I. 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.

Tartalom

1. fejezet - Java könyv

A következő bekezdésekben a Java 2 útikalauz programozóknak 5.0 első kötetének élménybeszámolóját olvashatják:

A Java programozási nyelv hasonló a C nyelvhez, azonban újabb és több lehetőséggel áll rendelkezésünkre. Nyelvünk objektumorientált, ami annyit tesz, hogy egy program objektumokra, azon belül osztályokra van felbontva, míg a C és C++ nyelvek eljárásorientáltak. Az osztályokon belül használunk változókat, illetve metódusokat. Ez utóbbi felel azért, hogy milyen műveletet hajtunk végre az adatokon. Míg a C-nél mutatókat használunk, a Java-ban már referenciákat. Változókat épp úgy adhatunk meg mint C vagy C++ nyelveknél, először a változó típusa (int, string, double, stb…), majd „=” operátor és az érték amit meg szeretnénk adni. Amikor Java kódot írunk érdemes megjegyzéseket hozzá tenni a programunkhoz (épp úgy mint bármelyik másik nyelvnél), ezt a „//” egyetlen sor esetében, ha több sorban szeretnénk megjegyzést írni akkor a „/*” nyitó részként, és a „*/” záró részként szolgáló jelöléssel tehetjük meg. Osztályokat a class kulcsszóval vezetünk be, amiken belül tetszőleges sorrendben vehetjük fel annak metódusait, és adatait. Fontos eldönteni, hogy kinek a számára legyen látható az adott elem. Így pl.: egy public osztály mindenki számára látható, míg private csak az osztályon belülre vonatkozik. Osztályon belül a new operátorral hozhatunk létre újabb objektumokat. Ügyelnünk kell a kivételkezelésre (bizonyos programok esetében pl.: nullával való osztás), ezt a try és catch segítségével oldhatjuk meg. A Java sok kétdimenziós grafikai elemmel rendelkezik, ilyen pl az awt, amivel saját magunk készíthetünk két dimenziós alakzatokat.

A legelterjedtebb karakterkészlet az ASCII kódrendszer, ami 8 biten ábrázolja a karaktereinket. Ebből azonban sok nemzet által gyakran használt karakterek hiányoznak (nálunk pl.: ű és ő betűk). Az ASCII-tól kétszer annyi biten ábrázoló Unicode azonban már rendelkezik a számunkra szükséges karakterekkel. Java-ban az azonosítók betűvel kell, hogy kezdődjenek és betűvel vagy számmal kell folytatódniuk. Tetszőleges hosszúak lehetnek, de nem tartalmazhatják a nyelv kulcsszavait (pl.: int, bolean, for, return, stb…). Az egyszerű típusokat és objektumokat literálokkal inicializáljuk, ezek pedig: objektum, logikai érték, egész szám, lebegőpontos szám, karakter, szöveg, és végül osztály. Változódeklarációnál kell hogy megadjunk egyetlen típust és legalább egy változónevet. Tömböket hasonlóan adunk meg mint C-ben, itt viszont a tömb igazi típus lesz, nem pedig mutató. Ha struktúrákat szeretnénk elérni, azokra ponttal hivatkozhatunk.

Az utasítások két fajtája: kifejezés-utasítás és deklaráció utasítás. Elágazásokat az if szerkezettel, összetettebbeket a switch-el oldhatunk meg. A Java-ban a következő ciklusok ismertek: elöltesztelő (while), hátultesztelő (do, while), léptető (for, while), és bejáró (for). Egy ciklusból a break parancs segítségével léphetünk ki, ilyenkor a program automatikusan fut tovább a következő kódsorra. Hasonlóan a continue parancs is ezt teszi, azonban ezt már használhatjuk metódusokban és inicializáló blokkokban. Visszatérési értéket C-hez hasonlóan a return parancs fog nekünk adni, viszont a Java-ból kikerült a goto utasítás, a biztonság növelése érdekében.

Java-ban az osztályok a legkisebb önálló egységek. Egy osztály egy adott tulajdonságú halmaz elemeit tartalmazza (pl.: emberek, tárgyak). Itt figyelnünk kell, hogy egy adott osztálynak csak egy célja legyen, és ne terheljük túl az osztályokat. Osztályon belüli változódeklarációnál ügyelnünk kell arra, hogy mekkora legyen a változó „hatósugara”, hogy más osztályok ne használhassák egymás változóit. Osztályon belüli metódusokat metódusdefiníciók árják le, amiknek fej és törzs része van. Metódushíváskor nem elég csak a nevet megadni, annak paramétereit is megkell. Figyelnünk kell a metódustúlterhelésre is. Ugyanis egy osztályon belül lehet több azonos nevű metódus is. Ilyenkor a fordítóprogram a paraméterek száma és típusa alapján választja ki a számunkra megfelelő metódust. Objektumokat a new operátorral hozunk létre, a new után pedig megkell adnunk, hogy melyik osztályt példányosítjuk. Példányosításkor memóriát foglalunk le, ahol az objektum változói lesznek tárolva, és visszaadja a kezdőcímét. Programozáskor problémát szokott jelenteni, hogy az objektumok feleslegesen foglalnak memóriát, mert egyszerűen nem hivatkozik rájuk semmi. Java-ban ez nem

jelent problémát, mert a rendszer automatikus felszabadítja azokat helyeket, amire nincs hivatkozás. Más nyelveknél ezt a programozónak magának kell megtennie. A metódusokra nagyban hasonlítanak a konstruktorok. Ezek végrehajtása a példányosításkor azonnal megtörténik. Egy konstruktornak meg kell egyeznie az osztály nevével. Csak példányosításon keresztül meghívhatóak. Néhány esetben nem árt értesülni egy-egy objektum megsemmisüléséről. Ez osztály szinten a classFinalize, metódus szinten simán finalize metódus (ez egy destruktor) fogja nekünk megmondani. Polimorfizmusnak nevezzük amikor egy változó olyan módon van deklarálva, hogy a leszármazottak is hivatkozhatnak rá. Polimorfizmusnál megkülönböztetünk statikus és dinamikus változókat. A statikus változó típusa változatlan, még a dinamikus változhat a változó által hivatkozott tényleges típusra.

Az osztályok mellett másik fontos építőköve a nyelvnek az interfész, ami olyan referenciatípus, amely absztrakt metódusok deklarációjának és konstans értékeknek az összege. Valódi használata az implementációján keresztül történik, így egy absztrakt program konkréttá válik. Interfészek között is van öröklődés. Deklarálni az interface kulcsszóval lehet, hasonlóan mint az osztályoknál a class. Deklarációval egy vázat hozunk létre, és ebben implementációkat helyezünk el, amik osztáylokat helyettesítenek. Fontos a fordítási hiba elkerülése végett, hogy publikus implementációkat adjunk az interfész összes metódusához. Az interfész egy újabb referenciatípus. Úgy használjuk mint egy osztályt. Az interfészek öröklődését kiterjesztésnek nevezzük, és az extend paranccsal tudjuk megvalósítani. Ez szintén úgy működik mint az osztályoknál. Konstansokat a következő képpen vezetünk be: módosítók, konstans típusa, azonosító, inicializáló kifejezés és ; zárjuk le, mint a legtöbb sorunkat. Deklarálásukhoz a módosítókat kell használjunk. Az interfészek használata az őket használó osztályok hívják meg. Ez csak akkor tud megvalósulni, ha az összes implementáló metódus szignatúrája és visszatérési értéke megegyezik az interfészével, különben error.

