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.

Magasszintű programozási nyelvek 2 DEIK PTI BSc 4 órás laborok

Feladatgyűjtemény a mérési jegyzőkönyv kidolgozásához

A Prog2 tematika tartalmi megtöltése

A jelen dokumentumban konkrét labor és otthoni feladatokat rendelünk a (számunkra hivatalból előírt, betartandó) heti bontású tematika tételeihez. A heti labormunka az adott héthez rendelt feladatok teljesítésén alapszik. A hallgatók a laboron önállóan dolgoznak, az oktató rövid indító iránymutatása (például a feladatok pontosítása) után, ám a feladatokon otthon is dolgozni kell.

Egy „feladatcsokor” (egy csokorból 5 feladatot kell választani, lásd például később a „Helló, Arroway!” egy csokor) feladatok teljesítésére időben az adott hét áll rendelkezésre.

- Az induláskor alkalmazunk egy 2 hetes időbeli puffert, a szeptember 21-i heti laborodon a „Helló, Berners-Lee!” csokor megkezdését kell bemutatnod, a következő héttől pedig folytatólagosan a „Helló, Arroway!”-től kezdve. Ha adott laborközösségnek laborja elmarad, akkor ez a „hol tartunk számlálót” sem léptetjük.
- Az a feladat fogadható el megkezdettnek, amelyről részletesen tudósít a hallgató a jegyzőkönyv pdf-jében (vagy magáról a feladatról, ha kész, vagy a nehézségről, amibe beleütközött és gátolja a megoldásban).
- A jegyzőkönyvet DocBook-ben kell megvalósítani.

Közös munka

Minden labormérést lehet egy a félév során fix, két fős csoportban is végezni. Ennek előkészülete, hogy a két hallgató a prog1-es jegyzőkönyvéből minden prog1-es feladatból kiválasztja a jobb megoldást, ezt teszi be az új mérési jegyzőkönyvbe és a jelen prog2 részt ebben az anyagban valósítja meg. Az új feladatoknál szerepeltetni kell egy bekezdést, ami rögzíti, ki mit csinált a feladatban. Védeni egyénileg kell.

Értékelési szempontok

A jegyzőkönyvnek nyilvános repóban (és DocBook XML 5 forrásokban is) elérhetőnek kell lennie. Ha a DocBook források nem validak, vagy plágium van bennük, akkor az értékelés automatikusan elégtelen.

A gyakorlati jegyet az utolsó laborokon a laborjegyzőkönyvre és a védésre adjuk.

A jeles szükséges (de nem elégséges) feltétele, hogy

- minden héten mind az 5 feladatra legyen megoldásunk bemutatva a jegyzőkönyvben. Legalább egy feladat megoldása, annak elmagyarázása, bemutatása legyen kistreamve vagy legalább YB videóban kitéve.
- Legalább 5 hallgatótárs feltünteti a jegyzőkönyvében, hogy a szóban forgó hallgató tutorja volt vagy volt legalább 50 megnézés videón.

A jó szükséges (de nem elégséges) feltétele, hogy minden héten az 5 feladatból legyen 4 feladatra megoldásunk bemutatva a jegyzőkönyvben.

A közepes szükséges (de nem elégséges) feltétele, hogy minden héten az 5 feladatból legyen 3 feladatra megoldásunk bemutatva a jegyzőkönyvben.

A 9 heti bontásból 1-nél lehet három feladatnál kevesebb megoldás, ha 2 vagy több olyan hét van, ahol három feladatnál kevesebb megoldás van, az automatikusan elégtelen gyakorlati jegyet eredményez.

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.

> Ha bármely hétnél 1 megoldás van vagy egyetlen megoldás sincs, az elégtelen gyakorlati jegyet eredményez.

A félév utolsó laborjait a védésnek szenteljük, amely a jegyzőkönyvből az oktató által kiválasztott feladat gép melletti bemutatásából áll. A jegyzőkönyv és a védés alapján adja a laborvezető a gyakorlati jegyet. Ezt időben támogatandó a tematika néhány párját összevontuk az alábbiak szerint.

Háttér

Minden feladatot megcsináltam már, ezeket eléred az UDPROG közösségben (vagy a repóban, vagy az évkönyvben vagy a fészes csoportban) vagy egyéb célrepókban.

A zöld, piros és kék (lásd később) feladatok nem kötelezőek.

- A „deprecated” (zöld) feladatokat is lehet választani, de azok mivel régiek, kevés a támogatottság, tipikusan nehézséget okozhat a megoldásuk… (mert például a felhasznált API-k sokat változtak és már nem gondoztam a kódokat…). De olyan is van, melyet már meghaladtál, például az első részben is feldolgozhattál…
- A piros feladat kidolgozásával az egész csokor kiváltható, ezek tipikusan kutatási feladatok, beszállhatsz velük akár kutatási kéziratokba társszerzőként, akár TDK-zhatsz, szakdolgozhatsz belőlük.
- A kék olyan mint a piros, de a tartalmazó és egy másik csokor is kiváltható vele.

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.

A feladatcsokrok

Minden csokor alapértelmezésben kiegészül egy opcionálisan választható előre egyeztetett Red Flower Hell[1] (Python vagy C++ megvalósítású Mineacraft MALMÖ MI ágens) fejlesztéssel és egy nyelvi csokorral, ahol a tematika egyszerű elemeinek (pl. az első héten: Osztály, objektum, példányosítás) megfelelő rövid kis kódcsipeteket mutatunk be.

EPAM

Az idei kurzus kvintesszenciája, hogy a Java platformra koncentrálva maga az EPAM is visz egy labort. Minden csokorban az "**EPAM:** " prefix után találjátok az "EPAM-os" feladatokat. Ezek ugyanúgy választhatók feladatok bármely más kurzusbeli laborcsopornak. (Ezeknek a feladatoknak a listája és a referencia megoldásukra adott tippek, segítség, megoldás-forrás linkek az első pár hétben még bővülni, frissülni fognak.)

A referencia megoldások: https://github.com/epam-deik-cooperation/epam-deik-prog2

0. hét - „Helló, Berners-Lee!”

A szokásos olvasónapló feladat:

C++: Benedek Zoltán, Levendovszky Tihamér Szoftverfejlesztés C++ nyelven

Java: Nyékyné Dr. Gaizler Judit et al. Java 2 útikalauz programozóknak 5.0 I-II.

- Ebből a két könyvből pár oldalas esszé jellegű kidolgozást kérek, Java és C++ összehasonlítás mentén, pl. kb.: kifejezés fogalom ua., Javában minden objektum referencia, mindig dinamikus a kötés, minden függvény virtuális, klónozás stb.

Python: Forstner Bertalan, Ekler Péter, Kelényi Imre: Bevezetés a mobilprogramozásba. Gyors prototípus-fejlesztés Python és Java nyelven (35-51 oldal)

- Itt a kijelölt oldalakból egy 1 oldalas élmény-olvasónaplóra gondoltam.

Java és C++ Összehasonlítása:

A Java nyelv és környezet tervezői a szintaxis számára a C és a C++ nyelvek szintaxisát vették alapul. Számos C++ kifejezés, utasítás szintaktikailag helyes Java-ban is, és sokszor ma jelentésük is hasonló. Azonban természetesen a hasonlóság nem azonosság.

A Java mint nyelv szűkebb a C++-nál, ugyanakkor szabványos osztálykönyvtárai szélesebb területet fednek le. Nyelvi szinten támogatja a thread-eket, a grafikus felhasználói felület programozását, a hálózati programozást, a perzisztenciát, különböző processzek közötti osztott objektumelérést, adatbázisok tartalmának elérését stb. A szabványos kiegészítő csomagokkal ez a terület még tovább bővül a kriptográfiával, elektronikus levelezéssel stb. Ezeket a fogalmakat persze C++-ban is lehet kezelni, de a C++ nyelv, illetve a szabványos könyvtár közvetlenül nem terjed ki ezekre a részekre. Valamely külső könyvtár segítségét kell igénybe vennünk.

A C++-al írhatunk forrásszinten hordozható programokat, amelyet a célgépen újrafordítva és szerkesztve helyes, futó programot kapunk. Az összeszerkesztett bináris kód azonban nem hordozható. Ezzel szemben a Java célkitűzései között szerepel a platformok közötti bináris hordozhatóság. A bájtkóddá lefordított programot átvihetjük más gépekre ahol szabványos Java virtuális gép környezetet használunk.

Az objektummodellek különbözőségéből, alapvető különbségek adódnak a C++ és a Java között. A C++ nyelv az objektumokat, mint a memória egy összefüggő területén elhelyezkedő bájtsorozatot fogja fel. Ennek a memóriakiosztása ismert, ennek megfelelően manipulálja a lefordított program az objektumot. A C++ nyelvben a mutatók révén közvetlenül manipulálhatjuk a memóriát.

Ezzel szemben a Java programok virtuális gépen futnak: a memóriát közvetlenül nem tudjuk elérni, hanem csak szimbolikusan, hivatkozásokon keresztül. A Java esetén nincs linker, ami címekké oldaná fel a hivatkozásokat. Egy osztály egy önálló class fájllá fordul le. A class fájl formátuma szabványos és platform független.

A Java virtuális gép az objektumokat egy automatikus szemétgyűjtő mechanizmus által felügyelt véletlen elérésű tárterületen tárolja. A dinamikus tárban lefoglalt memória programozói felszabadítása közvetlenül nem lehetséges. A Java nyelvből hiányzik a destruktor mechanizmus.

A C++ nyelv többparadigmás. Írhatunk procedurális programokat, egymást hívó függvényekkel, változókkal. Alkalmazhatjuk az objektumorientált elveket; akár objektumokat létrehozva, akár bonyolult osztály-hierarchiák, öröklődés, virtuális függvények és egyéb technikák használatával. Végül írhatunk generikus elvű programokat is, könyvtárak, intenzív template-használattal. Ezeket az eszközöket vegyíthetjük is.

Ezzel szemben a Java csak az objektumorientált programozást támogatja. Nincsenek globális változók és függvények, csak osztályokhoz tartozó attribútumok, változók, konstansok és metódusok.

A Java tömbjei tudják méretüket, ezért a main metódus megadásánál szükségtelen a C++-nál megszokott argc megadása. Elég csak egy String tömb értékű argumentum. C és C++-al ellentétben a Java-ban a main első argumentuma nem tartalmazza a program nevét. Ha létezik 0 indexű paraméter, akkor az az első parancssori argumentumot tartalmazza. C++-ban gyakran nem deklaráljuk a main argumentumait, ha azokat nem használjuk a programból. Az üres argumentumú main a Java-ban is lefordul, egyéb célokra használhatjuk is, de futási idejű hibát okoz, ha az osztály belépési pontjául szánjuk.

A Java-ban a C++-ban megszokott /* blokk kommenten */ és a // sorkommenten kívül a blokk-komment speciális eseteként létezik a /** jelekkel elindított úgynevezett dokumentációs komment. Ezekből a javadoc segítségével HTML-formátumú dokumentáció generálhatunk.

A C és C++ forrást fordításkor először az előfordító dolgozza fel. Ilyen előfordító a Java környezetben nincsen.

A java fordítás fájllállományok beemelését nem támogatja, a makrókat sem, de a szimbólumbehelyettesítést olyan értelemben igen, hogy a szimbólumként való használatra szánt azonosítókat valamilyen osztályban vagy interfészben final static adattagokként definiálhatjuk. Ezen adattagok értékét a fordító figyelembe veszi fordításkor.

A C és C++ nyelvekben a fordító az egyes forrásállományokat egymástól függetlenül fordítja le. Ahhoz, hogy a több forrásból is hivatkozott objektumokat konzisztensen használjuk, az állományokban azonosan kell deklarálnunk őket. Az ilyen közös deklarációkat általában egy angolul header-nek nevezett fejállományban helyezzük el.

Ilyen mechanizmusra Java-ban nincs szükségünk. Java-ban az osztály interfészre és implementációs részre való szétválasztását nem kell azzal kihangsúlyoznunk, hogy a deklarációkat külön állományba helyezzük.

C++-ban egy osztály deklarációja és definíciója elkülönül, és a deklaráció végső zárójele után pontosvessző van. Java-ban erre nincs szükség, de nem is hiba.

Ha egy osztály definícióit egy másik osztályból fel szeretnénk használni, az import paranccsal tehetjük láthatóvá a szükséges neveket.

A C és C++ nyelvben a beépített típusok pontos mérete és értéktartománya nincs definiálva, így az egyes implementációk saját igényeik szerint járhatnak el. A program más platformokon újrafordítva használja ki az adott platform minden előnyét.

Ezzel szemben a Java, amely egy bináris kompatibilitást ígérő nyelv, szigorúbb definíciókkal él. Az egyes típusok mérete és értéktartománya platform függetlenül definiált. A Java erős korlátozásokkal él a primitív típusokkal szemben. Ilyen típusokhoz tartozó változókat nem hozhatunk létre futási idő alatt.

A java nyelvben nincsenek külön objektumok és mutatók. Az objektumok a dinamikus tárterületen jönnek létre, és csakis hivatkozásokon keresztül érjük el őket, külön mutató vagy referencia szintaxis alkalmazása nélkül. Ugyanígy nincsenek sem függvénymutatók, sem tagfüggvényre, vagy osztály adattagjára mutató pointerek. Ezek helyett a Java-ban objektum-referenciákat, visszatérési értékeket, tömböket, interfészeket használhatunk, vagy ha minden kötél szakad az önelemzp java.lang.reflect csomagot.

