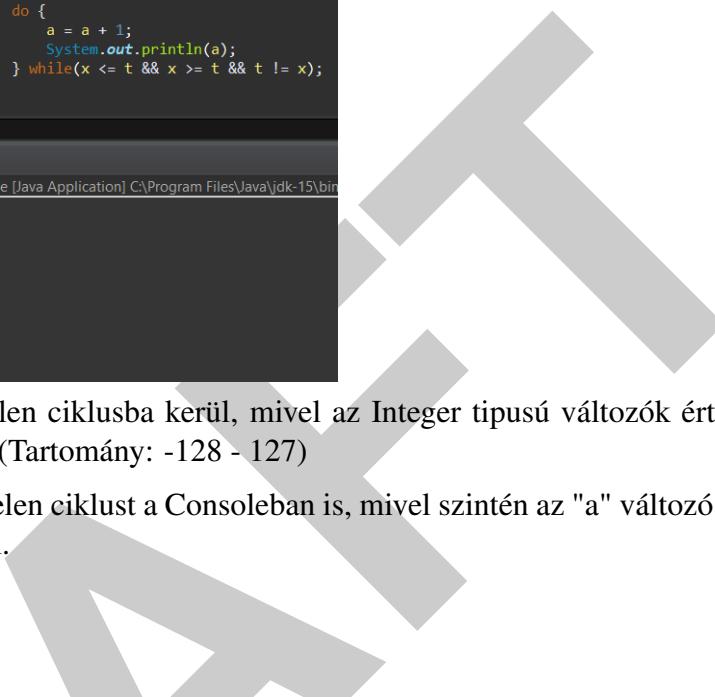


```
public class Gagyi_out_range {
    public static void main(String[] args) {
        Integer x = -129;
        Integer t = -129;

        System.out.println (x);
        System.out.println (t);

        int a = 0;

        do {
            a = a + 1;
            System.out.println(a);
        } while(x <= t && x >= t && t != x);
    }
}
```



```
1 package Feladatok;
2
3 public class Gagyi_out_range {
4
5     public static void main(String[] args) {
6
7         Integer x = -129;
8         Integer t = -129;
9
10        long a = 0;
11
12        do {
13            a = a + 1;
14            System.out.println(a);
15        } while(x <= t && x >= t && t != x);
16    }
17 }
```

Console X  
Gagyi\_out\_range [Java Application] C:\Program Files\Java\jdk-15\bin  
11563232  
11563233  
11563234  
11563235  
11563236  
11563237  
11563238  
11563239  
11563240  
11563241

Amikor teljesül a while() feltétele és végtelen ciklusba kerül, mivel az Integer típusú változók értéke az előre definiált pool tartományon kívül esik. (Tartomány: -128 - 127)

Ebben a programban már láthatjuk is a végtelen ciklust a Consoleban is, mivel szintén az "a" változó értéke minden ciklus lefutás után növekszik eggyel.

## 12.3. Yoda Conditions

Ez egy programozási szintaxis.

A nevét a Csillagok Háborújából ismert Yoda mesterről kapta, aki másképp beszéli az angolt, mivel nem szokványosan beszéli az angol nyelvet, hanem a szavakat mondattanilag helytelenül használja. Tehát felcseréli a szavak sorrendjét, viszont ettől eltekintve még érthető amit mondani akar.

Példa egy mondatra és Yoda mester aranyköpésére:

You have become powerful, I sense the dark side in you.

Become powerful you have, the dark side in you I sense.

Yoda Master Conditions Generator: <https://lingojam.command/EnglishtoYoda>

Ezt átvihetjük a programozásba is, mivel a java lehetővé teszi. A továbbiakban példát láthatunk ilyen fajta programozási szintaxisra is.

Megoldás forrása: [Yoda Condition](#)

A program ami java.lang.NullPointerException-re leáll, ha nem követjük a Yodaconditions-t:

```
public class Yoda {

    public static void main(String[] args) {

        String peldastring = null;

        if (peldastring.equals("example")) {
```

```
        System.out.println("code example");
    }
}
}
```

A programban az összehasonlító kifejezésben megcseréljük a változót és a konstanst. Eredetileg a konstanshoz hasonlítjuk a változót, azaz a konstans a bal míg a változó a jobb oldalon helyeszkedik el. De a Yoda feltételek szerint fordítva, azaz a konstans lesz a jobb oldalon és a változó a bal oldalon.

The screenshot shows a Java code editor with the following code:

```
1 package Feladatok;
2
3 public class Yoda {
4
5     public static void main(String[] args) {
6
7         String peldastring = null;
8
9         if (peldastring.equals("pelda")){
10             System.out.println("Pelda kod");
11         }
12     }
13 }
14
15 }
```

Below the code editor is a terminal window titled "Console". It shows the output of the program execution:

```
<terminated> Yoda (1) [Java Application] C:\Program Files\Java\jdk-15\bin\javaw.exe (2020. szep. 30. 14:16:31 – 14:16:32)
Exception in thread "main" java.lang.NullPointerException: Cannot invoke "String.equals(Object)" because "peldastring" is null
at Feladatok.Yoda.main(Yoda.java:9)
```

A java nem képes String tipust összehasonlitani null tipussal, ezért dob hibát. Ezzel a módszerrel elkerülhetjük a null Stringeket.

## 12.4. EPAM: Java Object metódusok

A javaban találhatunk beéített Object metódusokat amelyeket szabadon használhatunk, de ha számunkra egy picit más szeretnénk hogy végezzen akkor átírhatjuk, azaz felülírhatjuk a programunkban a már beéített metódust.

Nem minden esetben helyes számunkra az előre definiált metódus, mert nem biztos hogy optimálisan fog működni a programunk. Ekkor ajánlott és kötelező is felüldefiniálni a metódust.

Megoldás forrása: [Java Object Methods](#)

```
public class Methods {

    public Methods() {
        System.out.println("Methods Constructor.");
    }

    @Override
    public String toString() {
        System.out.println("Methods: " + getClass().getName() + " "
            ", toString()=" + super.toString());
        return "";
    }

    @Override
    public int hashCode() {
```

```
        return super.hashCode();
    }

    @Override
    public boolean equals(Object obj) {
        return super.equals(obj);
    }

    @Override
    protected Object clone() throws CloneNotSupportedException {
        return super.clone();
    }

    @Override
    protected void finalize() throws Throwable {
    }
}
```

A fenti kódban létrehoztunk egy Methods nevű osztályt amiben függvények vannak. Majd minden beéített függvény előtt egy Override kulcsszót, amely azt jelenti hogy az utána következő beéített függvény a programban felüldefiniált.

Mivel az osztályban nincs létrehozva az egy Constructoron kívül semmi más, ezért a beéített függvényeket nincs értelme felüldefiniálni. Ez a program csak egy osztálynak a váza.

Amikor egy osztályban több változót és függvényt hozunk létre akkor bizonyos metódusokat érdemes felüldefiniálni.

Például a hashCode() függvényt érdemes felüldefiniálni mivel minden objektumnak más kell legyen a hash kódja. Ebben az esetben ha egy hashcode alapú mapel dolgozunk akkor a felüldefiniálással elkerülhetjük az objektumok összekeverését.

```
public class Main {

    public static void main(String[] args) {

        Methods obj = new Methods();
        Methods obj1 = new Methods();

        obj.toString();
        System.out.println(obj.getClass());
        System.out.println(obj.equals(obj1)); // ←
            összehasonlitja a hash kódokat

        System.out.println(obj.hashCode());
        System.out.println(obj1.hashCode());
    }
}
```

A main() osztályban létrehozunk objektumokat és meghívjuk a definiált függvényeket.

Komplexebb program a beéített függvényekkel: <https://github.com/RubiMaistro/codes-prog2/blob/master/EPAM/Java%20Obj%20Methods/Ember.java>

A beípitett függvények optimalizálás okának a magyarázata:

A hashCode() metódus. minden objektum és változó amit létrehozunk rendelkezik egy azonosítóval, ezt nevezzük hash kódnak. (Egy komplex programban mindenkor felül kell definiálni a metódust, hogy optimálisan működjön a programra nézve.)

A toString() metódus. A függvény standard visszatérési értéke: class packageNév.osztalyNév és hash kód.

Ha változtatni szeretnénk a visszatérési értékén, akkor érdemes felüldefiniálni.

Az equals() metódus. Ez a függvény az összehasonlító függvény. Tehát ahogyan a hash kódnál úgy itt is, ha már a program komplex akkor több a hibalehetőség, így érdemes a programhoz optimalizálni.

A clone() metódus. Ez a függvény képes lemasolni egy objektumot teljesen, hogy a hash kódja is teljesen megegyezik a másolandó objektuméval.

A wait() metódus. Arra kényszeríti az aktuális processzor szálat hogy megvárjon egy másikat amíg az nem végez.

A notify() és notifyAll() metódusok. Az első egyet a második pedig az összes várakozó szálat felébreszti amelyeket a wait() függvényvel várakoztatunk a programban.

Lássunk példát egy néhány beépített java object metódusra:

```
75 public class Ember {  
76     ...  
77     public static void main(String[] args) throws CloneNotSupportedException {  
78         Human Atti = new Human("Attila", 22);  
79         Human Attila = (Human)Atti.clone();  
80         Human Laci = new Human("László", 21);  
81  
82         Laci.first.Name = "Nagy";  
83  
84         Attila.getNew();  
85  
86         System.out.println("\nToString");  
87         System.out.println(Atti.toString());  
88         System.out.println(Attila.toString());  
89         System.out.println(Laci.toString());  
90  
91         System.out.println("\nHashCode");  
92         System.out.println(Atti.hashCode() + " " + Attila.hashCode());  
93         System.out.println(Attila.hashCode() + " " + Laci.hashCode());  
94  
95         System.out.println("\nClone");  
96         System.out.println(Laci.name);  
97         System.out.println(Laci.first.Name + " " + Laci.name);  
98  
99         System.out.println("\nEquals");  
100        System.out.println("Atti = Attila ?: " + Atti.equals(Attila));  
101        System.out.println("Attila = Laci ?: " + Attila.equals(Laci));  
102  
103    }  
104 }  
105 }
```

<terminated> Ember [Java Application] C:\Program Files\Java\Attila  
ToString  
toString: Human : 1971585000  
toString: Human : 1971585000  
HashCode  
1971585000 1971585000  
1971585000 -1907811185  
Clone  
László  
Nagy László  
Equals  
Atti = Attila ?: true  
Attila = Laci ?: false

Részletesen a metódusok felüldefiniálásánál (Override) lehet követni a függvények működését, a linkben megtalálható a részletes programkód.

## 13. fejezet

# Helló, Liskov!

### 13.1. EPAM: Interfész, Osztály, Absztrak Osztály

A java egy objektum orientált nyelv. A javaban minden példányokhoz, vagy azaz objektumokhoz és osztályokhoz van társítva. Az osztály egy objektumkészítő, amellyel létrehozhatunk új objektumokat. Olyan dolgokat, tárgyakat, élőlényeket érdemes osztályban definiálni amelyek közös tulajdonságokkal rendelkeznek, mint például az autó, aminek van színe, súlya, kereke stb..

Az osztályok számunkra megkönnyítik a programozást mivel egy átláthatóbb osztályozott környezeten keresztül biztosítva bármely más programozó egy kis tanulmányozással megérte a programíró célját a programmal kapcsolatban. Egy osztályban érdemes olyan függvényeket létrehozni amelyeket sűrűn használ a programunk, ezzel egy helyre csoportosítva őket, programozói nézetben könnyebb tájékozódni a program kódban.

Az osztályoknak kifejtettségnek megfelelően három fő típusa sorolható.

Megoldás forrása: [Classes](#)

Osztály ( Simple Class ) :

Egy osztály a következőképpen néz ki:

```
class Worker_Simple {  
    public long payment;  
    public int worktime;  
  
    public Worker_Simple() {  
        payment = 210000;  
        worktime = 8;  
    }  
  
    public void getPayment() {  
        System.out.println("The payment of prog: " + payment + " Ft ←  
        .");  
    }  
  
    public void getWorktime() {
```

```
        System.out.println("The worktime of prog: " + worktime + " ↵
                         hour/day.");
    }
}
```

Egy osztály létrehozása a class kulcsszóval majd egy név megadásával történik. Létrehoztuk a Worker\_Simple osztályt ami tartalmazni fog két változót, a payment és a worktime változókat.

Az osztálynak létezik egy inicializáló függvénye is amely neve hivatkozik maga az osztály nevére, ezt konstruktornak nevezzük, jelen esetben kezdőértéket adunk a két változónak. Majd két függvény is helyet foglal amelyek feladata egy kiiratás lesz a változókra hivatkozva.

A teljes kód a main() fő függvényvel megtalálható a következő linken:

<https://github.com/RubiMaistro/codes-prog2/blob/master/EPAM/Class%20Types/SimpleClass.java>

Interfész ( Interface ) :

Egy interfész deklarációja a következőképpen néz ki.

```
interface Worker_interface {
    public void getPayment();
    public void getWorktime();
}
```

Létrehozása az interface kulcsszóval majd egy névvel történik. Majd mint egy osztályban függvényeket hozunk létre amelyeknek csak a deklarációja látható. A kifejtést egy örökített osztályban fogjuk végrehajtani.

```
class Programmer_interface implements Worker_interface {

    public long payment = 210000;
    public int worktime = 8;

    public void getPayment() {
        System.out.println("The payment of prog: " + this.payment + ↵
                           " Ft.");
    }

    public void getWorktime() {
        System.out.println("The worktime of prog: " + this.worktime ↵
                           + " hour/day.");
    }
}
```

Egy osztály az implements kulcsszóval fogja örökölni az interfész osztály tulajdonságait. Több interfészt örökölhet egy osztály, ebben az esetben az interfészek neveit veszszővel elválasztva felsorolhatjuk.

Jelen esetben egy interfészt örökolt az osztály amelyben már az előre definiált függvényeket kifejtettük.

A teljes programkód a main() függvénnnyel együtt megtalálható a következő linken:

<https://github.com/RubiMaistro/codes-prog2/blob/master/EPAM/Class%20Types/InterfaceClass.java>

Absztrakt ( Abstract class ) :

Egy absztrakt osztály a következőképpen néz ki:

```
abstract class Worker_abstract {  
    public abstract void getPayment();  
  
    public void getWorktime(int worktime) {  
        System.out.println("The worktime of prog: " + worktime + " ↵  
                           hour/day.");  
    }  
}
```

Az absztrakt osztály az előbb említett két osztály ötvözetén alapszik, hiszen ebben az osztály típusban van olyan függvény amely csak definiálva van, és van kifejtett is.

Azt a függvényt amelyik csak definiált, a létrehozásnál abstract kulcsszóval kell jelezni, amelyik pedig nincs jelezve azt kötelesek vagyunk kifejteni.

```
class Programmer_abstract extends Worker_abstract {  
  
    public long payment = 210000;  
    public int worktime = 8;  
  
    public void getPayment() {  
        System.out.println("The payment of prog: " + this.payment + ↵  
                           " Ft.");  
    }  
}
```

Egy osztály amelyik absztrakt osztály tulajdonságokat örökölt exteds kulcsszóval örököltetünk.

A Programmer\_abstract osztályban láthatjuk a getPayment() függvény kifejtését amit az absztrakt osztályban definiáltunk. Létrehozunk változókat amelyeket a függvények használhatnak. Ebben az osztályban ugyan nem jelenik meg az absztrakt függvényben kifejtett függvény de az osztály megörökli és használni tudja.

A teljes programkód a main() függvénnnyel együtt megtalálható a következő linken:

<https://github.com/RubiMaistro/codes-prog2/blob/master/EPAM/Class%20Types/AbstractClass.java>

Mindhárom osztály ugyan azt fogja csinálni ezért ugyan az a kimenetük is, viszont csak kódban térnek el. A Consoleban a futtatásuk után a következő figyelhető meg.

```
22 public class SimpleClass {  
23     public static void main(String[] args) {  
24         Programmer_Simple obj = new Programmer_Simple();  
25         obj.getPayment();  
26         obj.getWorktime();  
27     }  
28 }  
  
D:\AbstractClass.java x  
22 public class AbstractClass {  
23     public static void main(String[] args) {  
24         Programmer_abstract mypro = new Programmer_abstract();  
25         mypro.getPayment();  
26         mypro.getWorktime(8);  
27     }  
28 }  
  
D:\InterfaceClass.java x  
22 public class InterfaceClass {  
23     public static void main(String[] args) {  
24         Programmer_interface mypro = new Programmer_interface();  
25         mypro.getPayment();  
26         mypro.getWorktime();  
27     }  
28 }  
  
Console x  
<terminated> SimpleClass [Java Application] C:\Program Files\Java\jdk-15\bin\javaw.exe (2022-07-11T11:25:20)  
The payment of prog: 210000 Ft.  
The worktime of prog: 8 hour/day.
```

## 13.2. EPAM: Liskov féle helyettesíthetőség elve, öröklődés

A Liskov elv alapján az örököltetett osztályoknak cserélhetőknek kell lenniük az alaptípusukra, azaz az alaposztályra való hivatkozásnak helyettesíthetőnek kell lennie egy örököltetett osztályra, anélkül, hogy ez befolyásolná a viselkedést.

Az elv lényege, hogy ha egy B típusú osztály az A típusúnak egy alosztálya akkor a B típusú objektumok helyettesíthetőek A típusú objektumokkal, így megtartva a program tulajdonságait.

Az öröklődés fogalma egy Java programozónak nem idegen, ez tulajdonképpen kulcsfontosságú a java programokban. Az öröklés két osztály között jön létre, így hivatkozhatunk egy ōs osztályra és egy alosztályára. minden ami az ōs osztályban létrejött azt öröklí az alosztálya is, ezért a Liskov elv felhasználásával és érvénybelépéssel helyettesíthetjük egy osztály objektumát egy főbb osztályéval.

Megoldás forrása: [Liskov féle helyettesíthetőség elve](#)

Az örököltetést és a Liskov helyettesíthetőség elvet bemutató kódrész a következő:

```
class Vehicle {  
    public Vehicle() {  
        System.out.println("Vehicle.");  
    }  
  
    public void start() {  
        System.out.println("also.");  
    }  
}
```

```
class Car extends Vehicle {
    public Car() {
        System.out.println("Car.");
    }

    @Override
    public void start() {
        System.out.println("masodik.");
    }
}

class Supercar extends Car {
    public Supercar() {
        System.out.println("Supercar");
    }

    @Override
    public void start() {
        System.out.println("harmadik.");
    }
}
```

Láthatjuk az osztályok közötti összefüggéseket, az ősosztályokat és az alosztályokat. Létrehoztunk egy Vehicle osztályt aminek egy konstruktora van és egy start() függvénye, mindenkor egy egyszerű kiiratás. Majd létrehozunk egy új osztályt, a Car osztályt ami örököl a Vehicle osztály tulajdonságait, az örököltetést extends kulcsszóval végezzük el. Majd egy harmadik osztályt hozunk létre, a Supercar osztályt ami örökli a Car osztály tulajdonságait. Ezen utóbbi osztályokban is jelen lesz a start() függvény, ezeket viszont felül kell definiálni, ezzel is megkülönböztetve az osztályokat, a felüldefiniálást a függvény elő írt Override kulcsszóval végezzük el.

```
public class Inherit {
    public static void main(String[] args) {

        Vehicle firstVehicle = new Supercar();
        firstVehicle.start();
        System.out.println(firstVehicle instanceof Car);

        Car secondVehicle = (Car) firstVehicle;
        secondVehicle.start();
        System.out.println(secondVehicle instanceof Supercar);

        /* Nem lehet hivatkozni:
           Supercar thirdVehicle = new Vehicle();
           thirdVehicle.start();
        */
    }
}
```

}

A Main() metódusban először létrehozunk egy objektumot a Vehicle osztálynak ami Supercar osztály típusú objektum lesz. Ezen típusú példányosítás azt eredményezi, hogy a Vehicle osztályba létrehozott objektum végig fog haladni az alosztályokon, az az örökölés folyamon addig amíg el nem ér a Supercar osztályhoz. A fordító minden osztályban megvizsgálja, hogy a létrehozott objektum olyan típusú-e, ez annyit jelent, hogy megnézi, hogy a Supercar osztály megegyezik e a Vehicle osztályal, nem, akkor megy tovább, jön a Car osztály, ezzel megegyezik-e, nem, akkor megy tovább, jön a Supercar osztály, megegyezik-e a Supercar a Supercarral, igen akkor jó helyen járunk, létrejön az objektum, mivel létezik az osztály. Ezen hosszúnak tűnő folyamat közben a konstruktorknak lefutnak, innen a három kiíratás, azaz minden osztályban létrejöttek objektumok, viszont ezek fölülíródtak.

A program futtatás után a következő kimenetet produkálja:

```
35
36 public class Inherit {
37
38     public static void main(String[] args) {
39
40         Vehicle firstVehicle = new Supercar();
41         firstVehicle.start();
42         System.out.println(firstVehicle instanceof Car);
43
44         Car secondVehicle = (Car) firstVehicle;
45         secondVehicle.start();
46         System.out.println(secondVehicle instanceof Supercar);
47
48         /*
49         Supercar thirdVehicle = new Vehicle();
50         thirdVehicle.start();
51         */
52     }
53 }
54 <
```

Console x  
<terminated> Inherit [Java Application] C:\Program Files\Java\jdk-15\bin\javaw.exe (2020. s  
Vehicle.  
Car.  
Supercar  
harmadik.  
true  
harmadik.  
true

A második példányosításnál a seconVechicle Car osztály típusú objektum megkapja a firstVehicle objektum típusát, viszont már castoljuk Car osztály típusúvá és úgy adjuk át. A konstruktorknak is lefutnak, itt már csak a Car osztályig. Végül az utolsó példányosítást ami kommentben van nem tudjuk végrehajtani, mivel a Supercar osztálynak nem adhatunk meg magasabb osztályból típust, mivel az öröklés folyamon visszafelé nem lehetséges a példányosítás, hiszen magasabb szintű osztályból alacsonyabb osztályba objektumot nem lehet helyettesíteni.

### 13.3. Liskov helyettesítési elv sértése

A Liskov elvet fentebb megismertettük.

Az elv megsértése azt jelenti hogy az örököltettségi típusában hiba keletkezik az öröklődés miatt.

Tehát ha egy osztály örököli egy űrosztály típusát, akkor minden függvényt és változót ugyanúgy bármelyik osztályra hivatkozva használhatunk, azokon kívül amelyeket az örököltettségi osztályban definiáltunk, mert ezeket csak arra az osztályra hivatkozva használhatjuk amelyikben létrehoztunk őket.

Megoldás forrása: [Liskov elv sértése](#)

```
class Madar {  
    public void repul() {  
        System.out.println("A " + super.getClass().getSimpleName() ←  
            + " repül.");  
    };  
  
    class Golya extends Madar {};  
    class Strucc extends Madar {};
```

Jelen esetben a Strucc osztályban a Madar osztály repul() függvénye logikai hiba lesz a programban, mivel a strucc nem tud repülni.



The screenshot shows a Java code editor with a dark theme. The code is as follows:

```
19 public class LSPsértés {  
20  
21●  public static void main(String[] args) {  
22  
23      Prog kod = new Prog();  
24      Madar mad = new Madar();  
25      kod.fgv (mad);  
26      Golya golya = new Golya();  
27      kod.fgv (golya);  
28      Struc struc = new Struc();  
29      kod.fgv(struc);  
30  
31  }  
32 }  
33 }  
34 <
```

Below the code is a terminal window titled "Console". The output is:

```
<terminated> LSPsértés [Java Application] C:\Program Files\Java\jd...  
A Madar repül.  
A Golya repül.  
A Struc repül.
```

A repul() függvény kiiratja az osztály nevét aztán pedig, hogy repül. Ez a Strucc osztályban is le fog futni, ha pedig komplexebb programot szeretnénk konstruálni jelen programból, akkor hibásan működhet.

Általában a négyzet és téglalap örökkítési példájában szokták a Liskov elv sértését bemutatni, de ez egy olyan példa amelyet bárki megérthet.

A program elérhető C++ nyelven is a következő linken: <https://github.com/RubiMaistro/codes-prog2/blob/master/Liskov/LSP%20elv%20s%C3%A9rt%C3%A9s.cpp>

```
class Madar {  
    public:  
        void fly(string nev) {  
            nev = nev.erase(0, 6);  
            cout << "A " << nev << " repül.\n";  
        }  
  
        string nev = typeid(Madar).name();  
    };  
  
class Golya : public Madar{  
    public:  
        string nev = typeid(Golya).name();  
    };
```

```
class Struc : public Madar{
public:
    string nev = typeid(Struc).name();
};
```

A Java és C++ program csipet szintaxisban tér el egymástól. Ezért például az osztály egyszerű nevét C++-ban egy kicsit másképp kell lekérni a kódban.

## 13.4. Szülő-gyerek osztály

Az osztályok örökíthetőség által létrehozott gyerek osztályban bármilyen újabb megvalósított konstrukció már nem használható a nála feljebb lévő, szülő osztályban.

