* Java SE alapok - nyelvi elemek
* Java SE alapok - Java OO elemek
* Java SE haladó - Kivétel és fájlkezelés
* Java SE haladó - Kollekciók és osztálykönyvtárak
* Perzisztencia, adatbázis programozás JDBC technológiával

# Bevezetés a nyelvi eszközökbe

## Eszközkészlet

Java történeti háttér (background)

Elmélet

Szoftverkrízis fogalma már 1968-ban megjelent, mely a mai napig azt jelenti, hogy minél több, komplexebb alkalmazás fejlesztésére van igény, lehetőleg egyre kevesebb fejlesztői erőforrás igénybe vételével. Erre a problémára több megoldás is született több-kevesebb sikerrel, és ezek között szerepelnek különböző programozási paradigmák bevezetése, mint pl. objektumorientált programozás.

A Java nyelv 1991-ben a Sun titkos projektjeként született, James Gosling vezetésével. A cél egy olyan programozási platform, mellyel hatékonyan lehet alkalmazásokat fejleszteni olyan digitális eszközökhöz, mint pl. a televízió. Az első megjelenése egy Mosaic böngészőbe épített Java motor, mely egy molekulát forgatott három dimenzióban. Elképzelhető mekkora újítás volt ez akkor, mikor a weboldalak fehér alapon fekete betűk voltak, és az oldalakat kék színű linkek kötötték össze.



James Gosling

A Java kifejlesztésekor a következő célkitűzéseket fogalmazták meg:

* Egyszerű, könnyen tanulható és használható
* Objektumorientált
* Robusztus, azaz hibatűrő, az alkalmazás fejlesztője, vagy használója által vétett hiba ne befolyásolja a teljes alkalmazás működését, hanem kellően lokalizálható legyen
* Biztonságos, hiszen az internetről letöltött tartalmakban nem mindig lehet

megbízni

* Architektúra-semleges, hordozható, azaz a megírt alkalmazás ugyanúgy fusson eltérő platformokon, mint Windows vagy Linux
* Nagyteljesítményű
* Interpretált, utasításonként végrehajtott
* Többszálúságot nyelvi szinten támogassa
* Dinamikus, azaz a futáshoz szükséges kódrészeket futás közben töltse be, mindig azt, amire szükség van



Java logo

A Java programozási nyelv egy szabvány, melynek több implementációja

(megvalósítása) létezik. A hivatalos referencia implementáció az ingyenes OpenJDK, melyet az Oracle és az OpenJDK köré épült közösség fejleszt, több más cég közreműködésével. Azonban ezt nem javasolják éles használatra, hanem valamely erre épülő implementációt.

Történeti okok miatt az Oracle Java SE JDK a legelterjedtebb. Ez a Sun megvásárlásával került az Oraclehöz. Azonban ennek használatához a Java 11 óta fizetni kell, így megjelentek további implementációk, mint pl. az AdoptOpenJDK.

Mivel elterjedt, és oktatásra ingyenes, ezért az Oracle Java SE JDK-t használjuk a képzésen.

A Java elsődleges felhasználási területe nagyvállalati háttérrendszerek (backend). Mobil környezetben is találkozhatunk vele, hiszen az Android készülékekre először Javaban kellett fejleszteni.

A Google az Oracle-lel való jogi csatározások miatt Android területen kezd elfordulni a Java programozási nyelvtől, és helyette a Kotlin programozási nyelvet javasolja.

Felhasználói felületek fejlesztésére, számítógépre telepíthető, önállóan futó alkalmazások fejlesztésére (kliens alkalmazások) a Java nem annyira alkalmas. Bár több technológia létezik, Swing, NetBeans Platform, Eclipse Platform, nem igazán terjedt el. Főleg Java fejlesztőeszközöket implementálnak Java alkalmazásokként.

A JavaFX egy újabb technológia felhasználói felületek fejlesztésére. A Java 8-as fejlesztőkészletben jelent meg, azonban a Java 11-esből eltávolították. Eredetileg a Java részét képző Swing vastag kliens technológia leváltására jött létre.

Java áll több IoT (Internet of Things - Internetre kötött eszközök), és Big Data megvalósítás mögött. Az Oracle szerint jelenleg 10 millió Java fejlesztő van a világon és mintegy 15 milliárd Java kódot futtató eszköz.

A Java verziószámozása az újabb verziókban már konzisztens, a weboldalon letöltéskor megjelenő verziószám (pl. 12.0.1) megegyezik azzal, amit a fejlesztőkörnyezet is kiír.

A tananyag legutóbbi frissítésekor a Java legfrissebb verziója a 15-ös verzió.

További források

Java verziószámozását, és a különböző verziókban megjelenő újdonságot a Wikipedia részletesen taglalja.

Ellenőrző kérdések

* Milyen megoldások születtek a szoftverkrízisre?
* Kinek a nevéhez kötődik a Java programozási nyelv?
* Mely cég vett és vesz részt a Java fejlesztésében?
* Milyen irányelveket vettek figyelembe a Java nyelv kialakításakor?
* Milyen Java implementációkat ismersz?
* Melyik a legfrissebb Java verzió?

Teszt

Kérdés

Melyik az a jelenleg is létező cég, melyhez legjobban köthető a Java nyelv?

* Microsoft
* IBM
* ☒ Oracle
* Sun Microsystem

Az Oracle nevéhez ködődik manapság legjobban a Java nyelv. A Sun már megszűnt, felvásárolta az Oracle. A Microsoft ugyan részt vesz Java projektekben, de ők főleg a .NET keretrendszerrel és C# nyelvvel foglalkoznak. Az IBM szintén sokat tesz hozzá a Java projektekhez, de nem ő a fő irányító.

Kérdés

Mi a Java programozási nyelv legelterjedtebb felhasználási területe?

* Felhasználói felületek fejlesztése
* ☒ Nagyvállalati háttérrendszerek (backend) fejlesztése
* Mobil alkalmazások fejlesztése
* Matematikai programok fejleszése

A Java-t főleg nagyvállalati háttérrendszerek, webszolgáltatások fejlesztésére használják. Mobil fronton az Android irányban már inkább Kotlint érdemes használni. Felhasználói felületek fejlesztésére kevésbé alkalmas, ott a HTML, CSS, JavaScript hármas tűnik megfelelő választásnak. Ha matematikai programot kell fejleszteni, a Python alkalmasabb lehet. ### Oracle JDK telepítése (installjdk)

Feladat

Fontos megjegyezni, hogy a videók régebbi verziókkal kerültek felvételre, így lehetnek eltérések. Ezeket próbáljuk a leírásban külön jelezni.

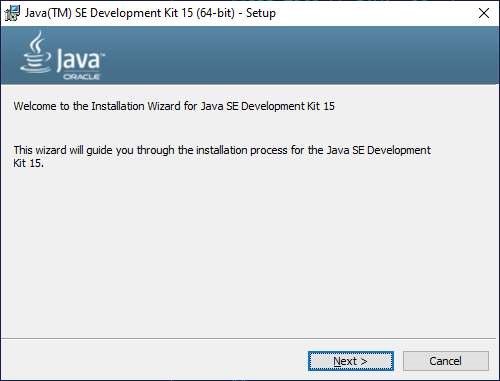
Ellenőrizd, hogy milyen JDK van telepítve a gépre parancssorban a java -version parancs kiadásával. Ha legfrissebb, akkor nincs további dolgod. Amennyiben van fenn korábbi, mondjuk 8-as, akkor el kell távolítani (Windowson a Programok telepítése és törlése kifejezésre keresve a Start menüben.)

A JDK legutolsó verzióját le kell tölteni a Java SE Development Kit 15 Downloads címről.

Jelenleg a legfrissebb a Java SE 15, a JDK Download linket kell kiválasztani. Ott a jdk-

15.0.1\_windows-x64\_bin.exe állományt kell kiválasztani, elfogadni a Licence feltételeket, és letölteni és elindítani az állományt.

A varázslóval értelemszerűen feltelepíthető.



Első képernyő



Második képernyő

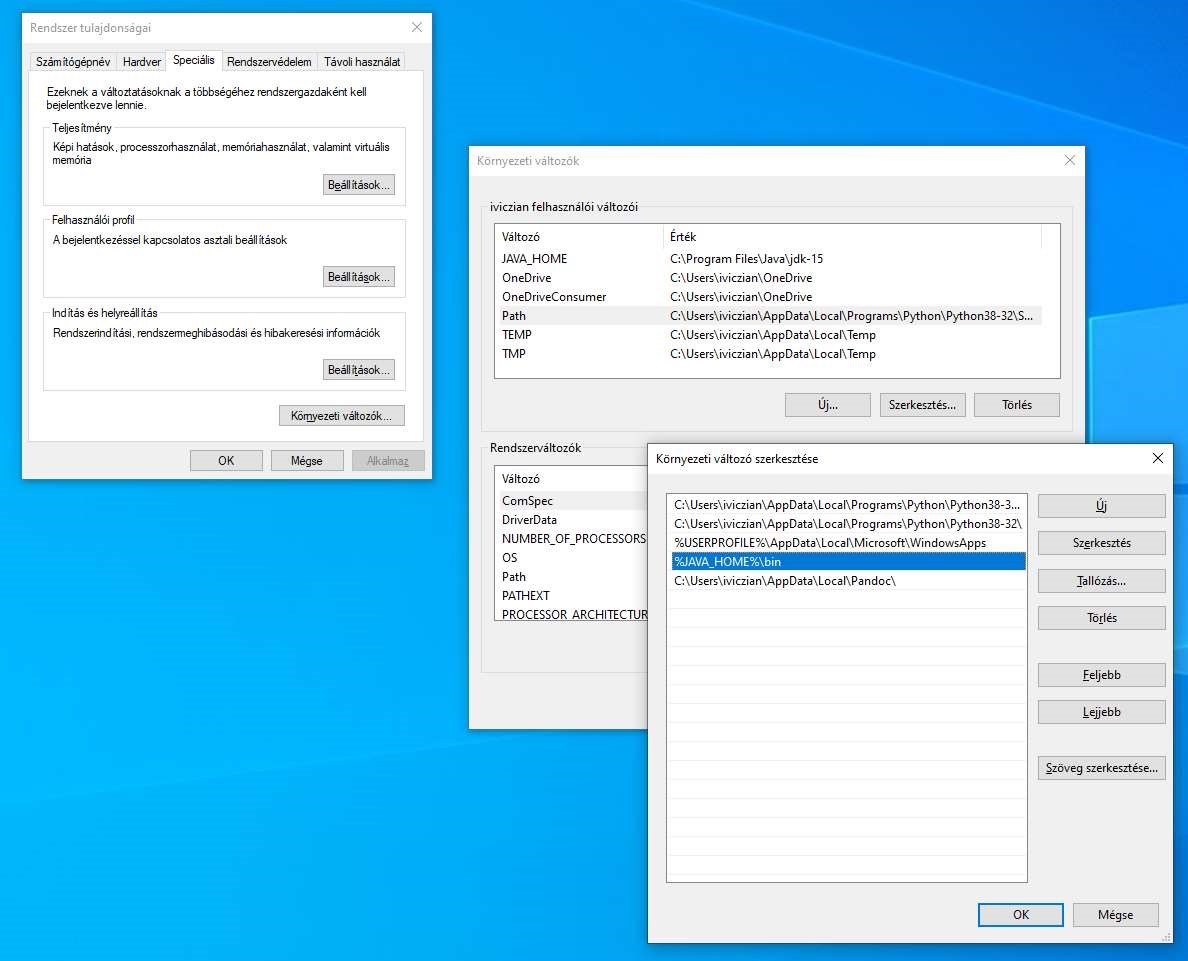


Harmadik képernyő

Alapértelmezetten a C:\Program Files\Java\jdk-15.0.1 könyvtárba telepíti.

Állítsuk be a JAVA\_HOME és PATH környezeti változókat!

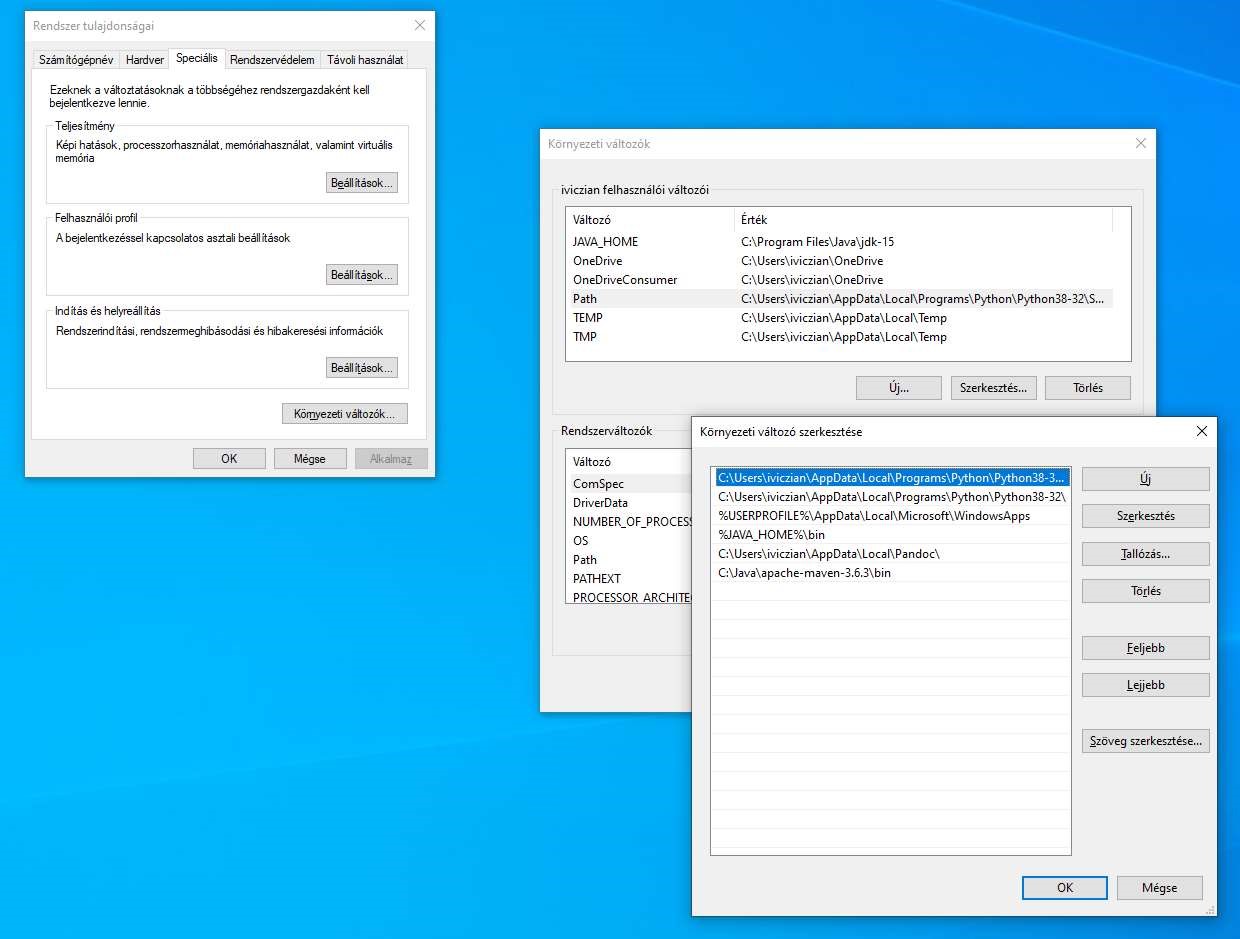
Ehhez a Windows Rendszer környezeti változóinak módosítása menüpontját kell kiválasztani a Start menüből. A megjelenő ablakban a Környezeti változók… gombra kell kattintani.



Környezeti változók

A JAVA\_HOME értéke könyvtár, ahova a JDK telepítve lett, tehát C:\Program

Files\Java\jdk-15.0.1, a PATH környezeti változó értéke %JAVA\_HOME%\bin legyen.



Java változók

Vigyázz, a környezeti változók szerkesztése után újra kell indítani a parancssort!

