

Univerzális programozás

Írd meg a saját programozás tankönyvedet!

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

Copyright © 2019 Bakos Bálint

Copyright (C) 2019, 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>

COLLABORATORS

	<i>TITLE :</i> Univerzális programozás		
<i>ACTION</i>	<i>NAME</i>	<i>DATE</i>	<i>SIGNATURE</i>
WRITTEN BY	Bakos, Bálint	2019. október 1.	

REVISION HISTORY

NUMBER	DATE	DESCRIPTION	NAME
0.0.1	2019-02-12	Az iniciális dokumentum szerkezetének kialakítása.	nbatfai
0.0.2	2019-02-14	Inciális feladatlisták összeállítása.	nbatfai
0.0.3	2019-02-16	Feladatlisták folytatása. Feltöltés a BHAX csatorna https://gitlab.com/nbatfai/bhax repójába.	nbatfai
0.0.4	2019-02-19	Aktualizálás, javítások.	nbatfai

Tartalomjegyzék

I. Bevezetés	1
1. Vízió	2
1.1. Mi a programozás?	2
1.2. Milyen doksikat olvassak el?	2
1.3. Milyen filmeket nézzek meg?	2
II. Második felvonás	3
2. Helló, Berners-Lee!	5
2.1. Olvasónapló: C++: Benedek Zoltán, Levendovszky Tihamér Szoftverfejlesztés C++ nyelven és Java: Nyékyné Dr. Gaizler Judit et al. Java 2 útikalauz programozóknak 5.0 I--II.II.	5
2.2. Olvasónapló: Python: Forstner Bertalan, Ekler Péter, Kelényi Imre: Bevezetés a mobil-programozásba. Gyors prototípus fejlesztés Python és Java nyelven	6
3. Helló, Arroway!	7
3.1. OO szemlélet	7
3.2. "Gagyi"	14
3.3. Yoda	15
3.4. Kódolás from scratch	17
4. Helló, Liskov!	23
4.1. Liskov helyettesítés sértése	23
4.2. Szülő-gyerek	23
4.3. Anti OO	25
4.4. Hello, Android!	25
4.5. Ciklomantikus komplexitás	25

III. Irodalomjegyzék	27
4.6. Általános	28
4.7. C	28
4.8. C++	28
4.9. Lisp	28

Táblázatok jegyzéke

4.1. Összehasonlítás	25
--------------------------------	----

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]

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 allokálni igény szerint pár robot programozót a felhőből? A programozók dolgozók lesznek vagy papok? Ki tudhatná ma.

Mindenesetre 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.

Hogyan forgasd

A könyv célja egy stabil programozási szemlélet kialakítása az olvasóban. Módszere, hogy hetekre bontva ad egy tematikus feladatcsokrot. 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 (hogyan lássuk mást is) példával.

Hogyan nyomjuk?

Ránts 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 találsz az informatikai szövegek jelölésére használható gazdag „API” elemenkénti bemutatását.

I. rész

Bevezetés

1. fejezet

Vízió

Mi a programozás?

Milyen doksikat olvassak el?

- Olvasgasd 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
- [[KERNIGHANRITCHIE](#)]
- [[BMECPP](#)]
- Az igazi kockák persze csemegéznek a C nyelvi szabvány [ISO/IEC 9899:2017](#) kódcsipeteiből is.

Milyen filmeket nézzek meg?

- 21 - Las Vegas ostroma, <https://www.imdb.com/title/tt0478087/>, benne a **Monty Hall probléma** bemutatása.

II. rész

Második felvonás

**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.

2. fejezet

Helló, Berners-Lee!

Olvasónapló: C++: Benedek Zoltán, Levendovszky Tihamér Szoftverfejlesztés C++ nyelven és Java: Nyékyné Dr. Gaizler Judit et al. Java 2 útikalauz programozóknak 5.0 I-II.II.

A C++ valamint a Java nyelv is magasszintű programozási nyelvnek számít. Mindkettő objektumorientált, azonban mivel a C++ nyelv hamarabb alakult ki így kisebb-nagyobb eltérések találhatók a két nyelv között.

A C++ a C nyelvtől örökölt gépközei konstrukciókat, ebből adódik sebessége. A Java nyelv pedig a C++-ból vett át sok mindent. Azonban van egy lényeges eltérés, hogy az Java nyelv a mutatók helyett referenciákat használ, így biztonságosabb, megbízhatóbb programokat lehet írni. Valamint a programok hordozhatósága is eltér. Például ha egy Linuxon írt C++ kódot akarunk átvinni Windowsra az nem biztos, hogy működni fog, de ha egy Java kódot akarunk futtani máshol az jól fog szuperálni feltéve, hogy van JVM(Java Virtual Machine) a gépen. Mivel ez a Java bájt kódot fogja futtani, így ez platformfüggetlen lesz.

Szintaktikában is találunk eltéréseket a két nyelv között.

Hasonlítsuk össze mondjuk ezt a két Hello world! programot.