A Java-ban nincs lehetőség felhasználói operátorok definiálására. Ami a primitív típusokra vonatkozik, a C++-ból jól ismert és a megfelelő primitív típusokkal manipuláló aritmetikai, logikai, bitmanipuláló, összehasonlító, kiválasztó operátorok mellet a Java az előjel-kiterjesztéses >> mellett ismeri a >>> zéró-kiterjesztéses jobbra léptetést, azaz a balról belépő bit mindig 0. Ellentétes <<< operátor nem létezik.

A const kulcsszó Java-ban ismert, de jelentése nincs, használata hibás. A C++-ban lehetőségünk van egy osztály tagfüggvényét konstansnak deklarálni, kifejezve, hogy az adott függvény konstans objektumra is alkalmazható. Egy adattagot mutable-nek deklarálva jelezhetjük, hogy ő konstans függvényben is módosítható.

A Java-ban ezek a lehetőségek nincsenek. Létezik a final kulcsszó, amely adattagra vonatkozva az adott változó értékének a változatlanságát, metódusra vonatkozva a metódus felülbírálhatatlanságát, osztályra vonatkozva azt jelenti, hogy nem lehet belőle származtatni

A C++-ban egy függvény argumentumának adhatunk alapértelmezett értéket. Ilyen lehetőség a Java-ban nincs.

A Java és a C++ utasításkészlete hasonlít. A deklarációk egy blokkon belül követhetnek nem deklarációs utasítást. Akárcsak C++-ban ezt a lehetőséget Java-ban is kihasználva a lehető legkisebbre szabhatjuk változóink hatókörét

A goto kulcsszó ismert a Java-ban, de nincs jelentése és használata hibás lesz. Azonban goto-t nem sokszor használunk C és C++-ban sem. Cymkék vannak, a Java-ban, de nem a goto utasítás, hanem a continue és break utasítások tárgyát adják meg.

A Java nyelvi szinten megkülönbözteti az osztályokat valamint az interfészeket. Az osztályok körében a Java egyszeres öröklődést támogat. és ha egy osztálynak nincsen a forráskódban megadott szülőosztálya, akkor implicit módon az Object osztály lesz az. Az öröklődést az extends kulcsszó jelzi. A C++-tól eltérően nincs különbség a protected, private és public öröklődés között.

Ezen kívül létezik az abstract kulcsszó is. ha egy osztályt abstract-nak deklarálunk, akkor egyrészt nem példányosítható, másrészt lehetnek benne tisztán virtuális, azaz csak deklarált, de nem implementált metódusok.

A Java a C++-ból ismert háromféle hozzáférési kategória mellett támogatja a külön kulcsszóval nem jelölt félnyílvános hozzáférési kategóriát, amelyik az illető osztályt tartalmazó csomag osztályai számára teszi lehetővé az elérést. A hozzáférési kategóriát a C++-tól eltérően nem címekéhez hasonló módon, hanem az egyes adattagok vagy metódusok minősítéseként adjuk meg.

Az egyes hozzáférési kategóriák az általuk engedett hozzáférés körének szűkülő sorrendjében: public, protected, félnyílvános, private.

A Java-ban nincs barátság, nincs friend deklaráció. Ha mégis szükségünk lenne hasonló funkcióra, azt a Java-ban beágyazott osztályok alkalmazásával érhetjük el. Ez formailag hasonlít a C++-ére, mégis lényegi különbség van a kettő között. A beágyazott Java osztály metódusai ugyanis elérik a beágyazás helyén látható változókat. A C++ beágyazott osztályának nincsen különleges hozzáférési joga a bennfoglaló osztály tagjaihoz, pusztán a névterek kerülnek beágyazásra.

A C nyelvben a struktóra tagjainak összegyűjtését, egy objektumba kapcsolását jelenti. A C++ nyelvben szigorúan véve a struktúra olyan osztályt jelent, amelyben az alapértelmezett elérési kategória publikus mind az attribútumokra, a metódusokra és az öröklődésre. A Java-ban nincsen külön struktúra, helyette osztályt kell alkalmaznunk.

Az 5-ös verziótól kezdve a Java is használhatja a C++ egyik népszerű típuskonstrukcióját, a felsorolást. Első ránézésre a Java nyelvbe épített felsorolás típus hasonlít a C++ megfelelőjére, azonban a Java felsorolástípusa egy teljes osztályt definiál. Tetszőleges új mezőket és metódusokat adhatunk ehhez az osztályhoz.

Bitmezők, uniók megfelelői Java-ban nem léteznek. Unió használata C++-ban két esetben indokolt: amikor helyet szeretnénk megtakarítani, és amikor konverziót szeretnénk elkerülni. Az első esethez kihasználhatjuk, hogy Java-ban minden osztálynak őse az Object osztály.

Java-ban az alapvető objektumokat tartalmazó java.lang csomagban található a beépített String osztály. A String osztály final, azaz nem származtathatunk belőle másik osztályt. így a metódusok felülbírálása révén sem változtathatjuk meg a viselkedését.

Hatókör operátor Java-ban nincs. Az osztály és az interfésznevek minősítőként alkalmazhatóak statikus tagok elérésére, ennek szintaxisa a tagkiválasztó ponton alapul.

Névtérnek nagyjából a Java csomagjai felelnek meg, habár a Java-ban nincsenek globális változók, konstansok vagy függvények, csak osztályokhoz és interfészekhez tartozók. A Java osztályok és interfészek csomagokba vannak rendezve. Egy csomagban további részcsomagok és/vagy osztályok, interfészek lehetnek.

A C++ nyelvhez hasonlóan a Java-ban is van kivételkezelés. A fő elemek Java-ban is a try blokk, ahol a kivétel kiváltását figyeljük, az ezt követő catch blokk(ok), ahol lekezeljük az eldobott kivételeket és a throw kifejezés, ahol a kivételt kiváltjuk. Ezen kívül a Java-ban létezik a finally kifejezés is.

A C++ kivételek tetszőleges típusúak lehetnek, azonban Java-ban az objektum-központú szemlélete miatt nem lehet akármilyen objektumot, mint kivételt dobni, csak Throwable (vagy ettől öröklő) osztályút. Ilyenek például az Error és az Exception osztályok leszármazottjai.

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

A Python egy általános célú programozási nyelv, amit Guido van Rossum alkotott meg 1990-ben. Tulajdonképpen egy szkriptnyelv, viszont rendkívül sok csomagot és beépített eljárást alkalmaz. A sok csomag megléte jó hír, hiszen így nem kell magunktól újra feltalálni a kereket, hanem építkezhetünk a már meglévő tudásra és módszerekre.

A Python egy interpreteres nyelv, ami abban különbözik az eddig tanult fordítóprogramos nyelvektől, hogy a programkódot az interpreter soronként értelmezi, nem pedig egyben az egészet. Ez azért jó, mert ha például van valami hiba a programunkban, akkor letesztelhetjük azt soronként is, így nem kell annyit keresgélnünk és találgatnunk a hiba hollétéről. Viszont a hátránya, hogy gyakran a programok futtatása lassabb, mint a fordítóprogramos nyelveknél.

Bár szerintem első ránézésre nem látszik rajta, a Python egy objektum-orientált nyelv, mint a C++ és a Java. Mind az immár redundáns kapcsos zárójeleket, mind az állítások végén lévő megszokott pontosvesszőket lecserélte négy darab szóközre, illetve kellemes meglepetést okozott nekem a szinte már pszeudokódra emlékeztető szintaxisa. Ennek függvényében nem emlékeztet az eddig tanult C, C++ és a főképp most tanulandó Java nyelvekre.

Habár szerintem nem ez a legfontosabb szempont, amit egy programnyelv megítélésénél figyelembe kell venni, szerintem a Python esztétikailag is megállja a helyét.

A Python segítségével tömör és mégis könnyen olvasható programokat készíthetünk, amelyek tipikusan sokkal rövidebbek , mint a velük ekvivalens C, C++ vagy Java programok. Ennek okai többek között, hogy a magas szintű adattípusok lehetővé teszik , hogy összetett kifejezéseket írjunk le egy rövid állításban, nincs szükség változó és argumentumdefiniálásra, továbbá a kódcsoportosítás egyszerű tagolással (új sor, tabulátor) történik, illetve mint ahogy az fent is írtam, nincs szükség nyitó és záró jelzésekre.
A Pythonban minden adatot objektumok reprezentálnak. Az adatokon végezhető műveleteket az objektumok típusa határozza meg. Pythonban nincs szükség a változók típusainak explicit megadására. A rendszer futási időben, automatikusan kitalálja a változók típusát a hozzárendelt érték alapján. Adattípusok lehetnek a sztringek, számok, ennesek (tuples), listák és a szótárak.

A nyelv támogatja a más nyelvekben megszokott if elágazást, és while ciklusokat. Ezek szintaxisa szinte megegyezik a C-ben lévő megfelelőjükével, a szokásos új sorokkal és tabulátorokkal. Újdonság viszont a for ciklus használata. Már végigmehetünk kulcs-értékpárokon is a for … in segítségével. Így egyetlen ciklusban megkaphatjuk a kulcsokat a hozzájuk tartozó értékkel. Továbbá használhatjuk még a range() függvényt, amely generál nekünk egy egész értékekből álló listát. Ennek egy változattát, az xrange()-et szintén alkalmazhatjuk egy for ciklussal, sőt, kifejezetten ehhez készítették.

A Pythonban függvényeket a def kulcsszóval definiálhatunk. A függvények rendelkeznek a szokásos paraméterekkel, amelyeknek adhatunk alapértelmezett értéket (paraméter = alapérték)

szintaxissal. A függvény hívásakor az egyes argumentumokat meg tudjuk adni a megszokott módon, abban a sorrendben, amelyben megadtuk azt a függvény definiálásakor. Van visszatérési értékük, amelyek lehetnek ennesek is.

A továbbiakban csak a nyelv OOP részéről fogok írni, hiszen a tantárgy fő vonulata – a Java – is egy főleg objektum-orientált nyelv.

A Python nyelv támogatja a klasszikus, objektumorientált fejlesztési eljárásokat. Definiálhatunk osztályokat, amik példányai az objektumok . Az osztálynak lehet attribútumaik,: objektumok, illetve metódusok. A Python támogatja az osztályok közötti öröklődést.

Az osztály metódusait ugyanúgy definiálhatjuk, mint a globális függvényeket (def használatával), azonban van egy kötelező első paraméterük, a self, amely értéke az adott objektumpéldány, amelyen a függvényt meghívják.

Az osztályok konstruktor tulajdonságú metódusa az __init__, amelynek első paraméterek ugyancsak a self, majd további paramétereket is megadhatunk, amelyek mibenléte az osztály attribútumaitól függ.

1. hét - „Helló, Arroway!”

1. hét Az objektumorientált paradigma alapfoglamai. Osztály, objektum, példányosítás.

OO szemlélet

A módosított polártranszformációs normális generátor beprogramozása Java nyelven. Mutassunk rá, hogy a mi természetes saját megoldásunk (az algoritmus egyszerre két normálist állít elő, kell egy példánytag, amely a nem visszaadottat tárolja és egy logikai tag, hogy van-e tárolt vagy futtatni kell az algot.) és az OpenJDK, Oracle JDK-ban a Sun által adott OO szervezés ua.!

Lásd még fóliák!

Ismétlés: https://arato.inf.unideb.hu/batfai.norbert/UDPROG/deprecated/Prog1_5.pdf (16-22 fólia)

Ugyanezt írjuk meg C++ nyelven is! (lásd még UDPROG repó: source/labor/polargen)

Homokózó

Írjuk át az első védési programot (LZW binfa) C++ nyelvről Java nyelvre, ugyanúgy működjön! Mutassunk rá, hogy gyakorlatilag a pointereket és referenciákat kell kiirtani és minden máris működik

---

1 https://github.com/nbatfai/RedFlowerHell

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.

(erre utal a feladat neve, hogy Java-ban minden referencia, nincs választás, hogy mondjuk egy attribútum pointer, referencia vagy tagként tartalmazott legyen).

Miután már áttettük Java nyelvre, tegyük be egy Java Servletbe és a böngészőből GET-es kéréssel (például a böngésző címsorából) kapja meg azt a mintát, amelynek kiszámolja az LZW binfáját![2]

„Gagyi"

Az ismert formális[3] „while (x <= t && x >= t && t != x);" 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 forrására[4], hogy a 128-nál inkluzív objektum példányokat poolozza!

Yoda

Írjunk olyan Java programot, ami java.lang.NullPointerEx-el leáll, ha nem követjük a Yoda conditions-t! https://en.wikipedia.org/wiki/Yoda_conditions

A kifejezések szokásos megadásánál a változó tagok mindig a jobb oldalon vannak, azaz:

```
if (value == 42) { /* .. */ }
```

A Yoda-condition, vagy Yoda-notation egy olyan programozási stílus, ahol a kifejezések két része a szoksásosnál fordítva van megadva, tehát a változó a jobb oldalon van.

```
if (42 == value) { /* .. */ }
```

A NullPointerException-t akkor dobja fel nekünk a program, ha nullpointerre hivatkozunk. Az alábbi program java.lang.NullPointerException-rel leáll. Ez azért következik be, mert a myString értékének egy nullpointert állítunk be. Így bár létrehoztunk egy változót, egy objektumot nem. Amikor az equals() metódus megpróbálja összehasonlítani a két Stringet, megpróbál átlépkedni a még nem létező objektumon, azonban ez nem lehetséges.

```
public class NullExceptionProgram {
	public static void main(String []args) {
		String myString = null;
		if (!(myString.equals("foobar"))) {
			System.out.println("It works");
		}
	}
}
```

Ezt úgy tehetjük működőképessé, ha a Yoda-conditions segítségével megfordítva a problémás kódrészt, nem egy nem létező Stringre hivatkozunk, és így elkerüljük a NullPointer Exception-t.

```
public class NullExceptionProgram {
	public static void main(String []args) {
		String myString = null;
		if (!("foobar".equals(myString))){
			System.out.println("It works");
		}
	}
}
```

Kódolás from scratch

Induljunk ki ebből a tudományos közleményből: http://crd-legacy.lbl.gov/~dhbailey/dhbpapers/bbp-alg.pdf és csak ezt tanulmányozva írjuk meg Java nyelven a BBP algoritmus megvalósítását!