Amikor programozunk törekedjünk a letisztult és átlátható formára. Révén, hogy objektumorientált nyelvről van szó, a programrészeket megfelelően tagoljuk. Erre lesznek segítségünkre a csomagok, amik tartalmazzák a fejlesztői környezetet és a kódunkat. Úgy is mondhatjuk, hogy a csomag a hozzáférési kategóriák használatának az eszköze. A csomagoknál az öröklődési szint hierarchikusan van jelen. Itt a gyermeket alcsomagnak hívjuk. Tartalmuk lehetnek típusok vagy további alcsomagok, melyeket ponttal választunk el. Itt egy fa struktúrát kell elképzeljünk, ahol nincs szorosabb kapcsolat az ős felé, mint bármelyik másik csomag iránt. Célja a programkód átláthatóságának növelése, vagyis a programozó munkájának könnyítése. Csomagokat fájlrendszerben (JDK) vagy adatbázisban tárolunk. A csomagok kódja a fordítási egységekben található. Deklarálásuk package kulcsszóval történik, amik a kód legelején kell hogy legyenek, már deklaráció után nem állhatnak. Továbbá lehet importdeklarációt alkalmazni aminek három fajtája a következő: egyszerű típusimport-deklaráció, igény szerinti típusimport-deklaráció, statikus típusimport-deklaráció (ez utóbbi Java 5-től kezdve van jelen). Ezek segítik a típusok használatát. Csomagot létrehozni úgy tudunk, hogy a elkészítjük a hozzá tartozó fordítási egységeket, ezeket deklaráljuk aszerint, hogy melyik csomaghoz tartoznak. Majd megírjuk a típus~ és importdeklarációkat a fordítási egységekben. Ügyeljünk a megfelelő névválasztásra, ugyanis csoportos projekt esetén egy rosszul megválasztott csomagnév sok fejfájást okozhat. Erre egy jó módszer ha az egész elérési útját adjuk meg névként.

Sokszor esünk bele abba a hibába, hogy azt hisszük mindent jól kódoltunk le, mégis hibás a program. Azt tudjuk, hogy bizonyos részek lefutnak, na de hol keressük a bugot? Erre használjuk a kivételkezelést (Java exception), ami Java-ban a hasonlóan történik mint C-ben. Ez egy olyan különleges helyzet, hogy valahol hibát dob a program, de mégis lefut, a hibás részt pedig egy kivételobjektum fogja kezelni, pontosabb információt kapva a hibáról. Ezt nevezzük kivétel kiváltásnak (throwing exception), és szintén három fajtája fordul elő: rendellenes dolog történt (pl.: 0-val való osztás), throw utasítás kivált egy kivételt(csomagokban vagy a kódban), aszinkron hiba (párhuzamos futásnál egyik szál megszakad). A kivétel kezelése oly módon történik, hogy a kivételkezelő megkeresi azt a helyet, ahol a kiváltott kivétel kezelése megtörténhet. A kiváltott típusnak meg kell egyeznie a kivétel típusával, vagy őse az osztályhierarchiában. A kivételkezelő dönti el, hogy mekkora az a blokk amit kivált. Ezeket az utasításokat kapcsoszárójelek közé teszi. A kivételt el is kell kapnunk valahogy, erre a célra a try lesz segítségünkre. Ehhez azonban hozzá kell tennünk még közvetlenül utána egy catch vagy finally blokkot, különben errort kapunk.

A generikusok a Java 5. újdonságai, feladatuk az osztályok vagy eljárások típusokkal való paraméterezése, ez által sokkal egyszerűbb megoldani a feladatokat. Céljai az egyszerűség, biztonság növelése és a hatékonyság. Típushelyettesítő forma a wildcard ami az alábbi formák egyike lehet: ?, ?

extends T, ? super T, itt T egy típust jelöl. Ügyelnünk kell, hogy a következők kivételével minden referencia-típus lehet generikus: névtelen belső osztály, kivétel típusok (oka, hogy futás közben a generikusok nincsenek tárolva, a throwable osztály márpedig futási időben él), felsorolási típusok (mivel statikus jellegűek ugyan az a probléma mint a kivételnél). A generikusok típustörléssel foglalkoznak, és objektumok terén végrehajtott ezért object típus lesz. Figyelnünk kell az instanceof kifejezésre, ugyanis nem mindig működik generikusokra, a típustörlés következményeként.

Gyűjteményeknek nevezzük azokat a típuskonstrukciós eszközöket, melyek feladata az egy vagy több típusba tartozó objektumok példányainak memóriában történő összefoglaló jellegű tárolása, lekérdezése és manipulálása. A legtöbb programozási nyelvben metalálhatók. Általában objektumok és tömbök mutatóinak vagy referenciáinak összekapcsolása. A Java 2-ben jelent meg, inkább a praktikusság jellemző rá, mintsem a teljesség. Alapvető gyűjtemények és leképezések osztályai szorosan összefüggnek, külön-külön nem használjuk őket. Az interfészeikhez több reprezentációs osztály is tartozhat, ezek közötti váltás egyszerű. A gyűjtemény (collection) egyik leszármazottja a halmaz (set), feladata a halmaz adattípus megvalósítása. Megszorításokat tesz a, nem tesz kiterjesztést a gyűjteményhez képest. Örökli a műveleteket, amik kifejezhetők halmazműveletekként. Másik leszármazottja a lista (list), és a lista adattípust valósítja meg. Itt az elemek duplikáltan is szerepelhetnek, és számít a sorrendjük. A lista ad kiterjesztést néhány gyűjtemény interfészhez. Néhány örökölt műveletet is megváltoztat, ez duplikálás és sorrend miatt fontos (pl.: a remove művelete csak az első elemet távolítja el a listából). A sor adatszerkezet lényege, hogy az első objektumot ami bekerül, azt vegyük ki elsőnek. A leképezés adatszerkezet kulcs-érték párokat tárol, a Map interfészen keresztül. Egy kulcshoz egy érték tartozik.

Egy szoftverrendszer felépítését az alapján kell megtervezni, hogy az adatokon milyen tevékenységek lesznek elvégezve. A legáltalánosabb a felülről lefelé haladó eljárás, hiszen gyors, könnyen áttekinthető. Azonban nem minden esetben ez a legmegfelelőbb eljárás. Ezt mindig a programozónak kell eldöntenie, hogy mi lesz a legmegfelelőbb a programhoz. Objektumorientált nylevként először mindig objektumokra bontjuk a feladatot. Ez reprezentálja a modellezendő világ egységeit. Az objektumokat osztályokba soroljuk azok tulajdonságai alapján (pl.: élőlény osztálynak van ember, állat és növény objektuma). Ezen belül lesznek az alosztályok (állatoknál pl.: emlős, madár ,stb…). Amikor egy objektumot nem tudunk egyértelműen besorolni egyetlen osztályba sem, olyankor kapcsolatokkal kötjük őket osztályokhoz. Minden osztály meghatározható egy másik osztály leszűkítésével vagy kiterjesztésével. Így jön létre az örökös és gyermek kapcsolat az osztályok között. Minden gyermek osztály örökli az ősosztály tulajdonságait, fordítva nem igaz. A programtervezés legfőbb célja, hogy jól átlátható kódot készítsünk, erre az objektum orientált nyelv az egyik legjobb módszer a fentebb említett példák miatt. Szintén három lépésre oszthatjuk: analízis (a probléma körvonalazása), rendszertervezés (a körünket részekre osztjuk), osztálytervezés (a rendszert további osztályokra bontjuk). Ha mindezzel megvagyunk akkor megkezdődhet az implementálás.