C++-ban:

```
#include <iostream>
using namespace std;
int main()
{
    cout << "Hello World!" << endl;
    return 0;
}
```

Java-ban:

```
public class HelloWorld {
    public static void main(String[] args) {
```

```
        System.out.println("Hello World!");  
    }  
}
```

Amint látható a Java nagyrészen osztályalapú. Az osztályoknak különböző elérése lehet: public, protected, private. Az osztályokon belül létrehozhatunk változókat, függvényeket.

Olvasónapló: Python: Forstner Bertalan, Ekler Péter, Kelényi Imre: Bevezetés a mobilprogramozásba. Gyors prototípus fejlesztés Python és Java nyelven

A Python támogatja a funkcionális valamint az imperatív nyelveket is. Legfőbb jellemzője, ami teljesen eltér a többi magasszintű programozási nyelvtől, hogy behúzásalapú a szintaxisa, tehát semmilyen kapcsos zárójelre vagy explicit kulcsszóra nincs szükség. Valamint a sorok végén már megszokott pontosvessző sem kell.

```
if 1 < 2  
    print("Nagyobb!")  
else:  
    print("Kisebb!")
```

A Python egy objektumorientált nyelv, tehát minden adatot objektumok reprezentálnak. A változók típusának explicit megadására sincs szükség, a rendszer futási időben dönti el a változók típusát.

Ilyen egyszerűen néz ki Pythonban egy deklarálás:

```
name = "Ádám"
```

A nyelvben definiálhatunk osztályokat is és ezeknek példányai az objektumok. Az osztályok attribútumai lehetnek objektumok vagy függvények is.

3. fejezet

Helló, Arroway!

OO szemlélet

A polártranszformációs generátor egy széles körben elterjedt random generátor. Olyannyira elterjedt formája ez a random szám generálásnak, hogy a `Java.util.Random` osztály is ezt a módszert alkalmazza.

Íme a Java kód teljes egészében:

```
public class PolarGenerator {

    boolean nincsTarolt = true;
    double tarolt;

    public PolarGenerator() {
        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;
    } else {
        nincsTarolt = !nincsTarolt;
        return tarolt;
    }
}

public static void main(String[] args) {

    PolarGenerator g = new PolarGenerator();

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

A következő sorokban részenként magyarázom el mi történik a kódban.

```
boolean nincsTarolt = true;
double tarolt;
```

A PolárGenerátor classban létrehozunk két változót. Az egyik boolean típusú, amely azt fogja megmondani, hogy éppen van-e tárolt értékünk. A másik maga a tárolt értéket tartalmazza, amit korábban kiszámítottunk.

```
public PolarGenerator() {
    nincsTarolt = true;
}
```

Ez a class publikus konstruktora, amely a nincsTarolt-at igazra állítja. Erre azért van szükség, hogy tudjuk éppen van-e tárolt érték vagy nincs.

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

Ez a program lelke ahol a `kovetkezo()` metódus végzi a meghatározó számítást. Ha a `nincsTarolt` értéke igaz, akkor számol két random értéket. Az egyiket elmenti a `tarolt` változóba, a másikat pedig visszaadja. Amennyiben a `nincsTarolt` értéke hamis, akkor pedig a tárolt értéket fogja visszaadni.

```
public static void main(String[] args) {

    PolarGenerator g = new PolarGenerator();

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

A `main` metódussal indul el a program. Itt példányosítjuk a `PolárGenerátor` classot. Ezután egy `for` ciklussal 10-szer kiíratjuk a meghívott `kovetkezo()` metódus értékét. Minden második érték a `tarolt` változóból visszatért érték lesz. Ezeket az értékeket generálta nekem:

```
bunyi@bunyi: ~/Documents
bunyi@bunyi:~/Documents$ javac PolarGenerator.java
bunyi@bunyi:~/Documents$ java PolarGenerator
-0.5518103598184344
-0.9108738027998466
-1.9560282490949241
0.5048225085719978
1.9118928849455419
-0.48283343220250585
1.7238209676208334
0.8937594137943287
-1.3956361879015262
0.8904870819477649
bunyi@bunyi:~/Documents$
```

Az érdekesség az még itt, hogy a Java fejlesztők egy nagyon hasonló módon oldották meg a `java.util.Random` osztályban a Random szám generálást. Íme:

```
public double nextDouble() {
    return (((long) (next(26)) << 27) + next(27)) * DOUBLE_UNIT;
}

private double nextNextGaussian;
private boolean haveNextNextGaussian = false;

synchronized public double nextGaussian() {
    // See Knuth, ACP, Section 3.4.1 Algorithm C.
    if (haveNextNextGaussian) {
        haveNextNextGaussian = false;
        return nextNextGaussian;
    } else {
        double v1, v2, s;
        do {
            v1 = 2 * nextDouble() - 1; // between -1 and 1
            v2 = 2 * nextDouble() - 1; // between -1 and 1
            s = v1 * v1 + v2 * v2;
        } while (s >= 1 || s == 0);
        double multiplier = StrictMath.sqrt(-2 * StrictMath.log(s)/s);
        nextNextGaussian = v2 * multiplier;
        haveNextNextGaussian = true;
        return v1 * multiplier;
    }
}
```