Sikeres telepítés és beállítás után parancssorba a java -version parancsot írva a következőt írja ki:

java version "15.0.1" 2020-10-20

Java(TM) SE Runtime Environment (build 15.0.1+9-18)

Java HotSpot(TM) 64-Bit Server VM (build 15.0.1+9-182, mixed mode, sharing)

Java platform (introjdk)

Elmélet

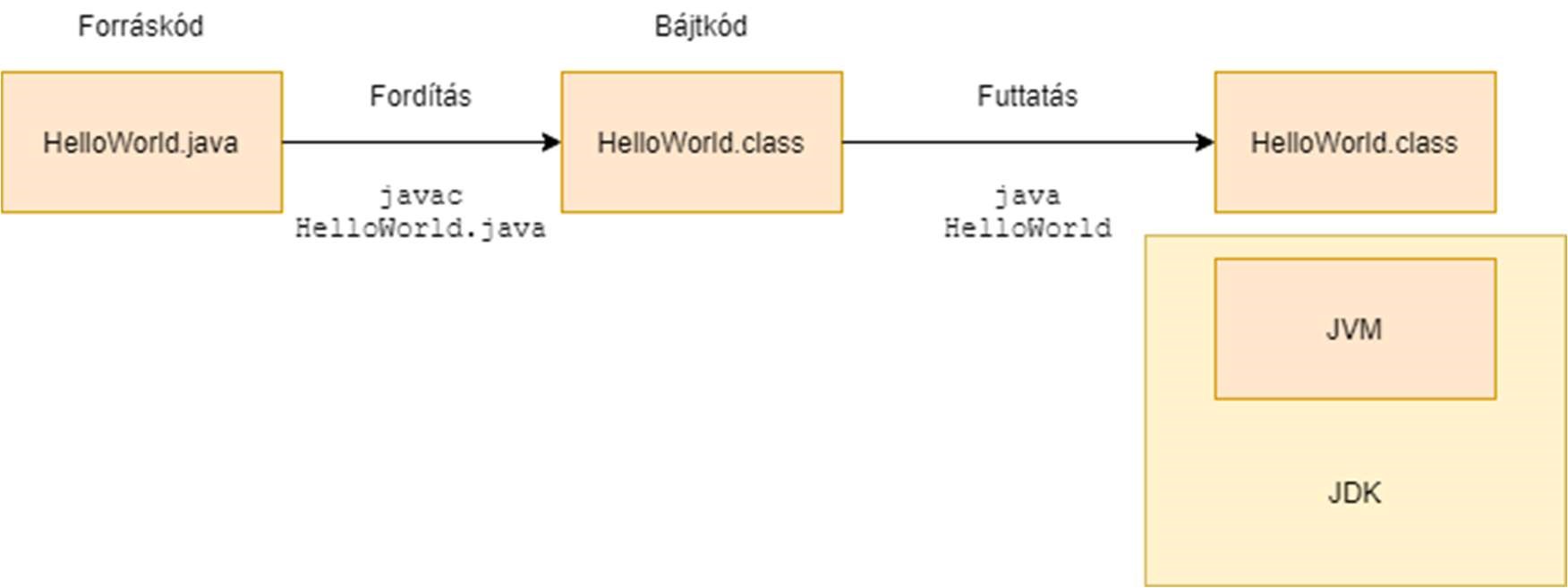
A Java imperatív programozási nyelv, ahol az alkalmazás utasítások sorozatából, lépésekből áll. Szöveges állomány a forráskód, melyet bájtkódra kell lefordítani. Ezt a fordítóprogram teszi meg, mely a JDK része. Ez a bájtkód a Java futtatókörnyezet, a Java Virtual Machine (továbbiakban JVM) gépi kódja. A JVM is a JDK része. A bájtkód gépi feldolgozása sokkal gyorsabb és biztonságosabb, mint a forrás közvetlen feldolgozása, ezért van szükség a fordításra.

A JVM felelős továbbá a platformfüggetlenségért is, ugyanis a bájtkódot másik platformra átmásolva az azon a platformon lévő virtuális gép módosítás nélkül tudja értelmezni, ezáltal elrejti a Java alkalmazás elől a tényleges platformot.

Az Oracle Java SE JDK-ban lévő JVM neve HotSpot.

A Java 11-es verziójától már nincs megkülönböztetve a JDK és JRE, a JRE-t már nem lehet külön letölteni. Régebben a JRE a JDK azon része volt, mely a futtatásért volt felelős, ezért a JVM-et tartalmazta, de a fordítóprogramot nem.

A forráskód létrehozásához elég egy szövegszerkesztő, és egy .java kiterjesztésű szöveges állományt kell létrehozni, mint pl. a HelloWorld.java. Ezt lefordítani a javac nevű parancssoros fordítóval lehetséges a javac HelloWorld.java parancs kiadásával. Ekkor megkapjuk a bájtkódot, ami a HelloWorld.class. Ezt lehet futtatni a java HelloWorld parancs kiadásával, mely elindítja a JVM-et.



Fordítás és futtatás folyamata

Ellenőrző kérdések

* Hogyan történik a fejlesztési folyamat?
* Hogyan biztosítja a Java a platform függetlenséget?
* Milyen eszközök szükségesek Java fejlesztéshez?

Feladat

Első Java program

Hozd létre egy üres introjdk könyvtárat, mondjuk a C:\training könyvtárban. A introjdk könyvtárban egy HelloWorld.java szöveges állományt! Ez a forráskód. Figyelj a kis- és nagybetűk közötti különbségekre! (Akár ki is másolhatod…) Ez a program a Hello World! szöveget írja ki a konzolra.

public class HelloWorld {

public static void main(String[] args) {

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

}

}

Fordítsuk le a javac HelloWorld.java paranccsal! Hatására létrejön a HelloWorld.class állomány.

Futtatni a lefordított állományt a java HelloWorld paranccsal lehet, mely hatására elindul a virtuális gép.

Teszt

Kérdés

A legszűkebb értelemben mi futtatja a Java bájtkódot?

* JDK
* JRE
* ☒ JVM
* Java fordító

A bájtkódot a JVM fordítja. A fordító a forráskódból készít bájtkódot. A JRE tartalmazza a JVM-et és az osztálykönyvtárat. A JDK az a JVM, osztálykönyvtár és olyan parancssori eszközök, mint pl. a fordító.

Kérdés

Hogyan lehet futtatni a Calculator.class fájlt?

* javac Calculator.java
* java Calculator.class
* javac Calculator
* ☒ java Calculator

A java parancs hívja meg a JVM-et futtatáshoz. A javac a fordítót hívja meg. A java parancsnak paraméterül csak az osztály nevét szabad megadni, a kiterjesztést nem. ### Maven (intromaven)

Elmélet

Sajnos a Java platform nem biztosít standard projektstruktúrát. A projekt nem más, mint a könyvtárak és fájlok halmaza. Itt főleg a forráskódokat tartalmazó állományokat tároljuk. A nyelv megjelenésekor minden projekt máshogy épült fel, más könyvtárakba csoportosították az alkalmazás felépítéséhez szükséges állományokat. A Java források fordításával, az így előállt bájtkód és az alkalmazás futtatásához szükséges egyéb állományok (un. erőforrás állományok, pl. szövegek, képek) összecsomagolásával jön létre maga az alkalmazás, ez egy jar kiterjesztésű állomány. Ezt a folyamatot nevezzük build folyamatnak.

A Maven egy kvázi szabványos eszköz a build folyamat megvalósításáért. Ezen kívül kezeli a függőségeket, hiszen egy alkalmazás fejlesztésekor számos más szervezet és gyártó által megírt nyílt és zárt forráskódú programot/könyvtárat használunk.

Ennél azonban többnek definiálja magát a Maven, un. software project management and comprehension tool. Betartja a Convention over configuration elvet, ami azt jelenti, ha a konvencióknak (megállapodásoknak) megfelelően járunk el, akkor az eszközt nem kell konfigurálni, hanem a build folyamatban az előzetes megállapodásoknak megfelelően fog eljárni.

Ez gyakorlatban annyit tesz, ha a fájlokat a megfelelő könyvtárakban helyezzük el, akkor különleges konfiguráció nélkül lefut a build folyamat, lefordításra kerülnek a forrás állományok, és összecsomagolásra az alkalmazás.

A build folyamat általában a következő nagyobb lépésekből áll:

* Forrás állományok fordítása
* Többi, ún. erőforrás állomány megfelelő helyre másolása
* Teszt esetekhez szükséges erőforrás állományok másolása
* Teszt esetek fordítása
* Teszt esetek futtatása
* Alkalmazás összecsomagolása

Jelen projektben még nincsenek automatikus tesztesetek, de későbbi projektekben készítünk ilyeneket is. (Az automatikus tesztesetek olyan programok, melyek az alkalmazás helyes működését ellenőrzik.)

A Maven projektet a pom.xml állomány írja le, mely a projekt gyökerében, azaz az állományainkat tartalmazó könyvtárban kell elhelyezni. A különböző állományokat konvenció szerint a következő könyvtárakba kell elhelyezni:

* src\main\java Java forráskódok
* src\main\resources Erőforrás állományok
* src\test\java Teszt esetek, nem része az alkalmazásnak
* src\test\resources Teszt esetekhez szükséges egyéb erőforrás állományok, nem része az alkalmazásnak

A következő példa bemutat egy minimális pom.xml állományt.

<?xml version="1.0" encoding="UTF-8"?>

<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">

<modelVersion>4.0.0</modelVersion>

<groupId>com.training360</groupId>

<artifactId>intromaven</artifactId>

<version>1.0-SNAPSHOT</version>

<properties>

<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>

<maven.compiler.source>15</maven.compiler.source>

<maven.compiler.target>15</maven.compiler.target>

</properties>

</project>

Ebben szerepelnek a projekt koordinátái, melyek egyedileg azonosítják a projektet, úgymint groupId, artifactId, version. Ezeket szabadon választhatjuk, a groupId tipikusan a cégünk neve, ahol dolgozunk, az artifactId a projekt neve. Az artifactId tipikusan megegyezik a könyvtár nevével, amely tartalmazza a projektet és pom.xml állományt.

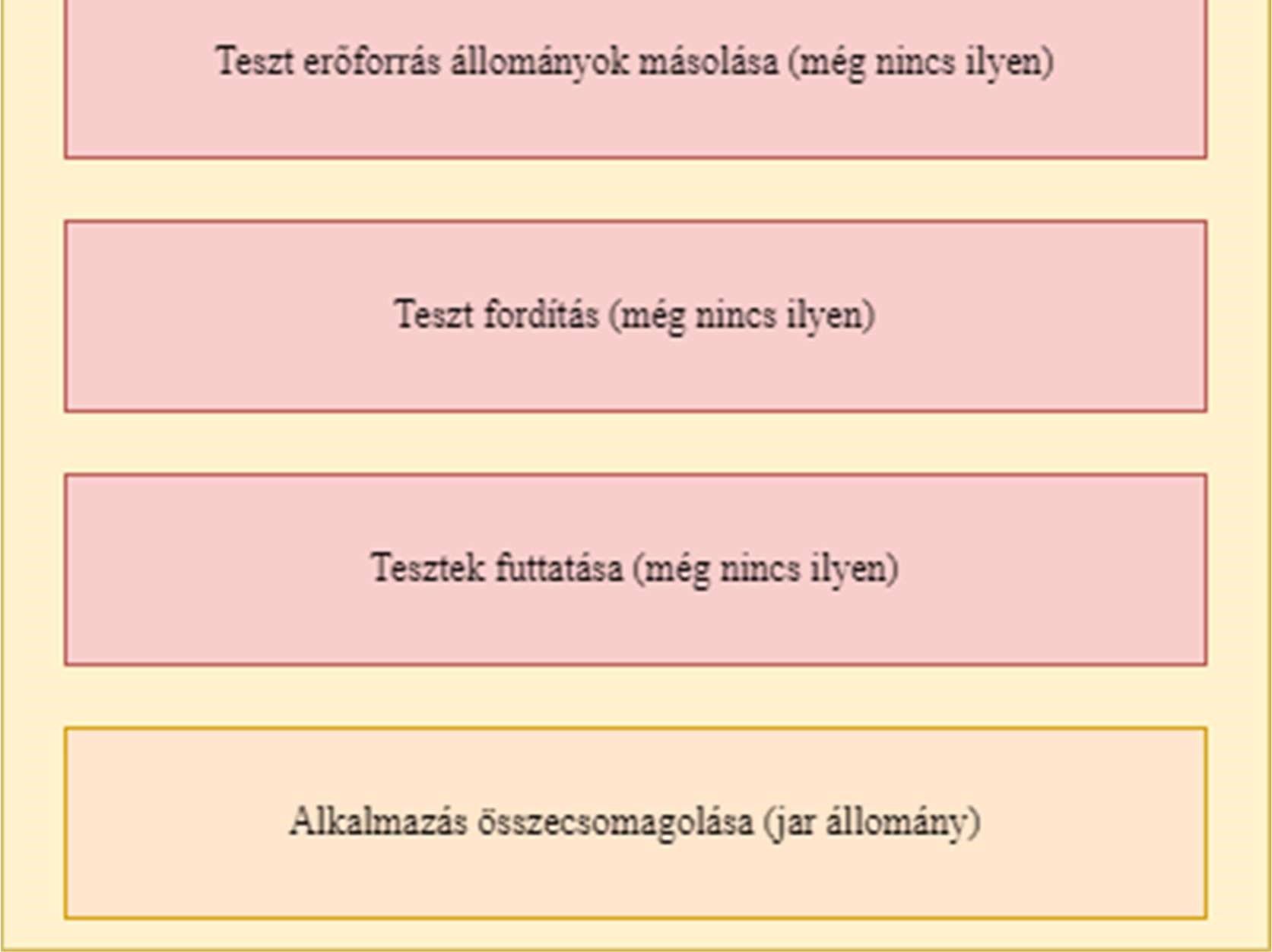
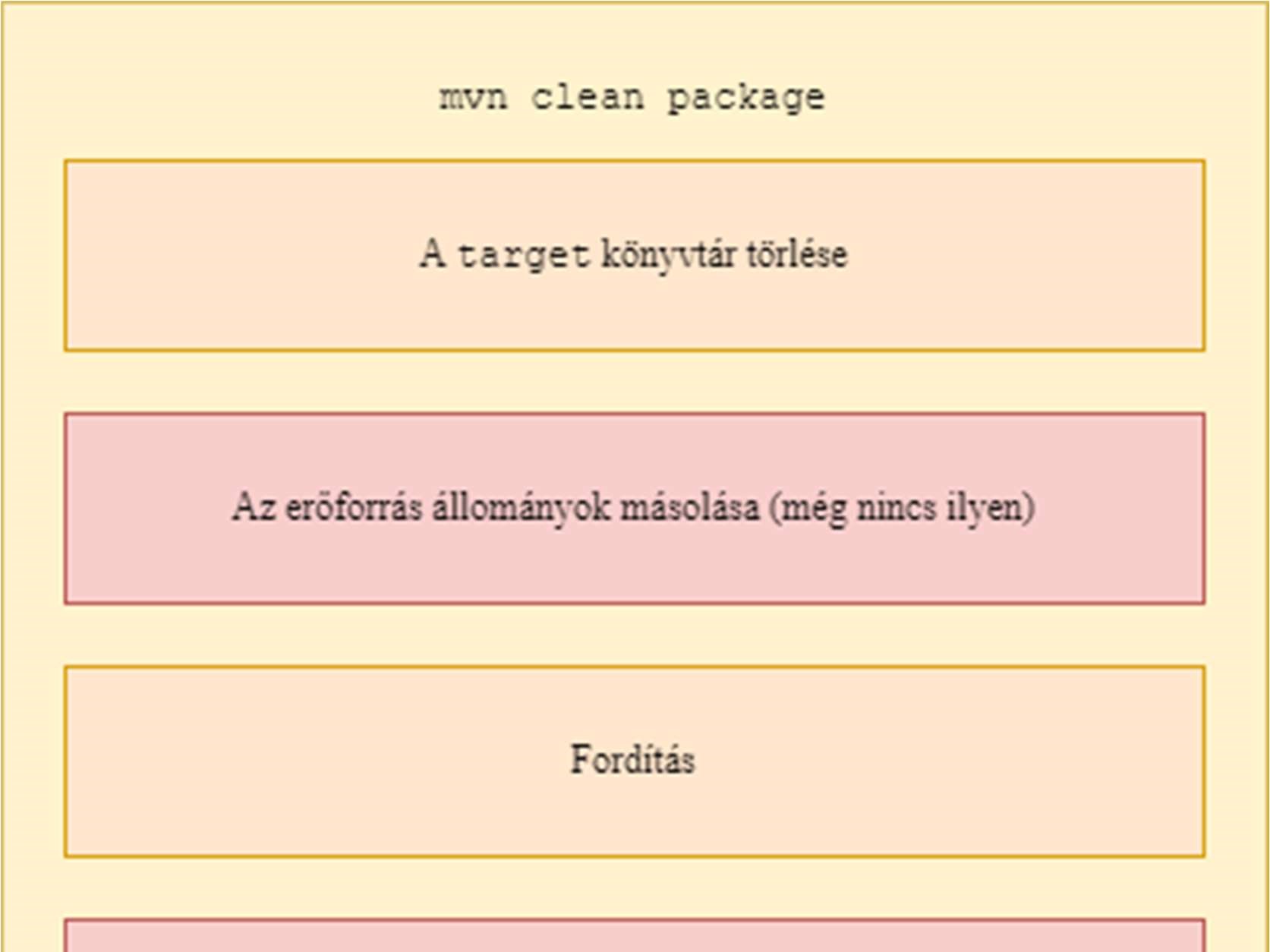
A verziószám pedig indulhat 1.0-SNAPSHOT verzióval. A SNAPSHOT azt jelenti, hogy az alkalmazás még fejlesztés alatt áll.