A C++ jelölésrendszeréből sok mindent átvett a Java. Egy C++-ban jártas programozónak nem okoz majd nagy nehézséget megtanulni a Java szintaktikai szabályait, mert nagy átfedés van a két nyelv között, mivel a Java szintaxisa a C és C++-ból fejlődött ki. Felépítésükben viszont eltérést látunk, mivel a Java egy teljesen ″ objektumorientált nyelv. Ez annyit takar, hogy minden változó és metódus egy osztálynak a része. Amikor megírunk egy programot, aztán fordítani és futtatni szeretnénk, akkor a C ++ fordítója a saját kódunkat gépi kóddá konvertálja. A gépi kódot már értelmezni tudja az eszközünk és így a programunk már futtatható. Ennek a módszernek az a hátránya, hogy egy másik számítógépen csak azonos platform mellet lesz garantált ugyanaz az eredmény. Ezt másnéven platform- függőségnek is nevezzük. Java-ban ha a típusokról van szó, akkor primitív és nem primitív típusokra oszthatjuk fel őket. A primitív típusok egy konkrét értéket tárolnak. Ezeket a primitív típusokat helyettesíthetjük csomagoló osztályokkal is. Minden primitív adattípus rendelkezik egy hasonló (vagy vele megegyező) névű, de nagy betűs csomagoló osztállyal. Például az int-et helyettesíthetjük az Integer-rel. Osztályokat úgy mint C++-ban, Java-ban is a class teremti meg. A C++-ban különféle beépített vagy akár saját osztályok eléréséhez header fájlokat include-olása szükséges. Ez Java-ban viszont a header fájlok hiányában nem lehetséges. Helyette csomagokat használunk, amelyeket az import-tal érhetünk el, vagy simán megadhatjuk a csomag elérési útvonalát.

2. fejezet - Helló, Arroway!

OO szemlélet

Feladatunk egy prolártranszformációs normális generátor megvalósítása Java nyelven. Tekintsünk el a matematikai háttér megértéséről. Számunkra inkább maga a program működése a lényeges.

A polártranszformációs generátor Java-ban:

```
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;
        }
}
```

Az elején láthatunk egy `nincsTarolt` nevű boolean típusú váltózót. Ez fogja meghatározni a későbbi sorokban, hogy a program generál-e számot, vagy pedig az eltárolt értéket adja vissza. Ez a változó alapértéke igaz, mivel az eddig nincs semmink, amit eltárolhatnánk. Amennyiben nincs tárolt számunk, a `kovetkezo()` függvény generál nekünk két random számot, az egyiket visszaadja, a másikat pedig eltárolja egy `tarolt` nevű double-ben. Ekkor a `nincsTarolt` hamis értéket kap. Ha már rendelkezünk tárolt értékkel, akkor a függvényünk egyszerűen kiírja a tárolt számot és a `nincsTarolt` igaz értéket kap.

Ha a fenti kódot összevetjük az OpenJDK Random.java állományában találhatóval, láthatjuk, hogy lényegében megegyezik a két forráskód.

```
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;
        }
```

Fellelhető ugyan egy apró eltérés a felcserélt if és else ágakban, de mivel `haveNextNextGaussian` alapértéke hamis, így az eredmény ugyanaz lesz.

A C++ megvalósítás is nagyban hasonlít a Java verziójához, de mint azt már tudjuk, a Java szintaktikája a C++-tól vett át sok mindent

```
#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;
    }
}
```

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

```
int
main (int argc, char **argv)
{
  PolarGen pg;
  for (int i = 0; i < 10; ++i)
    std::cout << pg.kovetkezo () << std::endl;
  return 0;
}
```

„Gagyi”

Az ismert formális tesztkérdéstípusra adj a szokásosnál (miszerint x, t az egyik esetben az objektum által hordozott érték, a másikban meg az objektum referenciája) „mélyebb” választ, írj Java példaprogramot mely egyszer végtelen ciklus, más x, t értékekkel meg nem! A példát építsd a JDK Integer.java forrására3 , hogy a 128-nál inkluzív objektum példányokat poolozza!

Vizsgáljuk meg a két alábbi kódcsipet eredményét.

```
public static void main (String[]args)
{
Integer x = 127;
Integer t = 127;
System.out.println (x);
System.out.println (t);
while (x <= t && x >= t && t != x);
}
```

```
public static void main (String[]args)
{
Integer x = 128;
Integer t = 128;
System.out.println (x);
System.out.println (t);
while (x <= t && x >= t && t != x);
}
```

Egy olyan ciklust szeretnénk létrehozni, ahol két számnak egyszerre kell kisebbnek (és egyenlőnek), nagyobbnak (és egyenlőnek), és nem egyenlőnek lenni. Ezt egy kis trükkel megtudjuk valósítani, ha ismerjük az `Integer` memóriafoglalási elvét. Az első példában nem következik be végtelen ciklus, mivel nem teljesültek a feltételek. Ellemben a második példában végtelen ciklusba torkollik a program, mert teljesül minden feltétel.

De hogyan teljesülhet egyszerre mind a három feltétel? A válasz abban rejlik, hogy Java-ban az `Integer` osztályban 127-ig pool-olva vannak a pozitív egészek értékei, tehát ha 127 vagy attól kisebb értéket adunk az objektumank, akkor ugyanarra a memoriórészre hivatkozik. Lényeges, hogy a != operátor nem az objektumok értékeit fogja összehasonlítani, hanem referenciákat. Mivel a 128 már nem pool-olt érték, ezért különböző memóriacímmel fog rendelkezeni a két objektum.

Yoda

Írjunk olyan Java programot, ami java.lang.NullPointerEx-el leáll, ha nem követjük a Yoda conditions-t!

A Yoda condition tulajdonképpen annyi, hogy feltétel sorrendje megfordul, és a bal oladlon fog elhelyezkedni a konstans, jobb oldalra pedig a változó kerül. Használtaával elkerülthetőek olyan hibák, mint például az ertékadó operátor(=) használata az összehasonlító helyet(==).

```
String myString = null;
if (myString.equals("foobar")) { /* ... */ }
```

Ezzel a kóddal NullPointerException-be ütközünk, mivel `null` értéket nem hasonlíthatunk konstashoz.

Nézzeük meg a Yoda conditions alakalmazásával:

```
String myString = null;
if ("foobar".equals(myString)) { /* ... */ }
```

Így viszont lefut a kódunk, mivel string konstanst hasonlíthatunk `null` értékhez. Ebben a példában hamis értéket ad a kifejezés, ahogy az várható.

Kódolás from scratch

A BBP, azazBailey-Borwein-Plouffe algoritmus segíttségével a Pi hexadecimális számjegyeit tudjuk meghatározni.