C++-ban így néz ki a teljes kód:

```
#include <iostream>
#include <tgmath.h>
#include <cstdlib>
```

```
#include <time.h>

using namespace std;

class PolarGenerator {
private:
    bool nincsTarolt;
    double tarolt;

public:
    PolarGenerator() {
        nincsTarolt = true;
        srand (time(NULL));
    }

    double kovetkezo() {
        if (nincsTarolt) {
            double u1, u2, v1, v2, w;

            do {
                u1 = rand() / (RAND_MAX + 1.0);
                u2 = rand() / (RAND_MAX + 1.0);

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

                w = v1 * v1 + v2 * v2;
            } while (w > 1);

            double r = sqrt((-2 * log(w)) / w);
            tarolt = r * v2;
            nincsTarolt = !nincsTarolt;

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

int main(int argc, char** argv) {
    PolarGenerator g;

    for (int i = 0; i < 10; ++i)
        cout << g.kovetkezo() << endl;
```

```
    return 0;
}
```

A következő sorokban részenként magyarázom el mi történik a kódban.

```
private:
    bool nincsTarolt;
    double tarolt;
```

Ez a Polárgenerátor class private része. Ez tartalmaz egy bool és egy double típusú változót. Ezek a változók csak az osztályon belül lesznek elérhetőek.

```
public:
    PolarGenerator() {
        nincsTarolt = true;
        srand (time(NULL));
    }

    double kovetkezo() {
        if (nincsTarolt) {
            double u1, u2, v1, v2, w;

            do {
                u1 = rand() / (RAND_MAX + 1.0);
                u2 = rand() / (RAND_MAX + 1.0);

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

                w = v1 * v1 + v2 * v2;
            } while (w > 1);

            double r = sqrt((-2 * log(w)) / w);
            tarolt = r * v2;
            nincsTarolt = !nincsTarolt;

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

Ez pedig a class public része, amelyben a változók és metódusok példányosítás után elérhetőek az osztályon kívül is.

```
PolarGenerator() {  
    nincsTarolt = true;  
    srand (time(NULL));  
}
```

Az osztály nevével megegyező metódust konstruktornak nevezzük. Az ebben lévő kódok példányosításkor hajtódnak végre.

```
double kovetkezo() {  
    if (nincsTarolt) {  
        double u1, u2, v1, v2, w;  
  
        do {  
            u1 = rand() / (RAND_MAX + 1.0);  
            u2 = rand() / (RAND_MAX + 1.0);  
  
            v1 = 2 * u1 - 1;  
            v2 = 2 * u2 - 1;  
  
            w = v1 * v1 + v2 * v2;  
        } while (w > 1);  
  
        double r = sqrt((-2 * log(w)) / w);  
        tarolt = r * v2;  
        nincsTarolt = !nincsTarolt;  
  
        return r * v1;  
    }  
    else {  
        nincsTarolt = !nincsTarolt;  
        return tarolt;  
    }  
}
```

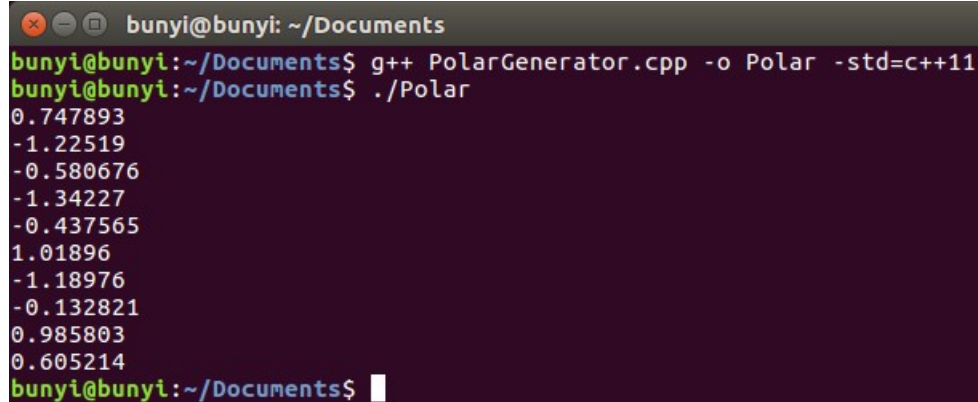
A kovetkezo metódusban szinte semmilyen lényegi eltérés nincs a Java kódhoz képest. Ugyanaz történik ha nincsTarolt igaz akkor generál randomot, ha pedig hamis, akkor visszaadja az eltárolt értéket.