Ha megakadsz, de csak végső esetben: https://www.tankonyvtar.hu/hu/tartalom/tkt/javat-tanitok-javat/apbs02.html#pi_jegyei (mert ha csak lemásolod, akkor pont az a fejlesztői élmény marad ki, melyet szeretném, ha átélnél).

EPAM: Java Object metódusok

Mutasd be a Java Object metódusait és mutass rá mely metódusokat érdemes egy saját osztályunkban felüldefiniálni és miért. (Lásd még Object class forráskódja)

```
public final Class getClass()
```

A getClass() függvény visszatéríti az adott objektum futási osztályát. Például Integer osztály esetén az eredmény:

```
class java.lang.Integer
```

```
public int hashCode()
```

A hashCode() egy hashcode-ot térít vissza. Ezt a függvényt érdemes átírni, hiszen így megszabhatjuk mi magunk, hogy hogyan számolódjon ki a hashcode. Ezt a metódust főleg az equals() metódussal együtt láttam használni.

```
public boolean equals(Object obj)
```

Az equals() metódus megnézi hogy az adott objektumunk megegyezik-e a paraméterként megadott objektumunkkal. Ez a metódus tranzitív, konzisztens, szimmetrikus, reflexív és minden x nem nulla referenciaértékre x.equals(null) visszatérési értéke false. Ezt is érdemes felülírni a hashCode()-dal együtt.

```
protected Object clone() throws CloneNotSupportedException
```

A clone() metódus az objektum teljes másolatát téríti vissza. A másolat precíz jelentése az adott objektumtól függ. Természetesen másolásra jó. Általánosan a következőek igazak:

```
x.clone() != x;
x.clone().getClass() == x.getClass();
x.clone().equals(x);
```

Ezek közül az utolsó nem feltétlenül igaz.

```
public String toString()
```

A toString() metódust is érdemes átírni. Segítségével könnyebben olvasható formátumban jeleníthetünk meg egy objektumot. Szerintem ezt is érdemes átírni, hogy egy általunk könnyen olvasható formában jelenítse meg az objektumot.

```
public final void notify()
```

Felébreszt egy szálat, amely az objektum monitorára vár. Ha bármilyen szál vár erre az objektumra, az egyik kiválasztódik, és az felébred. Ez a választás véletlenszerű. Egyszerre egy objektum monitorát csak egy szál birtokolhatja.

```
public final void notifyAll()
```

Felébreszti az összes objektum monitorára várakozó objektumot. Ezen kívül ugyanúgy működik, mint a notify(). A jelenlegi szálnak birtokolnia kell az objektum monitorát.

```
public final void wait(long timeout)throws InterruptedException
```

A jelenlegi szálat várakozásra készteti addig, amíg egy másik szál vagy a notify(), vagy a notifyAll() metódust meghívja, vagy egy bizonyos idő eltelik. Ezt az időt megadhatjuk a timeout segítségével.

```
public final void wait(long timeout,int nanos)throws
InterruptedException
```

Ugyanúgy működik, mint a fenti wait(), csak megadhatjuk neki, mennyi legyen az az eltelt idő nanoszekundumokban, miután felébred a szál. Ezzel sokkal precízebben irányíthatjuk a folyamatot.

```
public final void wait()throws InterruptedException
```

Ez a metódus a wait(0)-val egyezik meg.

```
protected void finalize()throws Throwable
```

Ezt a metódust az objektumon a szemétgyűjtő (Garbage Collector) hívja meg, amikor úgy állapítja meg, hogy nincs több hivatkozás az adott objektumra. Ezt az alosztály felül szokta írni, hogy megfelelően felszabadítsa a foglalt helyet, vagy egyéb takarítást végezzen.

EPAM: Eljárásorientál vs Objektumorientált

Írj egy 1 oldalas értekező esszé szöveget, amiben összehasonlítod az eljárásorientált és az objektumorientált paradigmát, igyekezve kiemelni az objektumorientált paradigma előnyeit!

Az objektumorientált programozás osztályokat és objektumokat használ arra, hogy modellezze a világot. Egy objektumorientált paradigmával íródott programban az objektumok meghíváskor üzeneteket küldenek, melyek különböző szolgáltatásokat és információkat kérnek le. Az objektumok képesek üzenetek küldésére, foglalására és az információ adatformátumú továbbítására.

Az objektum-orientált programozás előnye, hogy könnyebb a már meglévő kódot karbantartani és módosítani., mivel az újonnan készült objektumok képesek viselkedést örökölni a már meglévő objektumoktól. Ez leszűkíti a fejlesztési időt és könnyebbé teszi a program módosítását.

Ezzel szemben az eljárás-orientált programozás egy úgynevezett felülről lefelé stratégiát alkalmaz. Míg az objektumorientált programozás objektumokat és osztályokat használ, addig az eljárás-orientált paradigma esetén veszünk egy problémát, amit kisebb és kisebb problémákra bontunk, amelyekért egy-egy eljárás fog felelni.

Ezzel a módszerrel az a probléma, hogy ha valamit átszerkesztünk, akkor az összes olyan sor kódot is át kell szerkeszteni, amely hivatkozik az átszerkesztett kódrészletre.

Terminológia különbségek, hogy az eljárás-orientált paradigma eljárásainak (procedure) és függvényeinek (function) az objektum-orientált megfelelőit metódusoknak (method) nevezzük. Ugyanígy az eljárás-orientált paradigma adatszerkezeteire úgy szoktak hivatkozni, hogy rekordok (record). Ezeket OOP-ben objektumoknak (object) nevezzük.

A három lényeges különbség az eljárás- és objektum-orientált programozási paradigmák között, az öröklődés (inheritance), polimorfizmus és az beágyazás (encapsulation) jelenléte OOP-ben.

Az beágyazás adja magukat az osztályokat. A lényege, hogy az adatokat és a rájuk vonatkozó metódusokat tegyük egy helyre. OOP esetén ez a hely egy osztály.

Az öröklődés megkönnyíti a kódrészletek újrafelhasználását. Újonnan készített objektumok képesek a szülő objektumok tulajdonságait „örökölni”. Egy alosztály képes a szülő osztálya egyik metódusát felülírni. Egyszerre több osztályból is történhet öröklődés.

A polimorfizmus azt a koncepciót írja le, hogy különböző típusú objektumok elérhetőek ugyanazon az interfészen keresztül.

Az öröklődés, polimorfizmus és enkapszuláció szorosan kötődnek az objektumokhoz. Mivel az eljárás-orientált nyelvekben nincsenek objektumok, így ott ezekről a funkciókról sem beszélhetünk.

EPAM: Objektum példányosítás programozási mintákkal

Hozz példát mindegyik “creational design pattern”-re és mutasd be mikor érdemes használni őket!

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.

2. hét - „Helló, Liskov!”

2. hét Öröklődés, osztályhierarchia. Polimorfizmus, metódustúlterhelés. Hatáskörkezelés. A bezárási eszközrendszer, láthatósági szintek. Absztrakt osztályok és interfészek.

Liskov helyettesítés sértése

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

OOP programozás során a Liskov helyettesítési elv azt mondja, hogy ha S T-nek egy altípusa, akkor a T típusú objektumokat lecserélhetjük bármely S típusú objektumra anélkül, hogy a program kívánt tulajdonságait elvennénk. Az alábbi C++ kódcsipet megsérti a Liskov elvet.

```
// ez a T az LSP-ben
class Madar {
public:
     virtual void repul() {};
};

// ez a két osztály alkotja a "P programot" az LPS-ben
class Program {
public:
     void fgv ( Madar &madar ) {
          madar.repul();
     }
};

// itt jönnek az LSP-s S osztályok
class Sas : public Madar
{};

class Pingvin : public Madar // ezt úgy is lehet/kell olvasni,
                             //hogy a pingvin tud repülni
{};

int main ( int argc, char **argv )
{
     Program program;
     Madar madar;
     program.fgv ( madar );

     Sas sas;
     program.fgv ( sas );

     Pingvin pingvin;
     program.fgv ( pingvin ); // sérül az LSP, mert a P::fgv
                              // röptetné a Pingvint, ami ugye
                              // lehetetlen.

}
```

Ez a kódcsipet azért sérti a Liskov elvet, mert a programnak az a tulajdonsága, hogy helyes legyen sérül, amikor helyettesíteni akarjuk a Pingvin osztályunkat a Madar osztállyal, hiszen a Pingvinek nem tudnak repülni, viszont minden madárnak adtunk egy repül függvényt.

Annak érdekében, hogy ezt a sértést elkerüljük, egyszerűen jobban át kell gondolnunk a programunk tervezését, és ahelyett, hogy csak egy alap madár osztályt vezetünk be, írunk annak egy repülő madár alosztályát. Ezután megadjuk a Sas-t a repülő madár alosztályának, a Pingvint pedig az alap Madár-énak.

Ez a kódban a következőképpen valósul meg.

```
// ez a T az LSP-ben
class Madar {};

// ez a két osztály alkotja a "P programot" az LPS-ben
class Program {
public:
     void fgv ( Madar &madar ) {
          // madar.repul(); a madár már nem tud repülni
          // s hiába lesz a leszármazott típusoknak
          // repül metódusa, azt a Madar& madar-ra
          // úgysem lehet hívni
     }
};

// itt jönnek az LSP-s S osztályok
class RepuloMadar : public Madar {
public:
     virtual void repul() {};
};

class Sas : public RepuloMadar
{};

class Pingvin : public Madar // ezt úgy is lehet/kell olvasni,
hogy a pingvin tud repülni
{};

int main ( int argc, char **argv )
{
     Program program;
     Madar madar;
     program.fgv ( madar );

     Sas sas;
     program.fgv ( sas );

     Pingvin pingvin;
     program.fgv ( pingvin );

}
```

Ezeknek a programoknak Java-ba átíráskor csak az objektumok inicializálását és az alosztályok definiálását kell átírni a new és az extends kulcsszavak segytségével, illetve a megszokott main() fügvénnyel. A kódcsipet:

```
// ez a T az LSP-ben
class Madar {
      public void repul() {};
};

// ez a két osztály alkotja a "P programot" az LPS-ben
class Program {
      public void fgv ( Madar madar ) {
          madar.repul();
     }
};

// itt jönnek az LSP-s S osztályok
class Sas extends Madar
{};

class Pingvin extends Madar // ezt úgy is lehet/kell olvasni,
                            //hogy a pingvin tud repülni
{};

public class Main{

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

        Sas sas = new Sas();
        program.fgv ( sas );

        Pingvin pingvin = new Pingvin();
        program.fgv ( pingvin ); // sérül az LSP, mert a P::fgv
                                 //  röptetné  a  Pingvint,  ami
ugye
                                     // lehetetlen.
     }
}
```

Ezek amik sértik a Liskov elvet. A sértést elkerülő változat Java-ban ugyanúgy valósul meg, mint C++-ban az előbb említett változtatásokkal.

```
// ez a T az LSP-ben
class Madar {};

class RepuloMadar extends Madar {
    public void repul() {};
}

// ez a két osztály alkotja a "P programot" az LPS-ben
class Program {
    public void fgv ( Madar madar ) {}
};

// itt jönnek az LSP-s S osztályok
class Sas extends RepuloMadar {};

class Pingvin extends Madar {};
```

```
public class Main{

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

        Sas sas = new Sas();
        program.fgv ( sas );

        Pingvin pingvin = new Pingvin();
        program.fgv ( pingvin ); // sérül az LSP, mert a P::fgv
                                 //  röptetné  a  Pingvint,  ami
ugye
                                     // lehetetlen.
     }
}
```

https://arato.inf.unideb.hu/batfai.norbert/PROG2/Prog2_1.pdf (93-99 fólia)
(számos példa szerepel az elv megsértésére az UDPROG repóban, lásd pl. source/binom/Batfai-Barki/madarak/)

Szülő-gyerek

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

Lásd fóliák! https://arato.inf.unideb.hu/batfai.norbert/PROG2/Prog2_1.pdf (98. fólia)

Anti OO

A BBP algoritmussal[5] a Pi hexadecimális kifejtésének a 0. pozíciótól számított 10^6, 10^7, 10^8 darab jegyét határozzuk meg C, C++, Java és C# nyelveken és vessük össze a futási időket!
https://www.tankonyvtar.hu/hu/tartalom/tkt/javat-tanitok-javat/apas03.html#id561066

deprecated - Hello, Android!

Élesszük fel a https://github.com/nbatfai/SamuEntropy/tree/master/cs projektjeit és vessünk össze néhány egymásra következőt, hogy hogyan változtak a források!

Hello, Android!

Élesszük fel az SMNIST for Humans projektet!
https://gitlab.com/nbatfai/smnist/tree/master/forHumans/SMNISTforHumansExp3/app/src/main
Apró módosításokat eszközölj benne, pl. színvilág.

Hello, SMNIST for Humans!

Fejleszd tovább az SMNIST for Humans projektet SMNIST for Anyone emberre szánt appá!
Lásd az smnist2_kutatasi_jegyzokonyv.pdf-ben a részletesebb hátteret!

Ciklomatikus komplexitás

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

[5]https://www.tankonyvtar.hu/hu/tartalom/tkt/javat-tanitok-javat/apbs02.html#pi_jegyei

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.

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?

A Java 8 interface változtatások részei, hogy Java 8-ban már lehet interface-nek statikus és default metódusokat megadni. 8 előtt csak metódus deklarációkat adhattunk interface-eknek.