Vessünk egy pillantást a kódra.

```
public class PiBBP {
    public PiBBP(int d) {

        double d16Pi = 0.0d;

        double d16S1t = d16Sj(d, 1);
        double d16S4t = d16Sj(d, 4);
        double d16S5t = d16Sj(d, 5);
        double d16S6t = d16Sj(d, 6);

        d16Pi = 4.0d*d16S1t - 2.0d*d16S4t - d16S5t - d16S6t;

        d16Pi = d16Pi - StrictMath.floor(d16Pi);

        StringBuffer sb = new StringBuffer();

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

        while(d16Pi != 0.0d) {

            int jegy = (int)StrictMath.floor(16.0d*d16Pi);

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

            d16Pi = (16.0d*d16Pi) - StrictMath.floor(16.0d*d16Pi);
        }

        d16PiHexaJegyek = sb.toString();
    }
```

Az algroritmus alapján a {16^d Pi} = {4*{16^d S1} - 2*{16^d S4} - {16^d S5} - {16^d S6}} kiszámolásával kezdi a program. A {} jelölés a törtrész jelölésére szolgál. Mivel 16-os szsámrendszerben dolgozunk, és a Java nem tartalmaz hexa számjegyeket, így meg kell adnunk őket egy `hexaJegyek[]` tömbben.

```
    public double d16Sj(int d, int j) {

        double d16Sj = 0.0d;

        for(int k=0; k<=d; ++k)
            d16Sj += (double)n16modk(d-k, 8*k + j) / (double)(8*k + j);
        return d16Sj - StrictMath.floor(d16Sj);
```

Itt a {16^d Sj} részt számoljuk ki. A hexajegyek kiszámolása (d+1)-edik elemtől kezdődik.

```
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;
}
```

A 16^n mod k kiszámítása bináris hatványozással történik, ahol n a kitevő, k a modulus.

```
public String toString() {

        return d16PiHexaJegyek;
    }

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

A `toString()` függvény a kiszámolt hexajegyek visszaadására szolgál. A `main` függvényben példányosítunk egy PiBBP objektmot. Ebben a példában d=1000000, ezért 1000001. hexajegytől írjuk ki a Pi-t.

3. fejezet - Helló, Liskov!

Liskov helyettesítés sértése

Írjunk olyan OO, leforduló Java és C++ kódcsipetet, amely megsérti a Liskov elvet! Mutassunk rá a megoldásra: jobb OO tervezés.

A Liskov elv szerint minden osztály legyen helyettesíthető leszármozattaival anélkül, hogy ez befolyásolná a program helyes működését. Tehát ha T leszármazottja S, akkor behelyettesíthetjük T helyére S-t, és ezután is helyes eredmény kell kapnunk.

Nézzünk egy példát az elvet sértő programra:

```
class Madar {
public:
     virtual void repul() {};
};
class Program {
public:
     void fgv ( Madar &madar ) {
          madar.repul();
     }
};
class Sas : public Madar
{};
class Pingvin : public Madar
{};
int main ( int argc, char **argv )
{
     Program program;
     Madar madar;
     program.fgv ( madar );
     Sas sas;
     program.fgv ( sas );
     Pingvin pingvin;
     program.fgv ( pingvin );
}
```

A Madar lett a T osztályunk. Az S osztélyok pedig a Sas és Pingvin lettek. Mivel a Madar osztály tartalmazza a repülést, így az abból származtatott Sas és Pingvin osztályok is képesek leszenk rá a porgram szerint, de tudjuk, hogy a pingvin röpképtelen, ezért ez a valóságban nem következhet be.

Nézzünk egy megoldást a madaras példára a Liskov elv betartása mellett:

```
class Madar {};
class Program {
public:
     void fgv ( Madar &madar ) {
     }
};
class RepuloMadar : public Madar {
public:
     virtual void repul() {};
};
```

```
class Sas : public RepuloMadar
{};
class Pingvin : public Madar
{};
int main ( int argc, char **argv )
{
     Program program;
     Madar madar;
     program.fgv ( madar );
     Sas sas;
     program.fgv ( sas );
     Pingvin pingvin;
     program.fgv ( pingvin );
}
```

Láthatjuk, hogy a Madar osztálynak egy új leszármazottja is van, a RepuloMadar, és ebben található a repülés. A Pingvin úgyanúgy a Madar-ból származik, azonban a Sas-t a RepuloMadar-ból származtatjuk, így csak az lesz képes a repülésre.

Végül az elv teljesülése Java-ban. A gondoltamenet ugyanaz, a szintaktika kicsit eltér.

```
class Madar {}
class Program {
 public void fgv ( Madar madar ) {}
}
class RepuloMadar extends Madar {
 public void repul() {}
}
class Sas extends RepuloMadar {}
class Pingvin extends Madar {}
public class figyel{

 public static void main ( String[] args )
 {
      Program program = new Program();
        Madar madar = new Madar();
        program.fgv(madar);

        Sas sas = new Sas();
        program.fgv(sas);
        sas.repul();
        Pingvin pingvin = new Pingvin();
        program.fgv(pingvin);
 }
}
```

Szülő-gyerek

Írjunk Szülő-gyerek Java és C++ osztálydefiníciót, amelyben demonstrálni tudjuk, hogy az ősön keresztül csak az ős üzenetei küldhetőek!

Ebben a feladatban be kell mutatnunk azt, hogy az ős mutatóin vagy referenciáin keresztül csak az ős metódusait érhetjük el.

Nézzünk egy rövid példakódot:

```
#include <iostream>
using namespace std;
class Szulo
{};
class Gyerek: public Szulo{
public: void kiir(){
cout<<"gyerek"<<endl;
}
};
int main()
{
Szulo* sz= new Gyerek();
cout << sz->kiir() <<endl;
}
```

A Szulo gyermekosztályaként létrehozunk egy Gyerek osztályt. Ebben a leszármazottban definiálunk egy kiir nevű eljárást. Ez a kód nem fog lefordulni helyesen, hiszen a kiir() metódus az ősosztályban nem található meg.

Ha ezt a kódot Java-ban próbáljuk megvalósítani szintén problémába ütközünk, ugyanis nem hivható meg a leszármazott metódusa az ősön keresztül. Errort kapunk fordításnál, jogosan. Íme a kód Java-ban:

```
public class Szulo{
 public class Gyerek extends Szulo{
  public void kiir(){System.out.println("Gyerek");}
 }
 public  void main(String[] args)
 {
  Szulo szulo = new Gyerek();
  System.out.println(szulo.kiir());
 }
}
```

Ciklomatikus komplexitás

Számoljuk ki valamelyik programunk függvényeinek ciklomatikus komplexitását! Lásd a fogalomtekintetében a https://arato.inf.unideb.hu/batfai.norbert/UDPROG/deprecated/Prog2_2.pdf (77-79 fóliát)!

A ciklomatikus komplexitás vagy másnéven McCabe-komplexitás (alkotója után kapta ezt a nevet) a program forrásának komplexitását határozza meg egy konkrét számértékkel. A számítása gráfelméletre alapul. A képlet így néz ki: `M = E - N + 2P` , amelyben E a gráf éleinek száma, N a csúcsok száma és P az összefüggő komponensek száma.

Akkor nézzük is meg egy tetszőleges forráskód ciklomatikus komplexitás. Ez most az előzőekben már többször használt PiBBP algoritmus Java verziója lesz. Én a forrás ciklomatikus komplexitását a: Lizard [http://www.lizard.ws/] nevű open source elemzővel számoltattam ki. Tetszőleges kódot másolhatunk be a bal oldalra, majd kiválasztjuk a programnyelvet, és futtatjuk rajta az elemzőt. Az eredmény ta jobb oldalon kapjuk meg:

3.1. ábra - PiBBP.java ciklomatikus komplexitása

Function Name	NLOC	Complexity	Token #	Parameter #
PiBBP::PiBBP	20	3	192	
PiBBP::d16Sj	6	2	70	
PiBBP::n16modk	17	5	87	
PiBBP::toString	3	1	8	
PiBBP::main	3	1	22	

EPAM: Interfész evolúció Java-ban

Mutasd be milyen változások történtek Java 7 és Java 8 között az interfészekben. Miért volt erre szükség, milyen problémát vezetett ez be?

Vizsgáljuk meg az alábbi két kódot:

```
            package com.epam.training;

public interface InterfaceA {

 default public void method() {
  System.out.println("InterfaceA.method()");
 }

}
```

A Java 7, vagy korábbi verziókban nem lehetett implementálni az interfészekben default metódust. Ehhez képest a Java 8-ban újításnak hatott default metódusra azért volt szükség, hogy könnyebb legyen biztosítani a visszafele kompatibilitást az ebben a verzióban megjelent újításokkal kapcsolatban. Lássuk milyen problémát vetett is ez fel:

```
            package com.epam.training;

public interface InterfaceB {

 default public void method() {
  System.out.println("InterfaceB.method()");
 }

}
```

A példákban jól látható, hogy ha több interfész is rendelkezik ugyan azzal a default metódussal (és adnak hozzá default implementációt), akkor fellép egy többszörös öröklődési probléma. Erre kénytelenek vagyunk megadni egy explicit módon felüldefiniáló osztályt, ahol implementáljuk a két interfészt, és eldöntjük, hogy még is melyiket szeretnénk használni.

```
public class Implementation implements InterfaceA, InterfaceB {

 @Override
 public void method() {
  InterfaceA.super.method();
 }

}
```

4. fejezet - Helló, Mandelbrot!

Reverse engineering UML osztálydiagram

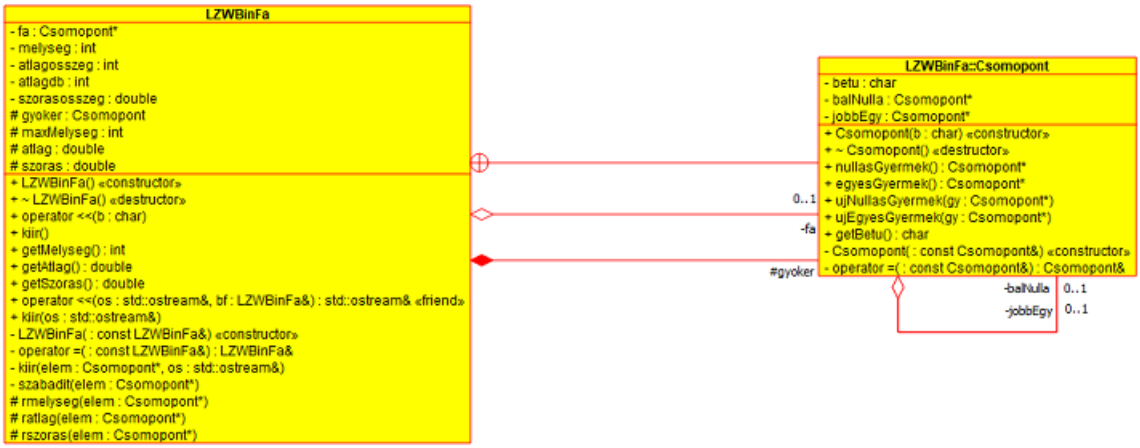
UML osztálydiagram rajzolása az első védési C++ programhoz. Az osztálydiagramot a forrásokból generáljuk (pl. Argo UML, Umbrello, Eclipse UML) Mutassunk rá a kompozíció és aggregáció kapcsolatára a forráskódban és a diagramon, lásd még: https://youtu.be/Td_nlERlEOs

https://arato.inf.unideb.hu/batfai.norbert/UDPROG/deprecated/Prog1_6.pdf (28-32 fólia) [https://arato.inf.unideb.hu/batfai.norbert/UDPROG/deprecated/Prog1_6.pdf]

Az UML (Unfied Modeling Language) egy általános célokra szánt modellező nyelv. Számunkra abban lesz hasznos, hogy ábrázoljuk az objektum orientált szemléletben írt kódunkat. Először is szükségünk lesz egy erre alkalmas programra. A választás az Umbrello-ra esett, mivel ez egy ingyenes UML diagram készítő.

A kódimportáláshoz a varázslóra lesz szükségünk. A `z3a7.cpp` ábrázolása a feladatunk, ezért a C ++ nyelvet kell kiválasztanunk. Miután kiválasztottuk a nyelevet, a kiterjesztések részt bővítsük ki a `.cpp`-vel, hogy improtálhassuk a kódunkat. Ezután a fastruktúra nézetből behúzzuk az osztályokat. Majd a fenti diasor alapján módosítjuk egy kicsit.

4.1. ábra - LZWBinFa UML



Végül egy pár szót az aggregáció és kompozíció kapcsolatáról. Ezek az egész-rész viszonynak alapvető formái. Az aggregációt üres rombusszal ábrázoljuk, a kompzíciót pedig teli rombusszal. Az előbbi hozzátartozik valamihez, de önmagában is létezhet, amíg az utóbbi csak valami részeként létezhet.

Forward engineering UML osztálydiagram

UML-ben tervezzünk osztályokat és generáljunk belőle forrást!

A feladathoz ismét az LZWBinfa C++ verzióját fogom felhasználni. Az Umbrello nem csak kódból készült diagram generálásra alkamlas, hanem fordítva is felhasználható. Ezúttal egy diagramból készítünk kódot. A kapott eredmény nem lesz futtatásra kész, ez csak egy váza lesz a programnak, az osztályok és függvények deklarációit tartalmazza. A `Code Generation Wizard` használatával ez könnyen kivitelezhető. Lássuk a kapott eredményt:

```
#ifndef LZWBINFA_H
```

```
#define LZWBINFA_H
#include <string>
class LZWBinFa
{
public:
  LZWBinFa ();
  virtual ~LZWBinFa ();
  void operator_ (char b)
  {
  }
  void kiir ()
  {
  }
  int getMelyseg ()
  {
  }
  double getAtlag ()
  {
  }
  double getSzoras ()
  {
  }
  std::ostream& operator_ (std::ostream& os, LZWBinFa& bf)
  {
  }
  void kiir (std::ostream& os)
  {
  }
protected:
  Csomopont gyoker;
  int maxMelyseg;
  double atlag;
  double szoras;
public:
protected:
public:
  void setGyoker (Csomopont new_var)   {
      gyoker = new_var;
  }
  Csomopont getGyoker ()   {
    return gyoker;
  }
  void setMaxMelyseg (int new_var)   {
      maxMelyseg = new_var;
  }
  int getMaxMelyseg ()   {
    return maxMelyseg;
  }
  void setAtlag (double new_var)   {
      atlag = new_var;
  }
  double getAtlag ()   {
    return atlag;
  }
  void setSzoras (double new_var)   {
      szoras = new_var;
  }
  double getSzoras ()   {
```