Megoldás forrása: [Parent Child Classes](#)

A program rámutat arra, hogy a gyerek osztály (Child{ }) rálát a teljes szülő osztályra (Parent{ }), viszont ez fordítva nem igaz mivel a gyerek osztályban létrehozott child() függvényt a szülő osztály képtelen használni, hiszen elérhetetlen számára.

```
class Parent {
    public void parent() {
        System.out.println("Parent class.");
    }
}

class Child extends Parent {
    public void child() {
        System.out.println("Child class");
    }
}

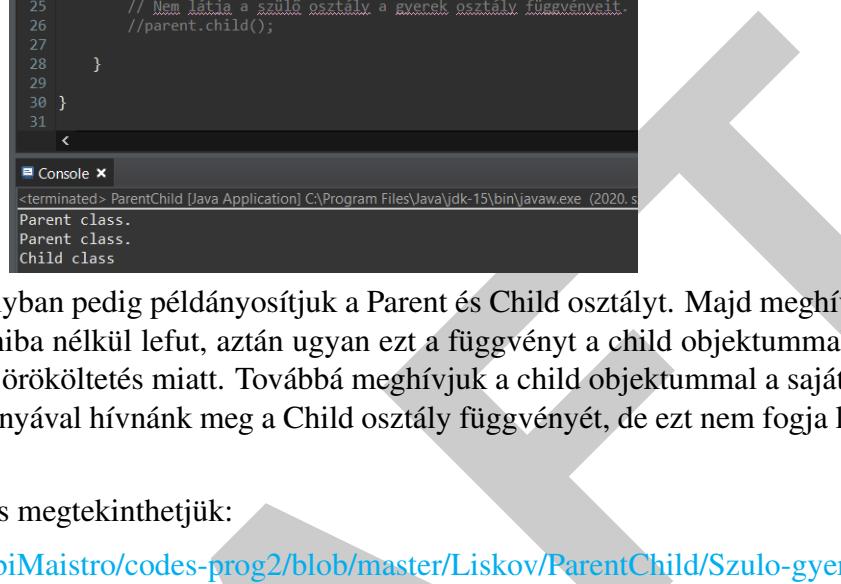
public class ParentChild {

    public static void main(String[] args) {

        Parent parent = new Parent();
        Child child = new Child();

        parent.parent();
        child.parent();
        child.child();
        // Nem látja a szülő osztály a gyerek osztály függvényeit.
        // parent.child();
    }
}
```

A programban létrehozunk egy Parent osztályt amit a Child osztály örökölni fog az extends kulcsszó révén. Majd mindkét osztályban létrehoztunk egy-egy függvényt.



```
14
15 public class ParentChild {
16
17     public static void main(String[] args) {
18
19         Parent parent = new Parent();
20         Child child = new Child();
21
22         parent.parent();
23         child.parent();
24         child.child();
25         // Nem látja a szülő osztály a gyerek osztály függvényeit.
26         //parent.child();
27
28     }
29
30 }
31 <
```

Console x  
<terminated> ParentChild [Java Application] C:\Program Files\Java\jdk-15\bin\javaw.exe (2020.s  
Parent class.  
Parent class.  
Child class

A main() fő függvény osztályban pedig példányosítjuk a Parent és Child osztályt. Majd meghívjuk a parent objektum függvényét, ami hiba nélkül lefut, aztán ugyan ezt a függvényt a child objektummal hívjuk, ami szintén hiba nélkül fut le az örököltetés miatt. Továbbá meghívjuk a child objektummal a saját függvényét, végül a Parent osztály példányával hívánk meg a Child osztály függvényét, de ezt nem fogja látni a felnőtt osztály, így hiba keletkezik.

A programot C++ nyelven is megtekinthetjük:

<https://github.com/RubiMaistro/codes-prog2/blob/master/Liskov/ParentChild/Szulo-gyerek.cpp>

```
#include <iostream>

using namespace std;

class Parent {
public:
    void printParent() {
        cout << "Parent.\n";
    }
};

class Child : public Parent{
public:
    void printChild() {
        cout << "Child.\n";
    }
};

int main() {

    Parent parent;
    Child child;

    parent.printParent();
    child.printChild();
    child.printParent();

    //parent.printChild();
}
```

```
    return 0;  
}
```

A Java és C++ kódok csak szintaxisban térnek el egymástól.

DRAFT

## 14. fejezet

# Helló, Mandelbrot!

### 14.1. EPAM: OO modellezés

A szoftverfejlesztésben lényeges momentum a modellezés. Amikor egy programot kódban szinten szeretnénk megvalósítani, több esetben előnyösebb előre megtervezni és átgondolni a program negyobb részeit. Ehez a program modelljét érdemes előbb létrehozni, hogy az esetleges nagyobb hibákat már a kódolás előtt orvosolni tudjuk.

Viszont jól modellezni egy programot nem olyan egyszerű, legalábbis ha nem akarunk sok hibát véteni már a kód megírásakor. Vannak bizonyos alapelvek amelyeket ajánlatos figyelembe venni egy modellezés megkezdése előtt. Ezeket az elveket szoftverfejlesztési filozófiáknak nevezzük.

Egy pár elv amelyek kulcsfontosságúak:

**DRY** ( Don't Repeat Yourself ) :

Amit rövidít az a "ne ismétled önmagad", szó szerint ez az elva alapja. Ez az alapelvek azt a lehetőséget kínálja, hogy használunk keretrendszeret és csökkentsük a szoftverminták ismétlődését. Tehát ha keretrendszer használunk akkor csak azokat a feladatokat végezzük kézzel amik a legfontosabbak. A DRY elv alapja, hogy ugyanazokat a dolgokat ne írjuk le több helyre mert akkor egy változtatáskor szinkornban kell tartanunk a programot és minden helyen át kell írnunk, hogy optimalizált legyen.

Mivel a **DRY** magyarra fordítva "**száraz**", így az elv megsértésére úgy hivatkoznak, hogy "**nedves**", ami angolra fordíva **WET**. Ezt többféleképpen feloldják, pl: "**Write Everythong Twice**" azaz "**írj minden kétszer**", vagy "**waste everyone's time**" azaz "**pazaroljuk mindenki idejét**".

**KISS** ( Keep It Simple, Stupid ) :

Az elvet az Egyesült Államok Haditengerészete által 1960-ban megfogalmazott programtervezési minta. Az alapelvek szerint egy rendszer akkor működik a legjobban ha azt egyszerű megfogalmazásban és kódban hagyják, tehát ami már jól működik azt nem kell feleseleges dolgokkal kibővíteni. A **KISS** kifejezést Kelly Johnson repülőgépmérnökhöz fűzik, ami lefordítva "**Tartsa röviden és egyszerűen**", vagy "**Tartsa kicsinek és egyszerűnek**", vagy "**Tartsd ostobán egyszerűnek**".

**YAGNI** ( You Aren't Gonna Need It ) :

Az elv lényege hasonlít az előző, azaz a KISS elvhez, hogy válasszuk az egyszerűséget. Tehát ha van egy jól működő prgoramunk akkor azt ne bonyolítsuk, hiszen nincs értelme, azaz válasszuk minden az egyszerű de még működő megoldást.

Ezt az elvet akkor szokták áthágni amikor egy bizonyos programban általánosítást szeretnének bevezetni, hiszen akkor optimális lesz a programuk működése, viszont ezzel el is bonyolódhat.

#### **SOLID** ( betűszó, több elv együttese ) :

- **S - SRP** ( Egyetlen felelősség ) :

Ezen elv alapja hogy minden metódus csak egyetlen dologról legyen felelős. Mivel ha van egy olyan metódusunk ami több dologgal is foglalkozik akkor ha meg akarjuk hívni és szükségünk van az egyik információra akkor nem csak azt hanem minden más dolgot is megkapunk ami számunkra haszontalan információ lehet. Ha pedig minden metódus csak egy valamire öszponosít akkor a programkód is átláthatóbb.

- **O - OCP** ( Nyitva zárva ) :

Az alapelv lényege, hogy ha egy régebben megírt programkódt bővíteni akarunk akkor a mindig meg kell néznünk a régebbi részeket, hogy nem-e kell azokban is változtatásokat végrehajtani. Tehát az alapja hogy minden rész a kódban szinkronizálva legyen az újabb kódrészkekhez, ezért vissza kell menni és megváltoztani.

- **L - LSP** ( Liskov helyettseítési elv ) :

A Liskov elv alapján az örökölhető osztályoknak cserélhetőknek kell lenniük az alaptípusukra, azaz az alaposztályra való hivatkozásnak helyettesíthetőnek kell lennie egy örökölhető osztályra, anélkül, hogy ez befolyásolná a viselkedést.

Az elv lényege, hogy ha egy B típusú osztály az A típusúnak egy alosztálya akkor a B típusú objektumok helyettesíthetőek A típusú objektumokkal, így megtartva a program tulajdonságait. Ez az elv a Liskov fejezetben bővebben kifejtve is megtalálható.

- **I - ISP** ( Interfész elkülönítési elv ) :

Ez az elv egyszerű, a Liskov elv fordítottja. A klienst nem kell arra kényszeríteni, hogy számára haszon-talan dolguktól függjön. Ha van egy metódus ami több dolgot képes elvégezni de nekünk csak egyre van szükségünk akkor csak azt az egyet tartalmazza, azaz az Interfészeket szét kell darabolni a lehetséges különböző kérésnek megfelelően.

- **D - DIP** ( Függőségi inverzió elve ) :

Ezen elv kimondja, hogy a magasabb szintű modulok nem függhetnek alacsonyabb szintű moduluktól. Az elv az Interfészek használatára hívja fel a figyelmet.

Egy példán keresztül, hogy egy főnök független attól hogy az alkalmazottai mit dolgoznak, tehát más szóval megbízható embereket vesz fel a munkahelyre és ha valakit pedig azért nem küldhet el mert azt a dolgot csak az az alkalmazott tudja végrehajtani, akkor már baj van hiszen függés alakult ki a magas rangban lévő cégevezető vagy főnök és az alacsonyabb rangú, alkalmazott között.

A jegyzethez használt segédanyagok:

<https://siderite.dev/blog/solid-principles-plus-dry-yagni-kiss.html/>

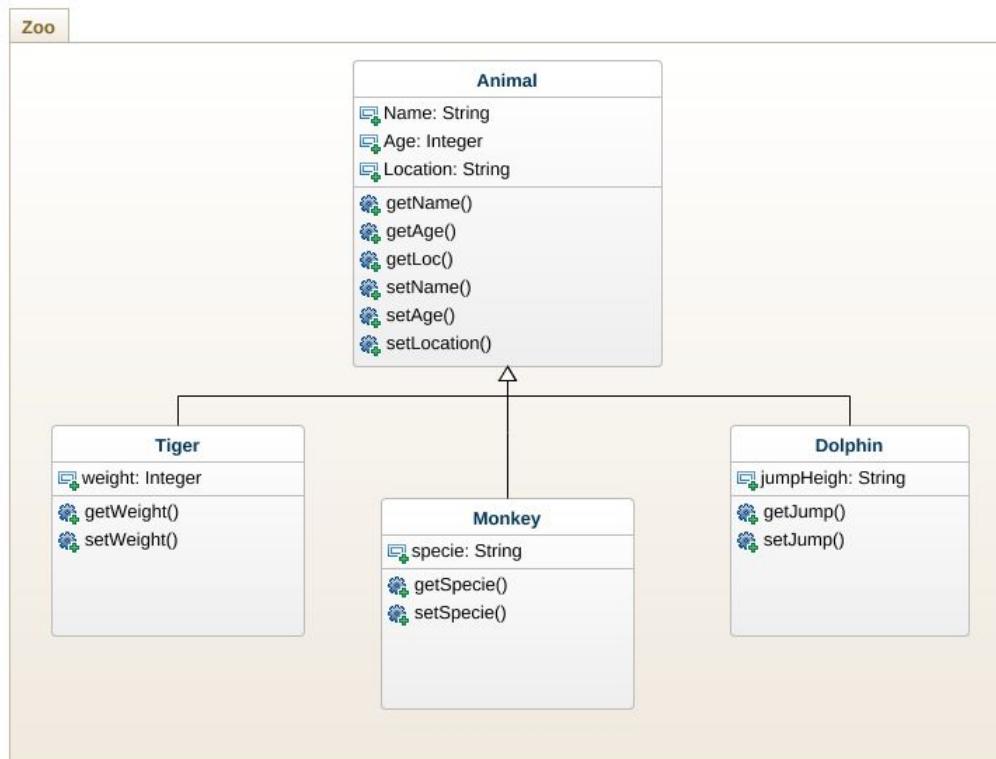
[https://hu.wikipedia.org/wiki/Szoftverfejleszt%C3%A9si\\_filoz%C3%B3fi%C3%A1k\\_list%C3%A1ja#Szoftverfejleszt%C3%A9si\\_filoz%C3%B3fi%C3%A1k](https://hu.wikipedia.org/wiki/Szoftverfejleszt%C3%A9si_filoz%C3%B3fi%C3%A1k_list%C3%A1ja#Szoftverfejleszt%C3%A9si_filoz%C3%B3fi%C3%A1k)

## 14.2. Forward engineering UML osztálydiagram

Egy egyszerű modellezés, amely tartalmaz osztályokat, azon belül értékeket és függvényeket. Van egy ős osztály és több származtatott osztály. Mindezt egy packagebe helyeztük el.

Megoldás forrása: [Animal UML](#)

A Diagaram:



Egy **Zoo** packageben létrehoztunk egy **Animal** osztályt amelyet örökoltettünk 3 másik osztályra, a **Tiger**, a **Monkey** és a **Dolphin** osztályokra. Majd az osztályokban látunk részeket és elválasztókat, a legfelső részben az attributumok helyezkednek el, majd alattuk a függvények. Az attributumoknak és a függvényeknek adhatunk tipust és nevet. A függvényeknek visszatérési értéket is beállíthatunk amit nem kell majd a kódoláskor.

A következő képen azok az osztályok látszanak amelyeket UML-ben terveztünk majd java kódba konvertáltunk.

```
Animal.java
1 package AnimalUML;
2
3 public class Animal {
4
5
6     public String Name;
7     public int Age;
8     public String location;
9
10    public Animal(){
11        super();
12    }
13
14    public void getName() {
15        // TODO implement me
16    }
17
18    public void getAge() {
19        // TODO implement me
20    }
21
22    public void getLocation() {
23        // TODO implement me
24    }
25
26    public void setName() {
27        // TODO implement me
28    }
29
30    public void setAge() {
31        // TODO implement me
32    }
33
34    public void setLocation() {
35        // TODO implement me
36    }
37
38
39 }
```

```
Monkey.java
1 package AnimalUML;
2
3 public class Monkey extends Animal {
4
5
6     public String specie;
7
8     public Monkey(){
9         super();
10    }
11
12    public void getSpecie() {
13        // TODO implement me
14    }
15
16    public void setSpecie() {
17        // TODO implement me
18    }
19
20 }
```

```
Dolphin.java
1 package AnimalUML;
2
3 public class Dolphin extends Animal {
4
5
6     public String jumpHeight;
7
8     public Dolphin(){
9         super();
10    }
11
12    public void getJump() {
13        // TODO implement me
14    }
15
16    public void setJump() {
17        // TODO implement me
18    }
19
20 }
```

```
Tiger.java
1 package AnimalUML;
2
3 public class Tiger extends Animal {
4
5
6     public int weight;
7
8     public Tiger(){
9         super();
10    }
11
12    public void getWeight() {
13        // TODO implement me
14    }
15
16    public void setWeight() {
17        // TODO implement me
18    }
19
20 }
```

Azt láthatjuk hogy teljesen üresek az osztályainkban a függvényeink. Modellezéskor érdemes a programunk osztályainak és azok függvényeinek csak a vázát létrehozni, aztán majd a kódban eldönthetjük, hogy hogyan működjenek a függvényeink.

Ebben a feladatban csak egy egyszerű UML modellezést szerettem volna szemléltetni. Ezt a kódot egy main() osztályal kiegészítve abban egy main() metódussal kiegészítettem, ami megtalálható az alábbi linken.

Link: <https://github.com/RubiMaistro/codes-prog2/tree/master/Mandelbrot/Animal>

Az UML-ből egy JAVA kódba konvertált program tényleges kifejtése és használata a következő feladatban mutatkozik meg igazán.

### 14.3. EPAM: Neptun tantárgyfelvétel modellezése UML-ben

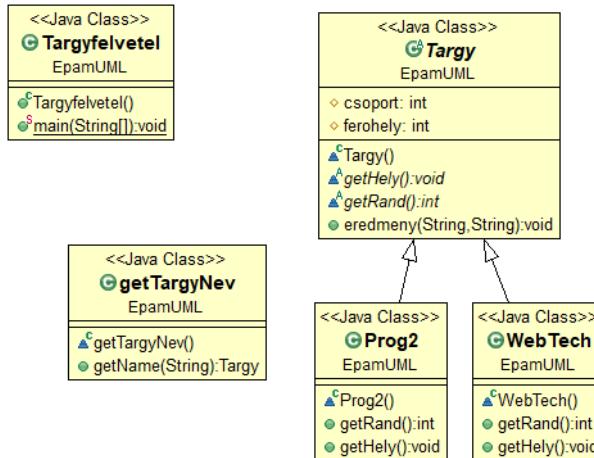
A modellezés fontosságáról és lényegéről az első feladatban, az **OO modellezés**-ben tudhatunk meg többet. Az első program amit modellezni fogunk az a jól írmert Neptunos Tárgyfelvétel lesz.

Modellezni már online is lehetőségünk van, de az **eclipse** applikáció ami a javat támogatja képes a modellezésre is, ha letöltünk modellező szoftvereket.

A modellünket a **Gyár (Factory)** tervezési mintára fogjuk alapozni, erről az arroway fejezetben a Creation Design Patterns részben lehet több információt megtudni.

A modellező szoftverben kiválaszthatunk bármit amit létre szeretnénk hozni, egy osztályt, egy függvényt, egy attributumot vagy akár packaget.

A modell amelyet létrehoztunk a következőképpen néz ki:



Modellezéskor számításba kell vennünk milyen osztályokat szerénénk létrehozni és hogyan. Jelen esetben a tárgyfelvételhez egy egyszerű ōs osztályt modellezünk, ami a **Tárgy** osztály lesz, majd megadunk az osztályhoz több tulajdonságot. Egy tárgyhoz tartoznak csoportok és ezekben férőhelyek, majd egy függvény amely visszaadja a tárgyhoz tartozó hallgatói kapacitást, ez a `getHely()` függvény amit abstractnak definiálunk hiszen majd kifejtjük minden tárgyra a összlétszámra való tekintetben. Majd ezt az osztályt örököltük további osztályoknak, ezek az osztályok már a bizonyos tárgyakat fogják definálni, jelen esetben két tárgyat hoztunk létre, a **Prog2**-t és a **WebTech**-et. Mivel ez a program egy szimulációnak felel meg ezért a tárgyfelvételben egyenlőre csak két tárgyat hoztunk létre, és hogy egy kicsit interaktív legyen a program, szimuláljuk majd a többi hallgató jelenlétét a tárgyfelvételben, de erről később.

Tehát létrehoztuk a két tárgyat, ezután egy segédosztályt hozunk létre a **getTargyNev** osztályt, ez az osztály fog kommunikálni a "felhasználóval és a rendszerrrel", vagyis ez az osztály a tárgyakért lesz felelős. Ez az osztály csak egy függvényt tartalmaz, tehát ezen keresztül kérjük le a bizonyos tárgy osztályokat.

A Tárgy ōs osztályban létrehozunk további függvényeket, az egyik a `getRand()` függvény ami, egy véletlenszerű számot fog létrehozni a bizonyos osztályok maximális létszámának függvényében. A másik függvény az `eredmény()` ami a tárgyfelvétel kimeneteléről fog információkat továbbítani a képernyőre.

A Prog2 és Webtech osztályokban kifejtjük a `getRand()` és `getHely()` függvényeket, a randomizálás fogja szimulálni a többi hallgató jelenlétét tárgyfelvételkor, nem bonyolítva túl a programot egy hálózatra kiterjedő verziójára, egyenlőre ennyi elég. Viszont a hallgatóknak sem foglalunk helyet egyik tárgyban sem, de ez majd látszani fog a kiiratáskor.

Azon részeket amelyeket később a kódban ki kell fejteni az a következő feladatban lesz bemutatva.

#### 14.4. EPAM: Neptun tantárgyfelvétel UML diagram implementálása

Mivel létrehoztuk a modellünket, azaz a program vázát, már csak a programkódban kell kifejteni a működésüket.

The screenshot shows the Eclipse IDE interface with two main panes. The left pane displays the Java code for `Targyfelvetel.java`. The right pane shows the `Console` output, which details a simulation of a school application. The console output includes messages about entering a subject name, selecting a group, and performing actions like 'Felvész' or 'Lead'. It also shows statistics for different groups and ends with a summary message.

```

1 Targyfelvetel.java x targyfelvetel.ucls
2
3 public class Targyfelvetel {
4
5     public static void main(String[] args) throws IOException {
6
7         getTargyNev targyMelyik = new getTargyNev();
8
9         BufferedReader targyfelvetel = new BufferedReader(new InputStreamReader(System.in));
10
11         System.out.println("Tárgyfelvételn!");
12         System.out.println("Ha végeztél írd be az 'exit' kulcsszót.");
13         boolean fut;
14
15         do {
16
17             System.out.println("Írd be egy tárgy nevét ( prog2 vagy webtech ) :");
18             String targynev = targyfelvetel.readLine();
19
20             System.out.println("Felvész vagy Lead?:");
21             String muvelet = targyfelvetel.readLine();
22
23             Targy t = targyMelyik.getName(targynev);
24
25             if(t != null) {
26
27                 t.getRand();
28                 t.getFerohely();
29                 t.eredmény(muvelet,targynev);
30
31             }
32
33             System.out.println("Ha nem akarsz további műveleteket végezni\nírd be az 'exit' kulcsszót.");
34             String exit = targyfelvetel.readLine();
35             if(exit.equalsIgnoreCase("EXIT")) {
36                 fut = false;
37             }
38             else {
39                 fut = true;
40             }
41
42         } while ( fut == true );
43
44         System.out.println("A tárgyfelvétel megtörtént.");
45
46     }
47
48 }

```

Egy tárgyfelvétel szimulálást láthatunk a fenti képen, bal oldalon a `main()` metódust láthatjuk, jobb oldalon pedig maga a tárgyfelvétel szimuláció egyik esetleges lefutási esetét lépésről lépésre. A **Console**-ban a kék színnel írt kéréseket, más szóval a felhasználótól kérte válaszokat láthatjuk.

A prgoramkódot a pedig a következőkben nézhetjük végig.

```

1 package EpamUML;
2
3
4 abstract class Targy {
5
6     protected int csoport;
7     protected int ferohely;
8     abstract void getHely();
9     abstract int getRand();
10
11
12     public void eredmény(String muvelet, String nev) {
13
14         if( muvelet.equalsIgnoreCase("FELVESZ") ) {
15
16             if( getRand() < (csoport*ferohely) )
17
18                 System.out.println("Sikeresen felvettet a " + ←
19                               nev + " tárgyat a " + (int) (Math.random() * ←
20                               (csoport - 1) + 1 ) + ". csoportba.");
21
22             else
23
24                 System.out.println("A kurzusok beteltek.");
25
26
27             for(int i=1; i<=csoport; i++) {
28
29                 System.out.println(i+". csoport: " + ferohely + ←
30                               "/" + ferohely);
31
32             }
33
34             else if(muvelet.equalsIgnoreCase("LEAD")) {
35
36                 System.out.println("Sikeresen leadtad a " + (int) ( ←
37                               Math.random() * (csoport - 1) + 1 ) + ". " + nev ←
38                               + " csoport tárgyat.");
39
40             }
41
42         else
43
44
45     }
46
47 }

```

```
        System.out.println("Nincs ilyen parancs. Próbáld ←  
        újra.");  
    }  
}
```

Az absztrakt Tárgy osztály amelyről modellezéskor beszélünk, ami az ōs osztály és ebben lesz minden tárgynak az alapja kifejtve. Először is egy pár változó, a csoport és a ferohely amelyek a tárgyakhoz tartozó létszámot definiálják.

Két absztrakt osztály is helyet foglal itt amit majd kifejtünk az örököltettségi osztályokban, majd egy **eredmeny()** függvény amit itt fejtünk ki, hogy egységesen minden örököltettségi osztályban optimális legyen a működése. Ez a függvény csak ki fogja iratni a tárgyfelvétel menetét, tahát if elágazásokon keresztül implementálva, egy tárgyat képesek vagyunk felvenni és leadni, valamint ha az előbbi szeretnénk akkor lehet hogy a tárgy felvétel nem lesz sikeres a férőhelyek betelése miatt.

Minden véletlenszerű, hiszen nem tároljuk el egyetlen hallagtó helyét sem, mert ez a program csak egy szimuláció, egyébként a randomizálást az egyes tárgyak osztályában láthatjuk majd a létszámal arányosan kiszámítva.

```
package EpamUML;  
  
class Prog2 extends Targy {  
    public int getRand() {  
        return (int) ((Math.random() * (120-60)) + 60);  
    }  
    public void getHely() {  
        ferohely = 18;  
        csoport = 5;  
    }  
}  
  
class WebTech extends Targy {  
    public int getRand() {  
        return (int) ((Math.random() * (100-70)) + 70);  
    }  
    public void getHely() {  
        ferohely = 22;  
        csoport = 4;  
    }  
}
```

Itt a két tárgy osztályát látjuk amelyekre örökölték az ōs osztály tulajdonságait. A **getHely()** és **getRand()** függvények kifejtését kell megtenni ezekben az osztályokban. Először is **getHely()** metódusban definiáljuk a tárgyhoz tartozó csoportok számát és a csoportonkénti létszámot, a Prog2-ben 5 csoport lesz és mindegyikben 18 hely, a WebTech-ben 4 csoport 22 helyekkel.