Érdemes még megadni a karakterkódolást, hiszen a fájl önmaga nem tartalmazza, hogy milyen karakterkódolással íródott a forráskód. Ez az ékezetes karakterek használatakor fontos. Ezt a project.build.sourceEncoding property-ben adható meg. Manapság érdemes mindenütt az UTF-8 karakterkódolást használni.

Tartalmaznia kell, hogy mely Java verzióval kompatibilis a forrás, és mely Java verzióra legyen fordítva. Ezeket a maven.compiler.source és maven.compiler.target property tartalmazza.

A videóban itt még a 1.8 érték szerepelt, használd itt a 15 értéket!

A build a mvn clean package paranccsal indítható. A target/classes könyvtárban létrejönnek a class kiterjesztésű állományok. A target könyvtárban létrejön a jar állomány.



Maven életciklus

Ellenőrző kérdések

* Mire használjuk a Maven-t?
* Tipikusan hogyan épül fel egy build folyamat?
* Milyen alkönyvtárakat tartalmaz a projekt könyvtár?
* Mi ír le egy projektet? Milyen elemek találhatók benne?

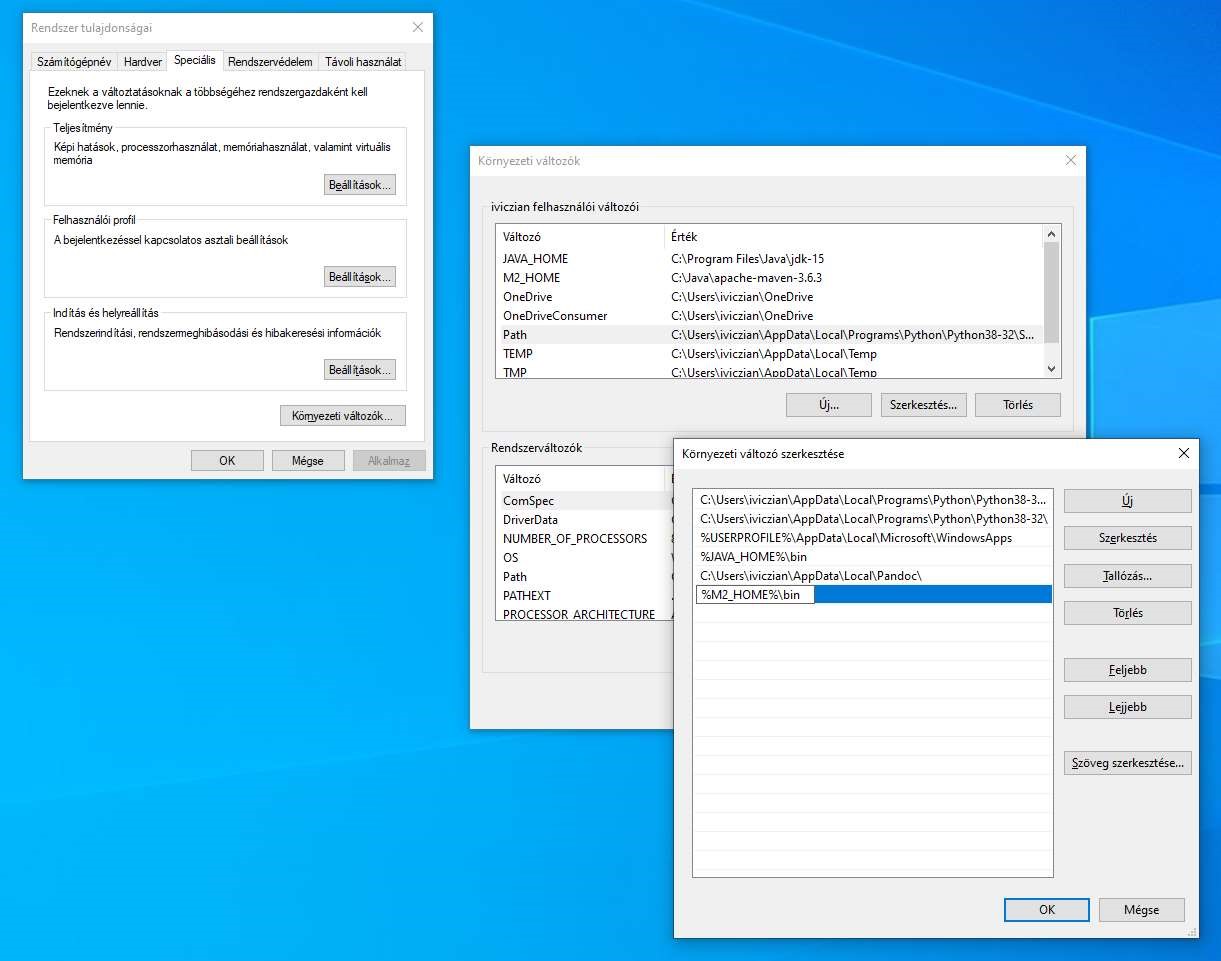
Feladat

Maven telepítése

Töltsd le a legfrissebb Maven Binary zip archive állományt a https://maven.apache.org/download.cgi címről, majd a letöltött .zip állományt tömörítsd ki a C:\Java könyvtárba!

A fájl neve pl. apache-maven-3.6.3-bin.zip.

Vedd fel az M2\_HOME környezeti változót, melynek értéke az a könyvtár, ahova a Maven ki lett csomagolva (példánkban C:\Java\apache-maven-3.6.3), és vegyük fel a PATH környezeti változóba a %M2\_HOME%\bin értéket is!



Környezeti változók

A telepítés és beállítás sikerességét a mvn -version parancs kiadásával ellenőrizzük! Valami hasonlót fog kiírni:

Apache Maven 3.6.3 (cecedd343002696d0abb50b32b541b8a6ba2883f) Maven home: C:\Java\apache-maven-3.6.3\bin\..

Java version: 15, vendor: Oracle Corporation, runtime: C:\Program

Files\Java\jdk-15

Default locale: hu\_HU, platform encoding: Cp1250

OS name: "windows 10", version: "10.0", arch: "amd64", family: "windows"

Első Maven projekt

Hozz létre egy üres könyvtárat intromaven néven, abban egy pom.xml állományt a következő tartalommal:

<?xml version="1.0" encoding="UTF-8"?>

<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">

<modelVersion>4.0.0</modelVersion>

<groupId>com.training360</groupId>

<artifactId>intromaven</artifactId>

<version>1.0-SNAPSHOT</version>

<properties>

<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>

<maven.compiler.source>15</maven.compiler.source>

<maven.compiler.target>15</maven.compiler.target> </properties>

</project>

Majd az src/main/java könyvtárban hozzunk létre egy HelloWorld.java állományt a következő tartalommal:

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

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

} }

Add ki az mvn clean package parancsot! Ha minden rendben történik, ezt írja ki:

[INFO] Scanning for projects...

[INFO]

[INFO] ---------------------< com.training360:intromaven >--------------------

[INFO] Building intromaven 1.0-SNAPSHOT

[INFO] --------------------------------[ jar ]-------------------------------- [INFO]

[INFO] --- maven-clean-plugin:2.5:clean (default-clean) @ intromaven ---

[INFO] Deleting C:\iviczian\Downloads\tmp\target

[INFO]

[INFO] --- maven-resources-plugin:2.6:resources (default-resources) @ intromaven ---

[INFO] Using 'UTF-8' encoding to copy filtered resources.

[INFO] skip non existing resourceDirectory

C:\iviczian\Downloads\tmp\src\main\resources

[INFO]

[INFO] --- maven-compiler-plugin:3.1:compile (default-compile) @ intromaven ---

[INFO] Changes detected - recompiling the module!

[INFO] Compiling 1 source file to C:\iviczian\Downloads\tmp\target\classes [INFO]

[INFO] --- maven-resources-plugin:2.6:testResources (default-testResources)

@ intromaven ---

[INFO] Using 'UTF-8' encoding to copy filtered resources.

[INFO] skip non existing resourceDirectory

C:\iviczian\Downloads\tmp\src\test\resources

[INFO]

[INFO] --- maven-compiler-plugin:3.1:testCompile (default-testCompile) @ intromaven ---

[INFO] No sources to compile

[INFO]

[INFO] --- maven-surefire-plugin:2.12.4:test (default-test) @ intromaven --

-

[INFO] No tests to run.

[INFO]

[INFO] --- maven-jar-plugin:2.4:jar (default-jar) @ intromaven --- [INFO] Building jar: C:\iviczian\Downloads\tmp\target\intromaven-1.0SNAPSHOT.jar

[INFO] -----------------------------------------------------------------------

[INFO] BUILD SUCCESS

[INFO] -----------------------------------------------------------------------

[INFO] Total time: 1.453 s

[INFO] Finished at: 2020-09-25T11:27:09+02:00

[INFO] -----------------------------------------------------------------------

A target/classes könyvtárban létrejön a HelloWorld.class állomány. Ez futtatható a következő paranccsal a projekt könyvtárában: java -classpath target\classes HelloWorld.

A target könyvtárban létrejön a intromaven-1.0-SNAPSHOT.jar állomány.

Teszt

Kérdés

Maven alkalmazása esetén konvenció szerint melyik könyvtárba kerülnek az alkalmazás forráskódjai?

* ☒ src\main\java
* src\java\main • src\test\java
* src\java\test

A konvenció szerint a forrásfájlok az src/main/java, az erőforrásállományok az src/main/resources könyvtárban vannak. A tesztesetekhez tartozó források a src/test/java, a tesztesetekhez tartozó erőforrásállományok az src/test/resources könyvtárban vannak.

Kérdés

Melyik állomány írja le egy Maven projekt tulajdonságait?

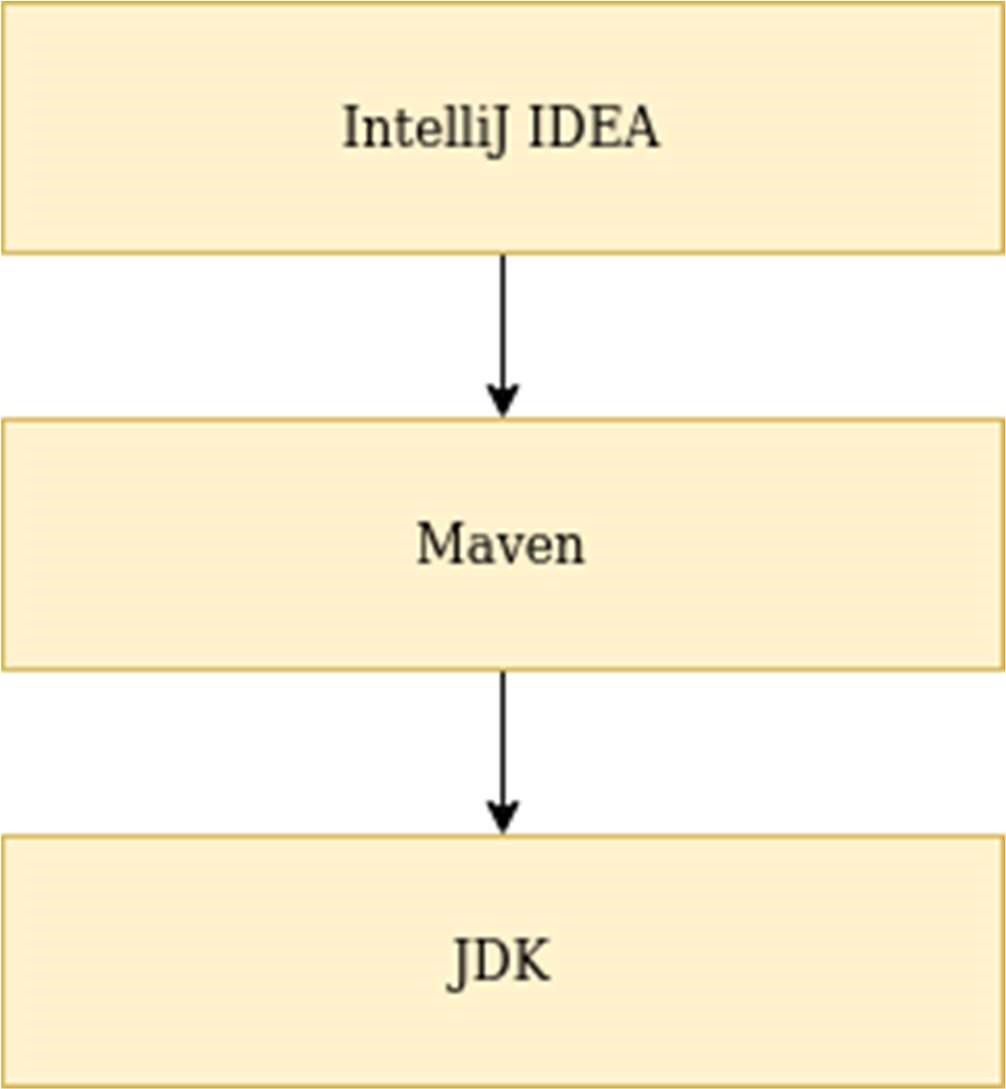
* mvn.java
* properties.mvn
* ☒ pom.xml
* pom.java

A Maven projekt leíró állománya a pom.xml XML formátumú állomány. ### IDE (introide)

Elmélet

Java fejlesztésre különböző fejlesztőeszközöket használunk, melyek ugyanúgy a JDK szolgáltatásaira épülnek, de grafikus felhasználói felülettel rendelkeznek, és egyszerűbbé teszik a forráskód szerkesztését, a build folyamat futtatását, és az alkalmazás tesztelését és elindítását.

A legelterjedtebb fejlesztőeszközök a NetBeans, Eclipse és a JetBrains IntelliJ IDEA. A NetBeans nyílt forráskódú, az Oracle berkeiben volt, de nemrég átkerült az Apache szervezethez. Az Eclipse fejlesztése régóta az Apache szervezeten belül történik, szintén nyílt forráskódú eszköz. A NetBeans egy kezdők számára jobban ajánlott eszköz, a különböző funkciók jobban integráltak, de kevésbé testre szabhatóak. Az Eclipse inkább haladó fejlesztőknek javasolt, pluginekkel tetszőlegesen bővíthető, és jobban testre szabható. A JetBrains egy fejlesztőeszközök gyártására szakosodott cég, és neki a terméke az IDEA, mely egy Java fejlesztőeszköz. Nagyon sok Javára épülő technológiát támogat. Két kiadása van, egy ingyen használható Community és egy nagyvállalati fejlesztésre szánt kereskedelmi Ultimate verzió.



Eszközök

A fejlesztőeszköz alapvető fogalma a projekt. Egy projekt állományok összessége: Java forrásállományok, erőforrás állományok stb. Létre lehet hozni új projektet, vagy meg lehet nyitni már létező projektet.

Egy projekt típusa lehet különbőző, attól függően, hogy milyen eszközöket szeretnénk használni.

Új projektet létrehozni a nyitóképernyőn, a New Project gombbal lehet. Maven projektet válasszunk ki. Majd meg kell adni a projekt nevét és a könyvtárát, amelyben tárolva lesz.

A videók az IntelliJ IDEA 2016.3.4-es verziójával lettek felvéve, azóta történtek változások. Az Artifact Coordinates megadása nem kötelező, hanem egy lenyíló panelen tudjuk ezeket megadni.

Ezután megjelenik a főablak. A projekt felépítését, fizikai elemeket (könyvtárakat, fájlokat) a jobb oldalon található project ablak mutatja.

Az alkalmazás új Java verzióval nem fog futni, mert be kell illeszteni a pom.xmlbe, hogy újabb Java verziót használunk.