```
int main(int argc, char** argv) {  
    PolarGenerator g;  
  
    for (int i = 0; i < 10; ++i)  
        cout << g.kovetkezo() << endl;
```

```
    return 0;
}
```

A main metódusban szintén, mint a Javanál példányosítunk és egy for ciklussal 10-szer visszadjuk a következo() metódus értékét.

C++-ban generált értékek:



```
bunyi@bunyi: ~/Documents
bunyi@bunyi:~/Documents$ g++ PolarGenerator.cpp -o Polar -std=c++11
bunyi@bunyi:~/Documents$ ./Polar
0.747893
-1.22519
-0.580676
-1.34227
-0.437565
1.01896
-1.18976
-0.132821
0.985803
0.605214
bunyi@bunyi:~/Documents$
```

"Gagyí"

```
while (x <=t && x>=t && t !=x);
```

Erre a tesztkérdésre kellett választ adnunk, hogy bizonyos számoknál miért jön létre végtelen ciklus és bizonyosnál miért nem.

```
public class Gagyí {

    public static void main (String[]args){

        Integer x = -128;
        Integer t = -128;

        while (x <= t && x >= t && t != x);
    }
}
```

Például itt -128-nál nem jön létre végtelen ciklus.

```
public class GagyInfinity {  
  
    public static void main (String[] args) {  
  
        Integer x = -129;  
        Integer t = -129;  
  
        while (x <= t && x >= t && t != x);  
    }  
}
```

Azonban -129-el már végtelen ciklus jön létre.

```
public static Integer valueOf(int i) {  
    if (i >= IntegerCache.low && i <= IntegerCache.high)  
        return IntegerCache.cache[i + (-IntegerCache.low)];  
    return new Integer(i);  
}
```

A válasz, hogy miért jön létre -129-el végtelen ciklus, míg -128-al semmi sem történik ebben a kódcsipetben rejlik, ami a `java.lang.Integer` osztályban található.

Elsősorban mindenképpen tudni kell, hogy a `!=`, `==` operátorok az objektumok címét hasonlítják össze, valamint a Java feltételezi, hogy a programok sokat dolgoznak, majd kis számokkal, így a poolban már előre elkészített számok vannak 127-től -128-ig. Tehát amikor létrehozunk két `Integer` objektumot és az a poolon belül van, akkor a két objektum címe meg fog egyezni. Pontosan ez történik a -128-nál, létrehozuk a két objektumot `x`-et és `y`-ot, azonban a poolból kapjuk meg mindkettőt egy már előre elkészített objektumot így a cím megegyezik. Ezért a `x != y` hamis értéket fog adni, így a `while` ciklus feltétele nem teljesül és nem jön létre végtelen ciklus.

```
return new Integer(i);
```

Ha nem esik bele viszont a poolba a szám akkor, itt látszik, hogy létrehoz egy új `Integer`-t. Mivel a -129 nem esik bele így két különböző című objektumot fog létrehozni és így a `while` ciklus feltétele igaz lesz és végtelen ciklust kapunk.

Yoda

A feladat az volt, hogy írjunk egy olyan Java kódot ami `NullPointerException` hibával kilép, ha nem követjük a Yoda conditionst. Íme a kód:

```
public class Yoda {  
  
    public static void main(String[] args) {
```



```
String myString = null;

if ("something".equals(myString)) {
    System.out.println("True");
} else {
    System.out.println("False");
}

//NullPointerException
if (myString.equals("something")) {
    System.out.println("True");
} else {
    System.out.println("False");
}
}
```

A Yoda conditions egy kódolási stílus, ahol a programkódot "fordítva" írjuk be, tehát az értékadásnál a konstans értéket írjuk balra és jobbra kerül a változó amibe elmentjük. A nevét is erről a szokatlan megfordított kódírásról kapta, Yoda-ról aki a Star Wars-ban hasonlóan nem szabályszerűen alkalmazza az angolt.

```
int érték = 3;
if( érték == 3) {
    System.out.println("Igaz");
}
```

Ez ahogy rendesen íránk egy kódot.