Majd a **getRand()** függvény amely szimulálni fogja a hallgatókat a tárgy felvételekor, ez abban nyilvánul meg, hogy létrehozunk egy véletlenszerű számot, ami arányos az összlétszámhöz azaz lesz olyan eset amikor betelt az összes kurzus amikor fel szeretnénk venni a tárgyat és nem sikerül és lesz olyan eset amikor fel tudjuk venni.

```
package EpamUML;

class getTargyNev {

    public getTargyNev(){
        ArrayList<String> targyak = new ArrayList<String>();
        targyak.add("webtech");
        targyak.add("prog2");
    }

    public Targy getName( String name ) {

        String vantargy = null;
        for(String i : targyak) {
            if(name.equalsIgnoreCase(i))
                vantargy = name;
        }
        if(vantargy != null) {
            if(name.equalsIgnoreCase("PROG2")) {
                return new Prog2();
            }
            else if(name.equalsIgnoreCase("WEBTECH")) {
                return new WebTech();
            }
        }
        else {
            System.out.println("Helytelen tárgynév vagy m ↵
                               ūvelet.");
        }
        return null;
    }
}
```

Először is egy listát létrehozva és ebbe elmentve a tárgyfelvételben szereplő lehetséges tárgyakat tárolja, ezt a konstruktor fogja mindenig lefuttatni, majd ha a függvényt meghívjuk akkor egy **for()** ciklusban végig megy a listán amelyben megnézni hogy a kért tárgy benne van-e a tárgyfelvételben, és ha igen értékül adja a tárgy nevét egy változónak ami majd **if()** elágazásokon keresztül eldönti melyik tárgyal szeretnénk foglalkozni, majd visszaad egy új példányt abból a tárgy osztályból, így a program vezérlés egyből ahoz a bizonyos tárgyhoz lép és folytatódik a tárgyfelvétel.

```
public class Targyfelvetel {

    public static void main(String[] args) throws IOException {

        getTargyNev targymelyik = new getTargyNev();

        BufferedReader targyfelvetel = new BufferedReader(new ←
            InputStreamReader(System.in));
    }
}
```

```
System.out.println("Tárgyfelvétel\n");
System.out.println("Ha végeztél írd be az 'exit' ←
    kulcsszót.");
boolean fut;

do {

    System.out.println("Írd be egy tárgy nevét ( prog2 ←
        vagy webtech ) : ");
    String targynev = targyfelvetel.readLine();

    System.out.println("Felvezet vagy Lead?: ");
    String muvelet = targyfelvetel.readLine();

    Targy t = targyMelyik.getName(targynev);

    if(t != null) {

        t.getRand();
        t.getHely();
        t.eredmeny(muvelet,targynev);
    }

    System.out.println("Ha nem akarsz további m ←
        ūveleteket végezni\nírd be az 'exit' kulcsszót ←
        egyébként nyomj entert:");
    String exit = targyfelvetel.readLine();
    if(exit.equalsIgnoreCase("EXIT"))
        fut = false;
    else
        fut = true;

    } while ( fut == true);

    System.out.println("A tárgyfelvétel megtörtént.");
}
}
```

A fő osztályban a Tárgyfelvétel-ben kap helyet a **main()** függvény ahol példányokat hozunk létre és ahol öszpontosítjuk a programot. Az első objektum a **getTargyNev()** osztályhoz tartozik, ami majd vissza ad nekünk egy tárgyat amit fel szeretnénk venni, erre egy kiiratásban kérdezünk rá, majd értéket adva a függvénynek visszaad egy tárgy objektumot.

A felhasználótól a bemenetet egy BufferedReaderrel fogjuk kezelní. Egy hárultesztelő ciklust hozunk létre amely egy "exit" kulcsszó beírására lezárja a tárgyfelvétel, erről egy kiiratásban a felhasználót is informáljuk. Majd a ciklusban bekérjük egy tárgy nevét, aztán a műveletet, hogy mit szeretnénk a tárggyal, felvenni vagy leadni. A Tárgy ős osztálynak létrehozunk egy példányt aminek alapján a létrehozott getTargyNev osztály példánya a **getName()** függvényben a bekért tárgy nevet megnézzi hogy van-e olyan tárgy, azaz van-e olyan osztály, ha nincs akkor **null** értéket ad a t objektumnak.

Az **if()** feltétel null értékre nem teljesül, csak ha van olyan tárgy osztály létrehozva mint amit bekértünk. Az

**if()**-ben lefuttatjuk az **getRand()**, **getHely()** és **eredmeny()** függvényeket arra az osztályra amelyet kapott a t objektum.

Majd informáljuk a felhasználót, hogy ha végzett a tárgyfelvétellel akkor irja be az **exit** kulcsszót, ami a logikai változó értékét **hamis/false** logikai értékre állítja, ami a ciklus futásának végét eredményezi, aztán egy kiiratásban tájékoztatjuk, hogy végeért a tárgyfelvétel.

DRAFT

## 15. fejezet

# Helló, Chomsky!

### 15.1. EPAM: Bináris keresés és Buborék rendezés implementálása

Egy java osztály implementálása, amely képes előre definált n darab egész szám tárolására. Ez az osztály az alábbi funkcionálitásokkal rendelkezik:

- Elem hozzáadása a tárolt elemekhez.
- Egy tetszőleges egész számról megállapíja, hogy el van-e tárolva.
- Az eltárolt elemeket rendezni tudja és visszatérni a rendezett elemekkel.

A porgram kód pedig a következő:

```
import java.util.Arrays;

public class IntTarolo {

    int array[];
    int index;
    int size = 0;
    boolean sorted = true;

    public IntTarolo(int size) {

        this.size = size;
        this.array = new int[size];
    }

    public IntTarolo(int array[]) {

        this.size = array.length;
        this.array = array;
        this.index = this.size;
        this.sorted = false;
    }
}
```

Az IntTarlo osztályban létrehozunk egy egész tipusú tömböt az értékek tárolásához, egy egész tipusú index nevű változtót majd a tömb méretének a vizesálatához, egy egész tipusú size nevű változót a tömb méretének a tárolásához, végül egy boolean, azaz logikai változót ami a tömb rendezettségét fogja definiálni.

Valamint létrehoztunk még két tipusú konstuktort, az első egy értéket kap amit értékül ad a size változónak, ez a tömb mérete. A második konstruktor egy tömböt kap paraméterül ami alapján leszűri a tömb információt és elmenti a létrehozott változókba, a tömb mértetét, magát a tömböt, az indexet, és a sorted változót hamisra állítja, azaz rendezelten a tömb.

```
public void add(int ertek) {
    if( size <= index ) {
        throw new IllegalArgumentException("Tele van.");
    }
    sorted = false;
    array[index++] = ertek;
}
```

A függvény amely segítségével új elemeket tárolunk el. A metódus paraméterként egy egész számot kap, majd ezt a számot egy if feltételben megvizsgálja, hogy ha a feltétel teljesül akkor megtelt a tömb és egy illegális argumentum tipusú hibát fog vissza adni, ami kiirja, hogy "Tele van". Ha hamis, tehát van még hely a tömbben akkor a rendezettséget, a sorted változót hamisra állítjuk és a tömb következő indexéhez ez értéket rendeljük.

```
public String taroltE(int ertek) {
    if(!sorted) {
        sort();
    }

    int left = 0, right = size - 1;

    while(left <= right) {
        int mid = left + (right - left) / 2;

        if(array[mid] == ertek) {
            return "A(z) " + ertek + " benne van.";
        }

        if(array[mid] < ertek) {
            left = mid + 1;
        } else {
            right = mid - 1;
        }
    }

    return "A(z) " + ertek + " nincs benne.";
}

public int[] sort() {
    for(int i=0; i < size - 1; ++i) {
        for(int j=0; j < size - i - 1; ++j) {
            if(array[j] > array[j + 1]) {
```

```
        int temporary = array[j];
        array[j] = array[j + 1];
        array[j + 1] = temporary;
    }
}

sorted = true;
return array;
}
```

A bináris kereső függvény amely Stringben lévő kifejezésnek megfelelően adja vissza, hogy egy elem tárolt-e vagy sem. A függvény paraméterként egy egész típusú értéket vár, majd a függvényben egy if feltétel megvizsgálja, hogy rendezett-e a tömb, ha nem rendezett akkor rendezi. Két változó segítségével keressük meg az értéket a tömbben, a tömb kezdő és végpontját tároljuk el. Majd egy while() ciklusban addig keresünk amíg az egyik változó át nem lépi a tömb középindexét, azaz túlép a tömb felén, hiszen a left nőni míg a right csökkeni fog. A mid változó megkapja a left és right változó értékei közé eső középindexet, ha ez megegyezik az értékkel akkor benne van a tömbben, ha pedig nem akkor a következő if feltétel megvizsgálja ez érték nagyságát és eldönti, a leftet vagy a rightot kell tologatni, azaz ha kisebb a szám akkor a right fog csökkeni, hiszen a lenti tartományban kell keresni, ellenkező esetben ha nagyobb akkor a left nőni fog, hiszen itt pedig a fenti tartományban kell tovább keresni.

A buborékrendezés alapján rendezést végező **sort()** függvényt is definiáltuk, amely növekvő sorrendbe állítja a tömb elemeit. A rendezést az egymás mellett lévő elemek összehasonlításával és cseréjével hajtja végre a függvény.

```
Console x
IntTarlo: [65, 23, 12, 44]
IntTarlo: [12, 23, 44, 65]
A(z) 44 benne van.
A(z) 12 benne van.
A(z) 13 nincs benne.

Main.java x IntTarlo.java CaesarCipher.java Main.java
1 package EPAM;
2
3 public class Main {
4
5     public static void main(String[] args) {
6
7         IntTarlo array = new IntTarlo(4);
8         array.add(65);
9         array.add(23);
10        array.add(12);
11        array.add(44);
12
13        System.out.println(array);
14        array.sort();
15        System.out.println(array);
16
17        System.out.println(array.taroltE(44));
18        System.out.println(array.taroltE(12));
19        System.out.println(array.taroltE(13));
20    }
21
22 }
```

A main() metódusban az IntTarlo osztályt használjuk egy tömb értékekkel való feltöltéséhez. Először létrehozunk egy tömböt, majd feltöljük tetszőleges értékekkel, aztán rendezzük a tömböt, végül kiiratjuk rendezés előtt és után is. Majd rákérdezhetünk értékre, hogy benne vannak-e a tömbben. Például a 44-et és a 12-öt felvettük értékül a tömbbe de a 13-mat már nem.

## 15.2. EPAM: Order of everything

Létezik egy érdekes és egyben nagyon hasznos dolog a Javaban, ez nem más mint a Comparable. Ezt függvények általánosítására használhatunk, ez egyébként C++-ban a Template osztályok és függvények

mutatják be nagyon jól. Mivel ez Javaban is létezik, ezért egy példa feladaton keresztül vizsgáljuk meg a használatát.

A feladatban egy olyan program működését fogjuk nézi amelyik képes bármilyen Collectionokat, azaz bármilyen struktúrát rendezni, például String vagy Integer tipusokat egyazon metódussal.

```
import java.util.Collection;
import java.util.Collections;
import java.util.List;
import java.util.Set;
import java.util.stream.Collectors;
import java.util.stream.Stream;

import org.junit.jupiter.params.ParameterizedTest;
import org.junit.jupiter.params.provider.Arguments;
import org.junit.jupiter.params.provider.MethodSource;

public class OrderOfEverythingTest {

    @ParameterizedTest
    @MethodSource("collectionsToSortDataProvider")
    public <T extends Comparable<T>>
    List<T> testOrder
    (List<T> input, List<T> expectedOutput) {

        //List<Integer> actualOutput = createOrderedList(input) ←
        ;
        List<T> actualOutput = createOrderedList(input);

        // Then
        //assertThat(actualOutput, equalTo(expectedOutput));
        return actualOutput;
    }

    private static Stream<Arguments> ←
    collectionsToSortDataProvider() {
        return Stream.of(
            Arguments.of(Collections.emptySet(), Collections.←
                emptyList()),
            Arguments.of(Set.of(1), List.of(1)),
            Arguments.of(Set.of(2, 1), List.of(1, 2))
        );
    }

    private <T extends Comparable<T>> List<T> createOrderedList ←
    (Collection<T> input) {
        return input.stream()
            .sorted()
            .collect(Collectors.toList());
    }
}
```

}

Az osztály elején importálunk sok szükséges osztályt, ezek a struktúrák kezeléséhez szükségesek, mint például a Collection, a Set vagy a legfontosabb a List, hiszen listában fogjuk elmenteni a rendezett elemeket. Egy függvényben elkérjük a felhasználótól a bementetet, azaz a rendezetlen Collectiont és azt a listát amibe rendezni szeretnénk, ez a testOrder() metódus. Ennek a metódusnak örökölheti a Comparable-t ami elősegíti, hogy a metódus bármilyen tipusú adatokat képes legyen feldolgozni. A metódusban létrehozunk egy listát ami megegyezik a paraméterben kapott lista típusával, ezt T tipusként értelmezünk, ennek a listának értékül adjuk a rendező függvényt ami az input, avagy bemenet alapján fogja létrehozni a rendezett listát amennyiben lehet rendezni.

Majd a rendezett listát létrehozott függvény a createOrderedList() lesz, ami Stream API-val egyszerűen rendez és visszaadja a listát. A Streamelést elősegítő metódus a collectionsToSortDataProvider(), azt adjuk meg a legelső metódusnak alapként, azaz a MethodSource-nek a metódus előtt, valamint kulcsszóként a ParameterizedTest kulcsszót is megadjuk.

```
@SuppressWarnings("serial")
public class ParentException extends RuntimeException {

    ParentException() {
        System.out.println("New Parent Class Constructor.");
    }
}

@SuppressWarnings("serial")
public class ChildException extends ParentException {

    ChildException() {
        System.out.println("New Child Class Constructor.");
    }
}
```

Ha egy függvény számára a Comparable definiálva van az azt jelenti, hogy illeszkedik rá bármilyen típus. Ez megegyezik a C++-ban lévő Template kulcsszóval.

```
import java.util.Arrays;
import java.util.List;

public class Main {

    public static void main(String[] args) {

        OrderOfEverythingTest obj = new OrderOfEverythingTest() ←
            ;

        // Integer
        List<Integer> array = Arrays.asList ←
            (420, 34, 1, -69, 72, 115, 276);
        List<Integer> list = array;

        System.out.println("Unordered:\n\t" + array);
    }
}
```

```
System.out.println("Ordered:\n\t" + obj.testOrder(array ←
    , list));

//String
List<String> str = Arrays.asList("Luke", "Sophie", "←
    John", "Jack");
List<String> list2 = str;

System.out.println("Unordered:\n\t" + str);

System.out.println("Ordered:\n\t" + obj.testOrder(str, ←
    list2));
}
}
```

A main osztályban egyszerűen létrehozunk az osztálynak egy objektumot amit használunk a rendezéshez, majd az objektummal meghívjuk az osztály rendező függvényét, aminek adunk egy I/O paramétert, azaz egy bemenetet, a rendezendő Collectiont, és egy kimenetet, ahova meg szeretnénk kapni a rendezett elemeket egy listában. Ez azért érdekes mert több tipusra definiálódott a rendező függvény ezért kipróbálhatjuk Integer és String tipusokra, de akár más tipusokra is.

```
Console x D Main.java
<terminated> Main (8) [Java Application] C:\Program Fil
Unordered:
[420, 34, 1, -69, 72, 115, 276]
Ordered:
[-69, 1, 34, 72, 115, 276, 420]
Unordered:
[Luke, Sophie, John, Jack]
Ordered:
[Jack, John, Luke, Sophie]
```

### 15.3. EPAM: HashMap implementáció

Egy java.util.Map implementáció, mely nem használja a Java Collection API-t. Az implementáció megfelel az összes megadott unit tesztnak, nem tud kezelni null értékű kulcsokat és a “keySet”, “values”, “entrySet” metódusok nem kell támogatják az elem törlést.

A Hash Map kulcs-érték párokat tárol, az első a kulcs a második pedig maga az érték. A feladat egy olyan HashMap implementáció ami képes különböző tipusú értékek tárolása egyetlen HashMapben.

Hogy a program képes legyen egymástól függetelenül különböző tipusok tárolására, általános típusokat kell használnunk, azaz a tipust is várva a felhasználótól, majd azzal dolgozva tovább.

```
package EPAM_HashMap;

import java.util.*;
import java.util.function.BiPredicate;

class ArrayMap<K, V> implements Map<K, V> {
```

```
private static final int INITIAL_SIZE = 16;
private static final String NULL_KEY_NOT_SUPPORTED = "This Map ←
    implementation does not support null keys!";

private int size = 0;
@SuppressWarnings("unchecked")
private K[] keys = (K[]) new Object[INITIAL_SIZE];
@SuppressWarnings("unchecked")
private V[] values = (V[]) new Object[INITIAL_SIZE];

@Override
public int size() {
    return size;
}

@Override
public boolean isEmpty() {
    return size <= 0;
}

@Override
public boolean containsKey(Object key) {
    Objects.requireNonNull(key, NULL_KEY_NOT_SUPPORTED);

    return searchItemInArray(key, keys, Object::equals) != -1;
}

@Override
public boolean containsValue(Object value) {
    int valueIndex = searchItemInArray(value, values, Object::←
        equals);
    return valueIndex > -1 && keys[valueIndex] != null;
}

@Override
public V get(Object key) {
    Objects.requireNonNull(key, NULL_KEY_NOT_SUPPORTED);
    if(size <= 0) {
        return null;
    }

    int keyIndex = searchItemInArray(key, keys, Object::equals) ←
    ;
    if (keyIndex > -1) {
        return values[keyIndex];
    }

    return null;
}
```

```
@Override
public V put(K key, V value) {
    Objects.requireNonNull(key, NULL_KEY_NOT_SUPPORTED);

    int keyIndex = searchItemInArray(key, keys, Objects::equals);
    if (keyIndex < 0) {
        keyIndex = findFirstEmptyPlace();
        if (keyIndex < 0) {
            expandArrays();
        }
        keyIndex = size;
    }

    V prevValue = values[keyIndex];

    keys[keyIndex] = key;
    values[keyIndex] = value;
    size++;

    return prevValue;
}

@Override
public V remove(Object key) {
    Objects.requireNonNull(key, NULL_KEY_NOT_SUPPORTED);

    int keyIndex = searchItemInArray(key, keys, Object::equals);
    if (keyIndex > -1) {
        V prevValue = values[keyIndex];

        keys[keyIndex] = null;
        values[keyIndex] = null;
        size--;

        return prevValue;
    }

    return null;
}

@Override
public void putAll(Map<? extends K, ? extends V> m) {
    m.forEach(this::put);
}

@Override
public void clear() {
    Arrays.fill(keys, null);
```

```
        Arrays.fill(values, null);
        size = 0;
    }

    @Override
    public Set<K> keySet() {
        Set<K> result = new HashSet<>();
        for(K i : keys) {
            if (i != null) {
                result.add(i);
            }
        }

        return result;
    }

    @Override
    public Collection<V> values() {
        Collection<V> result = new ArrayList<>();
        for(V i : values) {
            if (i != null) {
                result.add(i);
            }
        }

        return result;
    }

    @Override
    public Set<Entry<K, V>> entrySet() {
        Set<Entry<K, V>> result = new HashSet<>();
        for(int i = 0; i < keys.length; ++i) {
            K key = keys[i];
            if (key != null) {
                V value = values[i];
                result.add(new AbstractMap.SimpleEntry<>(key, value));
            }
        }

        return result;
    }

private <I> int searchItemInArray(I item, I[] array, BiPredicate<I, I> equalFunction) {
    for (int index = 0; index < array.length; index++) {
        if (equalFunction.test(item, array[index]))
            return index;
    }
}
```

```
        return -1;
    }

    private int findFirstEmptyPlace() {
        return searchItemInArray(null, keys, Objects::equals);
    }

    private void expandArrays() {
        int expandedSize = size * 2;

        keys = Arrays.copyOf(keys, expandedSize);
        values = Arrays.copyOf(values, expandedSize);
    }

}
```

Egy alapértéket állít be minden új HashMapnek, ez az érték a 16, de azt később meg lehet változtatni. A containsKey() metódus egy key objektumot vár majd vissza ad egy logikai értéket, hogy van-e benne olyan. A containsValue() matódus pedig egy értéket vár és vissza ad egy logikai értéket, hogy van-e benne olyan. A null értékeket nem kezeli, ezt minden metódusban jelezni kell a **NONE\_KEY\_NOT\_SUPPORTED** feltételle.

Több metódust szükséges volt felüldefiniálni, hogy ne támogasson null kulcsokat. Valamint több olyan metódus van a HashMap osztályban amit az Eclipse automatikusan tud generálni a meglévő kód alapján, ezek a következők: a get(), put(), set(), stb.

A put() metódus eltárol egy értéket úgy, hogy megadunk egy tetszőleges kulcsot, ami hivatkozik, majd az értékre. A get() parancs egy kulcsot vár paraméterként, ami alapján megkeresi a kulcsra tartozó értéket és ezt adja vissza.

A remove() metódus egy kulcs alapján tud egy értéket törölni. Mivel null kulcsokat, nem tud tárolni ezért nem is tudja őket kitörölni. A values() metódus kilistázza az értékeket egy Collectionban megjelenítve. Az entrySet() metódus az kulcs-érték párokat listázza ki egy Collectionban. A clear() matódus az egész HashMapet kiüríti.

A searchItemInArray() metódus vissza adja egy keresett item indexét, ha az benne van. Az isEmpty() metódus logikai értéket ad vissza HashMap üreségének a vizsgálatából. A size() metódus pedig egyértelműen a HashMap méretét adja vissza.

```
package EPAM_HashMap;

public class Main {

    @SuppressWarnings("rawtypes")
    public static void main(String[] args) {

        ArrayMap<Integer, Comparable> map = new ArrayMap< Integer, Comparable>();

        map.put(17, "$$$");
        map.put(22, "Különböző Collectionok nélkül használható.");
    }
}
```

```
map.put(35, "Ez egy Hash Map implementáció.");
map.put(52, "Alkalmazkodik az elsőként kapott tipushoz. ←
");

System.out.println(map.get(35));
System.out.println(map.get(22));
System.out.println(map.get(52));

System.out.println(map.values());

ArrayMap<Comparable, Comparable> map2 = new ArrayMap<←
Comparable, Comparable>();

map2.put("first", "first");
map2.put("2", 3.14);
map2.put(2, 54321);
map2.put(3.14, "Kiskacsa");

System.out.println("\nTetszőleges Hash Map:");
System.out.println(map2.get("first"));
System.out.println(map2.get("2"));

System.out.println("Egy HashMap kulcs-érték párjai: " + ←
    map.entrySet());
System.out.println("Egy HashMap értékei: " + map2.←
    values());

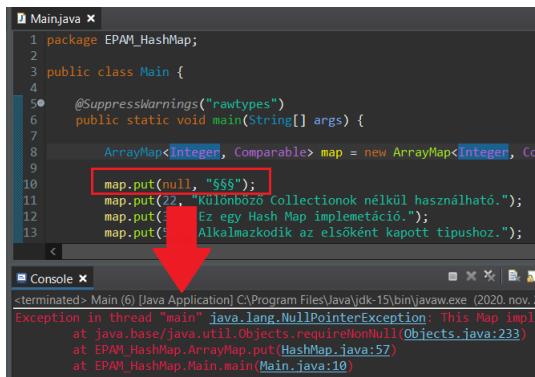
}

}
```

Egy main osztályban használjuk a HashMap implementációt, ahol két hash mapet hozunk létre. A elsőben a kulcsok Integer, az értékek pedig Comparable, azaz tetszőlegesek lehetnek. Majd a put() metódussal adunk a hashMaphez kulcsokat és értékeket, a get() metódussal pedig a kulcsaikra hivatkozva lekérhetjük őket. A values() metódus pedig vissza is adta. A következő képen a main() metódus futatás eredményét látjuk egy másik hashMapel együtt.

```
Console x HashMap.java D Mainjava <terminated> Main (6) [Java Application] C:\Program Files\Java\jdk-15\bin\javaw.exe (2020. nov. 26. 14:32:12 – 14:32:13)
Ez egy Hash Map implementáció.
Különböző Collectionok nélkül használható.
Alkalmazkodik az elsőként kapott tipushoz.
[§§§, Különböző Collectionok nélkül használható., Ez egy Hash Map implementáció., Alkalmazkodik az elsőként kapott tipushoz., 3.14, 54321, first, 22]
Tetszőleges Hash Map:
first
3.14
Egy HashMap kulcs-érték párok: [52=Alkalmazkodik az elsőként kapott tipushoz., 17=§§§, 22=Különböző Collectionok nélkül használható., Ez egy Hash Map implementáció., Alkalmazkodik az elsőként kapott tipushoz., 3.14, 54321, first, 22]
Egy HashMap értékei: [first, 3.14, 54321, Kiskacsa]
```

Majd egy olyan HashMapet is létrehozunk aminek a kulcsai és az értékei is Comparable tipusúak, ez azt jelenti, hogy képesek bármilyen értéket felvenni kulcsnak és képesek bármilyen értéket eltárolni. Négy kulcs-érték párt vettünk fel, string-string, string-int, int-int és int-String tipusokat vettünk fel ugyanabban a hashMapben. Majd a get() metódusba egy értékre a kulcsával hivatkozhatunk. kipróbálhatjuk az entrySet() metódust, ami kiiratja együtt a kulcs-érték párokat egy listában. Majd vegül kiirathatjuk csak az értékeket a values() metódussal.



```
1 package EPAM_HashMap;
2
3 public class Main {
4
5     @SuppressWarnings("rawtypes")
6     public static void main(String[] args) {
7
8         ArrayMap<Integer, Comparable> map = new ArrayMap<Integer, Comparable>();
9
10        map.put(null, "$$$");
11        map.put(2, "Különböző Collectionok nélkül használható.");
12        map.put(3, "Ez egy Hash Map implementáció.");
13        map.put(4, "Alkalmazkodik az elsőként kapott tipushoz.");
14    }
15 }
```

```
<terminated> Main (6) [Java Application] C:\Program Files\Java\jdk-15\bin\javaw.exe (2020. nov. 26)
Exception in thread "main" java.lang.NullPointerException: This Map implementation does not allow null keys
at java.base/java.util.Objects.requireNonNull(Objects.java:233)
at EPAM_HashMap.ArrayMap.put(ArrayMap.java:57)
at EPAM_HashMap.Main.main(Main.java:10)
```

Ezen a futási képen java.lang.NullPointerException errort generáltunk egy null kulcs érték megadásával, ez is bizonyítja hogy null kulcsok tárolására nem lehet alkalmazni ezt a HashMapet.

## 15.4. Full Screen

Ez a feladat egy kicsit érdekesebb, mivel egy teljes képernyős java program felépítéséről lesz szó. A hangsúly a teljes képernyőn van, vagyis inkább az ablak megjelenítésen. A java lehetőséget ad erre a JFrame segítségével. A program létrehoz egy kezdetleges Bejelentkező ablak prototípust.