A pom.xml-t kell módosítani, hogy frissebb Java verziót használunk:

<properties>

<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>

<maven.compiler.source>15</maven.compiler.source>

<maven.compiler.target>15</maven.compiler.target> </properties>

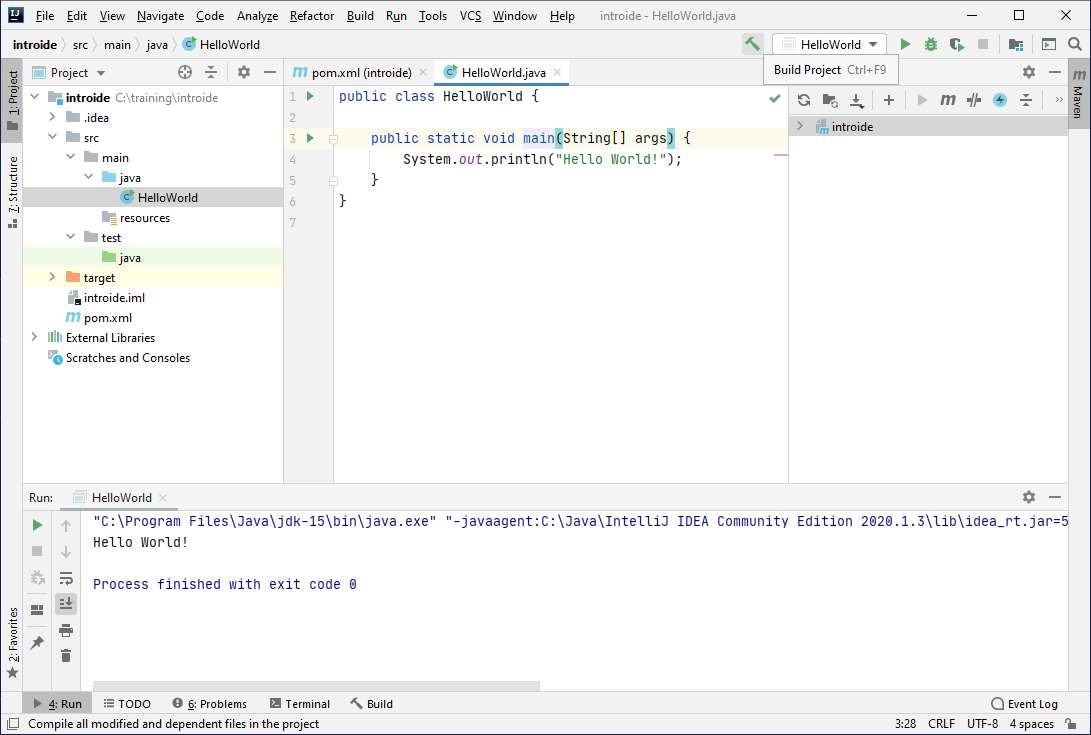
Az IDEA fejlesztőeszköz képes a Maven kezelésére, azonban a saját projekt fogalma, és a Maven projekt fogalma eltér. Ezért a kettőt szinkronizálja. Választhatunk, hogy a fejlesztőeszköz automatikusan tegye ezt, vagy kérdezzen rá, és nekünk kell a frissítést elvégezni, amennyiben módosítjuk a pom.xml állományt.

Azaz a pom.xml módosítása után a jobb oldalon lévő Maven panelt nyissuk ki, és nyomjuk meg a Reload All Maven Projects gombot!

Új .java állomány úgy hozható létre, hogy a bal oldalon található Project fülre kell kattintani. Itt le kell nyitni a projektet, azon belül az src\main\java könyvtárat, és jobb klikk New / Java class menüpontot kell kiválasztani. Ugyanis a .java állományok valójában osztályokat tartalmaznak.

A fejlesztő a legtöbb időt a kódszerkesztő ablakban tölti, mely különböző kisegítő funkciókkal támogatja a forráskód írását, ilyen a kód színezése, vagy az automatikus kódkiegészítés.

A fejlesztőeszköz lehetőséget biztosít az alkalmazás és a tesztesetek futtatására is.



IntelliJ IDEA

Fontos megjegyezni, hogy az IDEA-ban az src/main/java könyvtárnak Sources Root-nak kell lennie, míg az src/test/java könyvtárnak Test Sources Root könyvtárnak. Amennyiben a projekt megnyitásakor ezek már léteznek, az IDEA felismeri ezeket. Amennyiben később kerülnek létrehozásra, frissíteni kell a Maven konfigurációt. Ehhez a View / Tool Windows / Maven Projects ablakot kell előhozni, és ott megnyomni a Reimport All Maven Projects gombot.

Az IDEA leggyakrabban használt billentyűzet kombinációit a Help/Keymap reference tartalmazza, mely egy nyomtatható PDF állomány.

A leghasznosabb a Ctrl + Space, ami automatikus kódkiegészítést végez, valamint a Alt + Enter, mely a leggyakoribb hibákra próbál megoldási javaslatot adni. Gyakran használt kódrészleteket nem kell mindig begépelni, hanem erre un. code template-ek állnak a rendelkezésre. Ilyenkor egy rövidítést kell beírni, majd a Tab billentyűt lenyomni. A .java fájlon belül egy osztály található. Azon belül a main-t egy metódusnak nevezzük. A psvm + Tab a main() metódust generálja le. A sout + Tab billentyűkombinációval a System.out.println utasítást generáljuk le.

Ellenőrző kérdések

* Milyen Java IDE-kről hallottál?
* Miben segít nekünk az IDE?
* Mi a fejlesztés alapegysége, (a könyvtár, amiben a fájlok találhatóak)?
* Mondj néhány billentyűkombinációt!

Feladat

IDEA telepítése

Ellenőrizd, hogy az IntelliJ IDEA fejlesztőeszköz telepítve van-e a gépeden! Az Asztalon, Tálcán, vagy a Start menüben meg kell jelennie.

Amennyiben nincs, az IDEA letölthető a https://www.jetbrains.com/idea/#chooseYourEdition címről, és a Community verziót kell letölteni és feltelepíteni.

Érdemes a C:\Java könyvtárba telepíteni.

A telepítés történhet az alapértelmezett beállításokkal, azon módosítani nem szükséges.

Új projekt

Hozz létre egy új projektet introide néven a C:\training\introide könyvtárba! Módosítsd megfelelően a pom.xml fájlt! Hozz létre benne egy HelloWorld osztályt, mely kiírja a Hello World! szöveget! Futtasd az alkalmazást!

Teszt

Kérdés

Melyik nem Java fejlesztőeszköz?

* IntelliJ IDEA
* NetBeans
* ☒ Microsoft Visual Studio
* Eclipse

A három legelterjedtebb Java fejlesztőeszköz a NetBeans, Eclipse és IntelliJ IDEA. A Microsoft Visual Studio a Microsoft fejlesztőeszköze, mely leginkább C# programozásra használható.

Kérdés

Mire kell odafigyelni Maven fejlesztéskor?

* Be legyen állítva a pom.xml-ben a Java verziója
* Ha már létező projektet töltünk be, és kézzel hozzuk létre a könyvtárakat, akkor frissítsük a Maven projektet
* Ha módosítjuk a pom.xml fájlt, akkor újra kell töltenünk a projektet, ha nem automatikus újratöltést választottunk
* ☒ Mindháromra

Ahhoz, hogy 5-ösnél újabb Javaban fejlesszünk, be kell állítani azt a pom.xml-ben. Ha nincs src könyvtár a projektben, akkor előbb azt létre kell hozni, majd frissíteni kell a projektet. Amennyiben módosítjuk a pom.xml állományt, és nem az automatikus újratöltést választottuk, akkor frissíteni kell a projektet, hogy az IDE újraolvassa a pom.xml fájlt. ### Git használata IDE-ben (introgit)

Elmélet

Az IDEA fejlesztőeszköz beépítetten támogatja a Git használatát. Lehetőség van új projekt esetén azt verziókezelő rendszerben tárolni, vagy egy projektet úgy megnyitni, hogy közvetlen verziókezelő rendszerből kerüljön letöltésre és megnyitásra.

Ahhoz, hogy egy új projekt Git alatt legyen tárolva, a git init parancsot kell kiadni. Ez felületről is megoldható a VCS / Enable Version Control Integration menüpont használatával.

Alul található a Git panel. Ezen ablak mutatja az újonnan létrehozott, módosított és törölt állományokat. Itt lehet az állományokat a Githez hozzáadni, commitolni. A Revert művelettel lehet a saját módosításainkat elvetni.

Amennyiben a Git fülnél a Local Changes opció nem látszik, következő beállítással lehet előcsalogatni: File – Settings – Version Control – Commit lapon a Use non-modal commit interface pipát kiszedve megjelenik a videóban látható ablak.

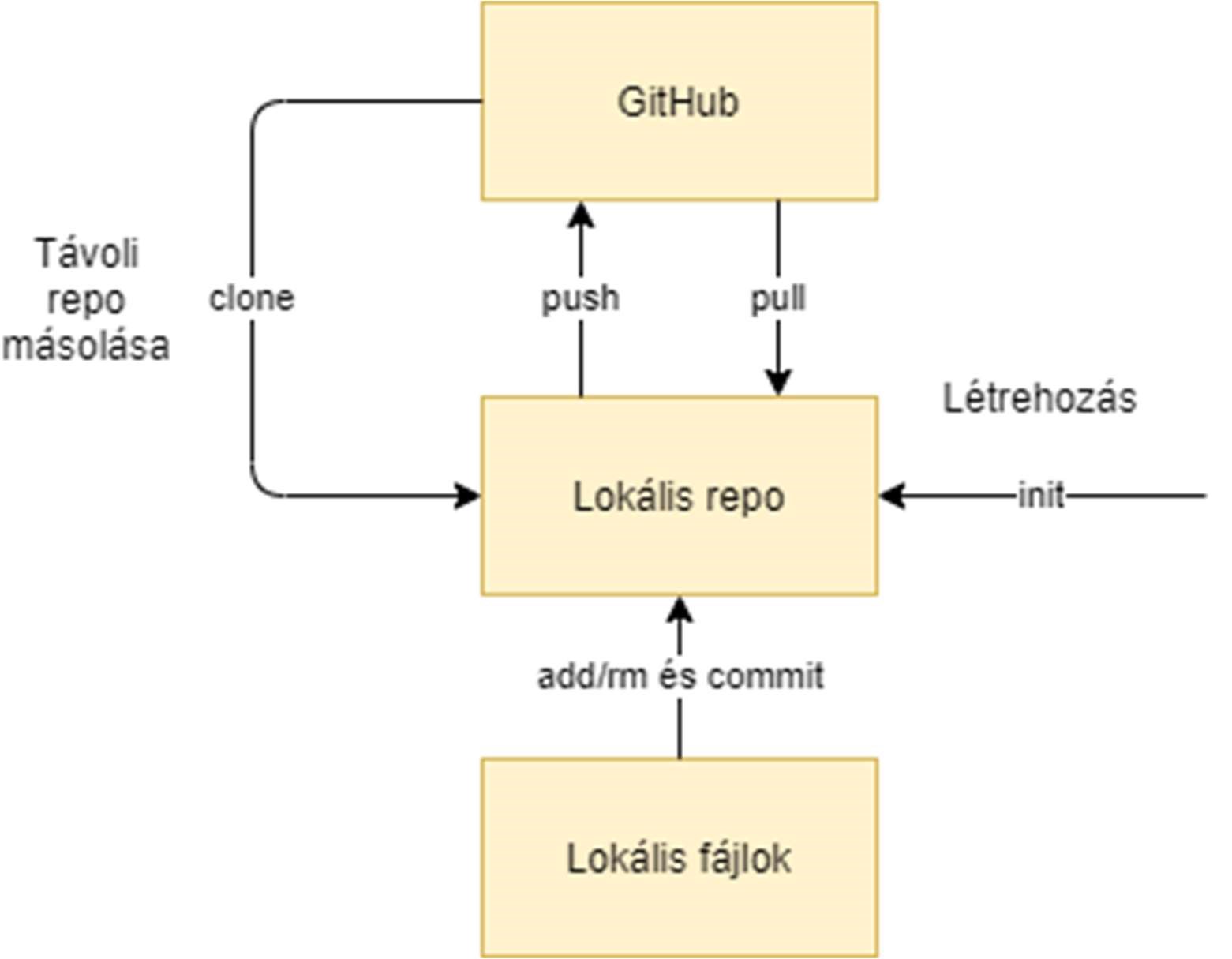
A pom.xml, .gitignore, és a \*.java állományokat adjuk hozzá. A target könyvtárba dolgozik a Maven, az .idea könyvtár tartalma és a .iml fájl az IDEA-hoz tartozik, azt sose commitoljuk. Ehelyett helyezzük el a .gitignore fájlban.

target .idea

\*.iml

A verziókezelő rendszerhez, azaz esetünkben a Githez kapcsolódó parancsokat a VCS / Git menüpont alatt lehet megtalálni, ebből a Remotes…, Pull… és Push… a leggyakrabban használt.

Alul megtalálható a Git panel is, mely információkat ad a Git műveletekről és a repository-ról.



Git műveletek

Bár érdemes az IDEA fejlesztőeszközt használni a Git műveletek végrehajtására, tisztában kell lenni azzal, hogy parancssorban hogyan lehet ezeket elvégezni. Gyakran lehet olyan, hogy vagy nincs kéznél fejlesztőeszköz, vagy parancssorból kell ellenőrizni, hogy a fejlesztőeszköz jól működik-e, esetleg speciális parancsokat kell kiadni.

Ellenőrző kérdések

* Hogyan támogatja a fejlesztőeszköz a Git használatát?

Feladat

Első commitok

Hozz létre egy új projektet introgit néven, abban egy HelloGit Java osztályt, ami kiírja, hogy Hello Git!!

Hozz létre egy lokális Git repository-t, és commitold a megoldásodat!

Módosítsd a Java osztályt, hogy most már azt írja ki, hogy Hello IDEA and Git!.

Commitold a megoldásodat!

Teszt

Kérdés

Melyik állományokat érdemes beírni a .gitignore állományba?

* target könyvtár
* idea könyvtár és a \*.iml fájl
* ☒ Mindkettő
* egyik sem

A target könyvtár a Maven munkakönyvtára. Az .iml állomány és az idea könyvtár tartalmazza a fejlesztőeszköz beállításokat és a konfigurációt, ez fejlesztőnként egyedi lehet. Ezeket nem szabad feltölteni a Git repo-ba, ezért érdemes betenni a .gitignore fájlba.

Kérdés

Melyik paranccsal lehet a lokális repository-ban lévő fájlokat eljuttatni a távoli repository-ba?

* ☒ push
* pull
* commit
* add

Az add hozzáadja az állományokat a repo-hoz. A commit a lokális repo-ba tölti fel. A pull a távoli repo-ból tölti le. A push tölti fel a módosításokat a távoli repo-ba. ### Feltöltés GitHubra (introgithub)

Elmélet

Ahhoz, hogy a projektet meg lehessen osztani GitHubon, a webes felületen létre kell hozni egy új üres projektet. Ennek az URL-jét kell kimásolni, és a VCS / Git/ Remotes menüpontban kell origin néven felvenni a lokális repositoryban távoli repository-ként.

Ahhoz, hogy működjön az azonosítás, a Settings menüpontban, a Git ablakban a Use credential helper checkboxot be kell klikkelni. Majd újra a \_VCS / Git / Push\_\_ menüpontban lehet a módosításokat push-olni a távoli repository-ba.

Commitnál is lehet egyből push-olni, ha a Commit gomb melletti nyillal kiválasztod, hogy Commit and push.

Ellenőrző kérdések

* Milyen lépések kellenek, hogy a GitHubon meg tudd osztani a projektedet?
* Mi az a push? • Mi az az origin?
* Mi a különbség a local és a remote repository között?

Feladat

Első push

Hozz létre egy új projektet introgit néven, abban egy HelloGit Java osztályt, ami kiírja, hogy Hello Git!!

Ha még nem vagy regisztrálva, regisztráld magad a GitHubon!

Hozz létre egy local Git repository-t, és commitold a megoldásodat! Majd hozz létre egy GitHub repository-t, vedd fel origin néven távoli repository-ként és push-old a megoldásodat!

Második push

Írd át a HelloGit osztályban, hogy azt írja ki, hogy Hello IDEA and Git!!

Commitold és egyben push-old a megoldásodat!

Teszt

Kérdés

Melyik paranccsal lehet a lokális repository-ban lévő módosításokat eljuttatni a távoli repository-ba?

* ☒ push
* pull
* commit
* add

Az add hozzáadja az állományokat a repo-hoz. A commit a lokális repo-ba tölti fel. A pull a távoli repo-ból tölti le. A push tölti fel a módosításokat a távoli repo-ba. ### Eszközkészlet összefoglalás (summarytools)

Ebben a leckében áttekintettük a Java nyelv kialakulásának történetét, és hogy milyen eszközöket is fogunk használni.

A Java egy objektumorientált programozási nyelv, melyet leginkább nagyvállalati backend rendszerek fejlesztésére használnak.

A Java nyelv mögött álló legnagyobb cég jelenleg az Oracle. És ennek a Java SE Java Development Kit futtató és fejlesztőkörnyezetét fogjuk használni.

Fejlesztés közben a forráskódot kell megírni, és azt a fordítóval lefordítani bájtkóddá. Ezt a bájtkódot futtatja a Java virtuális gép, a JVM.

A projektkezelést a Maven végzi. A konfigurációs fájlja a pom.xml, ebben lehet definiálni a projekt tulajdonságait, és a függőségeket. A Maven vezérli a build folyamatot is, fordít, futtatja a teszteseteket és csomagol.