Java-ban a default metódusok készítése a default kulcsszóval történik.

```
package com.journaldev.java8.defaultmethod;

public interface Interface1 {

	void method1(String str);

	default void log(String str){
		System.out.println("I1 logging::"+str);
	}
}
```

A fenti kódban a default metódus Interface1-ben a log(String str). Így, ha egy osztály implementálja ezt az interfészt, muszáj annak az interfésznek a default metódusaihoz implementációt biztosítani. Ez segíti azt, hogy az interfészeket további metódusokkal lássuk el.

Tegyük fel, hogy van még egy interfészünk:

```
package com.journaldev.java8.defaultmethod;

public interface Interface2 {

	void method2();

	default void log(String str){
		System.out.println("I2 logging::"+str);
	}

}
```

ilyenkor felmerül az úgynevezett „diamond problem”, aminek a lényege, hogy ha van egy osztályunk, ami implementálná a fenti két interfészt, de nem implementálna egy default metódust, akkor a program nem tudná eldönteni, hogy melyik interfész default metódusát használja.

Most már, ha egy osztály mindkét interfész implementálni akarja, akkor kötelező implementálnia a fdefault metódusokat is. Pl.:

```
package com.journaldev.java8.defaultmethod;

public class MyClass implements Interface1, Interface2 {

	@Override
	public void method2() {
	}

	@Override
	public void method1(String str) {
	}

	@Override
	public void log(String str){
		System.out.println("MyClass logging::"+str);
		Interface1.print("abc");
	}
}
```

A statikus metódusok hasonlóak a default-okhoz, azzal a különbséggel, hogy nem tudjuk őket felülírni. Ez segít elkerülni nem kívánt eredményeket nem elégséges implementációk esetén.

EPAM: Liskov féle helyettesíthetőség elve, öröklődés

Adott az alábbi osztály hierarchia.
`class Vehicle, class Car extends Vehicle, class Supercar extends Car`
Mindegyik osztály konstruktorában történik egy kiíratás, valamint a `Vehicle` osztályban szereplő `start()` metódus mindegyik alosztályban felül van definiálva.
Mi történik ezen kódok futtatása esetén, és miért?

```
Vehicle firstVehicle = new Supercar();
firstVehicle.start();
System.out.println(firstVehicle instanceof Car);
Car secondVehicle = (Car) firstVehicle;
secondVehicle.start();
System.out.println(secondVehicle instanceof Supercar);
Supercar thirdVehicle = new Vehicle();
thirdVehicle.start();
```

```
Vehicle firstVehicle = new Supercar();
firstVehicle.start();
System.out.println(firstVehicle instanceof Car);
```

A fenti kód esetén a következő lesz kiírva:

```
Creating vehicle!
Creating car!
Creating supercar!
Supercar is starting!
true
```

Ez azért van, hiszen a Supercar a Car alosztálya, ami a Vehicle alosztálya. Supercar nem létezhet Car osztály nélkül, és az sem Vehicle nélkül. Miután elkészült a Supercar objektum, elindítjuk a

start függvényét, és a várt eredményt adja vissza nekünk. A printlnban megvizsgáljuk, hogy a firstVehicle Car-nak egy instance-a, azaz a Supercar objektum egy Car objektum is egyben.

```
Car secondVehicle = (Car) firstVehicle;
secondVehicle.start();
System.out.println(secondVehicle instanceof Supercar);
```

A fentire a következőt kapjuk:

Supercar is starting!
true

Látható hogy itt már nem történik új objektum létrehozása, hiszen csak castoljuk, és az eredményt megadjuk a secodVehicle-nek. Azonban az eredmény marad a Supercar is starting, hiszen ahogy a következő true eredmény is mutatja, a secondVehicle instance-a a Supercar-nak. Ez azért történt, mert természetesen a Supercar is egy Car, tehát Supercar Car-á castolása során valójában nem szükséges semmit sem csinálnunk.

```
Supercar thirdVehicle = new Vehicle();
thirdVehicle.start();
```

Ez a rész szintaktikai hibát okoz, és nem engedi lefordulni a programot. Ezt a hibaüzenetet kapjuk.

```
error: incompatible types: Vehicle cannot be converted to Supercar
```

A probléma a Vehicle és a Supercar osztályok egymás közötti kapcsolatából függ. A Supercar a Vehicle alosztálya, tehát amíg minden Supercar Vehicle is egyben, addig nem minden Veghicle Supercar. Most a kódrészlet egy Supercar objektumot próbál létrehozni a Vehicle() függvénnyel.

A helyes kódrészlet ez lenne:

```
Supercar thirdVehicle = new Supercar();
```

EPAM: Interfész, Osztály, Absztrak Osztály

Mi a különbség Java-ban a Class, Abstract Class és az Interface között? Egy tetszőleges példával / példa kódon keresztül mutasd be őket és hogy mikor melyik koncepciót célszerű használni.

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.

3. hét - „Helló, Mandelbrot!”

3. hét Modellező eszközök és nyelvek. AZ UML és az UML osztálydiagramja.

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.

Lásd fóliák!

Forward engineering UML osztálydiagram

UML-ben tervezzünk osztályokat és generáljunk belőle forrást!

Egy esettan

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

BPMN

Rajzoljunk le egy tevékenységet BPMN-ben!
https://arato.inf.unideb.hu/batfai.norbert/PROG2/Prog2_7.pdf (34-47 fólia)

BPEL Helló, Világ! - egy visszhang folyamat

Egy visszhang folyamat megvalósítása az alábbi teljes „videó tutoriál” alapján:
https://youtu.be/0OnlYWX2v_I

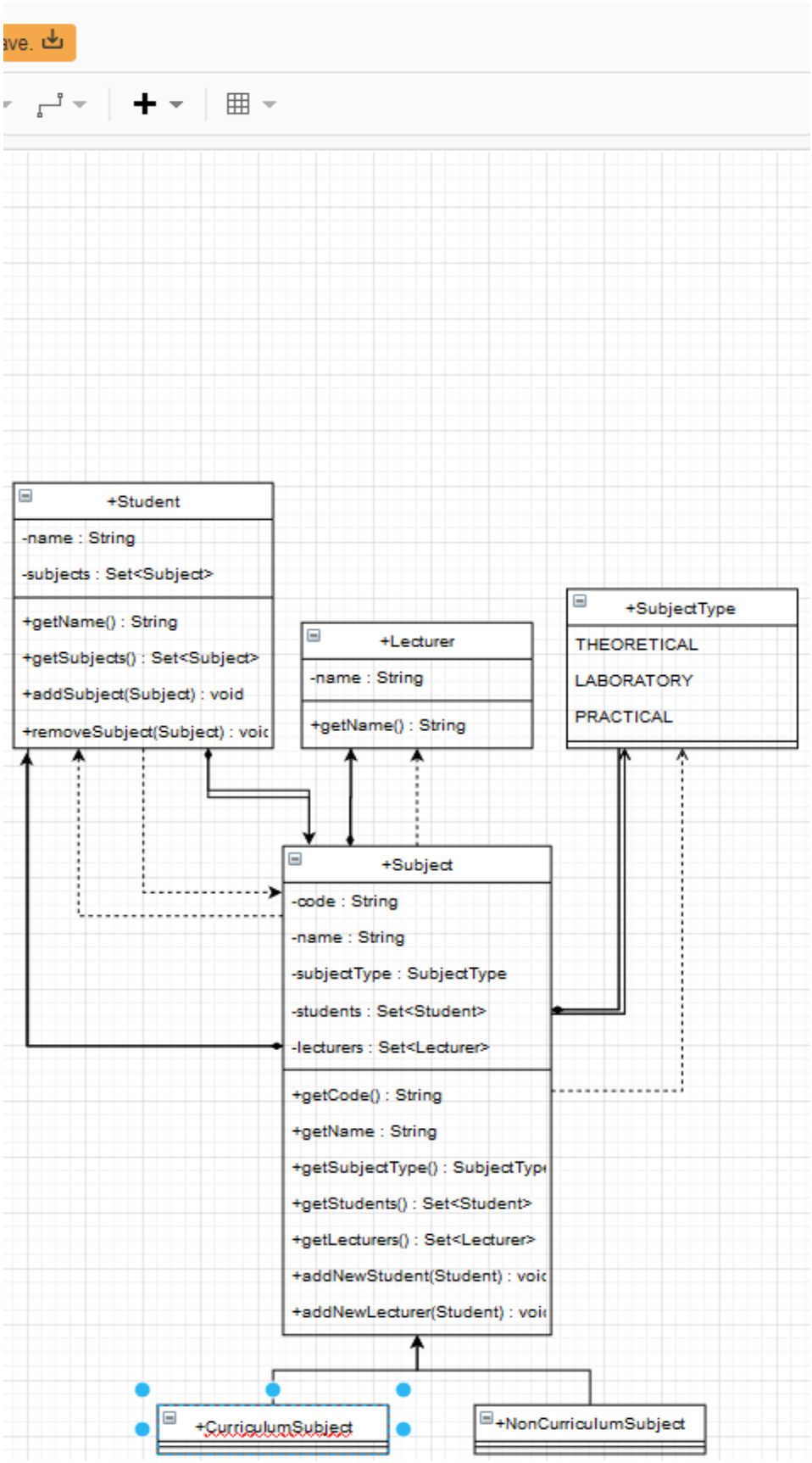
TeX UML

Valamilyen TeX-es csomag felhasználásával készíts szép diagramokat az OOCWC projektről (pl. use case és class diagramokat).

EPAM: Neptun tantárgyfelvétel modellezése UML-ben

Modellezd le a Neptun rendszer tárgyfelvételéhez szükséges objektumokat UML diagramm segítségével.

A felvétel UML diagramja:



A modellben összesen 6 darab osztály van.

A diagrammon minden osztály és függvény publikus, és minden változó privát.

Ezek közül a legegyszerűbbek a CurriculumSubject és a NonCurriculumSubject publikus osztályok, azaz a kötelező és nem kötelező tárgyak. Mindketten a Subject (tantárgy) alosztályai.

Minden tantárgynak van privát kódja, neve, típusa, illetve hozzá tartozó hallgatók és oktatók. Ezek közül a kód és név karakterláncok, a students és lecturers pedig Student (hallgató) és Lecturer (oktató) objektumokból álló halmazok. Természetesen mindegyik értékhez tartozik egy szokásos get metódus az érték eléréséhez. Ezen kívül itt található még az új hallgatók és oktatók hozzáadásáért felelős függvények is.

A SubjectType egy külön osztályban van definiálva. Ez egy enum típus ami lehet elméleti, labor vagy gyakorlati. Ez adja a Subject subjectType nevű változójának a typusát.

Ezen kívül még a Student és Lecturer osztályok vannak. Az oktatónak neve van, amit meg le lehet kérni, a hallgatónak viszont neven kívül van felvett tárgyai is. Mindkettőt le lehet kérni, illetve a tárgyakhoz hozzá lehet adni, illetve lehet belőlük elvonni.

EPAM: Neptun tantárgyfelvétel UML diagram implementálása

Implementáld le az előző feladatban létrehozott diagrammot egy tetszőleges nyelven.

A fenti diagramnak a Java-ban való implementációja:

A Lecturer osztály:

```
package com.epam.training.lecturer;

public class Lecturer {
    private String name;

    public String getName() {
        return name;
    }
}
```

Ez egy egyszerű osztály. Áll egy név változóból és a hozzátartozó getterből.

A Student osztály:

```
package com.epam.training.student;

import com.epam.training.subject.Subject;
import java.util.Set;

public class Student {

    private String name;
    private Set<Subject> subjects;
```

```
    public String getName() {
        return name;
    }

    public Set<Subject> getSubjects() {
        return subjects;
    }

    public void addSubject(Subject subject) {
        this.subjects.add(subject);
    }

    public void removeSubject(Subject subject) {
        this.subjects.remove(subject);
    }

}
```

Ez az osztály összeköttetésben áll a Subject-tel, hiszen a subjects változó Subject objektumok halmazából áll.

A SubjectType osztály

```
package com.epam.training.subject;

public enum SubjectType {

    THEORETICAL,
    PRACTICAL,
    LABORATORY

}
```

Egy egyszerű enum osztály benne három értékkel.

A Subject osztály:

```
package com.epam.training.subject;

import com.epam.training.lecturer.Lecturer;
import com.epam.training.student.Student;
import java.util.Set;

public class Subject {

    private String code;
    private String name;
    private SubjectType subjectType;
    private Set<Student> students;
    private Set<Lecturer> lecturers;

    public String getCode() {
```

```
        return code;
    }

    public String getName() {
        return name;
    }

    public SubjectType getSubjectType() {
        return subjectType;
    }

    public Set<Student> getStudents() {
        return students;
    }

    public Set<Lecturer> getLecturers() {
        return lecturers;
    }

    public void addNewStudent(Student student) {
        this.students.add(student);
    }

    public void addNewLecturer(Lecturer lecturer) {
        this.lecturers.add(lecturer);
    }

}
```

Ez a legnagyobb osztály. illetve ez áll a modell középpontjában, hiszen minden másik osztállyal közvetlenül összeköttetésben áll.

A CurriculumSubject és NonCurriculumSubject osztályok:

```
package com.epam.training.subject;

public class CurriculumSubject extends Subject {
}

package com.epam.training.subject;

public class NonCurriculumSubject extends Subject {
}
```

Azon kívül, hogy a Subject alosztályai, nincs bennük semmi különös. Csupán a megkülönböztetés miatt léteznek.

EPAM: OO modellezés

Írj egy 1 oldalas esszét arról, hogy OO modellezés során milyen elveket tudsz követni (pl.: SOLID, KISS, DRY, YAGNI).