A forráskód a következő.

```
import javax.swing.JFrame;
import javax.swing.JTextField;
import javax.swing.JPasswordField;
import javax.swing.JLabel;
import java.awt.FlowLayout;

@SuppressWarnings("serial")
public class SimpleFullScreen extends JFrame {

    public static void main(String[] args) {
        SimpleFullScreen screen = new SimpleFullScreen();

        screen.Screen();
    }
}
```

Egy osztályt létrehozva benne a main() metódusban példányosítás után az objektum meghívja azt a függvényt amelyik megjeleníti a teljes képernyős ablakot. minden importra szükségünk van. A JFrame osztály tulajdonságait örököl meg az osztály, amely egy ablak megjelenítés és elrendezést elősegítő osztály. A FlowLayout egy fajta elrendezést jelent majd. A JLabel segítségével egyszerű szövegdobozt lehet létrehozni. A JTextField segítségével, olyan szövegdobozt lehet létrehozni, amibe bármilyen felhasználó egyszerű szöveget gépelhet, a JPasswordField segítségével, pedig olyan szövegdobozt lehet létrehozni, ami password tipusú, azaz a bele írt karakterek rejttek lesznek, ahogyan egy egyszerű Bejelentkező oldalon is használják.

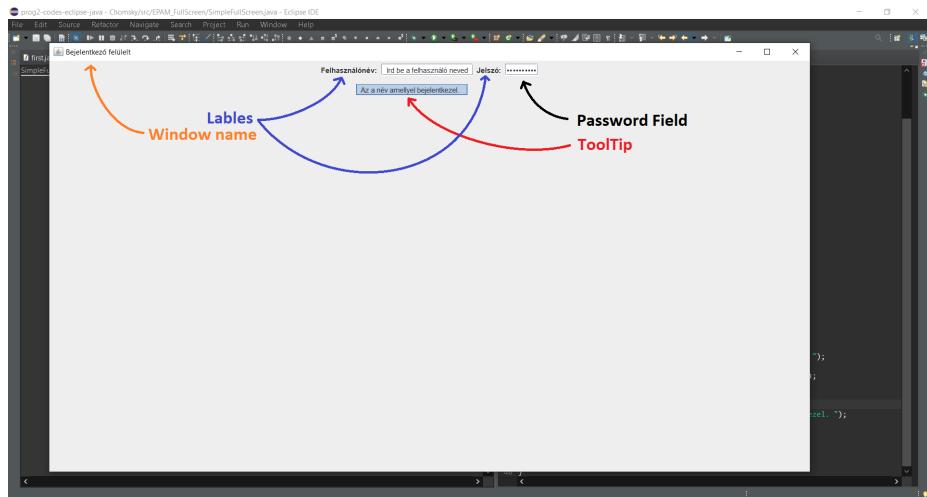
```
SimpleFullScreen() {
    super("Bejelentkező felülelt");
}
```

```
public void Screen() {  
  
    JLabel felhLabel;  
    JLabel jelszoLabel;  
    JTextField felhField;  
    JPasswordField jelszoField;  
  
    setLayout(new FlowLayout());  
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);  
    setSize(1280, 720);  
    setVisible(true);  
    setResizable(true);  
  
    felhLabel = new JLabel("Felhasználónév: ");  
    felhLabel.setToolTipText(" Az a név amellyel ←  
        bejelentkezel. ");  
    add(felhLabel);  
    felhField = new JTextField(" Ird be a felhasználó ←  
        neved ");  
    add(felhField);  
  
    jelszoLabel = new JLabel("Jelszó: ");  
    jelszoLabel.setToolTipText(" Az a jelszó amellyel ←  
        bejelentkezel. ");  
    add(jelszoLabel);  
    jelszoField = new JPasswordField(" Jelszó ");  
    add(jelszoField);  
}  
  
}
```

Maga a megjelenítés a Screen() függvényben történik, a konstruktőrban megadjuk az ablaknak a cím szavát amit megjelenít mint ablak nevet. A metódusban elsősorban létrehozunk négy változót. Az első két változó két egyszerű ablak, a harmadik egy szöveg mező és a negyedik egy jelszó mező.

A set tipusú metódusok végzik az alap beállításokat. Az elrendezést a setLayout() függvény FlowLayout-ra állítja, ami előre definiált, egy egyszerű középre állítás. A setDefaultCloseOperation() metódus a bezárhatóságot segíti elő. A setSize() metódus a feladat lényege, hiszen itt állíthatjuk az ablak méretét. A setVisible() metódus az ablak láthatóságát állítja be true értékre. A setResizable() metódus segítségével tudjuk kézzel is átméretezni az ablakot.

Az ablakba létrehozunk két Labelt, amik a felhasználónév és jelszó nevet kapták. Mindkét címkehez tudunk rendelni Fieldet, amibe írhatunk kezdeteleges segítő szövegeket, ezekbe a felhasználó tud gépelni. Majd létrehoztunk ToolTipet is amik rejtték magyarázatokat, ezek is segítenek a felhasználónak, ezek akkor jelennek meg amikor a kurzort a címkeire viszi a felhasználó. A Filedet és a ToolTipet mindenhol hozzá kell adni a címkehez, hogy kezelni tudja őket.



Egy érdekesség: A setResizable() metódus igazra állításával lehetőség van arra, hogy átméretezzük a képet, de ugyanakkor full screen-re állítani a jobb felső sarokban lévő Full Screen gomb segítségével.

```
9  @SuppressWarnings("serial")
10 public class SimpleFullScreen extends JFrame {
11
12•   public static void main(String[] args) {
13     SimpleFullScreen screen = new SimpleFullScreen();
14
15     screen.Screen();
16   }
17
18•   SimpleFullScreen() {
19     super("Bejelentkező felület");
20   }
21
22•   public void Screen() {
23
24     JLabel felhLabel;
25     JLabel jelszolabel;
26     JTextField felhfield;
27     JPasswordField jelszofield;
28
29     setLayout(new FlowLayout());
30     setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
31     setSize(1280,720);
32     setVisible(true);
33     setResizable(true);
34
35     felhLabel = new JLabel("Felhasználónév: ");
36     felhLabel.setToolTipText(" Ird be a felhasználó neved ");
37     add(felhLabel);
38     felhfield = new JTextField(" Ird be a felhasználó neved ");
39     add(felhfield);
40
41     jelszolabel = new JLabel("Jelszó: ");
42     jelszolabel.setToolTipText(" Az a jelszó amellyel bejelentkezel. ");
43     add(jelszolabel);
44     jelszofield = new JPasswordField(" Jelszó ");
45     add(jelszofield);
46   }
47
48 }
```

## 16. fejezet

# Helló, Stroustrup!

### 16.1. JDK osztályok

Ebben a feladatban a Boost C++ programot nézünk meg amely kilistázza a JDK összes osztályát.

Az első feladat a Boost könyvtárat telepíteni, ha még nincs a gépünkön akkor azt az alábbi módon tudjuk megtenni linuxon: **sudo apt-get install libboost-all-dev**. Azért van erre szükségünk, ugyanis az elérési utat és a kiterjesztést is ezel tudjuk elvégezni.

```
#include <iostream>
#include <string>
#include <map>
#include <iomanip>
#include <fstream>
#include <vector>
#include <boost/filesystem.hpp>
#include <boost/filesystem/fstream.hpp>
#include <boost/program_options.hpp>
#include <boost/tokenizer.hpp>
#include <boost/date_time posix_time posix_time.hpp>

using namespace std;
using namespace boost::filesystem;

int fajlok = 0;
void read_file ( boost::filesystem::path path, std::vector<std ::string> acts )
{
    if ( is_regular_file ( path ) ) {
        std::string ext ( ".java" );
        if ( !ext.compare ( boost::filesystem::extension ( path ) ) ) {
            std::cout << path.string() << "\n";
            std::string actpropspath = path.string();
        }
    }
}
```

```
        std::size_t end = actpropspath.find_last_of ( "/" ) ←
            ;
        std::string act = actpropspath.substr ( 0, end );

        acts.push_back(act);
        fajlok++;
    }
} else if ( is_directory ( path ) )
    for ( boost::filesystem::directory_entry & entry :
        boost::filesystem::directory_iterator ( path ) ←
            )
        read_file ( entry.path(), acts );
}

int main ( int argc, char *argv[] ) {
{
    string path="src";
    vector<string> acts;
    read_file(path, acts);
    cout<<"fajlok: "<< fajlok << endl;

    return 0;
}
}
```

A programnak egyszerűen csak a ".java" végzőséű fájlokat az elérési útjukkal együtt kilistázza, majd a végén összeszámolja, hogy hányat talált. A programban elején az alap szükséges includeok láthatók, valamint a telepített boost mappából a boost includeok. Egy egész tipusú változóval számoljuk meg, hogy hány fájlt talált, ez a fajlok változó. A read\_file() függvényben egy if() feltételben megvizsgáljuk az elérési út valódiságát, és beolvassuk a mappákat és fájlokat, majd egy másik if()-ben létrehozunk egy stringet aminek az értéke ".java" lesz, ami alapján vizsgáljuk a fájlok kiterjesztését. Ha talál a program egy ilyen fájlt, akkor a fajlok változót megnöveli eggyel. Egy vectorba fogjuk elmenteni az összes .java kiterjesztésű fájlt az elérési úttal együtt. Az else ágban vizsgáljuk, hogy ha mappát olvasott a program, akkor lépj a mappába.

A main() függvényben pedig csak az elérési utat definiáljuk egy stringben, majd egy vektort is létrehozva átadjuk a read\_file() függvénynek két paraméterként. Végül a fajlok változó értékét is kiiratjuk, ami a ".java" fájlok számát fogja jelenteni.

## 16.2. EPAM: It's gone. Or is it?

A feladatban rámutatunk arra az esetre amikor egy fájlba akarunk írni és az ehez szükséges matódust meg-hívva a fájlban nem törénik semmilyen változás, azaz nem sikerült a fájlba iratás, addig amíg be nem zárjuk a fájl írót.

```
import java.io.FileWriter;
import java.io.IOException;
import java.io.Writer;
```

```
public class FinalizeFixedExample {

    public static void main(String[] args) throws Exception {
        try (BugousStuffProducer stuffProducer = new ←
            BugousStuffProducer("src\\EPAM\\someFile.txt")) {
            stuffProducer.writeStuff();
        }
    }

    private static class BugousStuffProducer implements ←
        AutoCloseable {
        private final Writer writer;

        public BugousStuffProducer(String outputFileName) ←
            throws IOException {
            writer = new FileWriter(outputFileName);
        }

        public void writeStuff() throws IOException {
            writer.write("Stuff");
        }

        @Override
        public void close() throws Exception {
            writer.close();
        }
    }
}
```

A kód egyszerű, egy main metódusban egy try()-nak paraméterként adunk egy példányosítást. Ez a példány a BugousStuffProducer() osztály példánya lesz, amely fájlba iratással foglalkozik. A példányosításkor paramétert adunk az objektumnak, ez egy fájl elérési útvonala, neve és kiterjesztése lesz, amit a konstruktornak felhasználunk, hiszen inicializálunk erre az útvonalra egy fájl írót. Majd a try blockban meghívjuk a writeStuff() metódust ami a "Stuff" szót fogja beleírni a fájlba. A feldatok röviden ennyik, viszont elfelejtettük valamit, a fájlba írót bezárni. Viszont ez nem probléma mivel implementáltuk az osztálynak az AutoCloseable interfészét, ami elősegíti az objektumok automatikus bezárását. Egy egyszerű close() metódsban van a fájl írót bezáró close() függvény, amelyet ezért nem kell meghívni.

```
1 itsGoneOrIsIt.java x
2
3 public class itsGoneOrIsIt {
4
5     public static void main(String[] args) throws Exception {
6         try (BugousStuffProducer stuffProducer = new BugousStuffProducer("src\\EPAM\\someFile.txt")) {
7             stuffProducer.writeStuff();
8         }
9     }
10
11     private static class BugousStuffProducer implements AutoCloseable {
12
13         private final Writer writer;
14
15         public BugousStuffProducer(String outputFileName) throws IOException {
16             writer = new FileWriter(outputFileName);
17         }
18
19         public void writeStuff() throws IOException {
20             writer.write("Stuff");
21         }
22
23         public void close() throws IOException {
24             writer.close();
25         }
26     }
27 }
28
29
30
31
32 }
```

Kiírat

Automatikusan meghívja

someFile.txt x

1\$Stuff

## 16.3. EPAM: Kind of equal

A var kulcsszót ismerhetjük meg amely egy lokális változót hoz létre amely megkönnyíti a programozók dolgát, mivel nem kell típust adni a változónak.

Megoldás forrása:

A programkód a következő:

```
package com.epam.training;

import static org.hamcrest.CoreMatchers.is;
import static org.hamcrest.MatcherAssert.assertThat;

import org.junit.jupiter.api.Test;

public class KindofEqualTest {

    public void testKindofEqualExercise() {

        String first = "...";
        String second = "...";
        String third = new String(...);

        var firstMatchesSecondWithEquals = first.equals(second) ← ;
        var firstMatchesSecondWithEqualToOperator = first == ← second;
        var firstMatchesThirdWithEquals = first.equals(third);
        var firstMatchesThirdWithEqualToOperator = first == ← third;

        assertThat(firstMatchesSecondWithEquals, is(true));
        assertThat(firstMatchesSecondWithEqualToOperator, is( ← true));
    }
}
```

```
        assertThat(firstMatchesThirdWithEquals, is(true));
        assertThat(firstMatchesThirdWithEqualToOperator, is(←
            false));
    }
}
```

A program elején 3 szöveg (string) típusú változót hozunk létre amelyeknek az értékük "..." lesz. Aztán a **var** kulcsszóval létrehozunk 4 logikai (boolean) változót. Az elsőnek a adunk egy equals metódus által az első és második String összehasonlításából adódó logikai értéket, majd a másodiknak az első és második változóknak az == operátorral összehasonlított logikai értékét. A harmadiknak az első és harmadik változót összehasonlító equals() metódus logikai értékét. Végül pedig az első és harmadik változó egyszerű operátorral való összehasonlításának az értékét.

Végül az assertThat() metódusban minden a négy logikai változónak megadjuk ez értékét. Ne legyen egyenlő egy egyszerű String egy String típusú objektum értékével, ezért false értéket adunk annak a változónak az assertThat() metódus segítségével, még akkor is ha az értékeik megegyezhetnek.

```
import org.junit.runner.JUnitCore;
import org.junit.runner.Result;
import org.junit.runner.notification.Failure;

public class MainClass {

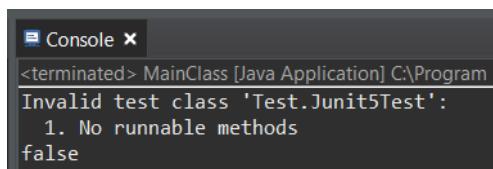
    public static void main(String[] args) {

        Result result = JUnitCore.runClasses(Junit5Test.class);

        for(Failure failure : result.getFailures()) {
            System.out.println(failure.getMessage());
        }

        System.out.println(result.wasSuccessful());
    }
}
```

A junit tesztelés a main() metódusban fog lefutni. A Result osztálynak létrehozunk egy objektumot, ami nek megadjuk, hogy a Junit5Test osztályt nézze a teszteléskor. Egy for ciklusban a Failure osztálynak létrehozunk minden ciklus futásakor egy objektumot ami a hibákat fogja megkapni, a resulthon keresztül a Junit5Test osztályból. Majd ezt minden kiiratjuk a konzolra, végül a resultra meghívjuk a wasSuccessful() metódust ami logikai értékként vissza adja, hogy sikeres volt-e a tesztelés.



## 16.4. EPAM: Java GC

A javaban léteznek úgynevezett Gargabe Collectorok, ami lefordíva szemetgyűjtőket jelent. A szemetgyűjtés megkönnyíti a dolgunkat, hiszen eltávolítja a memóriából azokat az objektumokat amelyeket már nem használunk.

Megoldás forrása: <https://medium.com/@hasithalgalage/seven-types-of-java-garbage-collectors-6297a1418e82>

A szemetgyűjtők típusai a következők:

A **Soros szemetgyűjtő** befagyasztja az összes alkalmazásszálat amikor fut, ezért nem alkalmas több szálon futó alkalmazásokban használni, ebből adódóan célszerű egyszálas környezetben alkalmazni. A soros szemetgyűjtő engedélyezéshez a következő argumentumot használhatjuk: **java -XX: + UseSerialGC -jar Application.java**

A **Párhuzamos szemetgyűjtő** már több szálat fog használni és hasonlóan a soros szemetgyűjtőhöz ez is lefagysztja az össze alkalmazásszálat a szemétszállítás alatt. Ezt a szemetgyűjtést általában olyan alkalmazásokban használják amely alkalmazások eltűrik az alkalmazás szüneteket.

A **CMS (Concurrent Mark Sweep) szemetgyűjtő** több processzor szálon dolgozik. Egy halmazba gyűjti azokat az alkalmazásokat amelyekben végre kell hajtani a szemetgyűjtést. Ez lényegében olyan alkalmazásokhoz használható amelyek képesek eltűrní a rövidebb szemetgyűjtési szüneteket. Futás közben megosszák a processzor erőforrásait a szemetgyűjtővel. Ez a fajta szemetgyűjtés több CPU-t használ mint a párhuzamos szemetgyűjtő, ezért hatékonyabbnak is mondható. Az összes alkalmazásszálat csak az objektumok megjelölése során és vagy bármely halmazváltozás változás során futtatja. Engedélyezéshez: **java -XX: + USeParNewGC -jar Application.java**

A **G1 (Garbage First) szemetgyűjtő** több processzoros gépeken futó alkalmazásokhoz készült. A memóriát széválasztja régiókra és párhuzamosan gyűjt bennük. Sokkal hatékonyabb mint a CMS gyűjtő. Két fázisa van a szemetgyűjtőnek, a jelölés és a söprés. A jelöléskor kupacokra vagy régiókra bontja a memóriát és mindegyiket átnézi és megjelöli. Majd a söpréskor tudja előre hol vannak az üres részek és itt kezd, ezért is nevezik First Garbage-nek és párhuzamosítva végzi a szemetgyűjtést. Engedélyezéshez: **java -XX:+UseG1GC -jar Application.java**

A **Epsilon szemetgyűjtő** egy passzív szemetgyűjtő, ez azt jelenti hogy nem gyűjti folyamatosan a fel nem használt objektumokat hanem csak akkor amikor az alkalmazások kifogynak a memóriából és akár már összeomlanak is. A célja az alkalmazások teljesítményének mérése és kezelése. Tehát meghatározott idő után kezdi a söprést, ez által megtudhatjuk mennyi a memória küszöbértéke és tudhatjuk mennyire szemetel a programunk. Ha minden teljesítményt ki kell szorítanunk az alkalmazásunkból, az Epsilon lehet a legjobb megoldás egy GC-hez.

A **Z szemetgyűjtő** előnye, hogy nem állítja le az applikációt több 10 msre a gyűjtés közben. Ez a gyűjtő tulajdonképpen alkalmas az alacsony késést igénylő vagy nagy kupacot használó programokhoz. Ezt a szemetgyűjtőt az Oracle fejleszette ki és mutatta be a Java 11-ben. Több terrabajton képes végezni a gyűjtést, a késés pedig átlagosan 1 ms, a G1 és a Parallel gyűjtőkkel szemben ami ezeknél akár 200 ms is lehetséges. Még kísérleti állapotban van ez a gyűjtő ezért még csak 64 bites Linux rendszereken érhető el. Adatbázis rendszerekben használja az Oracle ezt a tipusú szemetgyűjtést.

A ZGC három fázisban végzi a jelölést:

- 1. A világ rövid leállási fázisa (Short stop-the-world phase) :

Megvizsgálja a GC gyökereit, a halom többi részére mutató helyi változót. Ezeknek a gyökereknek a száma általában minimális, és nem változik a terhelés nagyságával, ezért a ZGC szünetei nagyon rövidek, és nem nőnek a halom növekedésével.

- 2. Egyidejű fázis (Concurrent phase) :

Bejárja az objektum grafikonját, és megvizsgálja a színes mutatókat, megjelölve az elérhető objektumokat. A teherzáró megakadályozza a GC fázis és az alkalmazás bármilyen tevékenysége közötti aktivitást.

- 3. Áthelyezési fázis (Relocation phase) :

A létező objektumokat a halom nagy részeinek felszabadítása érdekében mozgatja, hogy gyorsabbá váljon az allokáció. Amikor az áthelyezési fázis elkezdődik, a ZGC oldalakra osztja a kupacot, és egyszerre egy oldalon dolgozik. Miután a ZGC befejezte a gyökerek mozgatását, az áthelyezés többi része egyidejűleg történik.

A **Shenandoah szemétgyűjtő** egy nagyon alacsony szünetidővel rendelkező szemétgyűjtő amit a több munka elvégzésével segít elő. Ez a gyűjtő egyidejű tömörítést végez és memóriaterületeket foglal le a már használaton kívüli objektumok tárolására. Párhuzamos CPU-ciklusokat biztosít a szünetidő javítására, ezáltal több memóriát igényel, de ez a leggyorabb szemétgyűjtő. Egyidejűleg több kis fázisban dolgozza fel a halmot:

1. Előkészíti a halmot az egyidejű jelöléshez és beolvassa a gyökérkészletet és a hossza a gyökérkészlet méretének felel meg, nem pedig a kupacnak.

2. Ezután egy párhuzamos fázis bejárja a kupacot, és azonosítja az elérhető és elérhetetlen tárgyakat.

3. Befejezi a jelölés folyamatát a függőben lévő kupac frissítések törlésével és a gyökérkészlet újból beolvasásával.

4. Ezután egy másik párhuzamos fázis kimásolja az objektumokat az utolsó jel fázisban azonosított régióból. Ez a folyamat különbözteti meg Shenandoah-t a többi GC-től, mivel agresszíven tömöríti a kupacot az alkalmazásszálakkal párhuzamosan.

5. A következő fázis a ciklus harmadik (és legrövidebb) szünetét váltja ki. Ez biztosítja, hogy minden GC menetek kész kiürítését.

6. Egy párhuzamos fázis bejárja a kupacot, és frissíti a ciklus korábban áthelyezett objektumokra való hivatkozásokat.

7. A ciklus utolsó szüneteltetése befejezi a hivatkozások frissítését a gyökérkészlet frissítésével.

8. Végül az utolsó szakasz visszakéri a kiürített régiókat, amelyekben most nincsenek utalások.

A Shenandoah GC-t a háromféleképpen konfigurálhatjuk. Amikor a GC megkezdi a ciklusait, hogyan választja ki a régiókat az evakuáláshoz.

- Adaptív : Megfigyeli a GC ciklusokat, és elindítja a következő ciklust, így befejeződik, mielőtt az alkalmazás kimeríti a kupacot. Ez az alapértelmezett mód.
- Statikus : GC ciklust indít a kupac foglaltság és az allokációs nyomás alapján.
- Kompakt : GC ciklusokat folyamatosan futtat. Shenandoah új ciklust indít, amint az előző befejeződik, vagy az utolsó ciklus óta kiosztott kupac mennyisége alapján.

Shenandoah-nak gyorsabban kell gyűjtenie a kupacot, mint amennyit a kiszolgáló alkalmazás lefoglal. A az allokációs nyomás túl nagy, és nincs elegendő hely az új allokációhoz, akkor hiba lép fel. Shenandoah konfigurálható mechanizmusokkal rendelkezik ehhez a helyzethez.

**Pace** : Ha Shenandoah elkezdi lemaradni a kiosztás mértékétől, akkor leállítja a kiosztási szálakat, hogy utolérje. Shenandoah legfeljebb 10 ms késést vezet be. Ha az ütemezés nem sikerül, Shenandoah a következő lépésre lép: a degenerált GC-re.

**Degenerált GC** : Ha kiosztási hiba lép fel, Shenandoah elindítja a leállítás fázisát az aktuális GC ciklus befejezéséhez. Mivel a stop-the-world nem küzd az erőforrások alkalmazásával, a ciklusnak gyorsan be kell fejeződni és ki kell derítenie az elosztási hiányt. Gyakran a degenerált ciklus akkor következik be, amikor a ciklus legtöbb munkája már befejeződött. A GC napló teljes szünetként fogja jelenteni.

**Teljes GC** : Ha az ingerlés és a degenerált GC egyaránt kudarcot vall, Shenandoah teljes GC ciklusra esik vissza. Ez az utolsó GC garantálja, hogy az alkalmazás memórián kívüli hibával fog kudarcot okozni.

A Shenandoah ugyanazokat az előnyöket kínálja, mint a ZGC, nagy halmokkal, de több hangolási lehetőséggel. Szünetideje nem biztos, hogy olyan rövid, mint a ZGC, de kiszámíthatóbbak.

DRAFT

# 17. fejezet

## Helló, Gödel!

### 17.1. EPAM: Mátrix szorzás Stream API-val

Ebben a feladatban a java nyelv Stream API-jával ismerkedhetünk meg ami a Java 8-ban került be először a nyelvbe. A Stream API tulajdonképpen azért került be a javaba, hogy megkönnyítse a programozók dolgát különböző strukturákon való végig iteráláshoz. Ebben a feladatban a Stream API segítségével fogunk végig iterálni mátrixokon úgy, hogy ehez nem szükséges ciklusokat használnunk. A feladat lényege, hogy a mátrix szorzást implementáljuk streameléssel.

A Stream egy kollekció ami nem változtatja meg az adatok szerkezetét, csupán csak az eredményét adják egymás után láncolt metódusoknak. Ezeket a metódusokat operátoroknak hívjuk amelyeknek két tipusa létezik, a közbenső operátorok és a terminális operátorok. A közbenső operátorok streamelés közben végzik el a bizonyos feltételeiket, amelyek alapján streameljük a strukturát, tulajdonképpen feltétlekkétn tekintetük rájuk. A terminális operátorok minden az egymás után láncolt metódusok utolsó eleme, ezek zájják le a streamelést.

Előre nézzük meg, hogyan is fog a mátrix szorzás létrejönni.

The screenshot shows an IDE interface with two panes. The left pane displays the code for `MatrixMain.java`:

```
1 package epam.matrix.multiply;
2
3 public class MatrixMain {
4
5     public static void main(String[] args) {
6
7         int[][] matrix = {{1,-2,-3},{5,5,-5},{2,8,-9}};
8         int[][] matrix2 = {{-1,2,3},{-1,-2,5},{7,4,3}};
9
10        AbstractMatrix lm = new LambdaMatrix(matrix);
11        AbstractMatrix am = new LambdaMatrix(matrix2);
12        LambdaMatrix mm = (LambdaMatrix) lm.multiply(am);
13
14        System.out.println(mm.toString());
15    }
16
17 }
```

The right pane shows the `Console` output:

```
<terminated> MatrixMain [Java Application]
Matrix:
-20, -6, -16,
-45, -20, 25,
-73, -48, 19,
[rowsLength=3, columnsLength=3]
```

Annotations in red and blue highlight specific parts of the code and the output. Red arrows point from the variable declarations to the `AbstractMatrix` objects, and another red arrow points from the `multiply()` call to the resulting `LambdaMatrix` object. A large blue arrow points from the `toString()` call in the code to the printed matrix in the console.