Fejlesztőeszközként az IntelliJ IDEA Community verzióját fogjuk használni. Ez segít a forráskódok szerkesztésében, hibák megtalálásában, fordításban és futtatásban. A JDKra és a Mavenre épül. Érdemes a gyakori billentyűzetkombinációkat megjegyezni. Természetesen Git integrációt is tartalmaz. ## Java osztályok ### Kiírás és beolvasás konzolról (classstructureio)

A Java nyelv alap építőköve az osztály. Minden alkalmazás osztályokból épül fel. Tipikusan egy fájlba egy osztályt írunk, és a fájl neve meg kell egyezzen az osztály nevével. (Megj.: Ha több osztályt írunk egy fájlba, akkor csak egyetlen publikus osztály - public módosítószóval ellátott - lehet benne, és ennek a neve kell megegyezzen a fájlnévvel.)

Ez egy egyszerű osztály:

public class HelloWorld {

}

Látszik, hogy a Java forráskód struktúráját kapcsos zárójelekkel adjuk meg. Ezek un. blokkokat képeznek.

Sok osztályból álló alkalmazásnak csomagokkal tudunk belső struktúrát adni. A csomagok nem csak könnyebbé teszik az áttekintést, de az osztályok láthatóságát is szabályozzák. A csomagok fizikailag könyvtárként jelennek meg.

A csomagot meg kell adni az első utasításban a package kulcsszóval.

package training;

public class HelloWorld {

}

Az osztály tagjai között vannak, amelyek adatokat tárolnak és vannak, amelyek utasításokat fognak össze. Ez utóbbiak a metódusok.

Látható, hogy az eddigi utasítások hatására még nem történt semmi, az ilyen utasítások a deklarációs utasítások. Ezek csak arra valók, hogy struktúrát adjanak a programunknak, vagy valaminek nevet adjanak, amire később tudunk hivatkozni.

Az alkalmazáson belül azon osztályok futtathatóak, melyek tartalmaznak main() metódust. Egy alkalmazásban akár több ilyen is lehet, azonban tipikusan egy szokott lenni, ez az alkalmazás fő belépési pontja, ezzel indítjuk el az alkalmazást. Azonban tanuláskor több osztálynak is készíthetünk main() metódust, tesztelve osztályunk működését. A JVM a main() metódust csak akkor találja meg, ha megfelel bizonyos szabályoknak. Nézzük, hogy kell kinéznie egy main metódusnak az osztályon belül:

package training;

public class HelloWorld {

public static void main(String[] args) {

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

}

}

Látható, hogy itt a struktúra miatt a blokkokat egymásba ágyazzuk. Hiszen az osztály tartalmazza a metódust.

Ez az osztály futtatható, a training csomagban van és a fájl neve HelloWorld.java. Az osztály futtatásával a main() metódusban lévő utasítások szépen sorban végrehajtódnak. A HelloWorld osztály esetében ez egyetlen egy utasítást jelent, mely a konzolra kiírja a Hello World! szöveget.

Az IntelliJ IDEA fejlesztőeszköz számos template-et tartalmaz a gyakran használt kódrészletek gyors legenerálására. Ezek a template-ek rövid szavakkal elérhetőek, melyek begépelése után a Tab billentyű lenyomására az elmenett kódrészlet jelenik meg.

Ilyen például a main() metódust legeneráló psvm rövidítés. A System.out.println() utasítás szintén elérhető template-ből a sout rövidítéssel. További template-eket találsz a File / Settings… menüpont alatt megjelenő ablak Editor/Live Templates lapján.

Kommunikáció a felhasználóval

Az alkalmazások valamilyen felületen át kommunikálnak a felhasználóval. Az üzenetek

megjelenítésének legegyszerűbb módja, ha azt a konzolra kiírjuk. Ezt a System.out.println() metódus hívásával tehetjük meg. A zárójelei között paraméterként a kiírandó üzenetet kell megadnunk. Ez legtöbbször szöveg, de lehet más típusú adat is.

Ha szeretnénk valamilyen adatot bekérni a felhasználótól, akkor azt a Scanner osztállyal tehetjük meg. Ehhez először példányosítanunk kell egy objektumot az osztály alapján a new Scanner(System.in) utasítással, és a kapott objektumot el kell tárolni egy változóban. Ezek után a Scanner objektum metódusait használva különböző típusú adatokat tudunk beolvasni.

* nextLine(): az Enter lenyomásáig bevitt szöveget olvassa be
* nextInt(): egész szám beolvasására használható Egy szöveg beolvasható a következő kóddal:

package training;

import java.util.Scanner;

public class HelloWorld {

public static void main(String[] args) { Scanner scanner = new Scanner(System.in);

System.out.println("What's your name?");

String name = scanner.nextLine();

System.out.println(name);

}

}

A Scanner scanner egy változódeklaráció, ahol a scanner a változó neve, a Scanner annak típusa. A típust kötelező megadni, mert a Java egy szigorúan típusos nyelv, ami azt jelenti, hogy nem lehet akármilyen értéket adni egy változónak, csak olyat, amit a típus a megenged. Majd az egyelőségjellel értéket adunk neki.

A String name ugyanígy egy változódeklaráció, ahol a name a változó neve, a String annak típusa (karakterlánc). Majd az egyelőségjellel értéket adunk neki, méghozzá úgy, hogy beolvassuk a konzolról.

A kód tehát beolvassa a nevet, majd kiírja a konzolra. A nevet az IDEA-ban alul, a Run ablakban lehet megadni.

Mivel a Scanner osztály a java.util csomagban található, ezért importálni kell az osztály elején. Ezt nem szükséges beírni, hanem amikor az osztály nevét (Scanner) írjuk a main() metódusban, akkor az IDEA felajánlja, hogy automatikusan beimportálja. Ehhez nyomjuk meg az Alt + Enter billentyűkombinációt (amikor az osztály neve piros, és alá van húzva). Ekkor elhelyezi az IDEA az import utasítást az osztály elején a csomagdeklaráció (package) alatt.

A következő példa azt is megmutatja, milyen műveleteket lehet végezni a String és int típusú értékekkel.

package training;

import java.util.Scanner;

public class HelloWorld {

public static void main(String[] args) { Scanner scanner = new Scanner(System.in);

System.out.println("What's your name?");

String name = scanner.nextLine();

System.out.println("Hello " + name); // 1

System.out.println("What's your year of birth?"); int yearOfBirth = scanner.nextInt();

System.out.println("Year of birth: " + yearOfBirth); // 2

System.out.println(2019 - yearOfBirth); // 3

}

}



Kiírás és beolvasás

A int yearOfBirth utasítással egy int típusú, yearOfBirth nevű változót deklarálunk. Ennek értéke tehát csak egy egész szám lehet.

Az // 1 jelzéssel ellátott sor mutatja, hogy kell két karakterláncot összefűzni. A // 2 jelzéssel ellátott sor mutatja, hogy egy karakterlánchoz egy egész szám is hozzáfűzhető. Ekkor a szám először automatikusan szöveggé kerül átalakításra. A // 3 jelzéssel ellátott sor mutatja, hogy lehet két egész számmal kivonás műveletet elvégezni.

Egyszerűnek tűnik, ugye? Azonban akadhatnak problémák, amikbe előbb utóbb belefutsz. A nextInt() nem olvassa be az Enter leütésével odakerült sortörés karaktert, így az a következő nextLine() hívást megzavarja.

Mire figyelj a Scanner használatakor: a nextLine() a teljes szöveget beolvassa, de a sorvége jelet eldobja, míg a nextInt() csak az első láthatatlan karakterig (pl. szóköz, sorvége jel) olvas, és azt ott hagyja. Éppen ezért, ha szám beolvasása után egy szöveget akarsz olvasni, akkor azt fogod tapasztalni, hogy ez a szöveg üres lesz, és csak a második olvasás ad eredményt.

public static void main(String[] args) { Scanner scanner = new Scanner(System.in);

System.out.println("How old are you?"); int age = scanner.nextInt();

scanner.nextLine(); //Ez fogja az ottmaradt sorvége jelet beolvasni

System.out.println("What's your name?");

String name = scanner.nextLine();

System.out.println(name);

}

Ellenőrző kérdések

* Mik a Java alkalmazások alapvető építőkövei?
* Egy Java osztály mit tartalmazhat, amikben utasításokat lehet írni?
* Hogyan lehet konzolról adatokat bekérni? Melyik osztályra van ehhez szükségünk?
* Mivel a Scanner másik csomagban található, mit kell csinálni a használatához?
* Hogyan lehet a konzolra adatokat kiírni?
* Milyen probléma merülhet fel az egész szám beolvasásakor?

Feladat

A videóban szereplő feladatok a demos/src/main/java/classstructureio elérési úton vannak.

Hozz létre egy új projektet a C:\training\training-solutions könyvtárba, és innentől ebbe a projektbe dolgozz külön csomagokban! Hozz létre egy training-solutions repository-t a GitHub accountodon. Ide pushold a megoldásaidat!

Figyelj arra, hogy külön feladatmegoldások mindig külön commitban legyenek!

A feladatok megoldásai a solutions/classstructureio elérési úton vannak.

Számológép

Készíts egy Calculator osztályt a classstructureio csomagba! A main metódusban kérj be a felhasználótól két egész számot! Az első sorban írd ki a műveletet a következő formátumban: 5 + 10! A második sorban írd ki az eredményt (15)!

Regisztráció

Készíts egy Registration osztályt a classstructureio csomagba! A main metódusban kérd be a felhasználótól a nevét és az email címét, majd írd ki, hogy milyen adatokkal regisztrált!

Teszt

Kérdés

A Scanner osztály melyik metódusa tud bármilyen szöveget beolvasni?

* ☒ nextLine()
* nextInt()
* nextDouble()
* nextline()

A Scanner nextLine() metódusa olvassa be a következő sort. CamelCase a metóduselnevezés, azaz a szóhatáron nagybetűk vannak. Ez nem igaz az első karakterre, ugyanis az kisbetű, mivel a Javaban a metódusneveket kisbetűvel kell kezdeni. Ezért nem jó a nextline() metódus. A nextInt() metódussal egész számot, a nextDouble() metódussal lebegőpontos számot lehet beolvasni.

Kérdés

A Scanner osztály melyik metódusa tud egész számot beolvasni?

* nextLine()
* ☒ nextInt()
* nextNumber()
* nextInteger()

A Scanner nextLine() metódusa olvassa be a következő sort. A nextInt() metódusa egész számot olvas be. A másik két metódus nem létezik.

Kérdés

Melyik utasítással lehet a konzolra kiírni a Hello szöveget?

* System.in.writeln("Hello")
* System.out.writeln("Hello")
* System.in.println("Hello")
* ☒ System.out.println("Hello")

A System.out.println() metódus hívásával lehet szöveget kiírni a konzolra.

Objektumok és attribútumok (classstructureattributes)

Egy Java alkalmazás a futás közben létrejövő és egymással kommunikáló objektumok összessége. Ezen objektumok az osztályok alapján készülnek el. Az objektumok tárolhatnak adatokat, melyeket attribútumnak nevezünk. Az azonos típusú objektumok ugyanolyan mintára készülnek, ezért ugyanolyan típusú adatok tárolására alkalmasak.

Egy objektum állapotán az attribútumok aktuális értékeinek összességét értjük. Gondoljunk úgy az osztályra, mint egy tervrajzra, ami alapján elkészítjük, más szóval példányosítjuk az objektumokat. A példányosítás a new kulcsszóval történik. Ezért az objektumra a példány szót is szoktuk használni. Egy osztályhoz bármennyi példányt tudunk készíteni. (Hasonlóan ahogy egy tervrajz alapján több házat meg tudunk építeni.) Látható, hogy eddig a Scanner objektumot is példányosítottuk:

Scanner scanner = new Scanner(System.in);

Attribútumot a típusa és a neve megadásával deklarálhatunk egy osztályon belül:

public class Trainer {

String name;

int yearOfBirth;

}

Példányosítás után az adott objektumot értékül adjuk egy változónak, hogy később tudjunk rá hivatkozni. Az objektum attribútumát a pont operátorral érhetjük el.

Ahhoz, hogy kipróbáljuk az osztályunkat, egy új ClientMain osztályt hozunk létre. A main() metódust akár a Client metódusban is létrehozhatnánk, de így jobban látható, hogy hogyan használható egy osztály a másikból.

public class ClientMain {

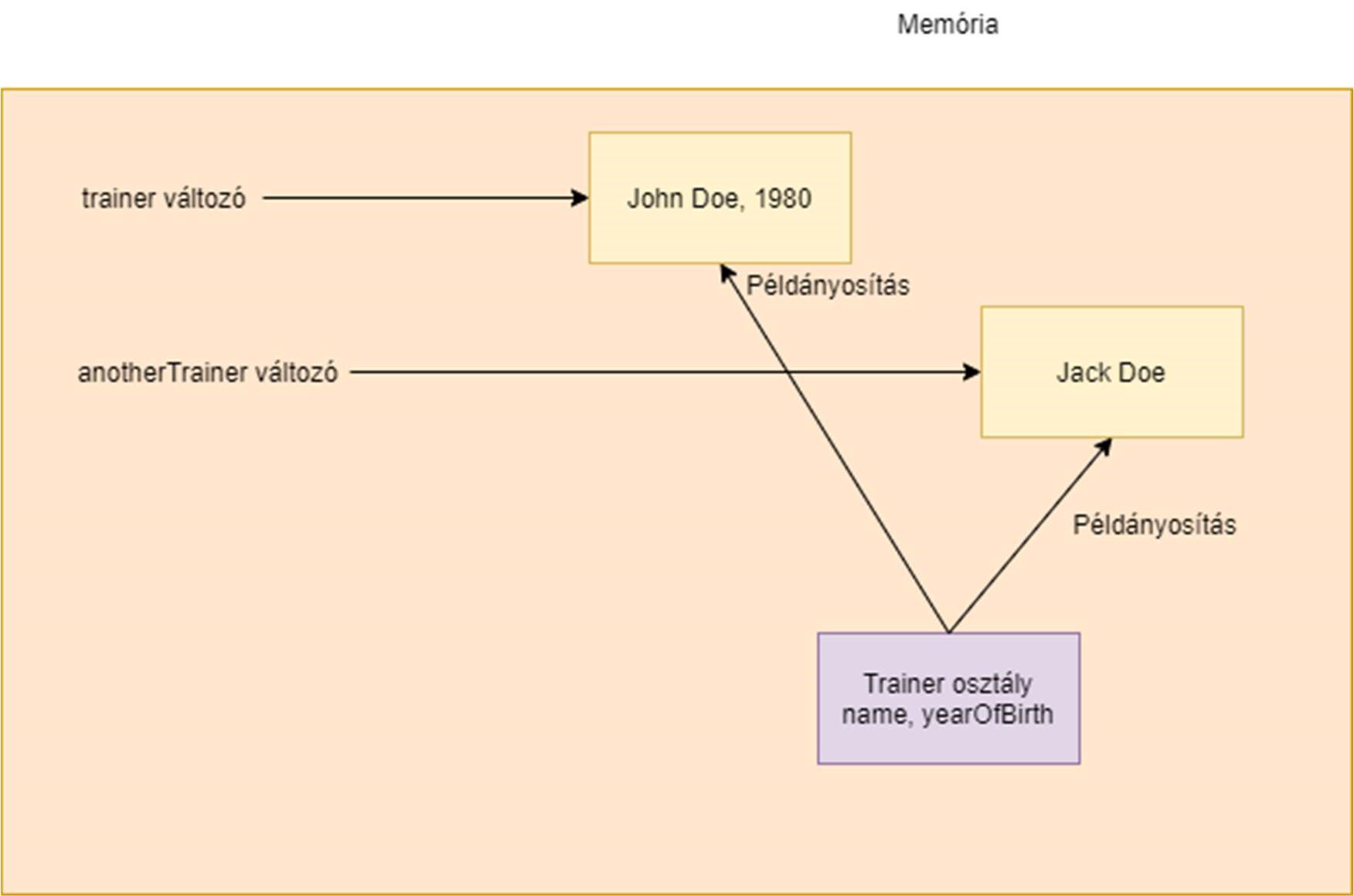
public static void main(String[] args) { Trainer trainer = new Trainer(); // 1 trainer.name = "John Doe"; trainer.yearOfBirth = 1980;

Trainer anotherTrainer = new Trainer(); // 2 anotherTrainer.name = "Jack Doe";

}

}

A példában látható, hogy az // 1 jelzéssel ellátott sorban példányosítunk egy Trainer objektumot, és beállítjuk a nevét és a születési évét. Ez lesz az objektum állapota. Majd a // 2 jelzéssel ellátott sorban példányosítunk egy másik Trainer objektumot, ennek más nevet állítunk be.