```
int érték = 3;
if( 3 == érték) {
    System.out.println("Igaz");
}
```

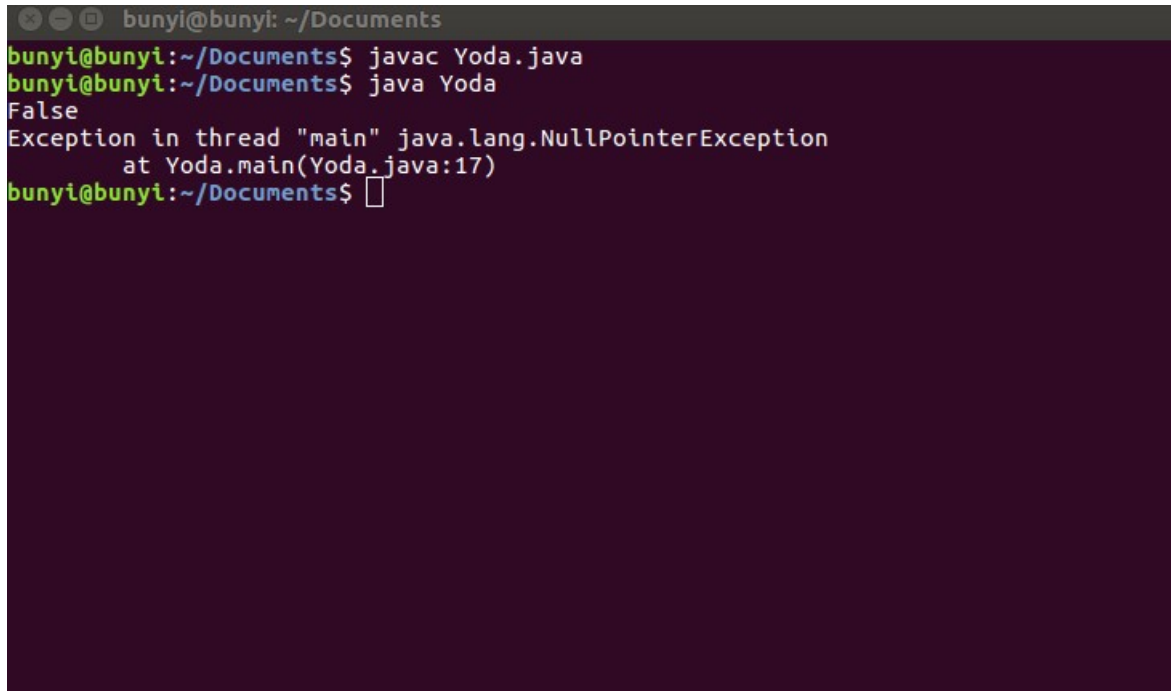
Ugyanaz a kód Yoda conditions-t használva. Mindakettővel teljesen normálisan fog működni a program.

```
if ("something".equals(myString)) {
    System.out.println("True");
} else {
    System.out.println("False");
}
```

Ez a rész egy tipikusan Yoda conditions-t használva lett megírva. Látszik, hogy az equals metódus bal oldalára került a konstans, jelen esetben egy string. Az equals ezt hasonlítja össze a myStringben lévő null értékkel. Ekkor semmilyen hiba nem fordul elő egyszerűen hamis lesz a visszatért érték.

```
if (myString.equals("something")) {
    System.out.println("True");
} else {
    System.out.println("False");
}
```

Ezek a sorok azonban nem használják a Yoda conditions-t, így NullPointerException-t dobnak a Java-ban. Így a Yoda conditions-al elkerülhető néhány nem biztonságos null viselkedés.

A terminal window with a dark background and light-colored text. The prompt is 'bunyi@bunyi: ~/Documents'. The user enters 'javac Yoda.java' and then 'java Yoda'. The output shows 'False' followed by an exception: 'Exception in thread "main" java.lang.NullPointerException at Yoda.main(Yoda.java:17)'. The prompt returns to 'bunyi@bunyi: ~/Documents\$' with a cursor.

```
bunyi@bunyi: ~/Documents
bunyi@bunyi:~/Documents$ javac Yoda.java
bunyi@bunyi:~/Documents$ java Yoda
False
Exception in thread "main" java.lang.NullPointerException
    at Yoda.main(Yoda.java:17)
bunyi@bunyi:~/Documents$
```

Amint látszik a console-on is NullPointerException hibával kilép.

Azonban a Yoda conditions bírálói nagyban panaszkodnak az olvashatóság elvesztésére, ha ezt a módszert alkalmazzuk.

Kódolás from scratch

Egy olyan feladatot kaptunk, hogy írjuk meg a BBP algoritmus megvalósítását. Ez egy olyan algoritmus ami kiszámítja a Pi hexadecimális számjegyeit egy megadott helyen. Íme a kód:

```
public class BBP {
    String HexaJegyek;

    public BBP(int d) {

        double HexPi = 0.0;

        double S1 = Sj(d, 1);
        double S4 = Sj(d, 4);
        double S5 = Sj(d, 5);
        double S6 = Sj(d, 6);

        HexPi = 4.0*S1 - 2.0*S4 - S5 - S6;

        HexPi = HexPi - Math.floor(HexPi);

        StringBuffer sb = new StringBuffer();
```

```
Character hexaJegyek[] = {'A', 'B', 'C', 'D', 'E', 'F'};

while(HexPi != 0.0) {

    int jegy = (int)Math.floor(16.0d*HexPi);

    if(jegy<10) {
        sb.append(jegy);
    } else {
        sb.append(hexaJegyek[jegy-10]);
    }

    HexPi = (16.0d*HexPi) - Math.floor(16.0d*HexPi);

}

HexaJegyek = sb.toString();
}

public String toString() {
    return HexaJegyek;
}

public double Sj(int d, int j) {

    double Sj = 0.0;

    for (int k = 0; k <= d; k++)
        Sj += (double)n16modk(d-k, 8*k + j) / (double)(8*k + j);

    return Sj - Math.floor(Sj);
}

public long n16modk(int n, int k) {

    int t = 1;
    while(t <=n)
        t *= 2;

    long r = 1;

    while(true) {

        if(n >= t) {
            r = (16*r) % k;
            n = n - t;
        }

        t = t/2;
    }
}
```

```
        if(t < 1)
            break;

        r = (r*r) % k;
    }

    return r;
}

public static void main(String[] args) {
    System.out.println(new BBP(1000000));
}
}
```

Nézzük részekre bontva:

```
String HexaJegyek;
```

Először is létrehozunk a BBP classban egy változót. Ebben a változóban fogjuk tárolni a végeredményt, tehát a Pi hexadecimális jegyeit az adott helyen.

Nézzük először a metódusokat, mert csak azután lehet megérteni a konstruktor működését.