A `main()` függvényben létrehozunk tömböket, amelyek a mátrixokat definiálják és objektumokat amikben a mátrixokat tároljuk. Az `lm` objektumban a `matrix`-t és az `am` objektumban pedig a `matrix2` tömböket, ezek egyszerű `AbstractMatrix` osztály tipusú példányok. A `LambdaMatrix` osztálynak is létrehozunk egy objektumot, ami megkapja az előbbi két mátrix szorzatát, amelyet az `lm` mátrixra hívjuk a szorzó függvényt, ami a `multiply()` metódus, és paramétreben megadjuk az `am` mátrixot. Végül kiiratjuk az `mm` objektum mátrixát, ezt a `toString()` metódus segítségével tesszük.

A programkód a következő:

```
public interface Matrix {  
  
    void setElement(int x, int y, int value);  
    Matrix multiply(Matrix input);  
}  
  
import java.util.Arrays;  
  
public abstract class AbstractMatrix implements Matrix {  
  
    protected final int[][] matrix;  
    protected final int rowsLenght;  
    protected final int columnsLenght;  
  
    public AbstractMatrix(int[][] matrix) {  
        this.matrix = matrix;  
        this.rowsLenght = matrix.length;  
        this.columnsLenght = matrix[0].length;  
    }  
  
    public AbstractMatrix(int rowsLenght, int columnsLenght) {  
        this.matrix = new int[rowsLenght][columnsLenght];  
        this.rowsLenght = rowsLenght;  
        this.columnsLenght = columnsLenght;  
    }  
}
```

Létrehoztunk egy Matrix interfészt amelynek van egy függvénye, később ezt hívjuk meg a számoláshoz. Majd egy AbstractMatrix osztályt hoztunk létre amelynek van három protected változója, ezek a mátrix, a sorai és az oszlopai amiknek értéket adunk a konstruktoron keresztül, amit a felhasználótól kértünk.

```
@Override  
public void setElement(int x, int y, int value) {  
    matrix[x][y] = value;  
}  
  
@Override  
public Matrix multiply(Matrix input) {  
    if (input instanceof AbstractMatrix) {  
        return multiply((AbstractMatrix) input);  
    }  
    throw new IllegalArgumentException("The input matrix should ←  
        be an instance of AbstractMatrix");  
}
```

Az első metódus három paraméter alapján beállít bizonyos értékeket a mátrixnak. A paraméterek a következők, az első a mátrix sora indexe, a második a mátrix oszlop indexe és a harmadik pedig maga az érték.

A második metódus később a mátrix szorzást végzi. Itt csak egy feltétel alapján eldönti, hogy a paraméterben kapott mátrix és a meglévő mátrix szorozható-e egymással. Ha ez nem lehetséges, akkor egy kivételkezeléssel dob egy hiba üzenetet.

```
@Override
public int hashCode() {
    final int prime = 31;
    int result = 1;
    result = prime * result + columnsLenght;
    result = prime * result + Arrays.deepHashCode(matrix);
    result = prime * result + rowsLenght;
    return result;
}

@Override
public boolean equals(Object obj) {
    if (this == obj)
        return true;
    if (obj == null)
        return false;
    if (getClass() != obj.getClass())
        return false;
    AbstractMatrix other = (AbstractMatrix) obj;
    if (columnsLenght != other.columnsLenght)
        return false;
    if (!Arrays.equals(matrix, other.matrix))
        return false;
    if (rowsLenght != other.rowsLenght)
        return false;
    return true;
}
```

Ezen, hashCode és equals metódusok felüldefiniálása szükséges, hogy a program optimálisan működjön. Ezen a felüldefiniálásokat az Eclipse automatikusan felajánlja az osztályban tárolt változók függvényen.

```
@Override
public String toString() {

    System.out.println("Matrix:");

    for (int row = 0; row < this.rowsLenght; row++) {
        for (int col = 0; col < this.rowsLenght; col++) {
            System.out.print(this.matrix[row][col] + ", ");
        }
        System.out.print("\n");
    }
    return "[rowsLenght=" + this.rowsLenght + ", columnsLenght= "
        "
```

```
        + this.columnsLength + "]\n";
    }

    abstract protected Matrix multiply(AbstractMatrix input);

}
```

A `toString()` metódus felüldefiniálását is automatikusan felajánlja az Eclipse, de ebben az esetben az nem lesz tökéletes. Ezért át van írva, hogy kiirassa a mátrix elemeit és vissza adja a mátrix sorainak és oszlopainak számát.

```
import java.util.Arrays;
import java.util.stream.IntStream;

public class LambdaMatrix extends AbstractMatrix {

    public LambdaMatrix(int[][] matrix) {
        super(matrix);
    }

    public LambdaMatrix(int rowsLength, int columnsLength) {
        super(rowsLength, columnsLength);
    }

    @Override
    protected Matrix multiply(AbstractMatrix input) {
        int[][] result = Arrays.stream(this.matrix)
            .map(r -> IntStream.range(0, input.columnsLength)
                .map(i -> IntStream.range(0, input.rowsLength)
                    .map(j -> r[j] * input.matrix[j][i]).sum())
                .toArray())
            .toArray(int[][]::new);
        return new LambdaMatrix(result);
    }
}
```

Egy `LambdaMatrix` osztályt létrehozva ami örököli az `AbstractMatrix` osztály tulajdonságait, definiáljuk a mátrix szorzást végző metódust, ez a `multiply()` metódus. Először két konstruktort fut le annak függvényében, hogyan adtunk meg mátrixt az objektumban. Majd a `multiply()` metódus paraméterként kap egy objektumot, ami egy mátrixot fog tartalmazni. A függvényben létrehozunk egy két dimenziós tömböt, aminek az értéke egy streamelés lesz, ez ajda meg az új mátrixnak a másik kettő szorzatát. Egyszerűen streameljük az elmentett mátrixot, aminek `map()`-ekkel megyünk végig először a sorain, majd az oszlopain egy-egy lambda-val és végül egy harmadik `map()`-ben egy lambda segítségével össze szorozzuk a tárolt mátrix sorainak az értékeit a paramétreben kapott mátrix oszlopában lévő elemekkel, majd ezeket össze adjuk. Végül a létrejött elemeket a `toArray()` metódussal a két dimenziós tömbbe rakjuk, amit vissza adunk egy új `LambdaMatrix` objektumként.

```
public class MatrixMain {

    public static void main(String[] args) {
```

```
int[][] matrix = {{1,-2,-3},{5,5,-5},{2,8,-9}};
int[][] matrix2 = {{-1,2,3},{-1,-2,5},{7,4,3}};

AbstractMatrix lm = new LambdaMatrix(matrix);
AbstractMatrix am = new LambdaMatrix(matrix2);
LambdaMatrix mm = (LambdaMatrix) lm.multiply(am);

System.out.println("lm matrix " + lm.toString());
System.out.println("am matrix " + am.toString());
System.out.println("mm matrix " + mm.toString());

}
```

Ez a main megegyezik a feladata elején bemutatott main függvényel, a különbség csak az, hogy kiiratjuk mind a három mátrix értékeit. Futtatás eredménye a következő.



```
Console ×
<terminated> MatrixMain [Java Application] C:\Program F
Matrix:
1, -2, -3,
5, 5, -5,
2, 8, -9,
lm matrix [rowsLength=3, columnsLength=3]

Matrix:
-1, 2, 3,
-1, -2, 5,
7, 4, 3,
am matrix [rowsLength=3, columnsLength=3]

Matrix:
-20, -6, -16,
-45, -20, 25,
-73, -48, 19,
mm matrix [rowsLength=3, columnsLength=3]
```

## 17.2. EPAM: LinkedList vs ArrayList

A Java nyelvben is léteznek úgynevezett tömbök, amelyeknek a célja, hogy ha programozók több változót használnak akkor azokat egy strukturában és egy helyre gyűjthetik, ezáltal könnyebb a használatuk és az elérésük. Viszont a tömbnek kötelezően meg kell adni a méretét amikor definiáljuk, ezért ha több elemre lenne szükségünk mint amennyit a tömb létrehozásakor definiáltunk, akkor egy problémába ütközünk, hiszen nem adhatunk a tömbhöz több elemet mint amennyit el tud tárolni.

Erre a problémára alkották meg a listát, amely szabadon bővíthető tetszőleges számú elemmel és ezzel párhuzamosan a lista mérete is nőni fog. Ebben a feladatban a Láncolt listát (LinkedList) és a Tömb listát (ArrayList) fogjuk összehasonlítani előnyeivel és hátrányaival együtt.

A listák és létrehozásunk:

ArrayList: Ez egy olyan lista amelynek a strukturája egy egyszerű tömbhöz hasonlít, a különbség viszont az, hogy ezt tetszőleges számú elemmel lehet bővíteni.

```
ArrayList<Integer> array = new ArrayList<Integer>();
```

LinkedList: Ez egy olyan lista amely elemei mutatókkal kapcsolódnak egymáshoz, ez azt jelenti, hogy egy elem csak azt tudja hogy milyen elem van előtte, tehát amelyik elem rá mutat, és azt hogy melyik következik utána, azaz melyikre mutat. A lista definíciója röviden, elemek láncolt sorozata.

```
LinkedList<Integer> linky = new LinkedList<Integer>();
```

A következőkben a létrehozott listákon különböző méréseket fogunk végezni:

Feltöljük a listákat elemekkel:

ArrayList:

```
// Upload
// ArrayList :
final long startTimeArrayList = System.nanoTime();
for(int i = 0; i < 10000; i++)
    array.add(i);
final long finishTimeArrayList = System.nanoTime();

long runningTimeArrayList = finishTimeArrayList - ←
    startTimeArrayList;

// LinkedList :
final long startTimeLinkedList = System.nanoTime();
for(int i = 0; i < 10000; i++)
    linky.add(i);
final long finishTimeLinkedList = System.nanoTime();

long runningTimeLinkedList = finishTimeLinkedList - ←
    startTimeLinkedList;
```

Mindkét listát egy-egy **for()** ciklusban az **add()** metódussal 10000 elemmel töltöttük fel, melyek értéke 1-től 10000-ig minden az index értékét kapta meg. Tehát egy azon elemek szerepelnek minden listában. A **System.nanoTime()** az aktuális időt tartalmazza, az időt a feltöltés előtt is és a feltöltés után is egy-egy változónak megadtuk, majd e két változó, tehát a kezdés időpontja és a befejezés időpontjának a különbözetéből kapjuk meg mennyi ideig tartott az elemek helyfoglalása és tárolása.

```
System.out.println("Array list fill in time: " + ←
    runningTimeArrayList);
System.out.println("Linked list fill in time: " + ←
    runningTimeLinkedList);
```

Kétszer futtattuk le:

```
<terminated> Lists [Java Application] C:\Program Files
Time of upload array list: 1640500
Time of upload linked list: 1204900

<terminated> Lists [Java Application] C:\Program Files
Time of upload array list: 2295900
Time of upload linked list: 3217299
```

A fenti két futási képen láthatjuk, hogy a listák feltöltési ideje váltakozó, egyszer az egyik egyszer a másik töltődik fel hamarabb, hiszen a memória lefoglalás véletlenszerűen törénik. Az ArrayList feltöléskor keres a memóriában egy rekeszt ahova elkezdi tárolni az elemeket, ha ez a memória rekesz megtelt de még érkeznek elemek a feltöltéshez akkor keres egy másik rekeszt amelyben belefér az eddig feltöltött elemek rekesze és hozzácsatolja a listához a soron következő többi elemet is. Tehát az adatokat egy helyen tárolja és együtt mozognak az adatok. A LinkedList pedig láncolt elemek sorozata, tehát a memória foglalás egyszerűen törénik. Elemenként foglalunk memóriában helyet az elemeknek, majd ezek egyszerűen mutatókkal lesznek összekapcsolva, ezért megtörténhet az is hogy az elemek nem egymás mellett lesznek a memóriában.

Végezzünk törlést a listából:

```
// Remove
// from ArrayList :
final long startTimeRemoveFromArray = System.nanoTime();
array.remove(200);
final long endTimeRemoveFromArray = System.nanoTime();

long arrayRemoveTime = endTimeRemoveFromArray - ←
                      startTimeRemoveFromArray;

// frome LinkedList :
final long startTimeRemoveFromLinkedList = System.nanoTime();
linky.remove(200);
final long endTimeRemoveFromLinkedList = System.nanoTime();

long linkyRemoveTime = endTimeRemoveFromLinkedList - ←
                      startTimeRemoveFromLinkedList;
```

```
<terminated> Lists [Java Application] C:\Program Files
Time of remove from array list: 23401
Time of remove from linked list: 11800
```

Egy egyszerű törlést végeztünk minden listából, ebben a kódban a 200. elemet töröltük ki minden listából, ehez a **remove()** metódust használtuk és az eltelt időt mértük minden esetben. Mivel az ArrayList-ből való törlés azt eredményezi, hogy a törölt elem után következő elemek az indexelésnek megfelelően egyel kisebb indexbe csúsznak, sokkal több idő kellet a törléshez mint a LinkedList-ből való törléshez, hiszen itt csak meg kell keresni az elemet amit törünk, majd az előtte lévő elem már nem a törölt elemre kell mutasson hanem a törlött elem után következő elemre.

```
array.remove(6000);
linky.remove(6000);
```

```
<terminated> Lists [Java Application] C:\Program Files\Java  
Time of remove from array list: 30101  
Time of remove from linked list: 101400
```

A két listából a két-két elem törlésének az idejét a fenti kódban használt módon mértük. A futási képen látható egy másik eset amikor a 10000 elemű listákból a 6000. elemeket töröltük. Ebben az esetben a LinkedList volt lassabb, hiszen meg kellett találja a memóriában a törlendő elemet, de ezt csak az előtte lévő összes elemen keresztül való végig iterálás során tehette meg. Az ArrayList esetében pedig a mérések alapján rájöhettünk, hogy egyre nagyobb indexű elem törlése esetén egyre kevesebb elemet kell átmozgatni a helyreállás elősegítése érdekében.

Még néhány számítás:

1 millió elemű listákból a 2000. elemeket törölve az ArrayList-ben 100x-os a futási idő:

```
<terminated> Lists [Java Application] C:\Program Files\Java  
Time of remove from array list: 10000501  
Time of remove from linked list: 73299
```

10 millió elemű listákból a 10000. elemeket törölve az ArrayList-ben 500x-os a futási idő:

```
Lists [Java Application] C:\Program Files\Java\jdk-15\bin\javav  
Time of remove from array list: 15634278000  
Time of remove from linked list: 33869799
```

```
array.get(200);  
linky.get(200);
```

```
<terminated> Lists [Java Application] C:\Program  
Time of get from array list: 7399  
Time of get from linked list: 13101
```

A két listából a két-két elem megtalálásának az idejét a törléskor használt módon mértük. A LinkedList ideje ebben az esetben is több, hiszen amíg el nem jut a keresett elemhez addig az elem előtt lévő elemeken végig kell iteráljon.

Még néhány számítás:

1 millió elemű listákban a 20000. elem megtalálása a LinkedList-ben 20x-os futási időt eredményez:

```
<terminated> Lists [Java Application] C:\Program File  
Time of get from array list: 15400  
Time of get from linked list: 315700
```

1 millió elemű listákban a 100. elem megtalálása a minden esetben hasonló futási időt eredményez:

```
<terminated> Lists [Java Application] C:\Program Fi  
Time of get from array list: 15400  
Time of get from linked list: 15000
```

ArrayList: Ezt a strukturát akkor érdemes használni ha kevesebb elemmel akarunk dolgozni és a listából több alkalommal akarunk elemeket megkeresni. Ez a struktúra több elem keresésre, akár elemek cseréjére

alkalmas a futási idő szempontjából. Alkalmatlan viszont az egyes elemek törlésére, a sok adat mozgatás szempontjából.

LinkedList: Ezt a struktúrát akkor érdemes használni amikor sok elemmel akarunk dolgozni és sokszor akarunk elemeket mozgatni, illetve törölni. Ez a struktúra nem alkalmas elemek többszöri keresésére. Elemek rendezésére és törlésre célszerűbb a használta.

### 17.3. EPAM: Refactoring

Ebben a feladatban kódrészleteket írunk át lambda kifejezések segítségével. A lambda kifejezések olyan egyszerű függvényeket helyettesítenek amelyeket nem szükséges külön definiálva megírni mert így időt és helyet spórolhatunk meg, egy lambda kifejezés igazából egy röviden megírt függvény. De ugyanakkor lambda kifejezéseket lehet definiálni, ekkor egy osztályt kell megadni típusként és a nevet amelyre hivatkozunk, így a lambdát többször is meghívhatjuk.

Lambda kifejezés implementációja:

```
// (paraméterek) -> { utsítás kifejezések }
// pl.

() -> System.out.println("Ez egy lambda kifejezés.");
```

Rendelkezésre áll egy kód amelyben lambda kifejezéseké kellene írni az egyes függvényeket:

```
package Lambda;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;
import java.util.function.Consumer;

public class LegacyRefactoring {

    public void legacy() {
        Runnable runnable = new Runnable() {
            @Override
            public void run() {
                System.out.println("Runnable!");
            }
        };
        runnable.run();

        Calculator calculator = new Calculator() {
            @Override
            public Integer calculate(Integer number) {
                return number * number;
            }
        };
        Integer result = calculator.calculate(3);
    }
}
```

```
System.out.println("Calculation result: " + result);

List<Integer> inputNumbers = Arrays.asList(1, null, 3, ←
    null, 5);
List<Integer> resultNumbers = new ArrayList<>();
for (Integer number : inputNumbers) {
    if (number != null) {
        Integer calculationResult = calculator.←
            calculate(number);
        resultNumbers.add(calculationResult);
    }
}

Consumer<Integer> method = new Consumer<Integer>() {
    @Override
    public void accept(Integer integer) {
        System.out.println(integer);
    }
};
System.out.println("Result numbers: ");
resultNumbers.forEach(method);

Formatter formatter = new Formatter() {
    @Override
    public String format(List<Integer> numbers) {
        StringBuilder sb = new StringBuilder();
        for (Integer number : numbers) {
            String stringValueOfNumber = String.valueOf ←
                (number);
            sb.append(stringValueOfNumber);
        }
        return sb.toString();
    }
};
System.out.println("Formatted numbers: " + formatter.←
    format(resultNumbers));
}
```

A program egy lista elemein számítást végéz majd az elemeket összefűzi és sztring típusá alakítja. Ebben a kódban egyetlen függvényben végzünk minden. Az egyes függvények amelyeket lambdákkal írunk át, a Runnable osztály példányosítása, Calculator példányosítása, Consumer metódus létrehozása és Formatter példányosítása majd az értékének a vissza adása.

Szükség van a következő interfészekre:

```
interface Print {
    public void simplePrint();
}

interface Formatter {
```

```
        String format(List<Integer> numbers);
    }

    interface Calculator {
        Integer calculate(Integer number);
    }
```

Az elsőt használni fogjuk kiiratáskor, a másodikat egy lista összefűzéshez, majd a harmadikat egy számoláshoz.

Lambda kifejezésekkel átírva a program kód a következő: Felül az könyvtárak importálása, majd a Refactored osztályban egy void típusú függvény amelyet majd meghívunk és ki fogja íratni a függvény értékét, majd maga a függvény lambdákkal kifejezve.

```
package Lambda;

//import java.util.Arrays;
import java.util.List;
import java.util.Objects;
import java.util.function.Consumer;
import java.util.stream.Collectors;

class Refactored {

    public void doSomething(Print obj) {
        obj.simplePrint();
    }

    public String refactored(List<Integer> inputNumbers) {

        Runnable runnable = createRun();
        runnable.run();

        //Calculator calculator = createCalculator();
        //System.out.println("Calculation: " + calculator. ←
        //    calculate(3));

        List<Integer> resultNumbers = inputNumbers
            .stream()
            .filter(Objects::nonNull)           // number -> ←
            .map(calculator::calculate)        // number -> ←
            .collect(Collectors.toList());

        Consumer <Integer> cons = createConsumerMethod();
        System.out.println("Numbers: ");
        resultNumbers.forEach(cons);

        Formatter formatter = createFormatter();
        String str = formatter.format(resultNumbers);
```

```
        return str;  
    }
```

A **refactored()** amely egy listát kap paraméterként függvényben létrehoztunk egy példányt a Runnable függvénynek, amely az első lambda kifejezés, és meghívjuk a **run()** függvényt, majd egy példányt hozunk létre a Calculatornak és meghívjuk a **calculate()** függvényét, ez a második lambda. Ebben a kódban nem használtam a Calculator osztályt, tehát nem számol a program semmit, ezért van kommentezve. Egy Integer típusú listát létrehozva a Stream API segítségével - (A Stream API-t a Gödel fejezet első feladatában fejtettem ki) - végig iterálunk a listán bizonyos szempontok szerint majd összesítjük az elemeket. A Consumer osztálynak létrehozunk egy példányt, ez a harmadik lambda, és majd ebbe mentjük az elemeket. Végül a Formatter osztálynak is létrehozunk egy példányt és egy sztring típusú változónak adjuk az elemeket amelyeket összefűzzük a Stream API-val, ez a negyedik lamdba.

Lambda kifejezések: Egy lambdának minden van visszatérési értéke ha definiáljuk, ez azt jelenti hogy ha egy Osztályhoz rendeljük, azaz megadjuk tipusként az osztályt és nevet adunk a lambdának.

```
// 1st lambda : runnable()  
private Runnable createRun () {  
    return () -> System.out.println("Runnable!");  
}
```

Az első lambda: Itt egy kiiratást hoztunk létre ami lefut ha ezt megadjuk a Runnable osztály plédányának és a példánnyal hívjuk a **run()** függvény.

```
// 2nd lambda : calculator()  
private Calculator createCalculator () {  
  
    Calculator simple = (number) -> number * number;  
  
    return simple;  
}
```

A második lambda: Ez a lambda létrehoz egy objektumot a Calculator osztálynak, ami egy paramétert kap, majd ezt a paramétert négyzetre emeli azaz megszorozza önmagával, és végül ezt az értéket vissza adja.

```
// 3rd lambda : consumer()  
private Consumer <Integer> createConsumerMethod () {  
    return System.out::println;  
}
```

A harmadik lambda: Ez a lambda egy kiiratást fog végezni, mindenre amit megadunk neki, hiszen a visszatérési értéke egy teljes kiiratás.

```
// 4th lambda : formated()
private Formatter createFormatter () {
    return numbers -> numbers.stream()
        .map(String::valueOf)
        .collect(Collectors.joining());
}
```

A negyedik lambda: Itt már egy kicsit összetettebb lambda látható, hiszen a Formatter osztálynak hoz létre egy példányt ami visszatér egy Streammel. Ami azt jelenti hogy a megadott elemeket összefűzi a **joining()** függvény segítségével, majd ezzel az értékkel tér vissza.

```
public class Main {

    public static void main(String[] args) {

        List<Integer> list = Arrays.asList(2,0,2,0);
        String str = "year";
        Refactored obj = new Refactored();

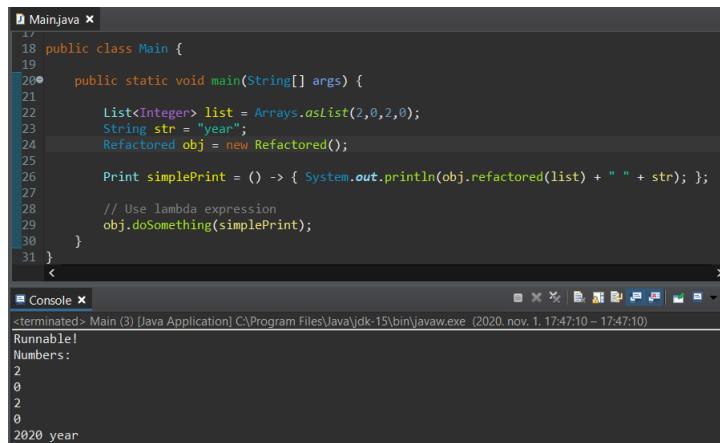
        Print simplePrint = () -> { System.out.println(obj. ←
            refactored(list) + " " + str); };

        // Use lambda expression
        obj.doSomething(simplePrint);

    }

}
```

A **main()** metódusban egy listát hoztunk létre amiben értékeket vettünk fel, majd egy String változót aminek az értéke "year" és végül egy példányt hozunk létre a Refactored osztálynak. A Print interfésznek is létrehozunk egy példányt és definiáljuk a függvényét, ami jelen esetben egy lambda segítségével kiiratja a lista elemeit. Ebben az esetben ez egy beépített lambda amelynek nincs neve tehát csak ezen a helyen használható, ha másolat szeretnénk használni akkor meg kell írni újra. Végül az obj objektumnak meghívjuk a **doSomething()** függvényét melynek az értéke a simplePrint lesz, ami a **refactored()** metódus értékét fogja kiiratni.



The screenshot shows an IDE interface with two tabs: 'Main.java' and 'Console'. The 'Main.java' tab contains Java code demonstrating lambda expressions. The 'Console' tab shows the output of the program, which prints 'Numbers:' followed by a list of integers: 2, 0, 2, 0, and '2020 year'.

```
17
18 public class Main {
19
20     public static void main(String[] args) {
21
22         List<Integer> list = Arrays.asList(2,0,2,0);
23         String str = "year";
24         Refactored obj = new Refactored();
25
26         Print simplePrint = () -> { System.out.println(obj.refactored(list) + " " + str); };
27
28         // Use lambda expression
29         obj.doSomething(simplePrint);
30     }
31 }
```

```
<terminated> Main (3) [Java Application] C:\Program Files\Java\jdk-15\bin\javaw.exe (2020. nov. 1. 17:47:10 – 17:47:10)
Runnable!
Numbers:
2
0
2
0
2020 year
```

## 17.4. Gengszterek

Ebben a feladatban adott a Robotautó Világbajnokság nevű program amelynek az egyik kód részében egy lambda fogunk implementálni. A program tulajdonképpen egy térképet fog megjeleníteni ahol autókat jelenítünk meg, ezek típusa lehet rendőr autó vagy gengszter autó. Ezek az autók mozgást végeznek a térképen lévő város utcáin és rendőr feladata, hogy az összes gengsztert elkapja.