Objektumok

A példányosítás során létrejön az objektum, és lefoglalásra kerül a JVM memóriájában. Ez a new Trainer() utasítás, és önállóan is megállja a helyét. Az értékadással (Trainer trainer =) csak hozzárendeljük egy változóhoz. Ezért is szerepel az ábrán csak egy címkeként, mert a változó önmagában csak egy név, mellyel az adott objektumra tudunk a későbbiekben hivatkozni.

A változón keresztül tudunk neki értéket adni. Amikor az objektum létrejön, még üres a name attribútumának értéke, és a yearOfBirth értéke 0. A trainer.name = "John Doe" értékadással adunk az attribútumának értéket, mely már így lesz letárolva a memóriában.

Ellenőrző kérdések

* Mi a kapcsolat az osztály és példány között?
* Mi az az attribútum?
* Mit értünk egy objektum állapota alatt?

Feladat

A videóban szereplő feladatok a demos/src/main/java/classstructureattributes elérési úton vannak.

A classstructureattributes csomagba dolgozz!

A feladatok megoldásai a solutions/classstructureattributes elérési úton vannak.

Ügyfél osztály

Hozz létre egy Client osztályt, melynek három attribútuma van: név (name), születési év (year) és cím (address). Típusaik rendre String, int és String.

Hozz létre egy main() metódust a ClientMain osztályba, amelyben kipróbálod a Client osztály működését. Példányosítani kell egy objektumot a Client osztály alapján, majd kérd be az attribútumok értékét a felhasználótól. Ellenőrzésképp írd ki minden attribútumának értékét a konzolra!

Zeneszámok

Készíts egy Song osztályt, melyben eltárolhatod egy dal előadóját (band), címét (title) és a hosszát (length) percben!

Készíts main() metódust egy Music osztályba, ahol kérd be a felhasználótól a kedvenc zeneszáma adatait! Ellenőrzésképp írd ki a megadott adatokat előadó - cím (hossz percben) formában, azaz Britney Spears - Oops!...I Did It Again (4 perc)!

Forrás

OCA - Chapter 1/Understanding the Java Class Structure, Writing a main() Method

Teszt

Kérdés

Válaszd ki a HAMIS állítást!

* Egy osztályból több példány hozható létre
* ☒ Egy példányból több osztály hozható létre
* Az objektum és példány ugyanaz
* A példányosítás kulcsszava a new

Egy osztályból több példány hozható létre. Példányból nem lehet osztályt létrehozni. Az objektum és a példány rokonértelmű szó. A példányosítás kulcsszava a new.

Kérdés

Válaszd ki a HAMIS állítást!

* Az osztály attribútumának mindig van típusa.
* Az osztály attribútumának mindig van neve.
* ☒ Az osztály attribútumai az osztály viselkedését írják le.
* Az osztály attribútumai az objektum állapotát írják le.

Az osztály attribútumainak mindig van típusa és neve, és az objektum állapotát írják le, nem a viselkedését.

Metódusok (classstructuremethods)

Az objektumok nem csak adatokat, hanem utasításokat is tartalmazhatnak, melyeket a metódusok fognak egybe. A metódusok feladata az objektum attribútumainak módosítása vagy azok lekérdezése. Ezzel tulajdonképpen pontosíthatjuk az osztály definícióját: az osztály a tulajdonságok (attribútumok) és műveletek (metódusok) összessége.

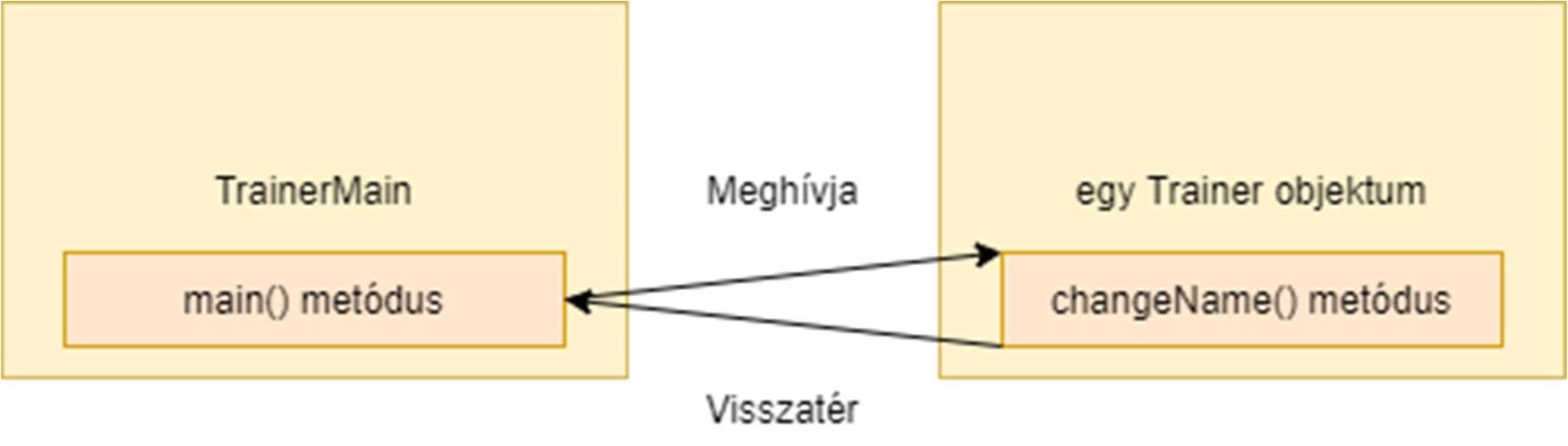
A metódus hívásakor az egyik metódus átadja a vezérlést a másik metódusnak, és átadja neki a paramétereket. A metódushívás a példányokon értelmezett.

Egy metódus készítésekor meg kell adnunk, hogy mi legyen a visszatérési típusa, a neve, valamint a paraméterek listája. Visszatérési típust akkor is kötelező megadnunk, ha a metódus nem ad vissza értéket. Ebben az esetben a void kulcsszót kell használnunk.

Amennyiben nem void a visszatérési típus, muszáj valamilyen értékkel visszatérni. Ehhez a return kulcsszó használandó és utána kell írni a visszatérési értéket. A return főleg a metódusok utolsó utasításaként szerepel.

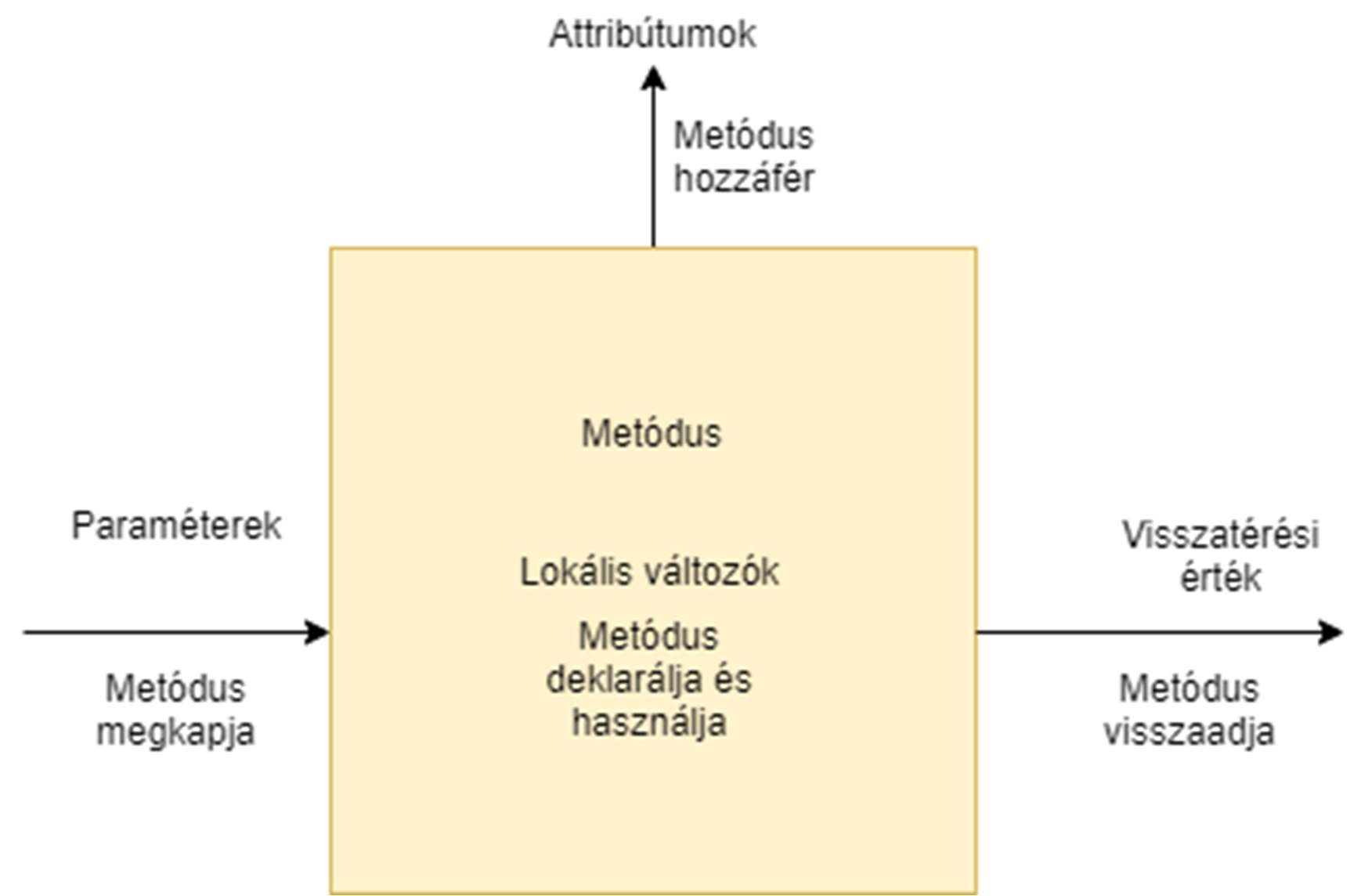
Metódusnak nem kell lennie paraméterének, ekkor csak egy nyitó és csukó zárójel szerepel. A paramétereket egymástól vesszóvel kell elválasztani. Mivel a Java erősen típusos nyelv, minden paraméter esetén meg kell adni annak típusát is.

A main() is egy metódus.



Metódushívás

A metódus hozzáfér az attribútumokhoz, valamint a paraméterekhez. A metóduson belül is deklarálhatunk változókat is, ezek a lokális változók. Előfordul, hogy a paraméterként kapott változó neve ugyanaz, mint az attribútumé. Ekkor a paraméter elfedi az attribútumot. Ha leírjuk magában a nevet, az a paraméterre vonatkozik. Úgy lehet az attribútumhoz hozzáférni, hogy az attribútum elé kitesszük a this minősítőt.



Metódus hozzáférései

Fontos objektumorientált alapelv, hogy az attribútumokhoz nem engedünk közvetlen hozzáférést kívülről, azokat mind lekérdezni, mind módosítani csak metóduson át lehet. Ez az information hiding alapelv. De hogyan tudjuk ezt elérni? A tagok láthatóságát módosító szavakkal szabályozhatjuk, melyet mindig a legelső helyen kell megadni.

* public - minden osztályból látható, elérhető
* private - csak az adott osztályon belül látható, érhető el

Javában az alapértelmezett láthatóság a package private, azaz azonos csomagon belül más osztályokból is elérhető a tag. Ekkor semmilyen módosítót nem használunk. Ezért deklaráljuk az attribútumainkat mindig privátként!

Az adott osztályban látható néhány metódus is.

public class Trainer {

private String name;

private int yearOfBirth;

public String getNameAndYearOfBirth() { return name + ": " + yearOfBirth;

}

public int getAge(int year) { return year - yearOfBirth;

}

public void changeName(String name) { this.name = name;

}

public void setYearOfBirth(int yearOfBirth) { this.yearOfBirth = yearOfBirth;

}

}

Egy másik osztályból már csak a Trainer metódusait érhetjük el, az attribútumait nem.

public class TrainerMain {

public static void main(String[] args) { Trainer trainer = new Trainer(); trainer.changeName("John Doe"); trainer.setYearOfBirth(1980);

String nameAndYearOfBirth = trainer.getNameAndYearOfBirth(); int age = trainer.getAge(2019);

}

}

Itt a main() metódusból meghívásra kerül a changeName() metódus, és paraméterül átadásra kerül a John Doe érték. Majd meghívásra kerül a setYearOfBirth() metódus, mely paraméterül kapja a 1980 egész számot. Ezek nem adnak vissza értéket. A getNameAndYearOfBirth() metódusnak nincs paramétere, de visszatér egy karakterlánccal, melyet eltárolukn a nameAndYearOfBirth változóban. A getAge() metódus visszaad egy egész értéket, melyet eltárolunk az age változóban.

Nagyon gyakran használunk olyan metódusokat, amelyek egyetlen attribútum értékét lekérdezik, illetve módosítják. A lekérdező metódus neve getter, a módosítóé setter.

Fejleszőeszköz támogatás

Látható, hogy nem mindig, de van olyan eset, mikor az IDEA megjeleníti a paraméter nevét. Ezt nem kell begépelni!

A fejlesztőeszköz csak segít abban, hogy szürkével megjeleníti a formális paraméter nevét. Ezt nem kell kiírni. Javaban a metódus hívásakor csak a paraméter neveket kell megadni sorban egymás után, vesszővel elválasztva.

Tehát az IDEA így jeleníti meg:

anotherTrainer.changeName(newName: "Joe Doe"); // Ez a Java kód nem fordul

le



IDEA paraméterek

Helyesen természetesen így kell meghívni:

anotherTrainer.changeName("Joe Doe");

A getter és setter metódusokat az IDEA-val nagyon könnyen előállíthatjuk: jobb klikk a szerkesztőben, majd Generate… (vagy ALT + Insert billentyűkombináció), azon belül Getter and Setter pont. Miután kiválasztottuk azokat az attribútumokat, amelyekhez szeretnénk gettert és settert generálni, az IDEA automatikusan beszúrja ezek kódját az osztályba.

A névhez generált getter és setter metódusok:

public String getName() { return name;

}

public void setName(String name) { this.name = name;

}

Ellenőrző kérdések

* Mi a feladata a metódusoknak?
* Mihez férhetnek hozzá a metódusok?
* Milyen részei vannak a metódus deklarációjának?
* Mik azok a getter és setter metódusok? Mi a nevük, paramétereik és visszatérési értékeik?

Feladat

A videóban szereplő feladatok a demos/src/main/java/classstructuremethods elérési úton vannak.

A classstructuremethods csomagba dolgozz!

A feladatok megoldásai a solutions/classstructuremethods elérési úton vannak.

Ügyfél osztály

Hozz létre egy Client osztályt, melynek három private attribútuma van: név (name), születési év (year) és cím (address). Típusaik rendre String, int és String!

Mind a három attribútumra legyen lekérdező és módosító metódus! Legyen egy public void migrate(String address) metódusa is, mely az ügyfél elköltözését implementálja, valójában beállítja a tárolt címet az új, paraméterként átadott címre.

Hozz létre egy main() metódust egy ClientMain osztályban, amelyben kipróbálod az osztály működését! Példányosítani kell egy objektumot a Client osztály alapján, majd be kell állítani az attribútumai értékét. Írd ki konzolra az összes adatát, majd hívd meg a migrate() metódust egy másik címmel! Jelezd vissza a felhasználónak a címváltozás sikerességét úgy, hogy kiírod az eltárolt új címet!

Jegyzeteljünk

Készíts egy Note osztályt, melyben a felhasználó rövid szöveges jegyzetét tárolod.

Tárolni kell a felhasználó nevét (name), a jegyzet témáját (topic) és szövegét (text). Fejlesztés során ezentúl mindig tartsd be az information hiding elvet, azaz az attribútumok legyenek privátak, és készíts hozzájuk gettereket és settereket! Készíts egy getNoteText() metódust, mely az osztály attribútumai alapján egyetlen szöveget ad vissza name: (topic) text formátumban!

Teszteld az osztályod a NoteMain osztály main() metódusából! A Note tartalmának megjelenítésekor használd a getNoteText() metódust!

Forrás

OCA - Chapter 1/Understanding the Java Class Structure, Writing a main() Method Teszt

Mi a metódus felépítése?

* ☒ Visszatérési érték típusa, metódusnév, paraméterek
* Metódusnév, paraméterek, visszatérési érték típusa
* Visszatérési érték típusa, paraméterek, név
* Paraméterek, név, visszatérési érték típusa

Mi a visszatérési típusa annak a metódusnak, amely nem ad vissza semmilyen értéket?

* main
* setter
* ☒ void
* String

Mi történik, ha egy metódus paraméter neve megegyezik egy attribútum nevével?