```
public String toString() {
    return HexaJegyek;
}
```

Legegyszerűbb a a toString() metódussal kezdeni. Ez egyszerűen visszaadja a végső eredményt egy string-ként, de ebben az esetben az eredményünk eleve string típusú így nincs más dolgunk csak azt visszaadni.

```
public long n16modk(int n, int k) {

    int t = 1;
    while(t <=n)
        t *= 2;

    long r = 1;

    while(true) {

        if(n >= t) {
            r = (16*r) % k;
            n = n - t;
        }

        t = t/2;

        if(t < 1)
            break;

        r = (r*r) % k;
```

```
    }  
  
    return r;  
}
```

Az `n16modk` metódusban számoljuk ki bináris hatványozással a $16^n \bmod k$ értékét.

```
public double Sj(int d, int j) {  
  
    double Sj = 0.0;  
  
    for (int k = 0; k <= d; k++)  
        Sj += (double)n16modk(d-k, 8*k + j) / (double)(8*k + j);  
  
    return Sj - Math.floor(Sj);  
}
```

Az `Sj` metódus egy `double` értékkel fog visszatérni. A BBP algoritmus képlet alapján fogja visszaadni ezt a számot.

```
public BBP(int d) {  
  
    double HexPi = 0.0;  
  
    double S1 = Sj(d, 1);  
    double S4 = Sj(d, 4);  
    double S5 = Sj(d, 5);  
    double S6 = Sj(d, 6);  
  
    HexPi = 4.0*S1 - 2.0*S4 - S5 - S6;  
  
    HexPi = HexPi - Math.floor(HexPi);  
  
    StringBuffer sb = new StringBuffer();  
  
    Character hexaJegyek[] = {'A', 'B', 'C', 'D', 'E', 'F'};  
  
    while(HexPi != 0.0) {  
  
        int jegy = (int)Math.floor(16.0d*HexPi);  
  
        if(jegy<10) {  
            sb.append(jegy);  
        } else {  
            sb.append(hexaJegyek[jegy-10]);  
        }  
  
        HexPi = (16.0d*HexPi) - Math.floor(16.0d*HexPi);  
  
    }  
}
```

```
HexaJegyek = sb.toString();  
}
```

Ez a rész a konstruktora a BBP classnak, ez mindenképpen le fog futni amikor példányosítják a függvényt. Nézzük ezt is részekre bontva:

```
double HexPi = 0.0;  
  
double S1 = Sj(d, 1);  
double S4 = Sj(d, 4);  
double S5 = Sj(d, 5);  
double S6 = Sj(d, 6);  
  
HexPi = 4.0*S1 - 2.0*S4 - S5 - S6;  
  
HexPi = HexPi - Math.floor(HexPi);
```

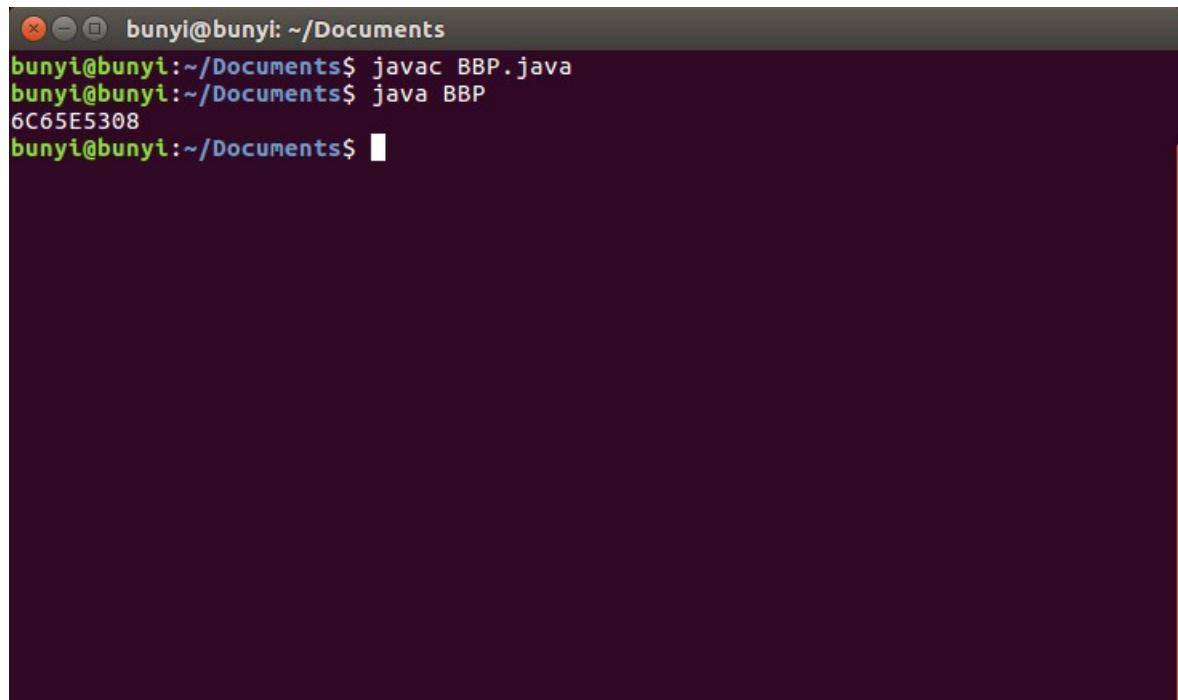
Először is létrehoz 5 db változót. A HexPi-t azért, hogy legyen miben tárolni a számot amit a képlet kiszámolása után megkapunk. Az S1, S4, S5, S6 változók részelemek az alatta lévő képletben. A d változó az a szám itt amit a felhasználó ad meg, hogy hányadik helyen számolja a Pi hexadecimális értékét.