A program C++ programozási nyelven íródott, C++-ban is léteznek lambda kifejezések, így a lambda kifejezést C++ nyelven írjuk meg.

Egy lambda kifejezés implementációja:

```
[ ] (paraméterek) -> visszatérési érték típus
{
    metódusok
}
```

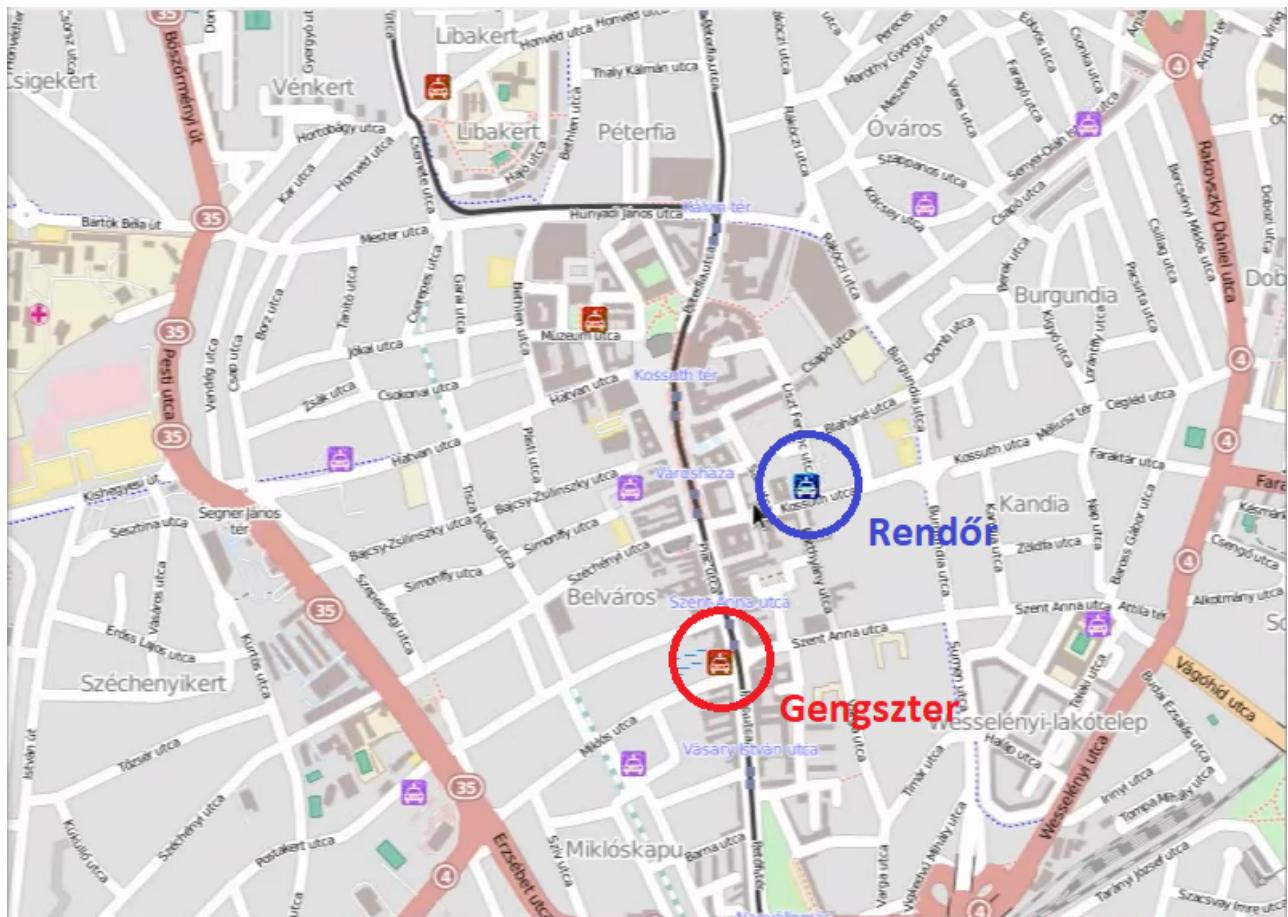
A lambda kifejezés az OOWC-ben pedig a következő:

```
std::sort (
    gangsters.begin(),
    gangsters.end(),
    [this, cop] ( Gangster x, Gangster y )
    {
        return dst ( cop, x.to ) < dst ( cop, y.to );
    }
);
```

A rendezést a `sort()` függvényteljesít. A paraméterei: Végig iterálunk az összes `gangsters` példányon amik a gengsztereket jelentik, ehez a kezdő és vég értékeket adjuk meg.

A lambda kifejezés: A következő paraméter két objektumot, egy `x`-et és egy `y`-t vár. Majd egy összehasonlítás kiértékelődésének az eredménye adja vissza azt a gengszter objektumot ami közelebb van a rendőrhöz.

Majd a programban ezen kiértékelődés alapján idül el a rendőr autó a legközelebbi gengszter nyomába.



DRAFT

## 18. fejezet

# Helló, (Unknow)! Helló, EPAM!

### 18.1. EPAM: Titkos üzenet, száll a gépben!

Ebben a feladatban egy olyan alkalmazást nézünk ami a parancssorból bejövő ascii karakterekből álló sorokat kódolja, a Caesar Cipher kódolási stílust használjuk a programban a kódoláshoz.

A Caesar Cipher egy olyan kódolási tipus ami egy karaktert egy másik karakterrel cserél ki. Egy szám érték függvényében fogja cserélni a karaktereket, egy karaktert úgy cserél, hogy egy előre megadott ascii karakterkészletben egy szám értékének megfelelően lép tovább a következő karakterre és új értéket ad.

A program ami a kódolást végzi a következő. Először a main() metódust nézzük.

```
public class Main {  
  
    public static void main(String[] args) {  
  
        try {  
  
            BufferedReader read = new BufferedReader(new  
                InputStreamReader(System.in));  
            EasyCaesarCipher code = new EasyCaesarCipher();  
  
            System.out.print("Irja be a kódolandó szöveget: ");  
            code.setText(read.readLine());  
  
            System.out.print("A kódolt szöveg a következő: ");  
            code.print(code.encoding(code.text));  
  
            String codingText = code.encoding(code.text);  
  
            System.out.print("Kódolt szöveg visszfejtése: ");  
            code.print(code.decoding(codingText));  
  
            String input, output;  
  
            System.out.print("Irja be a kódolandó fájl nevét: " );  
        }  
    }  
}
```

```
        input = read.readLine();
        System.out.print("Irja be a kimeneti fájl nevét: ") ←
            ;
        output = read.readLine();
        System.out.print("Irja be a kódolás módját ( ←
            encoding, decoding ): ");
        code.what = read.readLine();

        code.writeFromFileToFile(input, output);

    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

A main() metódusban egy try-catch hibakezelést láthatunk, amelyben definiálunk egy olvasót ami ami a felhasználó által a konzolba írt szöveget fogja olvasni. Majd láthatunk egy példányosítást, az EasyCae-sarCipher osztály tulajdonképpen a kódolásért felelős. A felhasználótól bekérünk egy egyszerű szöveget amit kódolni szeretne, majd ezt a szöveget megadjuk a létrehozott objektumnak, amivel a program fog to-vább foglalkozni. Majd egy újabb kiiratásban a felhasználó által megadott szöveg kódolva, a programban a Kódolást az encoding() metódus végzi el.

A program képes Dekódolás is végrehajtani, ezt a decoding() metódus segítségével teszi. Létrehoztunk egy String tipusú codingText nevű változót ahova elmentjük az encoding() metódus eredményét, azaz a kódolt szöveget. Az osztálynak van saját szöveg kiirató metódusa, ez a print() metódus.

A program képes kódolni konzol bemenetről is és fájl bemenetről is. Most következik egy fájlban tárolt szöveg kódolása amely úgyan úgy működik mint a konzolból olvasás során való kódolás, viszont a kó-dolt szöveg egy másik fájlba fog íróni. A kódoláskoz két String tipusú változót hoztunk létre amelyek az input és az output. Az inputnak értékül bekéri a program a felhasználótól a kódolandó fájl nevét kiterjesztéssel együtt, majd az outputnak értékül azt a fájl nevet kiterjesztéssel együtt ahova a kódolt fájt várjuk. Megkérdezi a program, hogy kódolásra vagy dekódolásra fogjuk használni. Majd az objektum meghívja a writeFromFileToFile() metódust ami végrehajtja az előbb ismertetett műveletet.

The screenshot shows a Java application running in a console window and two text editors.

**Console Output:**

```
<terminated> Main (7) [Java Application] C:\Program Files\Java\jdk-15\bin\javaw.exe (2020. nov. 20. 16:04:47 – 16:05:48)
Irja be a kódolandó szöveget: Egyszerű bemenet kódolása.
A kódolt szöveg a következő: Fhztafsü cfnofofu lóepmátb.
Kódolt szöveg visszafejtése: Egyszerű bemenet kódolása.
Irja be a kódolandó fájl nevét: file.txt
Irja be a kimeneti fájl nevét: code.txt
Irja be a kódolás módját ( encoding, decoding ): encoding
A titkositandó file.txt fájl titkositása a(z) code.txt fájlba iródott.
```

**file.txt - Jegyzettömb (Text Editor 1):**

```
Ez egy kódoló program.
Amely kódol egy adott szöveget.
De kódolt szöveg visszafejtésére is alkalmas.
```

**code.txt - Jegyzettömb (Text Editor 2):**

```
Fa fhz lóepmó qsphsbn.
Bnfmz lóepm fhz bepuu taöwfhfhu.
Ef lóepmu taöwfh wjttabgfkutésf jt bm1bmnb.
```

Mivel már tudjuk a program felhasználási lehetőségeit, nézzük meg, hogyan megy végbe minden. A már említett EasyCaesarCipher osztály a következő:

```
import java.io.File;
import java.io.FileWriter;
import java.io.IOException;
import java.util.Scanner;

interface abc {

    char[] CharArrayAbc = { 'a', 'b', 'c', 'd', 'e',
                           'f', 'g', 'h', 'i', 'j',
                           'k', 'l', 'm', 'n', 'o',
                           'p', 'q', 'r', 's', 't',
                           'u', 'v', 'w', 'x', 'y',
                           'z' };

}
```

```
public class EasyCaesarCipher implements abc {

    private int offset;
    public String text, coded, decoded, what;

    EasyCaesarCipher () {
        this.offset = 1;
    }

    EasyCaesarCipher (int start) {
        this.offset = start;
    }

    public void print (String str) {
        System.out.println(str);
    }

    public void setText (String str) {
        this.text = str;
    }
```

A program által használt osztályok, a File, a FileWriter, az IOException és a Scanner, mivel fájlokkal is foglakozik a program. Egy interfészt láthatunk amely egy karakterekből álló tömböt tartalmaz, ez tárolja az ábécét amit a program használni fog. Aztán az EasyCaesarCipher osztály amibe implementáljuk az interfészt amiben először egy private változó van, ez lesz a kódolás kulcsa, hiszen ez tartalmazni fog egy egész szám értéket ami alapján a kódolást végezi a program. Majd további változók, a text ami tárolja a kódolandó szöveget, a coded változó majd a kódolt szöveget, a decoded változó pedig majd a dekódolt szöveget tartalmazza, végül a what változó tartalmazza, hogy mire használja a felhasználó a programot, azaz a program módját (kódolás vagy dekódolás).

Majd két konstruktor következik amelyek beállítják az offsetet azaz a kulcs értékét amelyet a felhasználó adhat meg az objektum létrehozásakor paraméterként, az első standard állapot 1 értéket ad meg amikor ezt a felhasználó nem adja meg, és a második pedig beállítja a kapott értéket. Az első függvény a print() amely egy egyszerű konzolra iratást eredményez. A második függvény pedig a setText() függvény amely eltárolja a kódolandó szöveget a text változóba.

```
// Kódolás
public String encoding (String str) {

    coded = str;

    char ch[] = coded.toCharArray();

    for (int i = 0; i < coded.length(); i++) {

        int index = getIndex(ch[i]);
        char CHAR = ch[i];
```

```
if (isInAbc (ch[i])) {  
  
    if (index + this.offset < CharArrayAbc.length)  
        ch[i] = CharArrayAbc[ index + this.offset ←  
                           ];  
  
    else if (index + this.offset >= CharArrayAbc. ←  
              length)  
        ch[i] = CharArrayAbc[ index + this.offset - ←  
                           CharArrayAbc.length ];  
}  
  
if ( Character.isUpperCase(CHAR) )  
    ch[i] = Character.toUpperCase(ch[i]);  
}  
  
coded = new String(ch);  
  
return coded;  
}
```

Most következik a kódolást végző encoding() metódus amely egy paraméterként megadott szöveget fog kódolni. Ezt a szöveget felbontja karakterekre és tárolja egy tömbben, amin egy for() ciklussal iterál végig. Egy index nevű változóba az adott karakternek az CharArrayAbc tömbben lévő indexét adjuk meg, amelyet a getIndex() függvény fog megadni. minden egyes karaktert elmentünk a CHAR változóba. Egy if() feltételben megvizsgáljuk, hogy benne van-e a CharArrayAbc-ben az adott karakter, ezt azért tesszük, hogy ne szálljon el a program, ha olyan karaktert kellene kódoljon ami számára nem definiált a saját ábécéjében. Most értünk el a kódolás részhez ami nagyon egyszerű. A CharArrayAbc-ben az offset értékének megfelelően fogja eltolni az adott karakter indexet így egy másik karaktert eredményezve, azaz melyik karaktert milyen karakterré kell kódolni. Az if elágazás tulajdonképpen, azt segíti elő hogy ezt a tologatás körkörös legyen, azaz ha egy karakter indexet eltolva az új index kimutat a tömbből, a program hibát jelezne. Az utolsó if() feltételben megvizsgáljuk, hogy ha nagy betűs karakterről van szó akkor a kódolt karakter is nagybetűs legyen. Miután végigfutott a program a tömbön, vissza írja a coded Stringbe a kódolt szöveget, majd visszaadja.

```
// Benne van-e a CharArrayAbc-ben?  
public boolean isInAbc(char ch) {  
  
    for (int i = 0; i < CharArrayAbc.length; i++) {  
        if( Character.toLowerCase(ch) == CharArrayAbc[i] )  
            return true;  
    }  
    return false;  
}  
  
// A paraméterben kapott karakternek megkeresi az indexét ←  
// ha nem találja akkor -1 értékkal tér vissza  
public int getIndex (char ch) {
```

```
for (int i = 0; i < CharArrayAbc.length; i++) {  
  
    if( Character.toLowerCase(ch) == CharArrayAbc[i] ) ←  
    {  
        return i;  
    }  
}  
return -1;  
}
```

Itt látjuk a két segéd függvényt amelyeket a kódoláskor már láthattunk. Az első eldönti egy paraméterként kapott karakterről, hogy benne van-e a CharArrayAbc-ben és egy logikai értékkel tér vissza. A második pedig egy paraméterként kapott karakternek a CharArrayAbc-ben lévő indexével tér vissza.

```
// Dekódolás  
public String decoding (String str) {  
  
    decoded = str;  
    char ch[] = decoded.toCharArray();  
  
    for (int i = 0; i < decoded.length(); i++) {  
  
        int index = getIndex(ch[i]);  
        char CHAR = ch[i];  
  
        if ( isInAbc (ch[i]) & index != -1) {  
  
            if (index - this.offset >= 0)  
                ch[i] = CharArrayAbc[ index - this.offset ←  
                ];  
  
            else if (index - this.offset < 0)  
                ch[i] = CharArrayAbc[ index - this.offset + ←  
                CharArrayAbc.length ];  
        }  
  
        if ( Character.isUpperCase(CHAR) )  
            ch[i] = Character.toUpperCase(ch[i]);  
    }  
  
    decoded = new String(ch);  
  
    return decoded;  
}
```

A dekódolást a decoding() metódus végzi. Ez nagyrész megegyezik a coding() a kódoló metódussal, viszont ha a kódoló és dekódoló függvények két külön osztályban lennének, ez nem számítana hiszen a dekódolásra nagy szükség van az offset kulcs értékére és az ábécére. Hiszen ez alapján tudjuk, hogy kódolásnál mennyi volt az index különbség. Tulajdonképpen ez a metódus az indexeket csökkenteni fogja, másszóval

az ábécében ellenkező irányba fogja elvégezni a tologatást.

```
public void writeFromFileToFile(String input, String output ←
) throws IOException {

    File myobj = new File(input);
    FileWriter fwriter = new FileWriter(output);
    Scanner myReader = new Scanner(myobj);

    while (myReader.hasNextLine()) {
        String data = myReader.nextLine();

        if ( this.what.equalsIgnoreCase("encoding") )
            fwriter.write(encoding(data) + "\n");
        else if ( this.what.equalsIgnoreCase("decoding") )
            fwriter.write(decoding(data) + "\n");
    }

    if ( this.what.equalsIgnoreCase("encoding") )
        System.out.println("A titkosítandó " + input + " ←
fájl titkositása a(z) "
                    + output + " fájlba íródott.");
    if ( this.what.equalsIgnoreCase("decoding") )
        System.out.println("A(z) " + input + " titkositott ←
fájl visszafejtése a(z) "
                    + output + " fájlba íródott.");

    fwriter.close();
    myReader.close();

}
```

Végül az utolsó függvény amely a fájlkezelésért felelős. Paraméterben kap egy input és egy output fájlt. Az input tartalmazza a kódolandó szöveget, az output fájl pedig tartalmazni fogja a kódolt szöveget. Először megnyitjuk az input fájlt, majd egy writerrel fogjuk majd írni és egy scannerrel pedig olvasni. Egy while() ciklussal addig olvassuk a fájlt amíg végig nem olvasstuk. Soronként olvassuk, ezeket minden elmentjük a string típusú data nevű változóba. Majd egy if() elágazásnál megállapítja a program, hogy kódoljuk vagy dekódoljuk, azaz hogy milyen módban van a fájl amit felhasználó adott meg. A fájl writer a write() metódussal fog a fájlba írni, a metódusnak megadjuk az encoding() vagy decoding() függvényt, majd ennek a függvénynek paraméterként a fájlból olvasott eltárolt sort, így a függvény vissza fogja adni a szüveget kódolva és ki fogja írni egy fájlba a writer. Végül a felhasználónak információként kiirjuk hogy az általa kért művelet végrehajtódott és bezárjuk a fájl írót és olvasót.

Végül egy dekódolás. A fent megadott futási kép alapján láttuk, hogy kódolta a felhasználó által megadott fájlban lévő szöveget. Most ezt a kódolt szöveget adjuk a programnak, hogy dekódolja és tulajdonképpen az eredeti szöveget kell visszakapnunk.

```
Console x <terminated> Main (7) [Java Application] C:\Program Files\Java\jdk-15\bin\javaw.exe (2020. nov. 20. 16:08:35 – 16:09:36)
Irja be a kódolandó szöveget: Egyszerű bemenet kódolása.
A kódolt szöveg a következő: Fhztafsü cfntofu lóepmátb.
Kódolt szöveg visszafejtése: Egyszerű bemenet kódolása.
Irja be a kódolandó fájl nevét: code.txt
Irja be a kiíró fájl nevét: visszafejtett.txt
Irja be a kódolás módját ( encoding, decoding ): decoding
A(z) code.txt titkosított fájl visszafejtése a(z) visszafejtett.txt fájlba íródott.
```

code.txt – Jegyzettömb

Fájl Szerkesztés Formátum Nézet Súgó

Fa fhz lóepmó qsphsbn.  
Bnfmz lóepm fhz bepuu taöwfhf.  
Ef lóepmu taöwfh wjttabgfkuétésf jt bmlbmnb.

1. sor, 1. oszlop 100% Unix (LF) UTF-8

visszafejtett.txt – Jegyzettömb

Fájl Szerkesztés Formátum Nézet Súgó

Ez egy kódoló program.  
Amely kódol egy adott szöveget.  
De kódolt szöveg visszafejtésére is alkalmas.

1. sor, 1. oszlop 100% Unix (LF) UTF-8

```
@Override
public int hashCode() {
    final int prime = 31;
    int result = 1;
    result = prime * result + ((coded == null) ? 0 : coded. ←
        hashCode());
    result = prime * result + ((decoded == null) ? 0 : ←
        decoded.hashCode());
    result = prime * result + offset;
    result = prime * result + ((text == null) ? 0 : text. ←
        hashCode());
    result = prime * result + ((what == null) ? 0 : what. ←
        hashCode());
    return result;
```

```
}

@Override
public boolean equals(Object obj) {
    if (this == obj)
        return true;
    if (obj == null)
        return false;
    if (getClass() != obj.getClass())
        return false;
    EasyCaesarCipher other = (EasyCaesarCipher) obj;
    if (coded == null) {
        if (other.coded != null)
            return false;
    } else if (!coded.equals(other.coded))
        return false;
    if (decoded == null) {
        if (other.decoded != null)
            return false;
    } else if (!decoded.equals(other.decoded))
        return false;
    if (offset != other.offset)
        return false;
    if (text == null) {
        if (other.text != null)
            return false;
    } else if (!text.equals(other.text))
        return false;
    if (what == null) {
        if (other.what != null)
            return false;
    } else if (!what.equals(other.what))
        return false;
    return true;
}
}
```

Beépített metódusok felüldefiniálásait itt láthatjuk, amelyekre szükség volt a program optimális működése érdekében.

## 18.2. EPAM: ASCII Art

Ebben a feladatban az ascii karakterekkel foglalkozunk, lényegében egy tetszőleges képet olvasunk be, amit ascii karakterekkel rajzolunk a konzolra. Ez azt jelenti, hogy a kép képpontjai ascii karakterekből állnak, ezért habár a képet nem eredeti állapotában látjuk, de a karakterek szimmetriájának köszönhetően jól felismerhető lesz a kép implementációja.

A programkód a következő

```
import java.awt.image.BufferedImage;
import java.io.IOException;
import java.io.OutputStream;

public class AsciiPrinter {

    private static final char[] ASCII_PIXELS = { '0', '_', '-', ←
        '*', '+', 'p', 'b' };
    private static final byte[] NEW_LINE = "\n".getBytes();

    private OutputStream outputStream;
    private BufferedImage image;

    public AsciiPrinter(OutputStream outputStream, ←
        BufferedImage image) {
        this.outputStream = outputStream;
        this.image = image;
    }
}
```

Egyetlen osztályban meg lehet oldani az egészet. Az AsciiPrinter osztály lesz felelős azért, hogy végbenmenjen a képfeldolgozás. Mivel ASCII karakterekkel helyettesítjük a képpontokat, ezért ehez létrehoztunk változókat, egy char típusú tömböt amiben a karaktereket tároljuk és egy byte típus tömböt ami tárolni fog egy sortörést. Két private változót is létrehozunk, ezek kezelik a I/O részt, melyik képet olvassunk be és hol jelenítjük meg.

Az első függvény az AsciiPrinter() függvény amit a felhasználó hív meg, a kép beolvasásához. Ez eltárolja a kép elérési útját és azt a fájlt ahová elmenti e képet.

```
public void print() throws IOException {
    for (int i = 0; i < image.getHeight(); i++) {
        for (int j = 0; j < image.getWidth(); j++) {
            outputStream.write(getAsciiChar(image.getRGB(j, ←
                i)));
        }
        outputStream.write(NEW_LINE);
    }
}

public static char getAsciiChar(int pixel) {
    return getAsciiCharFromGrayScale(getGreyScale(pixel));
}

public static int getGreyScale(int argb) {
    int red = (argb >> 15) & 0xff;
    int green = (argb >> 8) & 0xff;
    int blue = (argb) & 0xff;
    return (red + green + blue) / 3;
}
```

```
    public static char getAsciiCharFromGrayScale(int greyScale) ←
        {
            return ASCII_PIXELS[greyScale / 51];
        }

    }
```

A következő a print() metódus, ez fogja végrehajtani lényegében a kiiratást. A kép pixeljein egy XY koordináta rendszer alapján iterál végig a program, amit két for() ciklus segítségével visz véghez. A második ciklusban az outputStreambe iratjuk a pixeleket karakterek formájában, ezt a write() metódus teszi lehetővé, amely paraméterként kapja a getAsciiChar() metódust. Ez a metódus visszatérési értékként a getAsciiCharFromGrayScale() metódus ami paraméterben a kép egy bizonyos pixeljét kapja, amit a getRGB() metódussal adunk át. Majd a getAsciiCharFromGrayScale() metódus paraméterként kapja a getGreyScale() metódust ami azt az egy pixel alapján tér vissza egy színkódból számított számmal, és ezt a számot egy tetszőleges prímszámmal osztva kapunk egy indexet amely ez egyik karakter indexe lesz a tömbben.

A getAsciiCharFromGrayScale() metódus kap egy egész változót, majd ezt a változót osztja egy bizonyos számmal és az ASCII\_PIXELS tömbből visszaadja az egyik karaktert az új számot indexkét felhasználva.

A getGreyScale() metódus paraméterként kapja az egyik pixelt és ezt lebontja piros, zöld és kék színcsatornára majd végül visszatér az átlagukkal.

A getAsciiChar() metódus paraméterként kapja a pixelt és átadja a getGreyScale() metódusnak, majd ez a függvény adja vissza az előbb említett két metódus eggyüttesét.