OO modellezéskor számos elvet tudunk követni. Ezen elvek ismerete fontos, hiszen segítenek minket a hatékonyabb munkában, illetve gyakori hibák elkerülésében. Ezek közül a KISS, SOLID, DRY és a YAGNI elvekről fogok itt írni.

A KISS (keep it simple, stupid vagy keep it stupid simple, magyar nagyjából: maradj az egyszerűségnél) a nevéhez híven a következőt állítja: A legtöbb rendszer akkor működik a legjobban, ha egyszerűek, nem lesznek túlbonyolítva, tehát az OO modellezés során az egyik legfontosabb tervezési cél az egyszerűség. A szükségtelen bonyolítástól pedig el kell tekintenünk.

A DRY (Don't Repeat Yourself – Ne ismételd magad) szerint az ismétléseket érdemes elkerülni, , tehát ha azon vesszük magunkat, hogy valamit már sokadjára leírtunk ugyanúgy, vagy kicsit másképpen, akkor valószínűleg érdemes lenne elgondolkodni egy lehetségéges absztrakción. A

DRY szerint minden információnak egyetlen, félreérthetetlen reprezentációjának kell lennie a rendszeren belül. Ezt az elvet használhatjuk többek között adatbázisoknál, tesztelésnél, sőt még a dokumentáció során is.

Ha sikeresen alkalmazzuk a DRY elvet, akkor egy elem megváltoztatásával nem fog más logikailag nem kapcsolódó részekben változás járni. A logikailag kapcsolódó részekben viszont ez a változás is logikus és kiszámítható lesz.

A YAGNI (You aren't gonna need it – Nem lesz rá szükséged) szerinte a programozó ne adjon olyan funkcionalitást a programhoz, amire még nincs szükség. Ezeket a funkcionalitásokat csak akkor kell implementálnunk, amikor tényleg szükségünk van rá, és nem akkor amikor úgy véljük, hogy előreláthatólag szükségünk lesz rá.

Végezetül mpedig a SOLID.A SOLID minden betűje egy elv kezdőbetűje.

Az első ilyen elv a Single Responsibility. A Single Responsibility szerint minden osztálynak csakis egy dologért szabad felelnie. Tehát egy osztályt csak egy cél érdekében szabad írnunk. Ha valami mást is szeretnénk csinálni, akkor azt tegyük meg egy másik osztállyal. Ha követjük ezt az elvet akkor minden osztály, amit írunk egyszerű, tömör és átlátható lesz.

A második elv az Open Closed. Az osztályokat mindig legyen nyílt a kibővítésre, de zárt a módosításra. Tehát, úgy tervezzük meg az osztályokat, hogy ha egy másik fejlesztő valamin változtatni szeretne, akkor elegendő legyen azt az osztályt kiegészíteni, és ne keljen semmin módosítani.

A harmadik a Liskov's Substitution. Eszerint a származtatott típusokat lehessen az alaptípusokkal helyettesíteni. Ennek az a lényege, hogy a más fejlesztők által készített alosztályok illeszkedjenek be tökéletesen az alkalmazásba. Ennek az a kötöttsége, hogy az alosztályok objektumainak ugyanúgy kell viselkedniük, mint a szülő osztály objektumainak.

A negyedik az Interface Segregation. Nem szabad arra kényszeríteni a klienseket, hogy olyan metódusokat implementáljanak, amelyeket nem fognak használni. Azaz, ha például van egy interfészünk, amelynek van két metódusa, amelyek közül a kliens csak az egyiket szeretné használni, akkor hiába csak az egyiket szeretné használni, akkor is mind a kettő metódust implementálnia kell. Ekkor a megoldás, hogy a létező interfészt osszuk meg.

Az ötödik pedig a Dependency Inversion. Tervezzünk olyan szoftvert, hogy a különböző modulok legyenek egymástól könnyen elválaszthatók és használjuk egy absztrakt réteget, amely össze köti őket.

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.

4. hét - „Helló, Chomsky!”

4. hét Objektumorientált programozási nyelvek programnyelvi elemei: karakterkészlet, lexikális egységek, kifejezések, utasítások.

Encoding

Fordítsuk le és futtassuk a Javat tanítok könyv MandelbrotHalmazNagyító.java forrását úgy, hogy a fájl nevekben és a forrásokban is meghagyjuk az ékezetes betűket!
https://www.tankonyvtar.hu/hu/tartalom/tkt/javat-tanitok-javat/adatok.html

OOCWC lexer

Izzítsuk be az OOCWC-t és vázoljuk a https://github.com/nbatfai/robocar-emulator/blob/master/justine/rcemu/src/carlexer.ll lexert és kapcsolását a programunk OO struktúrájába!

l334d1c4[6]

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!)

Full screen

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

Paszigráfia Rapszódia OpenGL full screen vizualizáció

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

Paszigráfia Rapszódia LuaLaTeX vizualizáció

Lásd vis_prel_para.pdf! Apró módosításokat eszközölj benne, pl. színvilág, még erősebb 3D-s hatás.

Perceptron osztály

Dolgozzuk be egy külön projektbe a projekt Perceptron osztályát!
Lásd https://youtu.be/XpBnR31BRJY

EPAM: Order of everything

Collection-ok rendezése esetén jellemzően futási időben derül ki, ha olyan típusú objektumokat próbálunk rendezni, amelyeken az összehasonlítás nem értelmezett (azaz T típus esetén nem implementálják a `Comparable<T>` interface-t). Pl. ClassCastException a Collections.sort() esetében, vagy ClassCastException a Stream.sorted() esetében.

---

[6] Lásd a C+lex megoldásom itt:
https://www.facebook.com/groups/udprog/permalink/942314465956443/?comment_id=942571282597428&comment_tracking=%7B%22tn%22%3A%22R3%22%7D

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.

Írj olyan metódust, amely tetszőleges Collection esetén vissza adja az elemeket egy List-ben növekvően rendezve, amennyiben az elemek összehasonlíthatóak velük azonos típusú objektumokkal. Ha ez a feltétel nem teljesül, az eredményezzen syntax error-t. Például:
`List<Integer> actualOutput = createOrderedList(input);`
Ahol az `input Collection<Integer>` típusú. Természetesen más típusokkal is működnie kell,
feltéve, hogy implementálják a Comparable interface-t.

```
public class OrderOfEverythingTest {

     @ParameterizedTest
     @MethodSource("collectionsToSortDataProvider")
     public void
testOrderShouldReturnExpectedListWhenCollectionIsPassed(Collection
<Integer> input, List<Integer> expectedOutput) {
        // Given as parameters

        // When
        // createOrderedList(List.of(new
OrderOfEverythingTest()));
        List<Integer> actualOutput = createOrderedList(input);

        // Then
        assertThat(actualOutput, equalTo(expectedOutput));
    }

    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());
    }
}
```

A createOrderedList egy olyan metódus, amely egy olyan listával tér vissza, amelynek elemeit össze lehet hasonlítani. Ezt a tulajdonságot a <T extends Comparable<T>>-vel lehet jelezni.
A megkapott inputot rendezi, majd összegyűjti egy listába.

A collectionsToSortDataProvider egy factory metódus, amely listák és setekből álló Stream objektumokat térít vissza. Ezeket a return értékeket a JUnit fel tudja használni inputként a teszteléshez a @MethodSource jelölés segítségével.

A teszt lényegi részében, az első függvényben pedig megtörténik maga az összehasonlítás. A teszt sikeres lesz, hiszen a createOrderedList csak olyan listákkal fog visszatérni, amelyek elemei összehasonlíthatóak egymással, és most pontosasn ezt vizsgáljuk.

Viszont a CreateIrderedList(List.of(new OrderOfEverythingTest)) már syntax error-t fog nekünk dobni, mert az OrderOfEverythingTest nem implementálja a Comparable<OrderOfEverythingTest>-et, azaz az elemei nem feltétlenül hasonlíthatóak össze.

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) struktúrával vissza tud térni (ehhez egy buborék rendezőt implementálj)

Kezdetnek van az IntegerCollection osztályunk:

```
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;
     }

     // ...

}
```

Amint láthatjuk, a típus igazából egy tömbből áll. Továbbá szükségünk van még egy indexre, amely alapból 0, azaz az első elemre fog mutatni, egy méretre és egy sorted booleanra, amely azt mondja meg nekünk, hogy rendezve van-e az objektumunk.

A setterek a megszokottak. Nincs bennük semmi különös.

Ami a feladat szempontjából lényeges, azok a contains() és a sort() algoritmusok

```
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;
}
```

Ez a metódus a bináris keresés implementációja. A bináris keresés lényege, egy rendezett számsor esetén meghatározzuk a sor középső számát. Ha ez a szám nem a keresett értékünk, attól függően, hogy a keresett érték nagyobb, vagy kisebb, eltoljuk a keresési intervallum jobb, illetve bal oldalát középre. Ha ez megvan, meghatározzuk az új bal vagy jobb érték segítségével az új közepet és folytatjuk így tovább a folyamatot amíg meg nem találjuk a keresett éltéket.

```
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;
}
```

A fenti kódrész pedig a buborékrendezés implementációja. A buborékrendezés valószínüleg a legegyszerűbb rendezési módszer, ugyanakkor elég lassú. Lényege, hogy végighaladva a számsoron összehasonlítunk minden szomszédos értéket egymással, és ha a bal oldali nagyobb a jobb oldalinál, megcseréljük őket.

Egy kicsit javítani lehet az alap módszeren azzal, hogy az átnézendő intervallumot mindig csökkentjük egyel, hiszen a egy sor végigiterálása után mindenképpen helyére kerül legalább egy érték, amely a legnagyobb azok közül, amelyek még nem voltak a helyükön.

EPAM: Saját HashMap implementáció

Írj egy saját java.util.Map implementációt, mely nem használja a Java Collection API-t.

Az implementáció meg kell feleljen az összes megadott unit tesztnek, nem kell tudjon kezelni null értékű kulcsokat és a “keySet”, “values”, “entrySet” metódusok nem kell támogassák az elem törlést.

Plusz feladatok:

1. az implementáció támogat null kulcsokat,

a “keySet”, “values”, “entrySet” metódusok támogatják az elem törlést.

A map Java-ban egy olyan objektum amely két tömbből áll. Ezek a kulcsok és értékek. Minden kulcshoz hozzárendel egy értéket. Egy kulcshoz maximum egy érték tartozhat

```
public class ArrayMap<K, V> implements Map<K, V> {

  private static final int INITIAL_SIZE = 16;

  private int size = 0;
  private K[] keys = (K[]) new Object[INITIAL_SIZE];
  private V[] values = (V[]) new Object[INITIAL_SIZE];

  // ...

}
```

A map olyan metódusokat tartalmaz, mint a put, get, remove, containsKey, putAll, clear, keySet, entrySet, és a values

A size() egy egyszerű metódus, amely visszaadja az ArrayMap méretét.

```
@Override
  public int size() {
    return size;
}
```

Az isEmpty() megmondja, hogy üres-e az objektum, azaz hogy a mérete nagyobb-e nullánál.

```
@Override
 public boolean isEmpty() {
  return size <= 0;
 }
```

A containsKey és a containsValue megmondják, benne van-e a megadott kulcs illetve érték az objektumunkban. Ezt mindkettő a searchItemInArray() függvénnyel teszi, amely visszatéríti, hogy egy megadott érték megtalálható-e egy értékkészletben.

```
@Override
  public boolean containsValue(Object value) {
  int valueIndex = searchItemInArray(value, values,
Object::equals);
  return valueIndex > -1 && keys[valueIndex] != null;
}
```

```
@Override
  public boolean containsKey(Object key) {
  Objects.requireNonNull(key, NULL_KEY_NOT_SUPPORTED);

  return searchItemInArray(key, keys, Object::equals) != -1;
}
```

A get() metódus végigiterálja az elemeket, amíg meg nem találja a megadott kulcsot. Ha az megvan, akkor visszatéríti a kulcshoz tartozó értéket. Egyébként null értéket ad vissza.

```
@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;
}
```

A put() betesz a containerünkbe egy adott kulcsú elemet. Ha a kulcs nem szerepel a containerünkben, akkor kibővíti azt. Ehhez hasonlóan a putAll() egyszerre több értéket tesz a Mapunkba.

```
@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 void putAll(Map<? extends K, ? extends V> m) {
     m.forEach(this::put);
}
```

A remove() eltávolít egy adott kulcsú elemet. Ennek a párja a clear(), amely az összes elemet eltávolítja.

```
@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 clear() {
     Arrays.fill(keys, null);
     Arrays.fille(values, null);
     size = 0;
}
```

A keySet() visszatér egy kulcsokat tartalmazó HashSet-tel, míg a values() egy értékeket tartalmazó ArrayList-et ad vissza nekünk.

```
@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 != 0) {
               V value = values[i];
               result.add(new AbstractMap.SimpleEntry<>(key,
value));
          }
     }
     return result;
}
```

Végezetül egy pár segítő függvények. a searchItemInArray() megkeres egy adott elemet a egy tömbben, a findFirstEmpty(), megkeresi az első üres elemet. Illetve az expandArrays() megkétszerezi a Map méretét.

```
private <I> int searchItemInArray(I item, I[] array,
BiPredicate<I, I> equalFunction) {
     for (int index = 0; index < array,length; index++) {
          return index;
     }
     return -1
}

private int findFirstEmptyPlace() {
     return searchItemInArray(null, keys, Objects::equals);
}

private void expandArrays() {
     int expandSize = size * 2;

     keys = Arrays.copyOf(keys, expandedSize);
     values = Arrays.copyOf(values, expandedSize);
}
```

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.

5. hét - „Helló, Stroustrup!”

5. hét Objektumorientált programozási nyelvek típusrendszere (pl.: Java, C#) és 6. hét Típusok tagjai: mezők, (nevesített) konstansok, tulajdonságok, metódusok, események, operátorok, indexelők, konstruktorok, destruktorok, beágyazott típusok.