* A metódusban csak az attribútumhoz lehet hozzáférni, a paraméterhez nem.
* ☒ A metódusban mind a paraméterhez, mind az attribútumhoz hozzá lehet férni, az attribútumhoz a this minősítővel.
* A metódusban csak a paraméterhez lehet hozzáférni, az attribútumhoz nem.
* A metódusban mind a paraméterhez, mind az attribútumhoz hozzá lehet férni, a paraméterhez a this minősítővel.

Konstruktorok (classstructureconstructors)

A konstruktorok felelősek az objektumok állapotának inicializálásáért. Amikor egy osztályt példányosítunk, akkor lefoglalásra kerülnek az attribútumai a memóriában. Ezek kezdőértéket kívülről konstruktoron át kaphatnak.

A konstruktor egy olyan speciális metódus, melynek nincs visszatérési értéke, a neve pedig megegyezik az osztály nevével.

public class Trainer {

private String name;

private int yearOfBirth;

public Trainer(String name, int yearOfBirth) { this.name = name;

this.yearOfBirth = yearOfBirth;

}

//getter és setter metódusok

}

Látható, hogy itt is a paraméterek elfedik az attribútumokat, ezért ha az attribútumra akarunk hivatkozni, akkor a this kulcsszót kell alkalmaznunk.

Amikor a new operátorral szeretnénk egy Trainer objektumot létrehozni, akkor tulajdonképpen a Trainer osztály konstruktorát hívjuk meg. (Ezt a paraméter nélküli konstruktort eddig legenerálta nekünk a fordító, nem magunknak kellett megírni, mi csak meghívtuk. Ez egy un. paraméter nélküli default konstruktor) Ha ez paramétereket vár, akkor azokat is meg kell adnunk.

public class TrainerMain {

public static void main(String[] args) {

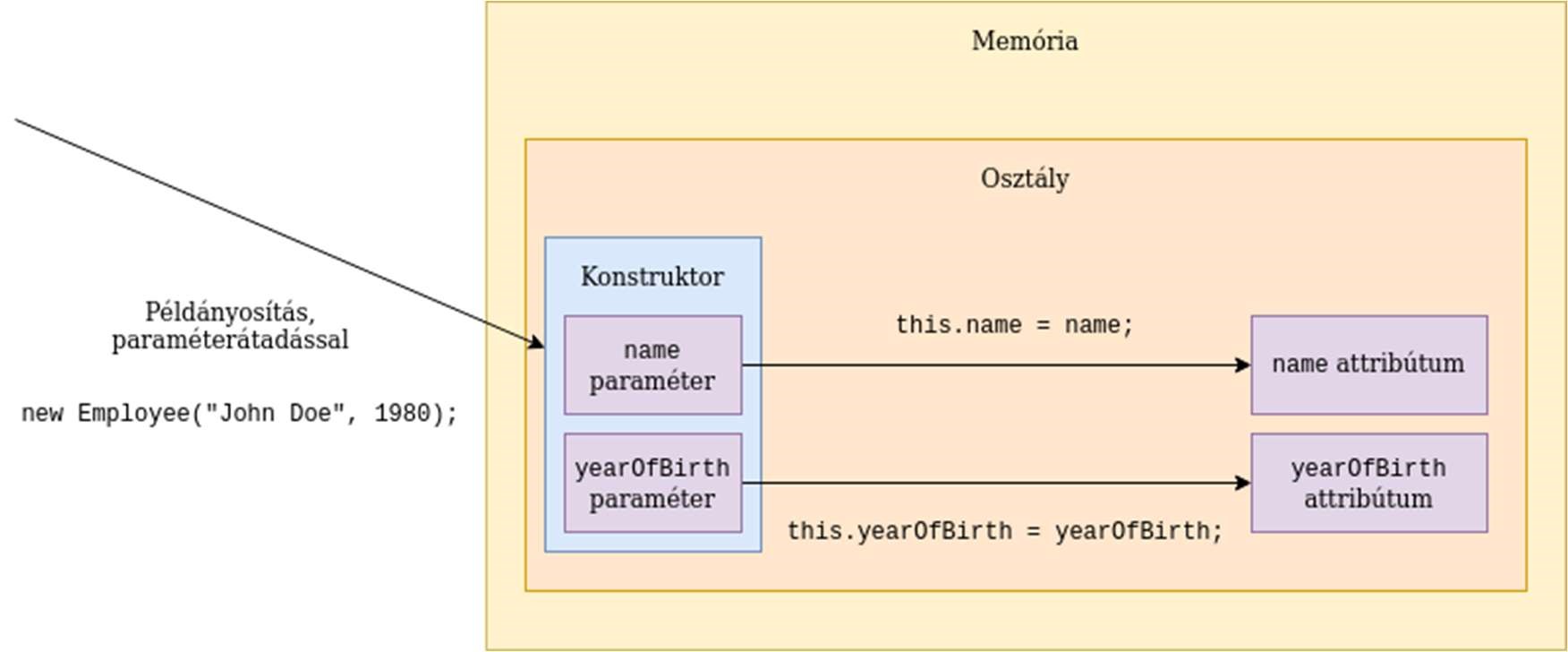
Trainer trainer = new Trainer("John Doe", 1980); //konstruktor hívása

System.out.println(trainer.getName()); //"John Doe"

System.out.println(trainer.getYearOfBirth()); //1980

}

}



Konstruktor hívása

Úgy képzeljünk, hogy a konstruktor hívásával az üres attribútumoknak adunk értéket a konstruktor paramétereken keresztül.

Amikor leírjuk a következő kódot:

Scanner scanner = new Scanner(System.in);

Már tudjuk, hogy a Scanner osztály konstruktorát hívjuk meg, és átadjuk neki paraméterül a System.in értéket.

Konstruktort a fejlesztőkörnyezet is tud generálni. Az IDEA-ban az ALT + Insert billentyűkombináció lenyomása után a Constructor menüpontot választva meg kell adnunk, hogy mely attribútumok kapjanak kezdőértéket, majd az IDE a konstruktor kódját automatikusan beszúrja.

Ellenőrző kérdések

* Mi a konstruktor feladata?
* Milyen megkötések vannak, amikor konstruktort készítesz? • Hogyan lehet az IDEA segítségével konstruktort létrehozni?

Feladat

A videóban szereplő feladatok a demos/src/main/java/classstructureconstructors elérési úton vannak.

A classstructureconstructors csomagba dolgozz!

A feladatok megoldásai a solutions/classstructureconstructors elérési úton vannak.

Könyv osztály

Hozz létre egy Book osztályt, melynek három privát attribútuma van: szerző (author), cím (title) és regisztrációs szám (regNumber), mindhárom típusa String. A Book példányosításakor csak a szerzőt és a címet kelljen megadni.

Legyen egy public void register(String regNumber) metódusa, mely a nyilvántartásba vételt implementálja, és ennek paraméterül kell megadni a regisztrációs számot.

Írj egy main() metódust a BookMain osztályba, amivel kipróbálod a működését! Az attribútumok kiolvasásához használj gettereket!

Raktár osztály

Készíts egy Store osztályt, mely egy raktárt modellez. A raktár jellemzője az, hogy miből (product) és aktuálisan mekkora mennyiséget (stock) tárol. (Ennek a raktárnak speciális jellemzője, hogy csak egyféle terméket tud tárolni.) Az első attribútuma String, a második int típusú.

A Store példányosításakor elég megadni a tárolt terméket, a mennyiség mindig 0, ezért a konstruktor csak a terméket kapja meg kívülről.

Készíts hozzá két metódust, mely a tárolt mennyiséget változtatja: a store() metódusa a paraméterként kapott mennyiséget eltárolja a raktárban növelve ezzel a készletet, míg a dispatch() metódusa az elszállítást modellezi, azaz a paraméterként kapott mennyiséggel csökkenti a készletet! (Most még nem kell ellenőrizned, hogy elszállításkor van-e a raktárban elegendő mennyiségű terméket.)

A StoreMain osztály main() metódusában készíts két Store példányt, és teszteld, hogy mindkét raktár helyesen és függetlenül működik be- és kiszállítás esetén is!

Teszt

Melyik állítás igaz?

* ☒ A konstruktor az osztály példányosításakor fut le.
* A konstruktor visszatérési típusa mindig ugyanaz, mint az osztály neve.
* A konstruktor feladata az objektum állapotának módosítása.
* A konstruktor az osztály metódusait inicializálja.

Miért írunk konstruktort?

* Az objektum állapotának módosítása.
* ☒ Az objektum állapotának inicializálására, azaz hogy az attribútumainak kezdőértéket adjunk.
* Hogy az objektum alapján egy másik objektumot tudjunk létrehozni.
* Hogy az osztály alapján egy másik osztályt létre tudjunk hozni. ### UML, példányok konzol íráskor/olvasáskor (classstructureintegrate)

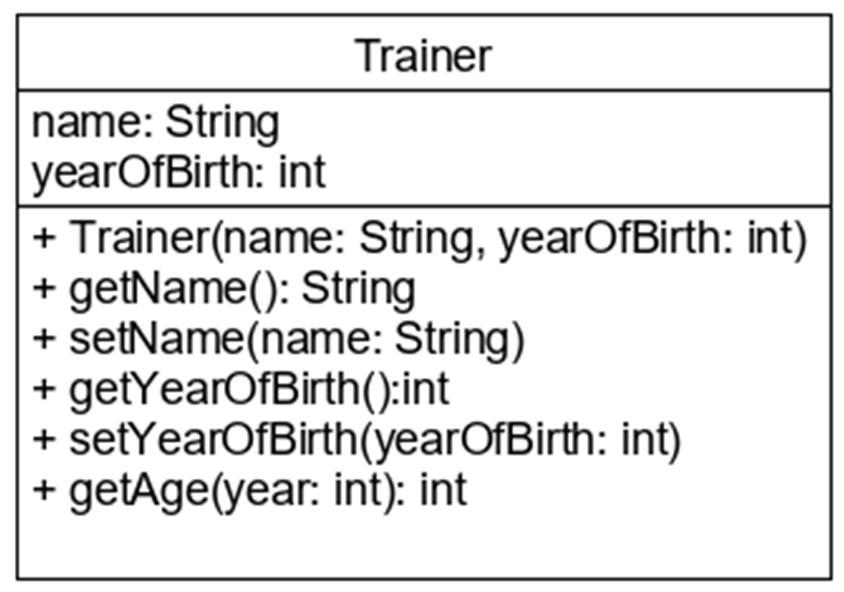
Az osztály egységbe zárja az összetartozó adatokat és a rajtuk végzett műveleteket. Háromféle tagját különböztetjük meg:

* attribútum: az adatok, tehát az állapot tárolására szolgáló változók
* konstruktor: az adatok inicializálására (kezdőérték adás) szolgáló speciális metódus, mely példányosításkor egyszer és csakis egyszer fut le
* metódus: az állapot lekérdezésére, módosítására szolgáló műveletek

UML (Unified Modeling Language)

Az UML szabványos, általános célú modellező nyelv, amely alkalmas nagy méretű rendszerek vizuális megjelenítésére, dokumentálására. A diagramok alkalmasak a rendszer struktúrájának és viselkedésének leírására. Egyik leggyakrabban használt diagramja az osztálydiagram, mely az osztályok és kapcsolataik grafikus ábrázolására szolgál. Az osztályokat egy három részre osztott téglalap jelöli. A felső részben található az osztály neve, a középsőben az attribútumok, a harmadikban a metódusok sora. A tagok láthatóságát speciális karakterek jelölik a tag neve előtt.

Metódusok esetén gyakran találkozunk azzal, hogy a paramétereknek csak típusa van, neve nincs. Ilyenkor tetszőleges, de beszédes nevet adhatunk nekik. Az UML osztálydiagram csak akkor tartalmazza a neveket is, ha több paraméter esetén nem egyértelmű azok szerepe. Például két String típusú paraméter is van.



UML osztály diagram

Konzol használata objektum létrehozásakor

Az objektum létrehozásához szükséges adatokat a felhasználótól Scanner objektum segítségével kérhetjük be, majd a létrehozott objektum aktuális állapotát

System.out.println() metódussal írhatjuk ki. Lássunk is egy teljes példát erre:

package training;

public class TrainerMain {

public static void main(String[] args) { System.out.println("What is your name?");

Scanner scanner = new Scanner(System.in);

String name = scanner.nextLine();

System.out.println("Year of birth?"); int yearOfBirth = scanner.nextInt();

Trainer trainer = new Trainer(name, yearOfBirth);

System.out.println(trainer.getName());

System.out.println(trainer.getNameAndYearOfBirth());

System.out.println("Age: " + trainer.getAge(2019));

}

}

Amennyiben két osztály van, a köztük lévő vonal jelenti, hogy kapcsolatban vannak egymással.



Osztályok közötti kapcsolatok

Egy osztálydiagrammon nem szükséges mindent kiírni, csak ami a megértés szempontjából fontos. Sőt komolyabb alkalmazások esetén, nem is szoktuk az alkalmazás minden osztályát megjeleníteni, hiszen az több száz vagy ezer is lehet.

Ellenőrző kérdések

* Mi az UML?
* Hogyan épül fel egy osztálydiagram?
* Milyen részekből áll egy osztály diagramelem?

Feladat

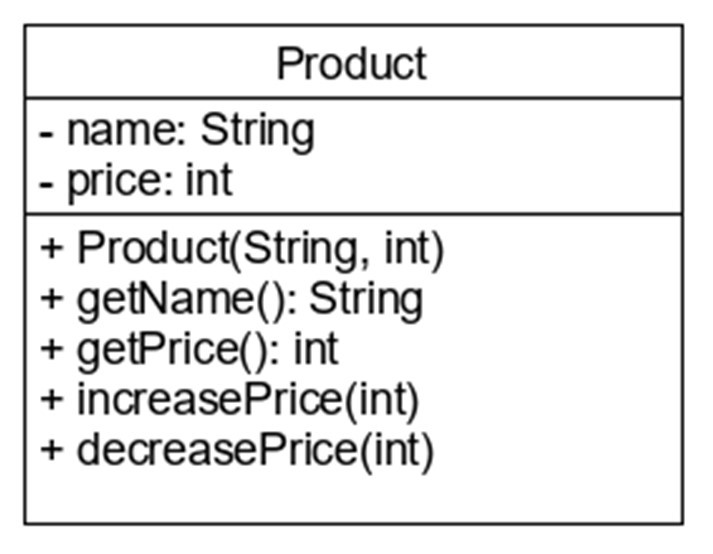
A videóban szereplő feladatok a demos/src/main/java/classstructureintegrate elérési úton vannak.

A classstructureintegrate csomagba dolgozz!

A feladatok megoldásai a solutions/classstructureintegrate elérési úton vannak.

Termék

Az alábbi diagram és a leírás alapján készítsd el a Product osztályt!



Product osztály diagram

Az attribútumok kezdőértéket a konstruktorban kapnak. Az increasePrice() a paraméter értékével növeli, a decreasePrice() pedig csökkenti az aktuális árat.

Próbáld ki az osztály működését main() metódusban, ahol a példány létrehozásához szükséges adatokat a felhasználótól kéred be!

Bankszámlák

Az UML diagram és a leírás alapján készítsd el a BankAccount osztályt!



BankAccount osztály diagram

Számlanyitáshoz mindhárom attribútum értékét meg kell adni. A számlára lehet befizetni (deposit()), lehet róla pénzt kivenni (withdraw()), illetve másik számlára át lehet utalni összeget (transfer()). Ez utóbbi esetben a számla saját egyenlege csökken, de a másik számla egyenlegére jóváírás történik.

A getInfo() metódus a számla adatait az alábbi formában adja vissza Stringként:

Tóth Kálmán (10073217-12000098-67341590): 103400 Ft

Készíts egy Bank osztályt, amely main() metódusában létrehozol két bankszámlát! Próbáld ki az összes elkészített metódust, hogy jól működik-e! A szükséges adatokat a felhasználótól kérd be! Átutalásnál ellenőrizd mindkét számla új egyenlegét!

Forrás

OCA - Chapter 1/Understanding the Java Class Structure, Writing a main() Method Teszt

Az UML osztálydiagramon az osztály diagramelem milyen három részre oszlik?

* Osztály neve, láthatóságok, tagok
* Osztály típusa, osztály neve, tagok
* Privát tagok, publikus tagok, osztály neve • ☒ Osztály neve, attribútumok, metódusok

Mi nem igaz egy osztálydiagrammra?

* Ábrázolja az osztályokat
* Ábrázolja az osztályok közötti kapcsolatokat
* ☒ Ábrázolja az osztályokat és példányokat
* Ábrázolja az osztály tagjait, mint pl. az attribútumok, konstruktorok és metódusok A Java alkalmazások Java osztályokból állnak. Ezek tervrajzok, ez alapján példányosítással hozhatóak létre a Java objektumok, azaz példányok.