```
StringBuffer sb = new StringBuffer();  
  
Character hexaJegyek[] = {'A', 'B', 'C', 'D', 'E', 'F'};  
  
while(HexPi != 0.0) {  
  
    int jegy = (int)Math.floor(16.0d*HexPi);  
  
    if(jegy<10) {  
        sb.append(jegy);  
    } else {  
        sb.append(hexaJegyek[jegy-10]);  
    }  
  
    HexPi = (16.0d*HexPi) - Math.floor(16.0d*HexPi);  
  
}  
  
HexaJegyek = sb.toString();
```

Itt létrehozunk egy StringBuffert amiben ideiglenesen elmentjük a hexadecimális számokat stringként. Egy Character típusú tömböt is alkotunk, ebben tároljuk a 16-os számrendszerben jelenlévő karaktereket. A while cikluson belül a stringBuffer-hez hozzáfűzzük a hexa számjegyeket, majd a legalján átadjuk a HexaJegyek nevű stringnek.

```
public static void main(String[] args) {  
    System.out.println(new BBP(1000000));  
}
```

A main metódusban a kiíratáson belül egy példányosítást láthatunk 10^6 értékkel. Tehát a program a Pi 1 milliomodik helyen lévő hexadecimális számjegyeit fogja visszaadni. Ahogy látható is:

A terminal window with a dark purple background and a title bar that reads 'bunyi@bunyi: ~/Documents'. The terminal shows the following commands and output:

```
bunyi@bunyi:~/Documents$ javac BBP.java
bunyi@bunyi:~/Documents$ java BBP
6C65E5308
bunyi@bunyi:~/Documents$
```

4. fejezet

Helló, Liskov!

Liskov helyettesítés sértése

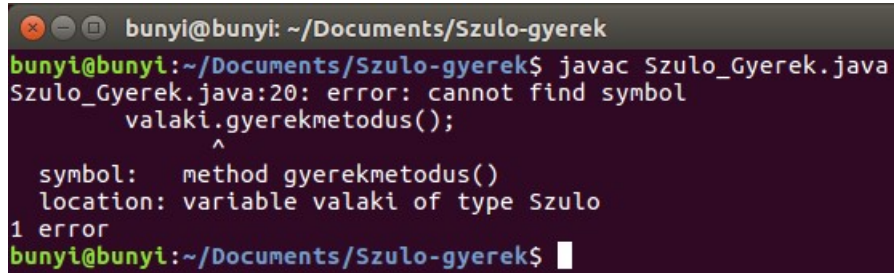
Szülő-gyerek

```
class Szulo {
    void szulometodus() {
        System.out.println("Szulo vagyok!");
    }
}

class Gyerek extends Szulo {
    void gyerekmetodus() {
        System.out.println("Gyerek vagyok!");
    }
}

class Szulo_Gyerek {
    public static void main(String[] args) {
        Szulo valaki = new Gyerek();

        valaki.szulometodus();
        //valaki.gyerekmetodus();
    }
}
```


A terminal window with a dark background. The title bar shows 'bunyi@bunyi: ~/Documents/Szulo-gyerek'. The prompt is 'bunyi@bunyi:~/Documents/Szulo-gyerek\$'. The command 'javac Szulo_Gyerek.java' has been executed. The output shows an error at line 20: 'error: cannot find symbol'. The code snippet 'valaki.gyerekmetodus();' is shown with an arrow pointing to the dot. Below it, the error details are: 'symbol: method gyerekmetodus()' and 'location: variable valaki of type Szulo'. The terminal ends with '1 error' and the prompt 'bunyi@bunyi:~/Documents/Szulo-gyerek\$' with a cursor.