Összevonva.

JDK osztályok

Írjunk olyan Boost C++ programot (indulj ki például a fénykardból) amely kilistázza a JDK összes osztályát (miután kicsomagoltuk az src.zip állományt, arra ráengedve)!

Másoló-mozgató szemantika

Kódcsipeteken (copy és move ctor és assign) keresztül vesd össze a C++11 másoló és a mozgató szemantikáját, a mozgató konstruktort alapozd a mozgató értékadásra!

Hibásan implementált RSA törése

Készítsünk betű gyakoriság alapú törést egy hibásan implementált RSA kódoló:
https://arato.inf.unideb.hu/batfai.norbert/UDPROG/deprecated/Prog2_3.pdf (71-73 fólia)
által készített titkos szövegen.

Változó argumentumszámú ctor

Készítsünk olyan példát, amely egy képet tesz az alábbi projekt Perceptron osztályának bemenetére és a Perceptron ne egy értéket, hanem egy ugyanakkora méretű „képet” adjon vissza. (Lásd még a 4 hét/Perceptron osztály feladatot is.)

Összefoglaló

Az előző 4 feladat egyikéről írj egy 1 oldalas bemutató „”esszé szöveget!

EPAM: It's gone. Or is it?

Adott a következő osztály:

```
public class BugousStuffProducer {
    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 finalize() throws IOException {
        writer.close();
    }
}
```

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.

Mutass példát arra az esetre, amikor előfordulhat, hogy bár a program futása során meghívtuk a `writeStuff()` metódust, a fájl, amibe írtunk még is üres.

Magyarázd meg, miért. Mutass alternatívát.

A következő main függvényben előfordulhat, hogy a fájl üres marad:

```
public static void main(String[] args) throws Exception {
        try (BugousStuffProducer stuffProducer = new
BugousStuffProducer("someFile.txt")) {
            stuffProducer.writeStuff();
        }
}
```

Ez azért fordul elő, mert nincs garancia arra, hogy lefut a finalize() függvény.

A finalizer akkor hívódik meg, ha a JVM szerint egy instance-hoz szükséges a garbage collection.

A finalizer feladata, hogy az objektumok által felhasznált erőforrásokat felszabadítsa, mielőtt azt kitörli a memóriából.

Viszont a finalizer sok hibába is ütközhet. Például lehet, hogy a takarítás végezete előtt kifutunk a rendelkezésre álló memóriából, amely rendszerösszeomláshoz vezethet. Ezen kívül a ez a folyamat nagy teljesítmény igényű. A JVM-nek sok műveletet kell végrehajtania, amikor nem üres finalizer-t tartalmazó objektumokat próbál konstruálni illetve dekonstruálni. Továbbá, ha nem kezeljük megfelelően a kivételeket, a finalize művelet megszakad, és az objektum minden jelzés nélkül meghibásodik.

Ahhoz hogy elkerüljük a finalize() hibáit, egyszerűen ne használjuk, és helyette implementáljuk a close() metódust. Ezt akkor tehetjük meg, ha osztály szinten implementáljuk az AutoCloseable tulajdonságot.

```
private static class BugousStuffProducer implements AutoCloseable
{
     ...
     @override
     public void close() throws Exception {
          writer.close();
     }
}
```

EPAM: Kind of equal

Adott az alábbi kódrészlet.

```
// Given
String first = "...";
String second = "...";
String third = "...";
// When
var firstMatchesSecondWithEquals = first.equals(second);
var firstMatchesSecondWithEqualToOperator = first == second;
var firstMatchesThirdWithEquals = first.equals(third);
var firstMatchesThirdWithEqualToOperator = first == third;
```

Változtasd meg a `String third = "...";` sort úgy, hogy a `firstMatchesSecondWithEquals, firstMatchesSecondWithEqualToOperator, firstMatchesThirdWithEquals` értéke `true`, a `firstMatchesThirdWithEqualToOperator` értéke pedig `false` legyen. Magyarázd meg, mi történik a háttérben.

A következő változtatás szükséges:

```
String third = new String("...")
```

Ekkor mindezek az assert állítások igazak lesznek:

```
// Then
assertThat(firstMatchesSecondWithEquals, is(true));
assertThat(firstMatchesSecondWithEqualToOperator, is(true));
assertThat(firstMatchesThirdWithEquals, is(true));
assertThat(firstMatchesThirdWithEqualToOperator, is(false));
```

Ez azért történik, mert a fenti first és second String ugyanarra a String objektumra referencia a String interning miatt, tehát egyenlők akkor is, ha az értéküket vizsgáljuk és akkor is, ha az objektumot.

A String.intern() Stringek sorozatára vonatkozóan biztosítja, hogy minden olyan String, amelynek ugyanaz az értéke, ugyanazon memórián fog tárolódni. Ezért lesz egyenlő a first és a second, akár az objektumot, akár az értéket nézzük.

Ezzel szemben, a third már egy új String objektumot kap meg értékként, tehát, már nem lesz ugyanaz az objektum, mint a másik kettő, ezért lesz false a negyedik.

EPAM: Java GC

Mutasd be nagy vonalakban hogyan működik Java-ban a GC (Garbage Collector). Lehetséges az OutOfMemoryError kezelése, ha igen milyen esetekben? Források:

Garbage Collector

A Java-ban hét fajta GC létezik: Serial Garbage Collector, Parallel Garbage Collector, GMS Garbage Collector, G1 Garbage Collector, Epsilon Garbage Collector, Z Garbage Collector, és a Shenandoah Garbage Collector.

A SGC a legegyszerűbb. Egyszálas környezetre tervezték. Ez az implementáció az összes alkalmazásszálat lefagyasztja. A szemétgyűjtéshez csak egy szálat használ, ezért egy szálon futó applikációkhoz javasolt.

A PGC hasonlít a SGC-höz, de több szálon fut. Hasonlóan az előzőhöz az összes alkalmazásszálat lefagyasztja. Azon alkalmazásokhoz lett kitalálva, amelyek kibírják a megállítást.

A GMSGC (Concurrent Mark Sweep Garbage Collector) több garbage collector szálat használ a szemétgyűjtéshez. Átnézi a heap memory-t, megjelöli azon példányokat, amelyeket be kell gyűjteni, majd begyűjti azokat. Azokhoz az alkalmazásokhoz tervezték, amelyek megengedhetik maguknak a futás közbeni GC-ral való processzor erőforrások megosztását.

A G1 GC felosztja a heap memóriát részekre, és párhuzamosan gyűjti be róluk a szemetet.

Az Epsilon egy passzív GC, memóriát allokál az alkalmazásnak, de nem gyűjti be a nem használt objektumokat. Az Epsilon megengedi az alkalmazásnak a memóriából való kifogyást, és összeomlást.

A ZGC minden erőforrásigényes operációt a program futásával egy időben végez el. Az alkalmazás szálait 10 ms-nál több időre nem állítja meg. Alkalmas a nagy heap-et használó alkalmazásokhoz. Az Oracle dokumentáció szerint több terabájtos heap-et is képes kezelni.

A Shenandoah egy nagyon kevés időre állítja meg csak a program szálait. A szemétgyűjtés nagy részét a program futásával egy időben végzi el.

Az OutOfMemoryError kezelése lehetséges, mégpedig akkor, ha egy allokációhoz nincs elegendő memória.

https://medium.com/@hasithalgamge/seven-types-of-java-garbage-collectors-6297a1418e82
https://stackoverflow.com/questions/2679330/catching-java-lang-outofmemoryerror

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.

6. hét - „Helló, Gödel!”

7. hét Interfészek. Kollekciók. és 8. hét Funkcionális nyelvi elemek. Lambda kifejezések.

Összevonva.

Gengszterek

Gengszterek rendezése lambdával a Robotautó Világbajnokságban https://youtu.be/DL6iQwPx1Yw (8:05-től)

C++11 Custom Allocator

https://prezi.com/jvvbytkwgsxj/high-level-programming-languages-2-c11-allocators/
a CustomAlloc-os példa, lásd C forrást az UDPROG repóban!

STL map érték szerinti rendezése

Például: https://github.com/nbatfai/future/blob/master/cs/F9F2/fenykard.cpp#L180

Alternatív Tabella rendezése

Mutassuk be a https://progpater.blog.hu/2011/03/11/alternativ_tabella a programban a java.lang Interface Comparable<T> szerepét!

Prolog családfa

Ágyazd be a Prolog családfa programot C++ vagy Java programba! Lásd para_prog_guide.pdf!

GIMP Scheme hack

Ha az előző félévben nem dolgoztad fel a témát (például a mandalás vagy a króm szöveges dobozosat) akkor itt az alkalom!

EPAM: Mátrix szorzás Stream API-val

Implementáld le a mátrix szorzást Java-ban for és while ciklusok használata nélkül.

package com.epam.training.matrix;

```
public interface Matrix {

     void setElement(int x, int y, int value);

     Matrix multiply(Matrix input);

}
```

Ez az interface leírja, hogy mit várunk el a mátrixunktól. A setElement(x, y, value) bele fog illeszteni egy mátrix x és y koordinátáiba egy value értéket. A multiply fogja elvégezni a mátrix szorzást.

A következő kód tartalmazza a mátrix alap implementálását:

```
import java.util.Arrays;
import com.epam.training.matrix.Matrix;

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;
	}

	@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");
	}

	...
}
```

A feladat lényegi része a mátrix szorzás funkcionális megoldása, ami a következő kódrészletben található:

```
import java.util.Arrays;
import java.util.stream.IntStream;

import com.epam.training.matrix.Matrix;

public class LambdaMatrix extends AbstractMatrix {

	public LambdaMatrix(int[][] matrix) {
		super(matrix);
	}

	public LambdaMatrix(int rowsLenght, int columnsLenght) {
```

```
            super(rowsLenght, columnsLenght);
      }

      @Override
      protected Matrix multiply(AbstractMatrix input) {
            int[][] result = Arrays.stream(this.matrix)
                        .map(r -> IntStream.range(0,
input.columnsLenght)
                                    .map(i -> IntStream.range(0,
input.rowsLenght).map(j -> r[j] * input.matrix[j][i]).sum())
                                    .toArray())
                        .toArray(int[][]::new);
            return new LambdaMatrix(result);
      }

}
```

A multiply() metódus először egy kétdimenziós result tömböt készít, amelyhez hozzáad egy a mátrixból (kételemű tömbből) készült streamet. Majd a map-ek segítségével végigiterálunk a sorokon és oszlopokon és elvégezzük rajtuk a sum() metódust. . Végezetül az eredményből készít egy új kétdimenziós tömböt, amit a függvény visszaad eredményül.

EPAM: LinkedList vs ArrayList

Mutass rá konkrét esetekre amikor a Java-beli LinkedList és ArrayList rosszabb performanciát eredményezhet a másikhoz képest. (Lásd még LinkedList és ArrayList forráskódja). Végezz méréseket is. (mit csinál az ArrayList amikor megtelik)

A mérések:

```
--- List measurement ---
Starting 'collection setup' measuring
ArrayList, Millisec: 1087
ArrayList, Millisec: 612
LinkedList, Millisec: 1959
Starting 'frequent change' measuring
ArrayList, Millisec: 1139
ArrayList, Millisec: 1412
LinkedList, Millisec: 16
Starting 'get' measuring
ArrayList, Millisec: 0
ArrayList, Millisec: 0
LinkedList, Millisec: 111
```

Az eredmények sorra a listához valü hozzáadás, listából való törlés, és listából való adott elem hozzáférése. Java-ban ArrayListet akkor érdemes használnunk a mérés alapján, ha gyors hozzáférést szeretnénk biztosítani az elemekhez. Viszont ha listánkból törölni is szeretnénk, akkor a LinkedList biztosít jobb teljesítményt.

LinkedListet akkor érdemes használni, ha végig akarunk iterálni rajta. Ha nem egy adott index értékével akarunk dolgozni, hanem végig akarunk menni a listán előről vagy hátulról.

Ezzel ellentétben az ArrayList gyors hozzáférést biztosít, bármelyik elemet szeretnénk elérni.

EPAM: Refactoring

Adott egy “legacy” kód mely tartalmaz anonymus interface implementációkat, ciklusokat és feltételes kifejezések. Ebben a feladatban ezt a “legacy” kódot szeretnénk átírni lambda kifejezések segítségével (metódus referencia használata előnyt jelent!)

```
Runnable runnable = new Runnable() {
                @Override
                 public void run() {
                     System.out.println("Runnable!");
                }
};
runnable.run();
```

A fenti legacy kód egyszerűen kiírja nekünk, hogy „Runnable!”. Ha ezt át szeretnénk íratni lambdák segítségével, akkor elegendő visszatérítenünk egy olyan lambda függvény eredményét, amely kiírja a konzolra azt, hogy „Runnable!”.

```
private Runnable createRunnable() {
          return () -> System.out.println("Runnable!");
}
```

A következő kódrészlet Calculator.calculate() függvénye visszaadja egy kapott szám négyzetét:

```
Calculator calculator = new Calculator() {
                @Override
                public Integer calculate(Integer number) {
                     return number * number;
                }
     };
```

Ezt a következőképpen lehet lambdákkal megoldani:

```
private Calculator createCalculator() {
          return number -> number * number;
}
```

Visszaadja a number paramétert, ahol a number megvan szorozva önmagával.

```
Consumer<Integer> method = new Consumer<Integer>() {
                @Override
                public void accept(Integer integer) {
                     System.out.println(integer);
                 }
      };
```

A Consumer.accept() kiírja a kapott számot a konzolra:

```
private Consumer<Integer> createConsumerMethod() {
          return System.out::println;
}
```

A System.out::println ugyanaz, mintha azt írnánk, hogy number -> System.out.println(number);

A Formatter.format() pedig kap egy Integer listát, majd kiírja öket egy számként, szóközök nélkül

```
private Formatter createFormatter() {
          return numbers -> numbers.stream()
                    .map(String::valueOf)
                    .collect(Collectors.joining());
 }
```

Itt a String::valueOf a number -> String.valueOf(number) lambdának felel me

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.