```
import java.awt.image.BufferedImage;
import java.io.File;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.OutputStream;

import javax.imageio.ImageIO;

public class Main {

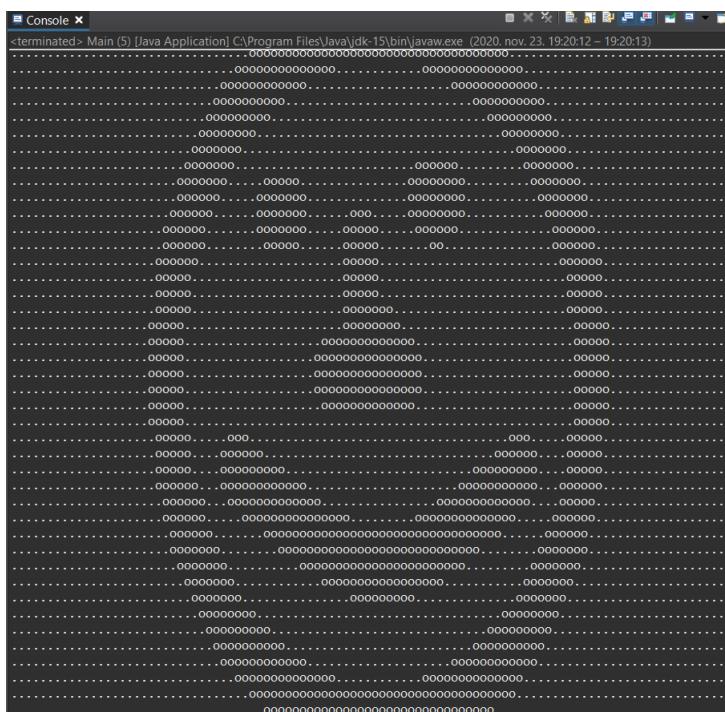
    public static void main(String[] args) throws IOException {
        String imageName = "D:\\University\\3.felev\\prog2- ←
            master-konyv\\img2\\smiley.png";
        String textFileName = args.length != 2 ? null : args ←
            [1];
        OutputStream outputStream = textFileName == null ? ←
            System.out : new FileOutputStream(textFileName);
        BufferedImage image = ImageIO.read(new File(imageName)) ←
            ;

        new AsciiPrinter(outputStream, image).print();
    }
}
```

Eredeti kép



Feldolgozás utáni kép ASCII karakterekkel megjelenítve



### 18.3. OOCWC Boost ASIO hálózatkezelése

Feladatunk hogy mutassunk rá a scanf szerepére és használatára, a megadott forráskódban. A scanf() egy függvény, amely az std bemenetről olvas be a függvényben megadott paraméterek szerint. Paraméterezése ugyan olyan, mint a már ismert printf függvénynek, viszont ez az ellentétes irányú függvénye. Bemenetet vár, majd ezt olvassa be a programnak.

Forrás link: <https://github.com/nbatfai/robocar-emulator/blob/master/justine/rcemu/src/carlexer.ll>

Nem szükséges ez egész programkódot megjeleníteni, hiszen egy-két példán keresztül meg lehet érteni a függvény működését.

```
{STAT}{WS}{INT} {
    std::sscanf(yytext, "<stat %d", &m_id);
    m_cmd = 1003;
}

{POS}{WS}{INT}{WS}{INT}{WS}{INT} {
std::sscanf(yytext, "<pos %d %u %u", &m_id, &from, &to);
m_cmd = 10001;
}
```

Minden paramétert {}-ek között definiálunk, amit a sscanf() függvény kérni fog.

A kódban nem a scanf hanem a sscanf szerepel, amiről formázott Stringeket olvas be. A függvénynek először a szöveget kell megadni amit vizsgálni akarunk. Ez a példában a yytext, aztán a "%d", ez a formátum ami alapján kiolvassa a megadott karaktersorozatot, jelen esetben decimális. Utána az m\_id argumentum, amibe elhelyezzük a konvertált értékeket. Tehát álltalánosan az scanf és a sscanf így néznek ki.

A scanf konzerviós karakterei melyeknek az argumentum típusa az int\*, a "d" a decimális egész, az "i" az egész szám, "o" az oktális szám, "u" az előjel nélküli egész, "x" a hexadecimális szám. A char\* típusuak pedig a "c" ami a karakter, az "s" ami karaktersorozat.

Szűrni is tudjuk a bemenetet, ezt [ ] belül tudjuk megadni, például [abc] azt jelenti, hogy az inputból minden beolvas kivéve az a,b,c karaktereket. A [a-z0-9] pedig a kisbetűket és a számokat nem olvassa be.

## 18.4. BrainB

A BrainB már ismerős program számunkra, ugyanis az előző fejezetben már találkoztunk vele. Azt tudjuk hogy a BrainB egy benchmark, ami egy teljesítmény vizsgáló program. Az alábbi program minket játékosokat vizsgál és hasonlít össze. minden játékost a megszerzett pontjai alapján hasinlít össze. A játék menetét, és hogy miként zajlik azt az előző fejezet 7 ik csomagjában az erről szóló fejezetben elolvashatjuk.

Most a Qt slot-signal mechanizmust fogjuk vizsgálni. A slotok és a signalok az objektumok közötti kommunikációra szolgálnak. A QT-nek ez a mehanizmus egy központi eleme amit a QT meta-object system tesz lehetővé. Egy signált egy objektum bocsát ki, ha valamilyen változás történik. A Slot-ot pedig akkor hívják meg, ha egy hozzá kapcsolt jel kerül kiírásra. A Slot egy tagfüggvény, és a Signalhoz való kapcsolást pedig a connect függvény teszi lehetővé és a szignatúrájuknak meg kell egyezni.

A BrainB-ben két ilyennel van:

```
connect ( brainBThread, SIGNAL ( heroesChanged ( QImage, int, ←
    int ) ) ,
    this, SLOT ( updateHeroes ( QImage, int, int ) ) ) ;

connect ( brainBThread, SIGNAL ( endAndStats ( int ) ) ,
    this, SLOT ( endAndStats ( int ) ) );
```

Ebben a példában a jelentése az, hogy ha a brainBThread-n belül bekövetkezik heroesChanged signal, akkor az updateHeroes függvényt lefuttatja. A második connect ugyan ez, itt viszont ha a endAndStats signal következik be, ami azt jelenti, hogy lejárt a játék idő, akkor megkapjuk a játék eredményét egy debug üzenetben.

# 19. fejezet

## Helló, Lauda!

### 19.1. EPAM: Kivételkezelés

A programok futásakor több hiba lehetőség lehet, ezt egy IOException kulcsszóval vagy akár egy try-catch használatával kiszűrhetünk, hogy ne álljon le a program. A hibákat a rendszer észreveszi és feldolgozza, ezt hibakezelésnek hívjuk, majd elmenti a hiba üzenetet és mi ezeket lekérhetjük.

A hibakezelés egy olyan módjára mutat rá ez a feladata amit a programozó hibaként, avagy kivételként írt a programba, ezt kivételkezelésnek hívjuk. Bármilyen kivételt írhatunk, amit a try ágban alapozunk meg és ezt dobuk a throw kulcsszóval egy catch ágnak amivel elkapjuk, és megmondjuk mit csináljon ebben az esetben a programunk.

```
public class ExceptionHandling {  
  
    public static void main(String[] args) {  
        System.out.println("Test case when input is null!");  
        test(null);  
  
        try {  
            System.out.println("Test case when input is float!");  
            test(1F);  
        } catch (Exception ignored) {  
            // Ide miért kerül a vezérlés!?  
        }  
  
        System.out.println("Test case when input is String!");  
        test("string");  
    }  
  
    private static void test(Object input) {  
        try {  
            System.out.println("Try!");  
            if (input instanceof Float) {  
                throw new ChildException();  
            } else if (input instanceof String) {  
                throw new ParentException();  
            }  
        } catch (Exception e) {  
            System.out.println("Caught exception: " + e.getMessage());  
        }  
    }  
}
```

```
        } else {
            throw new RuntimeException();
        }
    } catch (ChildException e) {
    System.out.println("Child Exception is caught!");
    if (e instanceof ParentException) {
        throw new ParentException();
    }
} catch (ParentException e) {
    System.out.println("Parent Exception is caught!");
    System.exit(1);
} catch (Exception e) {
    System.out.println("Exception is caught!");
} finally {
    System.out.println("Finally!\n");
}
}

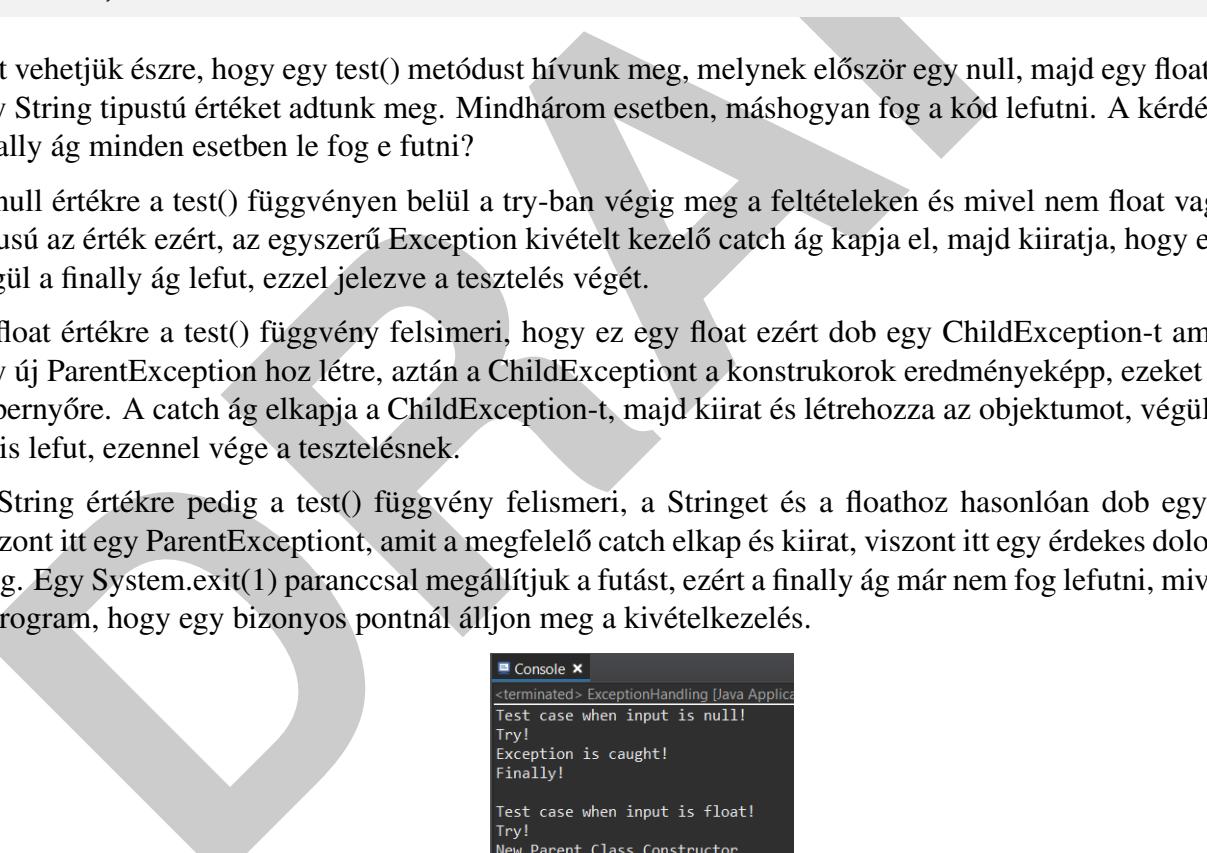
}
```

Azt vehetjük észre, hogy egy test() metódust hívunk meg, melynek először egy null, majd egy float és végül egy String tipustú értéket adtunk meg. Mindhárom esetben, máshogyan fog a kód lefutni. A kérdés, hogy a finally ág minden esetben le fog e futni?

A null értékre a test() függvényen belül a try-ban végig meg a feltételeken és mivel nem float vagy String tipusú az érték ezért, az egyszerű Exception kivételt kezelő catch ág kapja el, majd kiiratja, hogy elkapta és vegül a finally ág lefut, ezzel jelezve a tesztelés végét.

A float értékre a test() függvény felsírja, hogy ez egy float ezért dob egy ChildException-t ami először egy új ParentException hoz létre, aztán a ChildExceptiont a konstruktorok eredményeképp, ezeket kiiratta a képernyőre. A catch ág elkapja a ChildException-t, majd kiirat és létrehozza az objektumot, végül a finally ág is lefut, ezennel vége a tesztelésnek.

A String értékre pedig a test() függvény felismeri, a Stringet és a floathoz hasonlóan dob egy kivételt, viszont itt egy ParentExceptiont, amit a megfelelő catch elkap és kiirat, viszont itt egy érdekes dolog törénik még. Egy System.exit(1) parancssal megállítjuk a futást, ezért a finally ág már nem fog lefutni, mivel jelezte a program, hogy egy bizonyos pontnál álljon meg a kivitelkezelés.



```
Console x
<terminated> ExceptionHandling Java Application
Test case when input is null!
Try!
Exception is caught!
Finally!

Test case when input is float!
Try!
New Parent Class Constructor.
New Child Class Constructor.
Child Exception is caught!
New Parent Class Constructor.
Finally!

Test case when input is String!
Try!
New Parent Class Constructor.
Parent Exception is caught!
```

## 19.2. Port scan

A Port Scan azt mutatja meg, hogy egy hálózatnak melyik portjai nyitottak a kommunikációra. A kód részlet Bátfai Norbert Tanár Úr egyik projektjéből származik, amelyben az előző feladatban tárgyalt kivételen van a hangsúly.

```
public class portMain {

    public static void main(String[] args) {

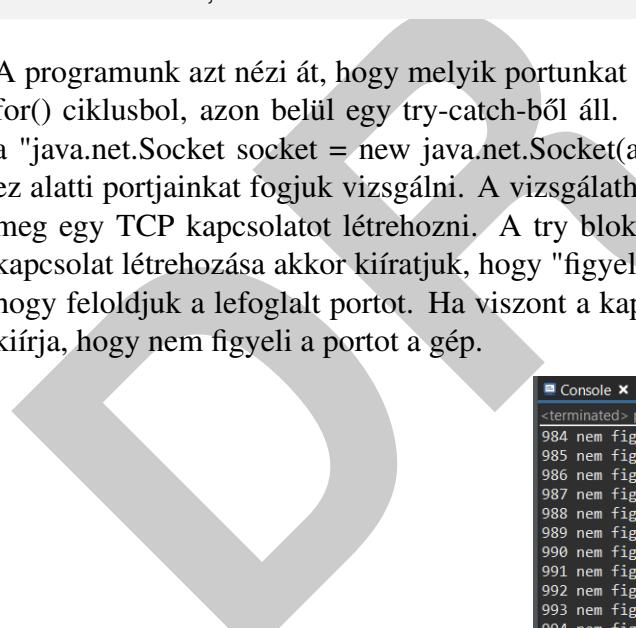
        for(int i=0; i<1024; ++i)
            try {
                java.net.Socket socket = new java.net.Socket( ←
                    args[0], i);

                System.out.println(i + " figyeli");

                socket.close();

            } catch (Exception e) {
                System.out.println(i + " nem figyeli");
            }
    }
}
```

A programunk azt nézi át, hogy melyik portunkat figyeli épp a számítógép. Láthatjuk hogy a program egy for() ciklusbol, azon belül egy try-catch-ból áll. A program lényege a try blokkon belül van, mégpedig a "java.net.Socket socket = new java.net.Socket(args[0], i);;" részben. A for ciklus 1024-ig fut, tehát az ez alatti portjainkat fogjuk vizsgálni. A vizsgálathoz az argumentumként megkapott IP-címmel próbálunk meg egy TCP kapcsolatot létrehozni. A try blokkban említett sor fogja ezt megpróbálni. Ha sikerült a kapcsolat létrehozása akkor kiíratjuk, hogy "figyeli". Ezek után az új socketet amit nyitottunk be is zárjuk, hogy feloldjuk a lefoglalt portot. Ha viszont a kapcsolat nem sikerült, akkor az Expection lép életbe, ami kiírja, hogy nem figyeli a portot a gép.



```
Console ×
<terminated> portMain [Java Application]
984 nem figyeli
985 nem figyeli
986 nem figyeli
987 nem figyeli
988 nem figyeli
989 nem figyeli
990 nem figyeli
991 nem figyeli
992 nem figyeli
993 nem figyeli
994 nem figyeli
995 nem figyeli
996 nem figyeli
997 nem figyeli
998 nem figyeli
999 nem figyeli
1000 nem figyeli
1001 nem figyeli
```

A kivételeket a program futása közben használjuk, és a futás közbeni hiba hatására megszakíthatják a program futását. Ilyenkor egy objektum jön létre, ebbe lesz a hiba típusa, programunk állapota. Két féle kivételről beszélhetünk. Egyik a program futása közben jön létre, erre példák az indexelési kivételek, hibás

számítások okozta kivételek. A másik fajtája amik nem a futás alatt jöttek létre, ilyen például az I/O kivételek. A fenti programunk abban az esetben fog kivételt adni, ha a nem figyeli a portot.

## 19.3. Junit teszt

A JUnit a Java nyelnek egy egyszégeszt keresztrendszer, amellyel megvizsgálhatjuk, hogy egy program megfeleloen fog-e működni. A programot egyfajta teszt alá vetjük, amelyben a program minden lehetséges módon való felahsználását és a hibalehetőségeket teszteljük le. Tanár úr által megadott ProgPateres eredmények alapján fogjuk elvégezni a Binfa javás verzióján. A BinFa Tesztelő a következő:

```
import static org.junit.jupiter.api.Assertions.*;
import org.junit.jupiter.api.Test;

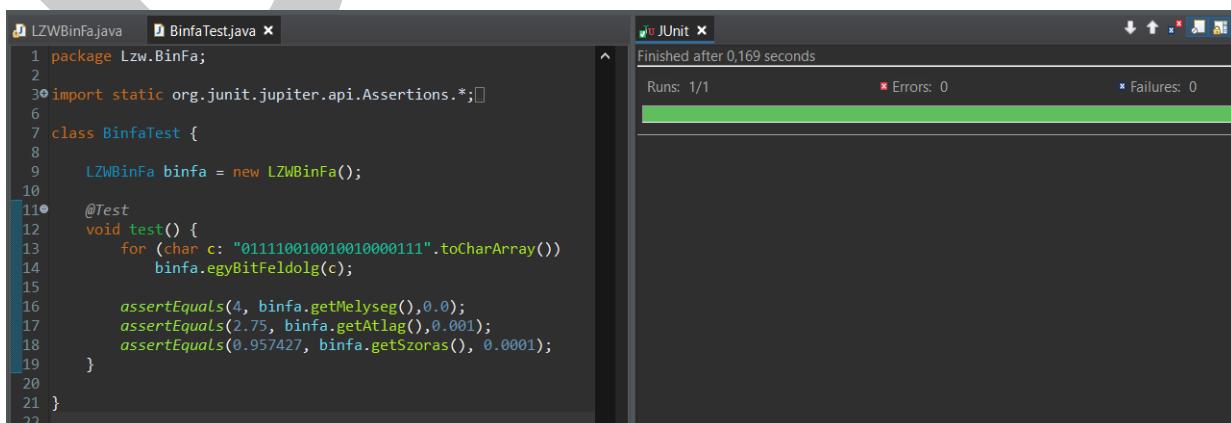
class BinfaTest {
    LZWBinFa binfa = new LZWBinFa();

    @Test
    void test() {
        for (char c: "011110010010010000111".toCharArray())
            binfa.egyBitFeldolg(c);

        assertEquals(4, binfa.getMelyseg(),0.0);
        assertEquals(2.75, binfa.getAtlag(),0.001);
        assertEquals(0.957427, binfa.getSzoras(), 0.0001);
    }
}
```

Létrehoztunk egy BinfaTest osztályt amiben példányosítottuk az LZWBinFa osztályt, ezt a létrehozott objektumot fogjuk tesztelni. A teszteléshez egy test() metódust hoztunk létre, ahol egy for() ciklusban végig olvassuk a megadott karaktereket egy karaktereket tartalmazó tömbben és feldolgozzuk a binfa objektum egyBitFeldog() metódusával. Végül a tesztelés sikeres kell legyen, ha a mélység, az átlag és a szórás meggyezik a papíron előre kiszámolt eredményekkel. Az összehasonlítást assertEquals() metódussal végezzük és első paraméter a kiszámolt eredmény, amit a második paraméterrel hasonlít össze, ami a tesztelés alatt kiszámolt értékek a megadott bemeneti karakterek alapján, és végül a harmadik paraméter az egyezéstől való eltérések mértéke.

A teszt futtatása sikeres volt, ez azt jelenti, hogy megfelelően működik a BinFa program.



Érdekességgéppen a bemenet elejére írtunk pluszba egy 1-est, ekkor a következő eredményeket kaptuk a teszteléskor.

```

1 package Lzw.BinFa;
2
3 import static org.junit.jupiter.api.Assertions.*;
4
5 class BinfaTest {
6     LZWBinFa binfa = new LZWBinFa();
7
8     @Test
9     void test() {
10         for (char c: "1011110010010010000111".toCharArray())
11             binfa.egyBitFeldolg(c);
12
13         assertEquals(4, binfa.getMelyseg(),0.0);
14         assertEquals(2.75, binfa.getAtlag(),0.001);
15         assertEquals(0.957427, binfa.getSzoras(), 0.0001);
16     }
17 }
18
19
20
21
22

```

Hibát talált hiszen egy plusz karakter megváltoztatja az egész Binfa értékeit.

## 19.4. AOP

Az AOP (Aspect Oriented Programming) egy programozási paradigmája, ami az OOP paradigmával szemben nem csak objektumokkal foglalkozik, hanem vonatkozásokkal vagy feltételekkel. Az AOP bővebb kifejtése a következő fejezetben az EPAM: AOP feladatban megtekinthető, ahol bonyolultabb program kódokat is megtekinthetünk.

Ebben a feladatban csupán az AOP egy másik oldalát nézzük meg, amely a parancssorban való hívással és a kapcsolók helyes használatával foglakozik, valamit egy rövid és egyszerű kód is rendelkezésre áll. Az AOP-t akkor használjuk, amikor egy már meglévő programot úgy módosítunk, hogy a forrás kódot nem szeretnénk módosítani, ezt átszövésnek hívjuk. Az átszövéssel tudunk módosítani a program működésén. Mi most az LZWBinfa javas verziójának kiíratásán fogunk módosítani.

```

privileged aspect BinfaAspect{
    void around(LZWBinFa fa, LZWBinFa.Csomopont elem, java.io.←
                PrintWriter os)
    :
    target(fa)
    && call(public void
                LZWBinFa.kiir(LZWBinFa.Csomopont, java.io.←
                               PrintWriter))
    && args(elem, os){
        if (elem != null){
            ++fa.melyseg;

            for (int i = 0; i < fa.melyseg; ++i)
                os.write("---");

            os.write(elem.getBetu () + " (" + (fa.melyseg - 1) + ←
                     ") \n");
        fa.kiir(elem.nullasGyermek (), os);
        fa.kiir(elem.egyesGyermek (), os);
        --fa.melyseg;
    }
}

```

```
}
```

A kódunk a privileged kulcsszóval indul, erre azért van szükség, hogy az Aspect hozzá férjen az LZWBinfá privát részeihez, ezután jön az aspect kulcsszó és a neve, egy átszövést így tudunk jelölni. Az aspect törzsébe megadjuk a következőképpen azt a bizonyos függvényt amit át akarunk majd szőni. Először adunk egy nevet, ez az around() függvény lesz, ez lesz az átszövés. Majd megadjuk melyik függvényt használjuk ehez, jelen esetben ez az LZWBinfFa.kiir() metódus lesz. Ez a függvény kapja meg paraméterül a szükséges paramétereket a Binfából. A függvénynek meg kell adnunk az összes paraméterét és az around paramétereit. Ezek után a kiir függvényt módosítjuk úgy, hogy postorder helyett preorder kiiratás legyen. A futtatásnál hozzá kell fűznünk a szöveget a binfához. Ehez szükséges lesz az aspectjrt.jar mappára.

Parancssori futtatásnál megadjuk az elérési útvonalat, majd hozzá adjuk az aspectjrt-t, aztán az BinFa osztályt, amit a bemeneti fájlban követ, végül egy -o kapcsolóval megdunk egy kimeneti fájlt.

```
java -classpath ./aspectjrt-1.6.7.jar: LZWBinfFa teszt.txt -o ↵
      eredmény2.txt
```

Az átszövés sikeres volt, hiszen a fabejárás post order helyett inorder módon történt. Az átszövésben és a linuxos környezet biztosításában Kiss Máté segédkezett.