```
bunyi@bunyi: ~/Documents/Szulo-gyerek
bunyi@bunyi:~/Documents/Szulo-gyerek$ javac Szulo_Gyerek.java
Szulo_Gyerek.java:20: error: cannot find symbol
    valaki.gyerekmetodus();
           ^
    symbol:   method gyerekmetodus()
    location: variable valaki of type Szulo
1 error
bunyi@bunyi:~/Documents/Szulo-gyerek$
```

```
#include <iostream>

class Szulo {
public:
    void szulometodus() {
        std::cout << "Szulo vagyok!" << std::endl;
    }
};

class Gyerek : public Szulo {
public:
    void gyerekmetodus() {
        std::cout << "Gyerek vagyok!" << std::endl;
    }
};

int main() {
    Szulo* valaki = new Gyerek();

    valaki->szulometodus();
    //valaki->gyerekmetodus();
}
```

```
bunyi@bunyi: ~/Documents/Szulo-gyerek
bunyi@bunyi:~/Documents/Szulo-gyerek$ g++ Szulo_Gyerek.cpp -o Szulo_Gyerek
Szulo_Gyerek.cpp: In function 'int main()':
Szulo_Gyerek.cpp:21:10: error: 'class Szulo' has no member named 'gyerekmetodus'
    valaki->gyerekmetodus();
           ^
bunyi@bunyi:~/Documents/Szulo-gyerek$
```

Anti OO

Össze kellett hasonlítani a BBP algoritmus kód futási idejét C, C++, Java és C# nyelven. Egy virtuális linux gépen futattam a kódokat. Ilyen eredményt kaptam:

	C	C++	Java	C#
10 ⁶	3.051	2.823	2.575	2.617
10 ⁷	34.460	33.332	28.978	30.589
10 ⁸	385.376	386.714	338.139	353.068

4.1. táblázat. Összehasonlítás

Ahogy látszik a C nyelv volt a leglassabb. Ez várható volt, hisz ez a legöregebb nyelv a négy közül. Leggyorsabb volt a Java kód, amely a 10⁸ pozíciónál 10 másodperccel leelőzte a C# is. Több oka is van, hogy a Java legyőzött mindenkit. Elsőnek lehet mondani, hogy a memória kiosztást sokkal jobban kezeli, mint a többi nyelv. Valamint a JVM jobban optimalizálja a metódus hívásokat. Futás időben dinamikus elemzést tud végezni, hogy mire van szükség és mire nem, így gyorsabb működést képes nyújtani, mint egy C++ fordítóprogram.

Hello, Android!

Ciklomantikus komplexitás

Ebben a feladatban ki kellett számolnunk valamelyik programunk függvényeinek ciklomantikus komplexitását. Ezt én egy online program segítségével oldottam meg, a Lizard-dal. A BBP java kódját elemzte a

program. Íme:

Try Lizard in Your Browser

.java

Analyse

```

public class BBP {
    String HexaJegyek;

    public BBP(int d) {

        double HexPi = 0.0;

        double S1 = Sj(d, 1);
        double S4 = Sj(d, 4);
        double S5 = Sj(d, 5);
        double S6 = Sj(d, 6);
    }
}

```

Egyszerűen csak ki kell választanunk a forráskód nyelvét, majd beillesztenünk magát a kódot.

Code analyzed successfully.

File Type

.java

Token Count

401

NLOC

54

Function Name	NLOC	Complexity	Token #	Parameter #
BBP::BBP	22	3	196	
BBP::toString	3	1	8	
BBP::Sj	6	2	70	
BBP::n16modk	17	5	87	
BBP::main	3	1	22	

A végeredményen a számok minél kisebbek annál jobb, hiszen ha túl bonyolultak a függvények nehezen olvasható a program.

Számítása a gráfelméleten alapul. A forráskód alapján határozza meg az egyes függvények ciklomantikus komplexitását. Ez a független utak számát jelenti, hogy a program mennyire bonyolult vezérlési szempontból. Két út akkor számít függetlennek, ha mindkettőben van olyan pont, amely nem eleme a másiknak.

III. rész

Irodalomjegyzék

Általános

[MARX] Marx, György, *Gyorsuló idő*, Typotex , 2005.

C

[KERNIGHANRITCHIE] Kernighan, Brian W. És Ritchie, Dennis M., *A C programozási nyelv*, Bp., Műszaki, 1993.

C++

[BMECPP] Benedek, Zoltán És Levendovszky, Tihamér, *Szoftverfejlesztés C++ nyelven*, Bp., Szak Kiadó, 2013.

Lisp

[METAMATH] Chaitin, Gregory, *META MATH! The Quest for Omega*, http://arxiv.org/PS_cache/math/pdf/0404/0404335v7.pdf , 2004.

Köszönet illeti a NEMESPOR, <https://groups.google.com/forum/#!forum/nemespor>, az UDPROG 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 érdeklődésükért és hasznos észrevételeikért.

Ezen túl kiemelt köszönet illeti az említett UDPROG közösséget, mely a Debreceni Egyetem reguláris programozás oktatása tartalmi szervezését támogatja. Sok példa eleve ebben a közösségben született, vagy itt került említésre és adott esetekben szerepet kapott, mint oktatási példa.