```
    return szoras;
  }
protected:
  void rmelyseg (Csomopont* elem)
  {
  }
  void ratlag (Csomopont* elem)
  {
  }
  void rszoras (Csomopont* elem)
  {
  }
private:
  Csomopont* fa;
  int melyseg;
  int atlagosszeg;
  int atlagdb;
  double szorasosszeg;
public:
private:
public:
  void setFa (Csomopont* new_var)   {
      fa = new_var;
  }
  Csomopont* getFa ()   {
    return fa;
  }
  void setMelyseg (int new_var)   {
      melyseg = new_var;
  }
  int getMelyseg ()   {
    return melyseg;
  }
  void setAtlagosszeg (int new_var)   {
      atlagosszeg = new_var;
  }
  int getAtlagosszeg ()   {
    return atlagosszeg;
  }
  void setAtlagdb (int new_var)   {
      atlagdb = new_var;
  }
  int getAtlagdb ()   {
    return atlagdb;
  }
  void setSzorasosszeg (double new_var)   {
      szorasosszeg = new_var;
  }
  double getSzorasosszeg ()   {
    return szorasosszeg;
  }
private:
   LZWBinFa (const LZWBinFa& )
  {
  }
  LZWBinFa& operator_ (const LZWBinFa& )
  {
  }
```

```
  void kiir (Csomopont* elem, std::ostream& os)
  {
  }
  void szabadit (Csomopont* elem)
  {
  }
  void initAttributes () ;
};
#endif
```

Ahogy azt láthatjuk, az UML diagramoknak nagy hasznát vehetjük már tervezési fázisban is. A kapott kód jól szemlélteti a program szerkezeti feképítését.

Egy esettan

A BME-s C++ tankönyv 14. fejezetét (427-444 elmélet, 445-469 az esettan) dolgozzuk fel!

Ennek a feladatnak is szerves részét képzik az UML diagaramok, ezért ismerkedjünk meg az alapvető dolgokkal az előzőekben látottak alapján. UML-ben téglalpokkal ábrázoljuk az osztályokat. Ezeket három részre bonthatjuk tovább. Fentről lefelé az első rész az osztály neve, majd ezt követik annak attribútumai, és végül a változókat láthatjuk. Függvények és változók előtt találhatók a láthatóságot jelölő '+'(public), '-'(privát) és '#'(védett).

Az esettanulmány a következő. Adott egy számítógep-alkatrész és számítógep-konfiguráció értékesítésével foglalkozó kereskedés. El kell készíteni egy olyan programot, amellyel megvalósítható a kereskedés alkatrészeinek és konfigurációinak a nyílvántartása. A szofvernek támogatnia kell a termékek állományból való betöltést, képernyőre kiírását, az állományba kiírását és a rugalmas árképzést.

Először is egy keretrendszer kialakítására van szükség osztálykönyvtárok által, ami alapszinten támogatja a termékcsaládokat. Majd erre a keretre építve egy alkatrészkezelő alkalmazás elkészítése vár ránk. A végső cél az, hogy a meglévő keretrendszerre változtatása nélkül más termékcsaládok támogatása is megvalósuljon.

A keretrendszer kialakításához kelleni fog egy `Product`osztály, amelyben termékek általános kezelését implementáljuk. Ez az osztály tárolja az általános jellemzőket, pl.: terméknév, beszerzési ár, beszerzési dátum. A különböző termékek a `Product` leszármazottai lesznek. A típusnév és típuskód a `Product` osztályban virtuális függvényként definiáljuk, mivel ezeknek a leszármazott osztályokban egyediek.

Az összetett termékek (például a számítógep-konfigurációk) kezelése a `CompositeProduct` osztályban valósul meg, amit szintén a `Product`-ból származtatunk. Vektorban fogjuk tárolni az alkatrészeket, amit az `AddPart` metódus segítségével bővíthetünk.

A termékek nyilvántartására létrehozzuk a `ProductInventory` osztályt. Arra szolgál, hogy betöltse a termékek listáját, a betöltött termékeklistákat tárolja a memóriában, a memóriából adatfolyamba írás és a formázások megjelenítése. A lista bővítésére az `AddProduct` szolgál.

Mivel a keretrendszer osztálykönvtár, ezért az általunk bevezetett termékosztályokat nem ismeri, így szükségünk van egy osztályra, amivel kiküszöbölhető ez a probléma. A megoldást számunkra egy `ProducFactoryt` osztály lesz. Ebben lesz egy`ReadAndCreateProduct` függvénye, amit a keretben definiálunk, de az általa meghívott `CreateProduct` függvény már virtuális.

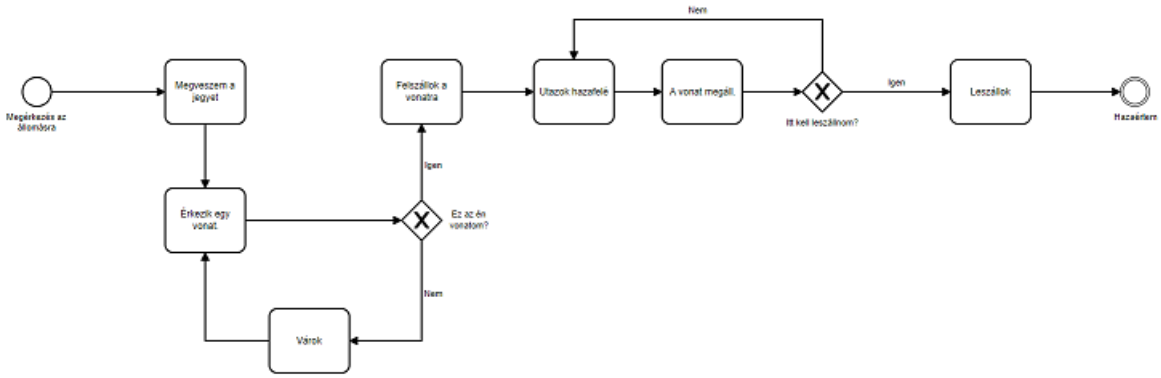
BPMN

Rajzoljunk le egy tevékenységet BPMN-ben!

A BPMN (Business Process Model and Notification) üzleti folyamatok modellezéséhez használatos folyamatábra jelölés. A célja az üzleti folyamatok egységes grafikus ábrázolása. Ilyen folyamatábrákat mi is tudunk készíteni. Én böngészőből csináltam a https://demo.bpmn.io/ oldalon.

Ahhoz, hogy mi is ábrákat készítsünk először ismerjük meg az alapvető jelöléseket. Az eseményeket(event) körrel jelöljük: kezdeti esemény simar körrel, és a végső esemény pedig vastag körrel. A tevékenységeket(activity) lekerekített sarkú téglalap jelöli. Az átjáróknak(gateway) trapéz a jele. Az elemek közti összekötést(association) nyilakkal jelölik.

4.2. ábra - BPMN



5. fejezet - Helló, Chomsky!

Paszigráfia Rapszódia OpenGL full screen vizualizáció

Lásd vis_prel_para.pdf! Apró módosításokat eszközölj benne, pl. színvilág, textúrázás, a szintek jobb elkülönítése, kézreállóbb irányítás.

A Paszigráfia Rapszódia célja, hogy lehetővé tegye homonkulusz és mesterséges homunkulusz közti kommunikációt. OpenGL segítségével valósul meg a vizualizáció. Szükségünk lesz `lib-boost` és `freeglut3` csomagokra. Ezeket könnyedén letölthetjük az alábbi két paranccsal

```
sudo apt-get install libboost-all-dev
sudo apt-get install freeglut3-dev
```

Az a feladat, hogy miután felélesztettük a programot, apróbb változtatásokat is hajtsunk végre benne. A színek változtatásához a `glColor3f` függvényt kell módosítanunk.