7. hét - „Helló, !”

9. hét Adatfolyamok kezelése, streamek és 11. hét I/O, állománykezelés. Szerializáció.

Összevonva.

FUTURE tevékenység editor

Javítsunk valamit a ActivityEditor.java JavaFX programon!
https://github.com/nbatfai/future/tree/master/cs/F6
Itt láthatjuk működésben az alapot: https://www.twitch.tv/videos/222879467

OOCWC Boost ASIO hálózatkezelése

Mutassunk rá a scanf szerepére és használatára! https://github.com/nbatfai/robocar-emulator/blob/master/justine/rcemu/src/carlexer.ll

SamuCam

Mutassunk rá a webcam (pl. Androidos mobilod) kezelésére ebben a projektben:
https://github.com/nbatfai/SamuCam

BrainB

Mutassuk be a Qt slot-signal mechanizmust ebben a projektben:
https://github.com/nbatfai/esport-talent-search

OSM térképre rajzolása[7]

Debrecen térképre dobjunk rá cuccokat, ennek mintájára, ahol én az országba helyeztem el a DEAC hekkereket: https://www.twitch.tv/videos/182262537 (de az OOCWC Java Swinges megjelenítőjéből: https://github.com/nbatfai/robocar-emulator/tree/master/justine/rcwin is kiindulhatsz, mondjuk az komplexebb, mert ott időfejlődés is van...)

EPAM: XML feldolgozás

Adott egy koordinátákat és államokat tartalmazó XML (kb 210ezer sor), ezt az XML-t feldolgozva szeretnék létrehozni egy SVG fájlt, melyben minden város megjelenik egy pont formájában az adott koordináták alapján (tetszőleges színnel)

Plusz feladat: A városokat csoportosíthatjuk államok szerint, és minden állam külön színnel jelenjen meg a térképen, így látszódni fognak a határok is.

Elvárt eredmény:

[7] Alternatívaként készíthetsz egy GoogleMaps alapú Androidos „GPS trackert”, 2007 óta csinálok ilyen példát: https://youtu.be/QStgBZ6JfAU az aktuális a Bátfai Haxor Stream keretében: https://bhaxor.blog.hu/2018/09/19/nandigps_ismerkedes_a_gps-el

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.

A main függvényben példányosítunk egy objektumot (SaxXmlReader), amely segíteni fog nekünk az xml file beolvasásában. Ezt egy FileInputStream-mel oldjuk meg. Ez az objektum byte-okat olvas be egy file-ból. A FileInputStream-et arra készítették, hogy úgynevezett raw byte-okat olvasson be, mint például valamilyen képadat.

```
import org.xml.sax.SAXException;

import javax.xml.parsers.ParserConfigurationException;
import java.io.FileInputStream;
import java.io.FileWriter;
import java.io.IOException;
import java.util.List;

public class XmlMapPlotter {
    public static void main(String[] args) throws IOException, SAXException,
ParserConfigurationException {
        XmlReader reader = new SaxXmlReader(new FileInputStream("input.xml"));
        List<City> cities = reader.getCities();
        MapImage image = new SvgMapImage(800, 600);
        cities.forEach(city -> city.plot(image));
        image.save(new FileWriter("map.svg"));
    }
}
```

Az SVG file készítését pedig egy külső GenericDOMImplementation könyvtár segítségével oldjuk meg az SvgMapImage file-ban

```
import org.apache.batik.dom.GenericDOMImplementation;
import org.apache.batik.svggen.SVGGraphics2D;
import org.w3c.dom.DOMImplementation;
import org.w3c.dom.Document;
```

```
import java.awt.*;
import java.io.Writer;

public class SvgMapImage implements MapImage {

    SVGGraphics2D svgGenerator;

    public SvgMapImage(int canvasWidth, int canvasHeight) {
        // Get a DOMImplementation.
        DOMImplementation domImpl = GenericDOMImplementation.getDOMImplementation();
        // Create an instance of org.w3c.dom.Document.
        String svgNS = "http://www.w3.org/2000/svg";
        Document document = domImpl.createDocument(svgNS, "svg", null);
        // Create an instance of the SVG Generator.
        svgGenerator = new SVGGraphics2D(document);
        svgGenerator.setSVGCanvasSize(new Dimension(canvasWidth, canvasHeight));
    }

    @Override
    public void addPoint(double x, double y) {
        svgGenerator.fillOval((int) x, (int) y, 2, 2);
    }

    @Override
    public void save(Writer targetStream) {
        try {
            svgGenerator.stream(targetStream);
        } catch (Exception e) {
            throw new RuntimeException("Failed to write image", e);
        }
    }
}
```

Ennek az osztálynak a két függvénye rak egy pontot egy bizonyos koordinátára (addPoint), illetve elmenti a képet (save).

A file beolvasása pedig a SaxXmlReader osztályon keresztül megy végbe.

```
import org.xml.sax.Attributes;
import org.xml.sax.SAXException;
import org.xml.sax.helpers.DefaultHandler;
import javax.xml.parsers.ParserConfigurationException;
import javax.xml.parsers.SAXParser;
import javax.xml.parsers.SAXParserFactory;
import java.io.IOException;
import java.io.InputStream;
import java.util.LinkedList;
import java.util.List;
import java.util.function.Consumer;
```

```
public class SaxXmlReader extends DefaultHandler implements XmlReader {

    private double currentXCoordinate;
    private double currentYCoordinate;
    private String currentState;

    private Consumer<String> actionOnCharacters = c -> {
    };

    private List<City> result = new LinkedList<>();

    public SaxXmlReader(InputStream input) throws ParserConfigurationException,
SAXException, IOException {
        SAXParser parser = SAXParserFactory.newInstance().newSAXParser();
        parser.parse(input, this);
    }

    @Override
    public void startElement(String uri, String localName, String qName, Attributes
attributes) throws SAXException {
        if (qName.equalsIgnoreCase("coordinateX")) {
            actionOnCharacters = c -> currentXCoordinate = Double.parseDouble(c);
        }
        if (qName.equalsIgnoreCase("coordinateY")) {
            actionOnCharacters = c -> currentYCoordinate = Double.parseDouble(c);
        }
        if (qName.equalsIgnoreCase("state")) {
            actionOnCharacters = c -> currentState = c;
        }
    }

    @Override
    public void endElement(String uri, String localName, String qName) throws
SAXException {
        if (qName.equalsIgnoreCase("city")) {
            result.add(new City(currentXCoordinate, currentYCoordinate, currentState));
        }
    }

    @Override
    public void characters(char[] ch, int start, int length) throws SAXException {
        actionOnCharacters.accept(new String(ch, start, length));
        actionOnCharacters = c -> {
        };
    }

    @Override
    public List<City> getCities() {
```

```
        return result;
    }
}
```

EPAM: ASCII Art

ASCII Art in Java! Implementálj egy Java parancssori programot, ami beolvas egy képet és kirajzolja azt a parancssorba és / vagy egy szöveges fájlba is ASCII karakterekkel.

A Main.java tartalma:

```
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;

import com.epam.training.asciiprinter.AsciiPrinter;

public class Main {

	public static void main(String[] args) throws IOException {
		String imageName = args[0];
		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();
	}

}
```

A main függvény megkapja parancssori argumentumként a kép nevét, majd az AsciiPrinter objektumon keresztül a print() metódussal elkészítjük és kiíratjuk az ASCII Artot.

Maga az AsciiPrinter:

```
import java.awt.image.BufferedImage;
import java.io.IOException;
import java.io.OutputStream;

public class AsciiPrinter {

	private static final char[] ASCII_PIXELS = { '$', '#', '*', ':', '.', ' '
};
	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;
      }

      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 >> 16) & 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];
      }

}
```

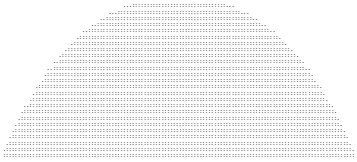
A print metódus végigmegy a kép minden pontján, majd minden pont helyén kiír egy ASCII karaktert. Ezt az outputStream-en keresztül történik meg, amely egy ByteStream.

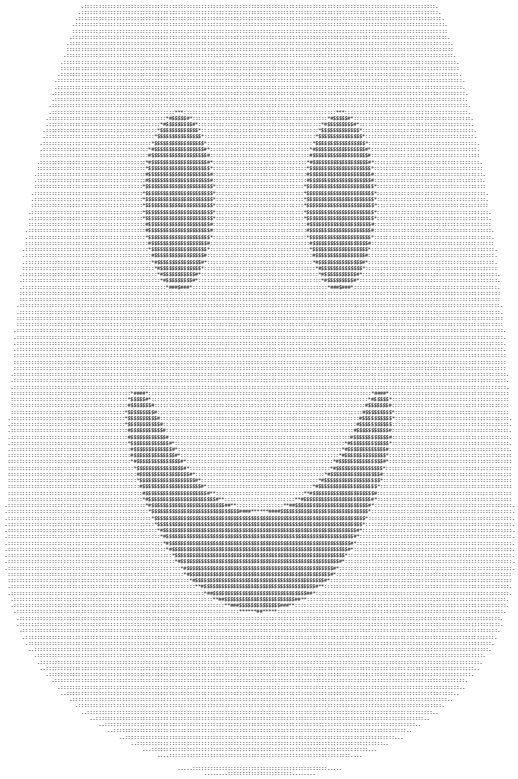
```
public void print() throws IOException {
            for (int i = 0; i < image.getHeight(); i++) {
                  for (int j = 0; j < image.getWidth(); j++) {
                        outputStream.write(getAsciiChar(image.getRGB(j, i)));
                  }
                  outputStream.write(NEW_LINE);
            }
      }
```

Az OutputStream write() metódusa kiír egy byte-ot a megnevezett stream-re

Az OutputStream minden output stream superosztálya. Az egyik alosztálya a FileOutputStream, amelyet adatok file-okba való írásához használunk. Ebben a programban is van erre példa, amikor a egy text file-ba íratjuk az ASCII karakterekből álló képünket a Main file-ban.
Egy példa a bemenetre:

A kimenet:





Az AsciiPrinter többi része azt állápítja meg, hogy milyen ASCII karakterrel helyettesítse az adott képpontot.

EPAM: Titkos üzenet, száll a gépben!

Implementájl egy olyan parancssori alkalmazást, amely a billentyűzetről olvas soronként ASCII karakterekből álló sorokat, és a beolvasott szöveget Caesar kódolással egy txt fájlba írja soronként.

A Main.java tartalma:

```
package caesar;

import java.io.IOException;

import caesar.coder.stream.ConsoleInputToFileCeasarEncoder;
import caesar.coder.stream.StreamEncoder;

public class Main {

	public static void main(String[] args) throws IOException {
		String fileName = args[0];
		int offset = Integer.valueOf(args[1]);
		try (StreamEncoder handler = new ConsoleInputToFileCeasarEncoder(fileName, offset)) {
			handler.handleInputs();
		}
	}

}
```

Ez a program megkap inputként egy file-t, majd annak a tartalmának betűt eltolja egy bizonyos számmal. Ez a Ceasar titkosítás.

A kódolás nagy részét a StreamEncoder osztály végzi, amelynek a forráskódja:

```
package caesar.coder.stream;

import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;
import java.util.Scanner;

import caesar.coder.Encoder;

public class StreamEncoder implements AutoCloseable {

	private static final byte[] NEW_LINE = "\n".getBytes();

	private Scanner inputScanner;
	private OutputStream outputStream;
	private Encoder encoder;

	public StreamEncoder(InputStream inputStream, OutputStream outputStream, Encoder encoder){
		this.inputScanner = new Scanner(inputStream);
		this.outputStream = outputStream;
		this.encoder = encoder;
	}

	public void handleInputs() throws IOException {
		String line;
		do {
			line = inputScanner.nextLine();
			String encodedLine = encoder.encode(line);
			outputStream.write(encodedLine.getBytes());
			outputStream.write(NEW_LINE);
		} while (!"exit".equals(line));
	}

	@Override
	public void close() throws IOException {
		inputScanner.close();
		outputStream.close();
	}
```

Ez az osztály a file-ok olvasására és írására az InputStream és OutputStream osztályokat használja.

Magát a kódolást és file-ba írást a StreamEncoder handleInputs() függvénye rendezi:

```
public void handleInputs() throws IOException {
		String line;
		do {
			line = inputScanner.nextLine();
			String encodedLine = encoder.encode(line);
			outputStream.write(encodedLine.getBytes());
			outputStream.write(NEW_LINE);
		} while (!"exit".equals(line));
	}
```

Soronként beolvassa a file-t és kódolja azokat.