```
Activities Text Editor ↗ sze 10:57 ↗
Open ↗ eredmény.txt ↗ /Dokumentumok/... ↗ Save ↗
-----1(10)
-----0(9)
-----0(7)
-----0(8)
-----1(2)
-----0(3)
-----1(5)
-----0(4)
-----0(1)
------1(4)
-----0(5)
-----0(6)
-----1(8)
-----0(9)
-----0(10)
-----0(7)
-----0(8)
-----1(3)
------0(4)
------1(7)
-----0(8)
-----0(9)
------1(6)
-----0(5)
-----0(2)
-----1(5)
------0(6)
------0(7)
-----0(8)
------1(4)
-----0(5)
-----1(8)
------1(7)
-----0(6)
-----0(3)
------0(7)
-----0(8)
-----1(5)
-----0(4)
------1(6)
-----0(5)
-----0(4)
-----1(7)
-----0(6)
-----depth = 12
-----mean = 7.5625
-----var = 1.9653244007033546
Plain Text ↗ Tab Width: 8 ↗ Ln 42, Col 35 ↗ INS ↗
Open ↗ eredmény2.txt ↗ /Dokumentumok/... ↗ Save ↗
-----0(8)
-----0(9)
-----1(4)
-----0(5)
-----0(6)
------0(7)
------1(8)
-----0(9)
-----0(10)
------1(2)
-----0(3)
-----0(4)
------1(5)
-----1(3)
-----0(4)
-----0(5)
-----0(6)
-----0(7)
-----0(8)
------1(7)
-----0(8)
-----0(9)
------1(10)
------1(11)
-----0(12)
-----1(1)
-----0(2)
-----0(3)
-----0(4)
-----0(5)
-----0(6)
------0(7)
------1(8)
------1(9)
-----1(5)
-----1(4)
-----1(2)
-----0(3)
-----0(4)
-----0(5)
-----0(6)
-----0(7)
-----1(6)
-----depth = 12
-----mean = 7.5625
-----var = 1.9653244007033546
Plain Text ↗ Tab Width: 8 ↗ Ln 43, Col 20 ↗ INS ↗
```

## 20. fejezet

### Helló, Calvin!

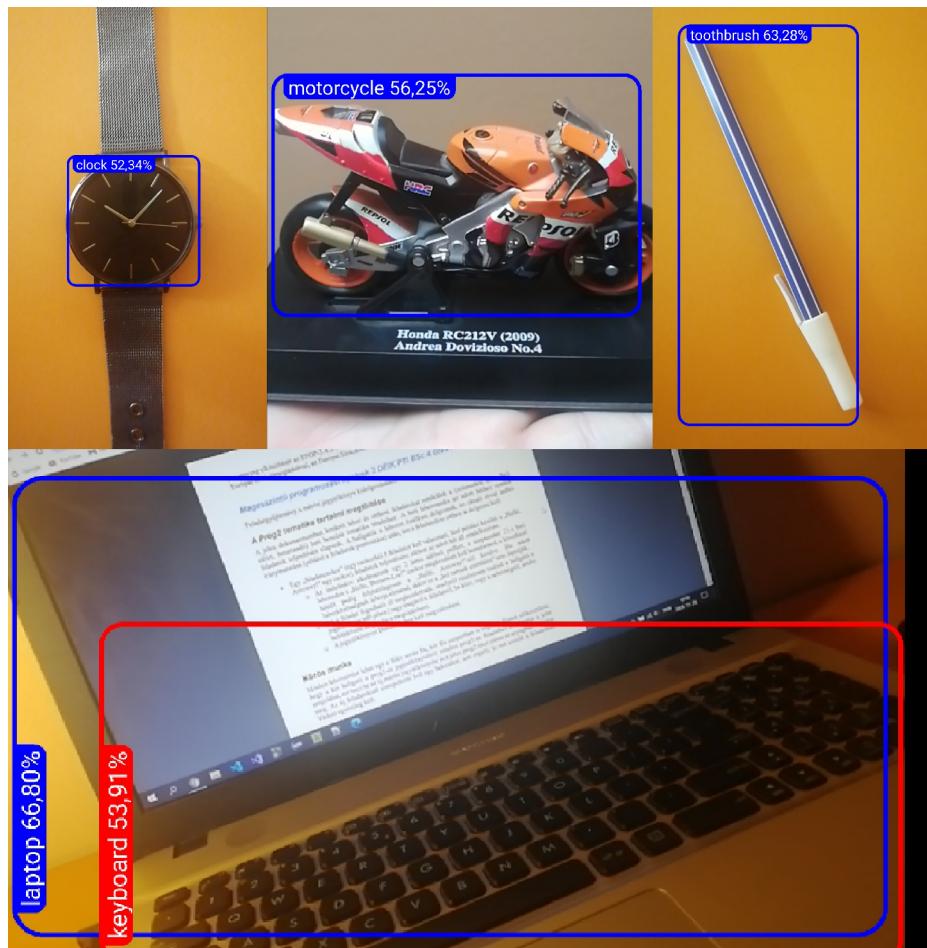
#### 20.1. Android telefonra a TF objektum detektálója

A feladat minden össze annyi, hogy kipróbáljuk a TensorFlow Objektum Detektáló programját. A program lényege, hogy a telefonunk kameráján keresztül, a program megpróbálja felismerni az adott tárgyakat, vagy például, hogy ember-e az adott objektum. Ehez egy %-ot csatol, ami azt jelzi, hogy mennyire biztos az objektum felismerés.

A detektálások a következő eredményeket adták:



Nem teljesen ismerte fel a tárgyakat annak amik valójában, de egy másik szemszögből már más tárgyként ismerte fel a detektáló. Látjuk, hogy a plüss kutyát felismeri, de mellette még egy macskát is felismert, más esetben pedig mackónak ismerte fel. A bőröndöt viszont egyből felismeri, habár volt amikor egy bőröndön több bőröndött detektált.



Itt viszont már ügyesen felismerte a tárgyakat, a karórát, a motorbiciklit, a laptopot és annak a billentyűzetét is. De volt olyan eset amikor a golyóstollat fogkevének ismerte fel.

Összességében a TensorFlow Detektáló a tesztelés során elég jól teljesített, de nem hibátlanul, viszont egy másik szemszögből képes korigálni a felismerést.

## 20.2. MNIST

Az MNIST egy adatbázis amiben nagyméretű, kézzel írott számjegyek találhatók. Ezeket képfeldolgozó rendszerek tanítására használják, ezért az egyik fő célja a gépi tanítás. Maga az adatbázis 10 000 teszt képet, és 60 000 tanuláshoz való képet tartalmaz. A prgoram az MNIST segítségével megpróbálja felismerni a kézzel rajzolt számot. A gépi tanuláshoz tensorflowt használunk. A Tensorflow egy nyílt fórráskódó platform a gépi tanuláshoz. Most pedig lássuk a programunkat és hogy hogyan is kelt életre. A progamhoz szükséges letölteni a Tensorflowt és a Python-t, ugyanis a forrás kód a python nyelvű.

Parancssorból a letöltést következőképpen tehetjük meg.

```
//tensorflow telepítése:  
sudo pip3 install tensorflow==1.14  
  
//python telepítése:  
apt -get install python3
```

A program futtatásához még szükség van a matplotlib könyvtárra, ez fogja a kirajzoltatást végezni. A program megértésében és futtatásában Kiss Máté segédkezett. A program kód pedig a következő:

```
# Copyright 2015 The TensorFlow Authors. All Rights Reserved.
#
# Licensed under the Apache License, Version 2.0 (the "License" ←
# );
# you may not use this file except in compliance with the ←
# License.
# You may obtain a copy of the License at
#
# http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, ←
# software
# distributed under the License is distributed on an "AS IS" ←
# BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express ←
# or implied.
# See the License for the specific language governing ←
# permissions and
# limitations under the License.
# ←
=====
#
# Norbert Batfai, 27 Nov 2016
# Some modifications and additions to the original code:
# https://github.com/tensorflow/tensorflow/blob/r0.11/←
# tensorflow/examples/tutorials/mnist/←
mnist_softmax.py
# See also http://progpater.blog.hu/2016/11/13/←
# hello_samu_a_tensorflow-bol
# ←
=====

"""A very simple MNIST classifier.
See extensive documentation at
http://tensorflow.org/tutorials/mnist/beginners/index.md
"""
from __future__ import absolute_import
from __future__ import division
from __future__ import print_function

import argparse

# Import data
from tensorflow.examples.tutorials.mnist import input_data

import tensorflow as tf
import matplotlib.pyplot
```

```
FLAGS = None

def readimg():
    file = tf.read_file("sajat8a.png")
    img = tf.image.decode_png(file)
    return img

def main(_):
    mnist = input_data.read_data_sets(FLAGS.data_dir, one_hot= ←
        True)

    # Create the model
    x = tf.placeholder(tf.float32, [None, 784])
    W = tf.Variable(tf.zeros([784, 10]))
    b = tf.Variable(tf.zeros([10]))
    y = tf.matmul(x, W) + b

    # Define loss and optimizer
    y_ = tf.placeholder(tf.float32, [None, 10])

    # The raw formulation of cross-entropy,
    #
    # tf.reduce_mean(-tf.reduce_sum(y_ * tf.log(tf.nn.softmax(y ←
    #     ))),
    # reduction_indices=[1]))
    #
    # can be numerically unstable.
    #
    # So here we use tf.nn.softmax_cross_entropy_with_logits on ←
    # the raw
    # outputs of 'y', and then average across the batch.
    cross_entropy = tf.reduce_mean(tf.nn. ←
        softmax_cross_entropy_with_logits(logits = y, labels = y_ ←
            )) # add name to args
    train_step = tf.train.GradientDescentOptimizer(0.5). ←
        minimize(cross_entropy)
    sess = tf.InteractiveSession()

    # Train
    tf.initialize_all_variables().run()
    print("-- A halozat tanítása")

    for i in range(1000):
        batch_xs, batch_ys = mnist.train.next_batch(100)
        sess.run(train_step, feed_dict={x: batch_xs, y_: ←
            batch_ys})

        if i % 100 == 0:
            print(i/10, "%")
```

```
print(" ←
----- ←
")

# Test trained model
print("-- A halozat tesztelese")
correct_prediction = tf.equal(tf.argmax(y, 1), tf.argmax(y_ ←
    , 1))
accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf. ←
    float32))
print("-- Pontosság: ", sess.run(accuracy, feed_dict={x: ←
    mnist.test.images, y_: mnist.test.labels}))
print(" ←
----- ←
")

print("-- A MNIST 42. tesztképeinek felismerése, mutatom a ←
    számot, a továbblepéshez csukd be az ablakat")

img = mnist.test.images[42]
image = img

matplotlib.pyplot.imshow(image.reshape(28, 28), cmap= ←
    matplotlib.pyplot.cm.binary)
matplotlib.pyplot.savefig("4.png")
matplotlib.pyplot.show()

classification = sess.run(tf.argmax(y, 1), feed_dict={x: [ ←
    image]})

print("-- Ezt a halozat ennek ismeri fel: ", classification ←
    [0])
print(" ←
----- ←
")

print("-- A saját kezi 8-asom felismerése, mutatom a számot ←
    , a továbblepéshez csukd be az ablakat")

img = readimg()
image = img.eval()

image = image.reshape(28*28)

matplotlib.pyplot.imshow(image.reshape(28, 28), cmap= ←
    matplotlib.pyplot.cm.binary)
matplotlib.pyplot.savefig("8.png")
matplotlib.pyplot.show()

classification = sess.run(tf.argmax(y, 1), feed_dict={x: [ ←
```

```

    image] })

print("-- Ezt a halozat ennek ismeri fel: ", classification -->
[0])
print(" -->
----- <-->
")

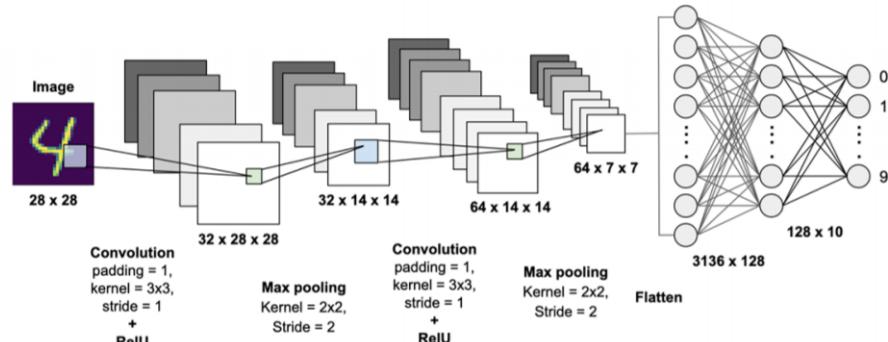
if __name__ == '__main__':
    parser = argparse.ArgumentParser()
    parser.add_argument('--data_dir', type=str, default='/tmp/ -->
tensorflow/mnist/input_data', help='Directory for -->
storing input data')

FLAGS = parser.parse_args()

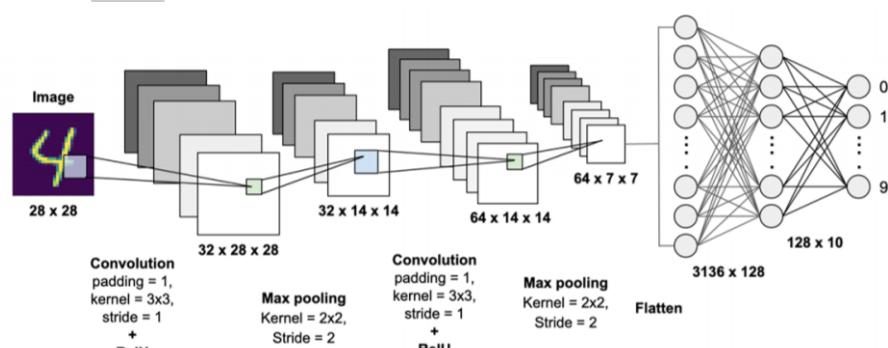
tf.app.run()

```

A kód elején beimportáljuk a tensorflow-ot és a matplotlib-et. Ezek után következik a rajzolt kép beolvasása. A következő kódrészletben történik az összehasonlítás, az mnist.test parancssal. Az alap képünk 28x28 ezért a Mnist teszt képet átkonvertálja 28x28 as képre, majd a sess.run() elvégzi az összehasonlítást, és ennek a vissza térsíni eredménye a felismert szám. A plt.imshow sorban megjelenítjük a képet, majd a plt.savefig sorban pedig, lementjük és kiíratjuk a felismert számot, itt ismét a sess.run() végzi a hasonlítást, majd kiiratjuk az eredményt.



Az MNIST folyamatainak megértésében segít az alábbi modelt (forrás: [towardsdatascience.com](http://towardsdatascience.com)).



Ahogy tudjuk, a számjegyek osztályozása egy konvolúciós neurális hálózat (azaz CNN) használatával történik. A modell meghatároz egy CNN-t amiben 2 konvolúciós réteg van, melyeket két teljesen összekapcsoltréteg követ

## 20.3. EPAM: Back To The Future

Ebben a feladatban egy kicsit betekintést kaphatunk az AOP (Aspect Orientated Programming) programozásba. Hiszen itt nem objektumokkal hanem feltételekkel, avagy vonatkozásokkal foglalkozunk.

Ez egy másfajta gondolkodás, ezért a porgam kód részekre van bontva.

```
package epam.backToTheFuture;

import java.util.concurrent.CompletableFuture;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public class BackToTheFuture {

    private static ExecutorService executorService =
        Executors.newFixedThreadPool(2);

    public static void main(String[] args) {

        CompletableFuture<String> shortTask =
            CompletableFuture.supplyAsync(
                () -> {sleep(500); return "Hi"; },
                executorService);

        CompletableFuture<String> mediumTask =
            CompletableFuture.supplyAsync(
                () -> {sleep(750); return "Hey"; },
                executorService);

        CompletableFuture<String> longTask =
            CompletableFuture.supplyAsync(
                () -> {sleep(1000); return "Hello"; },
                executorService);
    }
}
```

Először is létrehoztunk egy osztályt, aminek egy privát objektumot hozutnk létre egy ExecutorService példányt aminek a beállítottuk, hogy a 2 szálon fusson. Ez képes lesz Future(jóvőbeli) folyamatok kezelésére. Ehez létrehozunk három CompletableFuture változót, amelyek egy-egy folyamatot tartalmaznak, és inicializálva lesznek az **executorService** szerint. Mindegyik kap egy lambda kiejezést, amelyek először altatva lesznek és aztán vissza adnak egy-egy Stringet, az első altava lesz 500ms-t és vissza adja a "Hi" Stringet, a második 750ms-t és "Hey"-t, végül a harmadik 1000ms-t és "Hello"-t.

```
CompletableFuture<String> result =
    longTask.applyToEitherAsync(
        shortTask, String::toUpperCase,
        executorService);

result = result.thenApply(s -> s + " World");

CompletableFuture<Void> extraLongTask =
    CompletableFuture.supplyAsync(

```

```
(() -> { sleep(1500); return null; },
executorService);

result = result.thenCombineAsync(mediumTask,
(s1, s2) -> s2 + " " + s1,
executorService);
```

A lényegesebb rész most következik, hiszen létrehozunk egy negyedik CompletableFuture változót, ami nek értéket adunk a harmadikat, a longTaskot, viszont a shortTask szerint és egy function szerint, ami jelen esetben egy toUpperCase függvény ami nagybetűssé teszi a longTask-ot, majd átadjuk a Servicenek. A result változóhoz hozzáfűzzünk a végéhez a thenApply() metódussal a "World" Stringet, hogyha az előző interakció befejeződött. Majd egy extraLongTask változót több ideig alatalva (1500ms) vissza adunk egy null értéket, és megadjuk az executorService-t. Majd a result-ot összekombináljuk, a mediumTask értékét, a "Hey"-t adjuk meg amihez lambdaival hozzáadjuk a másik két tárolt értéket, végül megadjuk az executorService-t.

```
System.out.println(result.getNow("Bye"));
sleep(1500);
System.out.println(result.getNow("Bye"));

result.runAfterBothAsync(extraLongTask,
() -> System.out.println("After both!"),
executorService);

result.whenCompleteAsync(
(s, throwable) -> System.out.println("Complete: " + s),
executorService);

executorService.shutdown();
}

private static void sleep(int sleeptime) {}
```

A program ezen részén lesznek a kiiratások. Előbb a resultal egy egyszerű "Bye"-t íratunk ki, majd alatatás után ugyan ezt akarjuk kiiratni, de nem ezt kapjuk, sőt van olyan eset amikor egyik kiiratáskor sem a "Bye"-t kapjuk meg, hanem az összekombinált Stringet, ezek véletlenszerűek. Miután megvoltak a kiiratások, lesz egy folyamat ami akkor fut le ha az előtte lévők végrehajtódtak, ezt a runAfterBothAsync() metódussal érjük el, ami lambdaival kiiratja az "After both!" szöveget. Végül amikor elkészült teljesen a szinkornizáció, lefut a whenCompleteAsync() metódus ami, kiiratja az eldobható kombinációt amit egy lambda irat ki, majd lezárjuk az executorService-t.

Ezek a különböző véletlenszerű eredmények.

Console	Console	Console	Console
<terminated> BackToTheFuture Java App			
Hey, HELLO World	Bye	Bye	Bye
Hey, HELLO World	Hey, HELLO World	Bye	Hey, HI World
After both!	After both!	Bye	After both!
Complete: Hey, HELLO World	Complete: Hey, HELLO World	Complete: Hey, HELLO World	Complete: Hey, HI World

## 20.4. EPAM: AOP

Ebben a feladatban az AOP (Aspect Object Programming) paradigmát használjuk. Ez a fajta programozás eltér az OO (Object Orientated) programozástól, hiszen itt nem az objektumok a legfontosabb részei a programoknak. Ugyanakkor osztályokkal és azok objektumaival is foglalkoznunk kell. Lényegében ez a paradigma használatkor a program folyamatokból épül fel, ugyanakkor háttér folyamatoknak is nevezhető folyamatokból. Aspecteket, azaz vonatkozásokat, vagy feltételeket szabhatunk a programban lévő függvényekhez, vagy metódusokhoz.

Amikor meghívtunk egy függvényt, hogy elvégezze a feladatát társíthatunk mellé egy aspectet, ami azt jelenti, hogy az aspect valamelyen szinten függeni fog a függvénytől és annak futásától. Egy aspect egy olyan interakció amit függvényként is felfoghatunk. Az aspecteknek inicializálhatunk metódusokat amiktől függhetnek, valamint az aspektek tudják, hogy futnak-e a számukra lényeges függvények. Tulajdonképpen több fajta aspect létezik. Mivel egy aspekt tudja, mikor fut le az a függvény amelyik számára fontos, ezért vannak olyan aspektek amelyek a függvények előtt lefutnak, de akkor vannak olyanok is amik a függvények után futnak le.

De léteznek olyanok is amik egy függvényt körülölelnek, például ebben a feladatban is egy ilyet nézünk meg. A körülölelő vagy around aspecteknek két része van, az első amelyik a függvény előt fut le, a második pedig a függvény után fut le. Például egy függvény futási idejét meg lehet oldani egy around aspect segítségével.

Egy időmérés program, először le kell tölteni a SpringAOP és az AspectJ könyvtárakat majd ezeket megfelelően kell a programhoz importolni.

```
import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface CustomTimer {
}
```

Ebben a kódban tulajdonképpen ahogyan eddig is használtunk interfészket, így használhatunk itt is, ebben az esetben erre alapozzunk majd azt az osztályt amelyik az időmérést fogja végezni. Megadtuk az irányelvet a **Retention** kulcsszóban, azaz a RUNTIME (időmérés). Megadtuk azt is, hogy mikor kell ezt végezze a **Target** kulcsszóban, tehát a célja az időmérésnek a METHOD vagyis minden metódus.

```
import org.aspectj.lang.ProceedingJoinPoint;
import org.aspectj.lang.annotation.Around;
import org.aspectj.lang.annotation.Aspect;
import org.springframework.stereotype.Service;
import org.springframework.util.StopWatch;

@Aspect
@Service
public class CustomTimerAspect {

    @Around("@annotation(com.epam.training.CustomTimer)")
}
```

```
public Object timeAnnotatedMethod(ProceedingJoinPoint pjp) ←
    throws Throwable {
    // start stopwatch
    StopWatch stopWatch = new StopWatch(pjp.getTarget()) . ←
        toString());
    stopWatch.start();

    Object retVal = pjp.proceed();

    // stop stopwatch
    stopWatch.stop();

    System.out.println("Execution time for " + pjp. ←
        getTarget().toString() + ": " + stopWatch. ←
        getTotalTimeMillis() + " ms");

    return retVal;
}
}
```

Itt az Around Aspectet látjuk, amit az **Aspect** kulcsszóval jeleztük az osztály előtt előtt, tehát ez azt jelenti, hogy minden függvény ami ebben az osztályban szerepel azoknak aspectnek kell lenniük. A timeAnnotatedMethod() metódus lesz az egyetlen függvény ebben az esetben, ami előtt az **Around** kulcsszó szerepel. Ennek megadtuk az **annotation** kulcsszóval, hogy mely függvények futásakor hajtódjön végre az aspect. Itt megadhatunk általános annotációt is, például adhatunk meg függvényt úgy, hogy eléríjük a típusát is, ekkor pl egy "int" típusú függvény után egy \* karaktert használva, minden int típusú függvény futásakor az aspect is lefut. A függvény paramétere egy függvény lesz, amit egy viszonyítási pontként használunk, hiszen ezen függvény köré építjük fel az aspektet. Jelen esetben létrehozunk egy StopWatch osztály objektumot ami az időt méri. Majd elindítjuk és aztán az objektumnak is helyet adunk a futásra az Object példányosítással, majd leállítjuk az időt és végül kiiratjuk a függvényt és a hozzá tartozó futási időt.

```
import org.springframework.stereotype.Service;

@Service
public class ExampleService {

    @CustomTimer
    public void exampleMethodToTime() throws ←
        InterruptedException {
        System.out.println("Testee start!");

        Thread.sleep(1000);

        System.out.println("Testee end!");
    }
}
```

Ez csak egy példa a futásra aminek megadtuk a **CustomTimer** kulcsszóval a létrehozott interfészt, ezért lefut majd az aspect is. Tulajdonképpen szemléltettük is két kiiratással az aspect működését.

```
import org.springframework.aop.framework.autoproxy. ←
    ProxyCreationContext;
import org.springframework.beans.annotation.AnnotationBeanUtils ←
    ;
import org.springframework.core.SpringVersion;

@SpringBootApplication
@EnableAspectJAutoProxy
public class MainApplication {

    public static void main(String[] args) {
        SpringVersion.run(MainApplication.class, args);
    }

    @Bean
    public CommandLine commandLineRunner(ProxyCreationContext ←
        ctx) {
        return args -> ctx.getBean(ExampleService.class).←
            exampleMethodToTime();
    }

}
```

Ez a MainApplication osztály ahol létrehoztunk egy egyszerű metódust ami CommandLine osztály tipusú. A függvény egyszerűen csak vissza adja az argumentum bemeneteket. A main() metódusban csak meghívásra kerül a konzol bemenet olvasó, és mivel ez egy metódus, az aspekt interakcióba is fog lépni. Jelen esetben azt mérjük majd, hogy a felhasználó mennyi ideig gépelet a konzolban.

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
          xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 ←
              http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <parent>
        <artifactId>unideb-prog2-parent</artifactId>
        <groupId>com.epam.training</groupId>
        <version>1.0-SNAPSHOT</version>
        <relativePath>../../</relativePath>
    </parent>
    <modelVersion>4.0.0</modelVersion>

    <artifactId>aop-performance-test</artifactId>

    <dependencies>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-aop</artifactId>
        </dependency>
        <dependency>
```

```
<groupId>org.junit.jupiter</groupId>
<artifactId>junit-jupiter</artifactId>
</dependency>

<dependency>
    <groupId>org.hamcrest</groupId>
    <artifactId>hamcrest</artifactId>
</dependency>
</dependencies>
</project>
```

Fontos még az XML rész is ahol deklaráljuk az egész programot és honnan kezelní, avagy innen konfigurálhatjuk az aspecteket, hogy melyek működjenek és hogyan. A szükséges könyvtárakat itt implementáljuk és a Bean részeket is bele írhatjuk, valamint az aspecteket és a hozzá tartozó paramétereket.

DRAFT

**IV. rész**

**Irodalomjegyzék**

DRAFT

## 20.5. Általános

[MARX] Marx, György, *Gyorsuló idő*, Typotex , 2005.

## 20.6. C

[KERNIGHANRITCHIE] Kernighan, Brian W. & Ritchie, Dennis M., *A C programozási nyelv*, Bp., Műszaki, 1993.

## 20.7. C++

[BMECPP] Benedek, Zoltán & Levendovszky, Tíhamér, *Szoftverfejlesztés C++ nyelven*, Bp., Szak Kiadó, 2013.

## 20.8. Lisp

[METAMATH] Chaitin, Gregory, *META MATH! The Quest for Omega*, [http://arxiv.org/PS\\_cache/math/pdf/0404/0404335v7.pdf](http://arxiv.org/PS_cache/math/pdf/0404/0404335v7.pdf) , 2004.

Köszönet illeti a NEMESPOR, <https://groups.google.com/forum/#!forum/nemespor>, az UDPORG tanulószoba, <https://www.facebook.com/groups/udprog>, a DEAC-Hackers előszoba, <https://www.facebook.com/groups/DEACHackers> (illetve egyéb alkalmi szerveződésű szakmai csoportok) tagjait inspiráló érdeklődésekért és hasznos észrevételeikért.

Ezen túl kiemelt köszönet illeti az említett UDPORG közösséget, mely a Debreceni Egyetem reguláris programozás oktatása tartalmi szervezését támogatja. Sok példa eleve ebben a közösségen született, vagy itt került említésre és adott esetekben szerepet kapott, mint oktatási példa.