```
glColor3f ( 0.0, 1.0f, 0.0f );
```

Ez például zöld színt eredményez. Ha más színt szeretnénk megadni, akkor néhány alapvető színt megtalálhatunk például itt [https://pemavirtualhub.wordpress.com/2016/06/20/opengl-color-codes/].

```
void skeyboard ( int key, int x, int y )
{
        if ( key == GLUT_KEY_UP ) {
                cubeLetters[index].rotx -= 5.0;
        } else if ( key == GLUT_KEY_DOWN ) {
                cubeLetters[index].rotx += 5.0;
```

Ez pedig felcseréli a fel és le gombok funkcióját, tehát másik irányba fordulnak, mint eddig.

A program fordítása és futtatása:

```
g++ para6.cpp -o para -lboost_system -lGL -lGLU -lglut
./para 3:2:1:1:0:3:2:1:0:2:0:2:1:1:0:3:3:0:2:0:1:1:0:1:0:1:0:1:0:2:2:0:1:1:1:3:
```

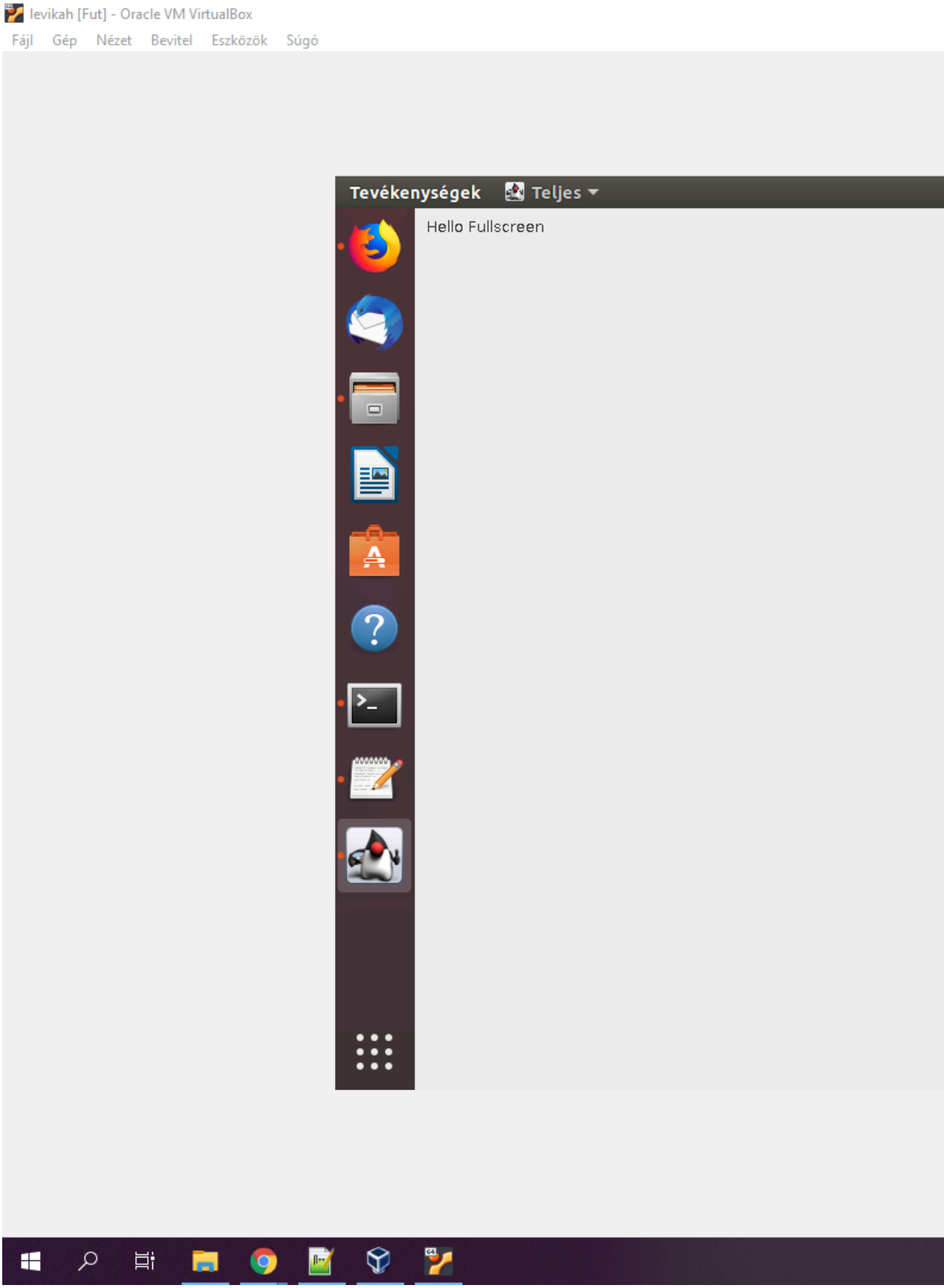
Full screen

Készítsünk egy teljes képernyős Java programot! Tipp: https://www.tankonyvtar.hu/en/tartalom/tkt/javat-tanitok-javat /ch03.html#labirintus_jatek [https://www.tankonyvtar.hu/en/tartalom/tkt/javat-tanitok-javat/ch03.html#labirintus_jatek]

Ebben a feladatban egy egyszerű kiírást fogunk kitenni teljes képernyőre. Néhány YouTube videó és kis böngészés után ez [http://www.java2s.com/Code/JavaAPI/java.awt/GraphicsdrawStringStringstrintxinty.htm] alapján készítettem el egy nagyon egyszerű változatot.

A végeredmény nem lett túl látványos, de a célnak megfelel:

5.1. ábra - Teljes Képernyő



A programot a szükséges osztályok improtálásával kezdjük.

```
import java.awt.*;
import javax.swing.JFrame;
import javax.swing.JPanel;
```

Létrehozzuk a `Teljes` osztályunkat a `Jpanel` leszármazottjaként.

```
public class Teljes extends JPanel {
```

A `paint` metódusban található `drawString` függvény lesz felelős kiírásért.

```
public void paint(Graphics g) {
    g.drawString("Hello Fullscreen", 200, 200);
  }
```

Elértünk a `main`-hez. Az elején példányosítunk egy `JFrame`-t. A többi sorba pedig magyarázó kommenteket tettem.

```
  public static void main(String[] args) {
    JFrame frame = new JFrame("Teljes"); //Zárójelek között a programnév
    frame.getContentPane().add(new Teljes());
    frame.setExtendedState(JFrame.MAXIMIZED_BOTH); //Itt lesz teljes képernyős
    frame.setUndecorated(true); //etávolítja a címsort
    frame.setVisible(true);
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);//Kilépéskor bezárja a
    }
}
```

l334d1c46

Írj olyan OO Java vagy C++ osztályt, amely leet cipherként működik, azaz megvalósítja ezt a betű helyettesítést: https://simple.wikipedia.org/wiki/Leet (Ha ez első részben nem tetted meg, akkor írasd ki és magyarázd meg a használt struktúratömb memóriafoglalását!)

A leet nyelvvel már volt dolgunk korábban a Prog1-es könyv során. Lényege, hogy a betűket (vagy akár számokat is) kicseréljük hozzáájuk hasonló karakterre vagy karakterláncra. A feladatban látható Java forrás alapját ez a kód [https://codehackersblog.blogspot.com/2015/06/leet-speak-convertor-with-java-code.html] adta. Ezen egy kicsit változtattam, mivel tartalmazott számunkra most lényegtelen részeket.

Itt is a szükséges könyvtárak improtálásával kezdtük.

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.util.HashMap;
import java.util.regex.Matcher;
import java.util.regex.Pattern;
```