A program nem ad meg outputFile-t mint kimeneti lehetőséget, de a következő test megállapítja, hogy a program helyesen működik és sikeresen végzi el a Ceasar titkosítást.

```
package caesar.coder.ceasar;

import static org.junit.jupiter.api.Assertions.assertEquals;
```

```
import static org.junit.jupiter.api.Assertions.assertThrows;

import org.junit.jupiter.api.Test;

import caesar.coder.caesarcoder.CeasarCoder;

public class CeasarCoderTest {

	private CeasarCoder underTest = new CeasarCoder(3);

	@Test
	public void testEncodeShouldWorkCorrectly() {
		// Given
		String input = "he told me i could never teach a llama to drive";
		String expected = "kh wrog ph l frxog qhyhu whdfk d oodpd wr gulyh";

		// When
		String actual = underTest.encode(input);

		// Then
		assertEquals(expected, actual);
	}

	@Test
	public void testDecodeShouldWorkCorrectly() {
		// Given
		String input = "kh wrog ph l frxog qhyhu whdfk d oodpd wr gulyh";
		String expected = "he told me i could never teach a llama to drive";

		// When
		String actual = underTest.decode(input);

		// Then
		assertEquals(expected, actual);
	}

	@Test
	public void
testConstructorShouldThrowIllegalArgumentExceptionWhenOffsetParameterIsGreaterThen127() {
		// Given

		// When
		assertThrows(IllegalArgumentException.class, () -> new CeasarCoder(128));

		// Then
	}

	@Test
	public void
testConstructorShouldThrowIllegalArgumentExceptionWhenOffsetParameterIsLessThen1() {
		// Given

		// When
		assertThrows(IllegalArgumentException.class, () -> new CeasarCoder(0));

		// Then
	}

}
```

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.

8. hét - „Helló, Lauda!”

10. hét Kivételkezelés. és 12. hét Reflexió. A fordítást és a kódgenerálást támogató nyelvi elemek (annotációk, attribútumok).

Összevonva.

Port scan

Mutassunk rá ebben a port szkennelő forrásban a kivételkezelés szerepére!

https://www.tankonyvtar.hu/hu/tartalom/tkt/javat-tanitok-javat/ch01.html#id527287

Maga a forrás:

```
public class KapuSzkenner {

    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 program megnézi, hogy a gépünk éppen milyen portokat figyel. Végigiterál a parancssorban megkapott nevű gép 1024 alatti TCP kapuin és megpróbál megnyitni egy TCP kapcsolatot.

Ha ez sikerül, az eredmény „figyeli”, egyébként egy kivételt kapunk, amit később a catch blokkban kezelünk, azzal, hogy kiítjuk „nem figyeli”

Az eredmény:

```
.
.
.
19 nem figyeli
20 nem figyeli
21 figyeli
22 figyeli
23 nem figyeli
24 nem figyeli
25 figyeli
26 nem figyeli
.
.
.
```

```
79 nem figyeli
80 figyeli
81 nem figyeli
82 nem figyeli
.
.
.
```

AOP

Szőj bele egy átszövő vonatkozást az első védési programod Java átiratába! (Sztenderd védési feladat volt korábban.)

Android Játék

Írjunk egy egyszerű Androidos „játékot”! Építkezzünk például a 2. hét „Helló, Android!” feladatára!

Junit teszt

A https://progpater.blog.hu/2011/03/05/labormeres_otthon_avagy_hogyan_dolgozok_fel_egy_pedat poszt kézzel számított mélységét és szórását dolgozd be egy Junit tesztbe (sztenderd védési feladat volt korábban).

OSCI

Készíts egyszerű C++/OpenGL-es megjelenítőt, amiben egy kocsit irányítasz az úton.

OSCI2

Készíts egyszerű C++/OpenGL-es megjelenítőt, amiben egy kocsit irányítasz az úton. A kocsi állapotát minden pillanatban mentsd le. Ezeket add át egy Prolog programnak, ami egyszerű reflex ágensként adjon vezérlést a kocsinak, hasonlítsd össze a kézi és a Prolog-os vezérlést. Módosítsd úgy a programodat, hogy ne csak kézzel lehessen vezérelni a kocsit, hanem a Prolog reflex ágens vezérelje!

OSCI3

Készíts egy OSM utakat megjelenítő C++/OpenGL-es progit!

EPAM: DI

Implementálj egy alap DI (Dependency Injection) keretrendszert Java-ban annotációk és reflexió használatával megvalósítva az IoC-t (Inversion Of Control).

A Dependency Injection, vagy magyarul függőségbefecskendezés az, amikor egy objektumnak vagy keretrendszernek függőséget adunk át. A Dependency Injetction megkönnyíti a tesztelést. A befecskendezés megtörténhet a konstruktoron keresztül

Például legyen fenti SomeClass() osztálynak a konstruktora:

```
public SomeClass() {
    myObject = Factory.getObject();
}
```

Ha a myObject objektum komplex feladatokban is részt vesz, akkor nehéz tesztelni a SomeClass-t. Erre a megoldás a dependency injection, azaz az objektum átadása a SineCkass-nak. Ezt Java-ban a this kulcsszóval tudjuk használni.

```
public SomeClass (MyClass myObject) {
    this.myObject = myObject;
}
```

A DI keretrendszerben található dependency injection a DiContextBuilder osztályban:

…

```
private class BeanDefinition {
        String name;
        Class type;
        Method builderMethod;
        List<BeanDefinitionParameter> dependencies;

        public BeanDefinition(String name, Class type, Method builderMethod,
                List<BeanDefinitionParameter> dependencies) {
            super();
            this.name = name;
            this.type = type;
            this.builderMethod = builderMethod;
            this.dependencies = dependencies;
        }
    }

    private class BeanDefinitionParameter<T> {
        String name;
        Class<T> type;

        public BeanDefinitionParameter(String name, Class<T> type) {
            super();
            this.name = name;
            this.type = type;
        }
    }
…
```

EPAM: JSON szerializáció

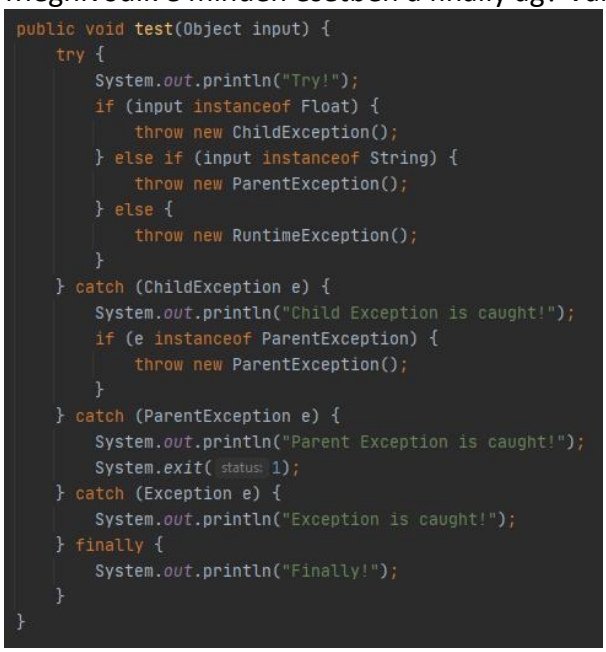
Implementálj egy JSON szerializációs könyvtárat, mely képes kezelni sztringeket, számokat, listákat és beágyazott objektumokat. A megoldás meg kell feleljen az összes adott unit tesztnek. Plusz feladat:

1. a könyvtár tudjon deszerializálni

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.

EPAM: Kivételkezelés

Adott az alábbi kódrészlet. Mi történik, ha az input változó 1F, “string” vagy pedig null? Meghívódik e minden esetben a finally ág? Válaszod indokold!



```
public void test(Object input) {
    try {
        System.out.println("Try!");
        if (input instanceof Float) {
            throw new ChildException();
        } else if (input instanceof String) {
            throw new ParentException();
        } 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( status: 1);
    } catch (Exception e) {
        System.out.println("Exception is caught!");
    } finally {
        System.out.println("Finally!");
    }
}
```

```
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) {

        }

        System.out.println("Test case when input is String!");
        test("string");
    }
...
```

A fenti kód a képen lévő kódrészlethez lesz hozzáadva. A tesztelést a képen definiált test() függvénnyel végezzük.

Ami érdekes, hogy az 1F tesztelése esetén szükség van kivételkezelésre a main függvényben is. Ha ezt nem tesszük meg, a következő eredményt kapjuk:

```
Exception in thread "main" com.epam.training.ParentException
        at com.epam.training.ExceptionHandling.test(ExceptionHandling.java:29)
        at com.epam.training.ExceptionHandling.main(ExceptionHandling.java:10)
```

A tesztelésünk String része nem fog lefutni.

Az Exception ignored kód azt csinálja, hogy ha elkap valamilyen kivételt, akkor azt figyelmen kívül hagyja.

A kód képen lévő része:

```
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();
            } 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");
        }
    }
```

A fenti kód leteszteli a megadott kódrészletet. Először null-lal, majd float és string értéket használva. Az eredmény a következő:

```
Test case when input is null!
Try!
Exception is caught!
Finally!

Test case when input is float!
Try!
Child Exception is caught!
Finally!

Test case when input is String!
Try!
Parent Exception is caught!
```

Amint az látható, a null input kivételt eredményez. Belépünk a test-en belüli try blokkba, majd dobunk egy új RuntimeException-t.

Hasonló az eredmény float (1F) esetén, ahol egy child exception-t kap el a függvény. A ChildExpection a ParentException alosztálya. A kódban ez a következőképpen néz ki:

```
public class ChildException extends ParentException {

}
```

A ParentException pedig a RuntimeException-höz tartozik.

```
public class ParentException extends RuntimeException {

}
```

A float esetén is meghívódik a finally ág.

Végezetül a string esetén dobunk egy ParentException-t, amit ha elkapunk, kilépünk a programból a System.exit(status 1) függvénnyel. Ezért a függvény miatt nem fog meghívódni a finally ág String esetén.

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.

9. hét - „Helló, Calvin!”

13. hét Multiparadigmás nyelvek és 14. hét Programozás multiparadigmás nyelveken.

Összevonva.

MNIST

Az alap feladat megoldása, +saját kézzel rajzolt képet is ismerjen fel, https://progpater.blog.hu/2016/11/13/hello_samu_a_tensorflow-bol Háttérként ezt vetítsük le: https://prezi.com/0u8ncvvoabcr/no-programming-programming/

Deep MNIST

Mint az előző, de a mély változattal. Segítő ábra, vesd össze a forráskóddal a https://arato.inf.unideb.hu/batfai.norbert/NEMESPOR/DE/denbatfai2.pdf 8. fóliáját!

CIFAR-10

Az alap feladat megoldása, +saját fotót is ismerjen fel, https://progpater.blog.hu/2016/12/10/hello_samu_a_cifar-10_tf_tutorial_peldabol

Android telefonra a TF objektum detektálója

Telepítsük fel, próbáljuk ki!

SMNIST for Machines

Készíts saját modellt, vagy használj meglévőt, lásd: https://arxiv.org/abs/1906.12213

Minecraft MALMO-s példa

A https://github.com/Microsoft/malmo felhasználásával egy ágens példa, lásd pl.: https://youtu.be/bAPSu3Rndi8, https://bhaxor.blog.hu/2018/11/29/eddig_csaltunk_de_innentol_mi, https://bhaxor.blog.hu/2018/10/28/minecraft_steve_szemuvege

EPAM: Reaktív programozás

Számoljuk ki az első 10 nem negatív egész szám összegét és átlagát.

1. Tegyük mindezt reaktív módon.
2. A számok előállítását végző komponensre "figyelhessenek" a különböző statisztikákat számító kom-ponensek, az egyes számítások pedig párhuzamosan, külön szálon menjenek végbe.
3. Ügyeljünk arra, hogy a számok előállítása során ne küldjünk több számot az összeget és átlagot szá-moló szálaknak, mint amit azok fel tudnak dolgozni, bármilyen lassú is legyen a számítás. Az egyes szá-mítások sebessége ne befolyásolja a számok előállításának és más számításoknak a sebességét.
4. Amennyiben egy számítást végző szál nem tudja fogadni a következő számot, azt mentsük el és kí-náljuk fel a szálnak amint kész új számot fogadni. Az így elmentett számok mennyisége legyen limitálva, ha túl sok számot kellene elmentenünk, töröljük azt, amelyik a legrégebben érkezett. Ne blokkoljuk a számok előállítását, ha van olyan számítást végző szál, amely nem tudja feldolgozni az előállítot szá-mot.
5. A számokat a számítást végző szálak az előállításuknak megfelelő sorrendben dolgozzák fel.

6. Igyekezzünk minimálisra csökkenteni a blokkolt szálak számát.
7. A számítást végző szálak fejeződjenek be, ha nincs több feldolgozandó szám.

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.

8. A megoldásunk legyen nyitott a bővítésre, de zárt a módosításra.
9. Legyen lehetőség új statisztikák bevezetésére úgy, hogy a meglévő osztályokat nem módosítjuk, illetve szükség esetén tudjunk hasonlóképpen új komponenseket létrehozni a számok előállítására is.

EPAM: Back To The Future

Adott az alábbi kódrészlet:

```
public class FutureChainingExercise {

  private static ExecutorService executorService = Executors.newFixedThreadPool(2);

  public static void main(String[] args) {
    CompletableFuture<String> longTask
        = CompletableFuture.supplyAsync(() -> {
      sleep(1000);
      return "Hello";
    }, executorService);

    CompletableFuture<String> shortTask
        = CompletableFuture.supplyAsync(() -> {
      sleep(500);
      return "Hi";
    }, executorService);

    CompletableFuture<String> mediumTask
        = CompletableFuture.supplyAsync(() -> {
      sleep(750);
      return "Hey";
    }, executorService);

    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);

    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();
  }

  /**
   *
   * @param sleeptime sleep time in
   milliseconds */
  private static void sleep(int sleeptime) {...}

}
```

Mi lesz kiíratva a standard kimenetre és miért?

EPAM: AOP

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.

Készíts két példa projektet, melyben egyes metódusok futási idejét méred majd kiíratod úgy, hogy a metódus futási idejének méréséhez AOP-t használsz. Az első projektben csak az AspectJ könyvtárat, a második esetében pedig Spring AOP-t használj.

Debrecen, 2020. szept. 14.
Dr. Bátfai Norbert, Lámfalusi Csaba (EPAM)