Deklaráljuk a `toLeetCode` függvényt. Létrehozunk egy `Patter` típust, aminek az elfogadott szövegmintát adjuk meg értékül, ami jelen esetben kis- és nagybetűkből állhat. A `StringBuilder` a végeredmény tárolására szolgál.

```
class L33tConvertor {

    private String toLeetCode(String str) {
```

```
        Pattern pattern = Pattern.compile("[^a-zA-Z]");
        StringBuilder result = new StringBuilder();
```

Ezt követően egy `HashMap`-ot csinálunk, ami kulcs érték párok tárolására alkalmas. Ez fogja tárolni a helyettesítéseket, amiket megadunk.

```
HashMap<Character, String> map = new HashMap<Character, String>();
        map.put('A', "@");
        map.put('B', "ß");
        map.put('C', "©");
        map.put('D', "ԁ");
        map.put('E', "€");
        map.put('F', "ƒ");
        map.put('G', "6");
        map.put('H', "#");
        map.put('I', "!");
        map.put('J', "¿");
        map.put('K', "X");
        map.put('L', "£");
        map.put('M', "M");
        map.put('N', "r");
        map.put('O', "0");
        map.put('P', "p");
        map.put('Q', "0");
        map.put('R', "®");
        map.put('S', "$");
        map.put('T', "7");
        map.put('U', "µ");
        map.put('V', "v");
        map.put('W', "w");
        map.put('X', "%");
        map.put('Y', "¥");
        map.put('Z', "z");
```

Maga az átalakítás egy `for` ciklusban valósul meg. Ez a ciklus bejárja a szöveget egyesével és nagybetűre alakítja őket, és megnézi, hogy van-e helyettesítése a vizsgált karakternek a mapunkban. Ha nem talál helyettesítést, akkor visszaadja az adott karaktert, egyébként pedig a `map.get` visszaadja a leet változatot. Az végeredményt stringre konvertálva kapjuk vissza.

```
for (int i = 0; i < str.length(); i++) {
            char key = Character.toUpperCase(str.charAt(i));
            Matcher matcher = pattern.matcher(Character.toString(key));
            if (matcher.find()) {
                result.append(key);
                result.append(' ');
            } else {
                result.append(map.get(key));
                result.append(' ');
            }
        }
        return result.toString();
```

A `main` függvényben példányosítunk egy `L33tConvertor` osztályt. Egy BufferReader fog beolvasni a standard bementről. A `leetWord` tárolja az általunk beírt szöveget, és erre hívjuk meg a toLeetCode függvényt.

```
public static void main(String[] args) throws IOException {
       L33tConvertor obj = new L33tConvertor();
       BufferedReader br = new BufferedReader(new InputStreamReader(System.in)
       String leetWord;

 System.out.println("\nEnter the English Words :-");
                  leetWord = br.readLine();
                  String leet = obj.toLeetCode(leetWord);
                  System.out.println("The 1337 Code is :- " + leet);

}
```

EPAM: Bináris keresés és Buborék rendezés implementálása

Implementálj egy Java osztályt, amely képes egy előre definiált n darab Integer tárolására. Ennek az osztálynak az alábbi funkcionalitásokkal kell rendelkeznie:

- Elem hozzáadása a tárolt elemekhez

- Egy tetszőleges Integer értékről tudja eldönteni, hogy már tároljuk-e (ehhez egy bináris keresőt implementálj)

- A tárolt elemeket az osztályunk be tudja rendezni és a rendezett (pl növekvő sorrend) struktúrávalvissza tud térni (ehhez egy buborék rendezőt implementálj)

Mivel halmazokkal fogunk foglalkozni, fontos, hogy importáljuk is azt: `java.util.Arrays`. Majd vegyük fel a "főosztályt" `IntegerCollection`, ahol lesz maga a halmazunk, egy index (mely utal majd a halmaz i-edik elemeire), egy logikai változós sorted (hogy tudjuk rendezve vannak-e már a számok), és kell egy size is ,ami a halmaz számosságát adja vissza.

```
package com.epam.training;

import java.util.Arrays;

public class IntegerCollection {

 int[] array;
 int index = 0;
 int size;
 boolean sorted = true;

 public IntegerCollection(int size) {
  this.size = size;
  this.array = new int[size];
 }

 public IntegerCollection(int[] array) {
  this.size = array.length;
  this.index = this.size;
  this.array = array;
  this.sorted = false;
 }
```

A következő rész felel az új elem felvételéért, ahol először megnézzük, hogy nem-e haladtuk meg a halmaz méretét. Amennyiben igen úgy errort dob a progi `The collection is full`, de ha nem ütközünk semmilyen problémába, úgy jöhet az új elem, de ekkor már a rendezést hamisra kell állítsuk.

```
public void add(int value) {
 if (size <= index) {
  throw new IllegalArgumentException("The collection is full");
 }
 sorted = false;
 array[index++] = value;
}
```

A második kritérium szerint el kell tudni döntenie, hogy a felvett érték szerepel-e már a kis halmazunkban. Tehát a `contains` osztály pontosan ezt fogja tenni. Végig megy az elemeken és egyesével összehasonlít. A végén pedig hamis értéket ad vissza.

```
public boolean contains(int value) {
 if (!sorted) {
  sort();
 }

 int left = 0, right = size - 1;
 while (left <= right) {
  int mid = left + (right - left) / 2;

  if (array[mid] == value) {
   return true;
  }

  if (array[mid] < value) {
   left = mid + 1;
  } else {
   right = mid - 1;
  }
 }

 return false;
}
```

Végül pedig a rendzés. Itt a két egymásba ágyazott for ciklus az egymás mellett álló elemeket nézi meg, és rendezi őket növekvő sorrendbe.

```
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 temp = array[j];
    array[j] = array[j + 1];
    array[j + 1] = temp;
   }
  }
 }
 sorted = true;
 return array;
}
```

```
 @Override
 public int hashCode() {
  final int prime = 31;
  int result = 1;
  result = prime * result + Arrays.hashCode(array);
  return result;
 }

 @Override
 public boolean equals(Object obj) {
  if (this == obj) {
   return true;
  }
  if (!(obj instanceof IntegerCollection)) {
   return false;
  }
  IntegerCollection other = (IntegerCollection) obj;
  return Arrays.equals(array, other.array);
 }

 @Override
 public String toString() {
  return "IntegerCollection [array=" + Arrays.toString(array) + "]";
 }

}
```

A kiíratás és értékfelvétel a mian.java-ban történik, ahol először felvesszük az `IntegerCollection` elemeit és tároljuk őket. Majd a felvehető elemeket. Végül kiíratjuk őket tárolt és növekvő sorrendben, illetve a felvett elemeket:

```
package com.epam.training;

public class Main {

 public static void main(String[] args) {
  IntegerCollection collection = new IntegerCollection(3);
  collection.add(0);
  collection.add(2);
  collection.add(1);
  System.out.println(collection);
  collection.sort();
  System.out.println(collection);
  System.out.println(collection.contains(0));
  System.out.println(collection.contains(1));
  System.out.println(collection.contains(2));
  System.out.println(collection.contains(3));
  System.out.println(collection.contains(4));
 }

}
```