Az osztályok tartalmazhatnak attribútumokat, metódusokat és konstruktorokat. Az attribútumok hordozzák az objektum állapotát. A metódusok az állapotot tudják visszaadni és módosítani. A konstruktorok tudják inicializálni az objektum állapotát. Az attribútumokat private módosítószóval látjuk el.

A metódusok névvel, paraméterekkel és visszatérési típussal rendelkeznek. Speciális visszatérési típus a void. Visszatérni a return kulcsszóval lehet. Kiemelt metódusok a getter és setter metódusok.

A konstruktorok olyan metódusok, melyek példányosításkor futnak le.

A felhasználónak kiírni valamit a System.out.println() metódussal lehet. Beolvasni a konzolról a Scanner osztály segítségével lehet.

Az osztályokat UML osztálydiagrammon tudjuk ábrázolni. ## A nyelv építőkövei ### Kódolási konvenciók (conventions)

A kódolási konvenciók arra valók, hogy az együtt dolgozó fejlesztők hasonló formátumú kódot írjanak, így ha el kell olvasni, netán módosítani kell egymás kódját, könnyebben megértsék azokat.

Kódformázás

Java szabványban nincs egységes kódolási szabálygyűjtemény az elnevezésekre, behúzásokra, sortörésekre stb. Az Oracle weboldalán megtalálható egy ajánlás, valamint a nagyobb cégek is, mint például a Google, mind rendelkeznek saját konvenciókkal. Ezeket az alábbi linkeken tanulmányozhatod át: http://www.oracle.com/technetwork/java/codeconvtoc-136057.html https://google.github.io/styleguide/javaguide.html

Az IDE-k tartalmaznak kódformázó menüpontokat, de ezek sem egységesek.

Behúzások, üres sorok, blokkok

Konvenció szerint beljebb kezdünk minden olyan utasítást, amely valamilyen másik utasításon, blokkon belül van. Így az osztályon belül beljebb kezdjük az attribútum és metódus deklarációkat, a metóduson belül a metódustörzset.

Az attribútumok és a metódusok között hagyjunk ki egy-egy üres sort, így a kódunk sokkal strukturáltabbá, olvashatóbbá válik.

Elnevezések

A Java nyelvben használt azonosítók bármilyen Unicode betűt, számot \_-t és $-t tartalmazhatnak, azonban kerüljük az ékezetes karakterek használatát. Legjobb, ha csak az angol ábécé betűit és számokat használunk. A több szóból álló elnevezés esetén minden szót nagybetűvel kezdünk (CamelCase) és egybe írunk. Mozaikszavak esetén is csak az első betűt írjuk naggyal, a többit kicsivel. Bármilyen elnevezés megengedett, kivéve a nyelv által lefoglalt kulcsszavakat, mégis kerüljük azokat, melyek megtévesztők. Például ne használjuk a string vagy az integer azonosítót, mert ezeket ugyan elfogadja a fordító, de összekeverhetőek a String és Integer típusokkal.

Megjegyzések

Háromféle megjegyzés létezik. Az egysoros megjegyzést // karakterekkel kezdjük és a sor végéig tart. Ezt akár egy utasítás mögé is írhatjuk. Ha többsoros megjegyzést szeretnénk írni, akkor azt a /\* jellel kezdjük és \*/ jellel zárjuk. A Javadoc megjegyzések olyan speciális elemek, amelyekből HTML-ben írt dokumentáció készíthető. Ezeket /\*\* jellel kezdjük és \*/ jellel fejezzük be, és konvenció szerint minden sor elején \* jel van. A Javadoc megjegyzés mindig azon elem fölé kerül, amelyikhez készítjük. Tartozhat osztályhoz, attribútumhoz és metódushoz is. Speciális jelöléseket használ például a metódus paraméterek vagy a visszatérési érték megadásához.

/\*\*

* Visszaadja összefűzve az oktató nevét és születési évét. \*
* @return Az összefűzött eredmény.

\*/

public String getNameAndYearOfBirth() {

return name + ": " + yearOfBirth; // Összefűzés

}

Ne használjunk feleslegesen megjegyzéseket. Amennyiben a kódból is egyszerűen megfejthető, hogy mit csinál, nem kell odaírni megjegyzésben is.

Forrás: http://www.oracle.com/technetwork/articles/java/index-137868.html Sorrendezés

A java fájlban először mindig a package deklaráció, utána az importok, végül az osztálydeklaráció következik. Ezek sorrendje kötött. Az, hogy az osztályon belül milyen sorrendben deklaráljuk az attribútumokat, metódusokat és konstruktorokat, az ránk van bízva, mégis konvencionálisan először adjuk meg az attribútumokat, utána a konstruktorokat, és csak legvégén a metódusokat. Ez utóbbiakat láthatóság szerint csökkenő sorrendbe szoktuk rendezni, azaz először írjuk a publikus, a legvégén pedig a privát metódusokat.

Ellenőrző kérdések

* Mire valók a kódolási konvenciók?
* Hogyan nevezzük el az osztályokat, attribútumokat és metódusokat?
* Mire figyeljünk a betűszavaknál?
* Milyen sorrendben következzenek egy osztályon belül a különböző tagok?
* Milyen típusú megjegyzések vannak Java nyelvben?

Feladat

SonarLint telepítés

A SonarLint az IDÉ-be beépülve ellenőrzi a kódod minőségét, azaz, hogy mennyire tartod be a konvenciókat, illetve mennyire jól olvasható a kódod.

Nyisd meg a File/Settings… menüpontot, és a megjelenő ablak bal oldalán válaszd ki a Plugins szakaszt! A MarketPlace keresőjében keress rá a SonarLint bővítményre, és telepítsd fel! A bővítmény az IDE újraindítása után lesz elérhető.

Nézd meg az eddigi gyakorlati feladatokkal kapcsolatban mit ír a SonarLint! Az ablak alsó részében kell látnod egy SonarLint gombot. Amennyiben nem jelenik meg, a View/Tool Windows/SonarLint menüponttal bekapcsolhatod.

Forrás

OCA - Chapter 1/Ordering Elements in a Class

Teszt

Mire valók a kódolási konvenciók?

* ☒ Hogy a fejlesztők könnyebben tudjanak együtt dolgozni.
* Hogy gyorsítsák a fordítást.
* Hogy ne legyenek fordítási hibák.
* Hogy segítsük a verziókezelő rendszer működését.

Megfelel a kódolási konvencióknak a Java java.net.URL vagy java.net.URLConnection osztály?

* Igen
* ☒ Nem

Literálok és lokális változók (localvariables)

Adattípusok

A Java erősen típusos nyelv, azaz minden változó deklarációjakor meg kell adnunk annak típusát, és ezt később nem változtathatjuk meg. Egy típus pl. a boolean mely egy logikai értéket tárol, true vagy false. Az int egész típus, a double lebegőpontos típus (a valós számok tárolási módja, lényeg, hogy ezzel lehet nem csak egész számokat ábrázolni). Ezek primitív típusok, úgy is megkülönböztethetőek, hogy kisbetűvel kezdődnek.

Egy változó lehet primitív típusú vagy referencia tipusú. A primitív típusú változók magát az értéket tárolják. A referencia típusú változók mindig egy objektumra mutatnak, de annak nem az állapotát, hanem csak egy hivatkozást (referencia) tárolnak. Egy objektum állapota csak közvetve, ezen a hivatkozáson, vagy referencián keresztül érhető el. Ilyen referencia típus például a String.

Bevezetés a literálok használatába

Literálnak hívjuk azokat a kifejezéseket, amelyeknek önmagukban is van jelentése. Ezekhez mindig implicit társul egy adattípus.

Az implicit ebben az esetben azt jelenti, hogy nem nekünk kell megadni, hanem a “háttérben”, közvetetten történik meg. (A fordító vagy a virtuális gép végzi el helyettünk.)

A true és false boolean típusú literál. Ha pl. leírjuk, hogy 12, az egy egész literál, melynek típusa int.

Amennyiben nagy egész számot használunk, az olvashatóság kedvéért írhatjuk a következő formátumban is: 100\_000.

Pl. a 1.5 egy lebegőpontos literál, melynek típusa double.

Speciális literál a null, mely referencia változóknak adható értékül, amikor semmilyen objektumra nem mutatnak.

A "John Doe" egy String típusú literál.

Lokális változók

Metóduson belül deklarált és paraméterként átadott változókat együttesen lokális változóknak nevezzük. Változókat deklarálni a típusukkal és a nevükkel lehet. Ezen kívül opcionálisan meg lehet adni kezdőértéket is, egyenlőségjel után.

double amount;

double sum = 1.5;

int i = 0;

Látható, hogy a deklaráció és az értékadás állhat külön, két utasításban, de állhat egyben is, egy utasításként.

A Java 10-ben jelent meg az a lehetőség, hogy a lokális változó típusát nem kell megadni, ha kezdőértékadás van, ekkor a kezdőérték típusa lesz a változó típusa.

Ilyenkor a var kulcsszót kell alkalmazni.

var sum = 1.5;

var i = 0;

A lokális változót a deklarálástól kezdve az adott blokk végéig lehet használni, ez a láthatósága.

A változónak bármennyiszer újra értéket tudunk adni.

A lokális változóknak nincs automatikus kezdőértékük, ezért használatuk előtt mindig nekünk kell az értékadásról gondoskodni. Kizárólag a deklarációtól azon blokk végéig léteznek, amelyben deklaráltuk.

Típuskonverzió

Alapszabály, hogy egy változónak csak olyan típusú érték adható, amilyen a deklarált típusa. Ha más típusú értéket szeretnénk benne tárolni, akkor azt konvertálni kell. Ez a típuskonverzió vagy típuskényszerítés. Ez a konverzió történhet automatikusan, implicit módon vagy a programozó által jelölten, explicit módon.

Hasonló típusok között (pl. számok) a kisebb automatikusan konvertálható nagyobbra, mint például int típusú érték double típusúra. A nagyobb típusú csak explicit, azaz kiírt konverzióval (cast) konvertálható kisebbre, és ez esetleges adatvesztéssel járhat.

double d = 12; // int --> double implicit int i = (int) 3.14; // double --> int kényszerítéssel, értéke 3

A videóban kerekítés hangzik el, de valójában levágja a tizedesjegyeket.

Például a matematikai kerekítés szerint 3.6 kerekített értéke 4 lenne, de Java típuskényszerítés esetén 3 lesz.

double d = 3.6; int i = (int) d; // i értéke 3 lesz

A Java nyelvben a logikai érték nem kapcsolható számértékhez, mint sok más nyelvben, ezért ezek nem konvertálhatók számmá.

Objektumok élettartama

Egy objektum a létrehozásától (konstruktor hívása) addig létezik, amíg használjuk. Nem kell nekünk megjelölnünk, hogy nem használjuk többet, hanem ezt a JVM automatikusan figyeli. Van egy mechanizmus, mely a memóriából kitörli a nem használt objektumokat, ezáltal helyet szabadítva fel, ez a szemétgyűjtő mechanizmus, garbage collector (GC).

Getter boolean esetén

Egy boolean típusú változó esetén a hozzá generált getter nem get, hanem is előtagot tartalmaz.

public class Employee {

private boolean fullTime;

public boolean isFullTime() { return fullTime;

}

}

Ellenőrző kérdések

* Mire valók a literálok?
* Milyen literálokat ismersz, és hogyan deklarálod őket?
* A hol definiált változókat nevezzük lokális változóknak?
* Szükséges-e típust definiálni? Milyen típusokat különböztetünk meg?
* Mi a lokális változó láthatósága?
* Mi a kezdőértéke egy lokális változónak?
* Mi az a szemétgyűjtő mechanizmus?

Feladat

A videóban szereplő feladatok a demos/src/main/java/localvariables elérési úton vannak.

A localvariables csomagba dolgozz!

A feladatok megoldásai a solutions/localvariables elérési úton vannak.

Literálok és típusok

A localvariables.LocalVariablesMain osztály main() metódusában hozd létre az alábbi lokális változókat!

Definiálj egy boolean típusú változót b néven, majd írasd ki az értékét!

Sikerül?

Adj értékül neki false értéket!

Definiálj egy int típusú változót 2 kezdőértékkel a néven!

Definiálj egy sorban két int típusú változót i és j néven 3 és 4 kezdőértékkel!

Definiálj egy int típusú változót k néven, és add neki értékül az i változó értékét!

Próbálj egy változót definiálása előtt kiírni! Sikerül?

Definiálj egy String típusú változót s néven! Adj neki "Hello World" értéket!

Definiálj egy String típusú változót t néven, és add értékül neki az s változó értékét!

Metóduson belül definiálj egy blokkot (kapcsos zárójelek között)! A blokkon belül definiálj egy int típusú x változót 0 kezdőértékkel!

Az értékét próbáld kiírni blokk után, a blokkon kívül! Fog sikerülni?

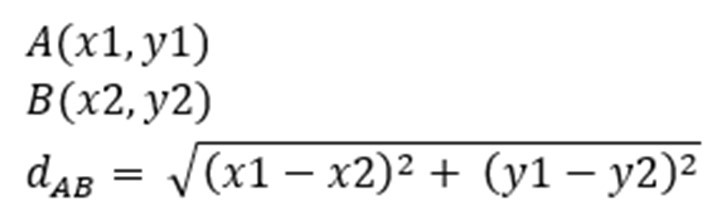
A blokkban próbáld meg kiírni a blokkon kívül, a blokk előtt definiált a változó értékét!

Távolság

Hozz létre egy Distance osztályt, mely tartalmazza a távolságot lebegőpontos számként, valamint azt, hogy a mért távolság pontos-e, egy boolean értékként! Csak getter metódusokat hozz létre!

A DistanceMain osztályban hozz létre egy Distance példányt, majd írd ki a távolságot, és hogy pontos-e!

Majd deklarálj egy int típusú változót, és add értékül neki a távolság egész részét! Majd írd ki ezt az értéket!



Distance

Forrás

OCA - Chapter 1/Understanding Default Initialization of Variables, Understanding Variable Scope

Teszt

Válaszd ki a helytelen értékadást!

* int x = 23;
* double d = 'a';
* double d = -1;
* ☒ int b = true;

Hogyan kell egy double d = 10.10; típusú változót int értékké konvertálni?

* int i = int d;
* int i = [int] d;
* int i = (double) d;
* ☒ int i = (int) d; ### Kifejezések és utasítások (statements)

A kifejezés operátorok (műveleti jelek) és operandusok (azok a literálok, változók vagy metódushívások, amelyekkel a műveletet elvégezzük) kombinációja, tipikusan egy érték kiszámítására.

Operátorok

* Matematikai operátorok: +, -, \*, /, % (összeadás, kivonás, szorzás, osztás, maradékos osztás)

* Példányosítás: new
* Értékadó operátor: =
* Összevont értékadó operátorok: +=, -=, \*=, /=, %=, &=, |=, ^=
* Összehasonlítás, egyenlőségvizsgálat: <, <=, >, >=, ==, !=
* Logikai operátorok: !, &, |, &&, ||
* Léptető operátorok (prefix és postfix): ++, -- • Háromoperandusú operátor: ?: Az összevont értékadó operátor működése:

int i = 5; i += 6; // i értéke 11, az i = i + 6 kifejezésnek felel meg Az összehasonlító operátorok eredménye egy boolean érték:

boolean eq = 5 == 10; // false érték, mert a két literál nem egyenlő

boolean gt = 10 > 5; // true érték

Összetett kifejezés esetén a műveletek elvégzésének sorrendje kötött, de megfelelő zárójelezéssel módosítható. Ehhez kerek zárójelet használunk.

int i = 2 + 3 \* 5; // 17, mert a szorzást előbb hajtja végre

int j = (2 + 3) \* 5; // 25, mert a zárójelek miatt az összeadást előbb hajtja végre

VIGYÁZZ! Ne keverd össze az értékadó = és az egyenlőséget vizsgaló == operátorokat!

Egy kifejezés operandusa lehet literál, egy változó és egy metódushívás is.

String name = trainer.getName(); // Értékadó operátor jobb oldalán egy metódushívás

Egy kifejezést át lehet adni paraméterként is.

System.out.println(4 + 5);

Utasítások

A Java nyelvben az utasítások:

* változó deklaráció int age;
* értékadás age = 10;
* példányosítás new Random();
* metódushívás

System.out.println("Hello World");

* vezérlő utasítás if (age < 18) {

System.out.println("Too young");

}

A Java nyelvben minden utasítást pontosvesszővel kell lezárnunk. Ez alól csak néhány vezérlő utasítás a kivétel.

Szövegek összehasonlítása

A szövegek nem hasonlíthatóak össze a == operátorral, hanem az equals() metódust kell hívni. Ennek használata a következő:

String name = "John";

System.out.println(name.equals("John")); // 1

System.out.println("Jack".equals(name)); // 2

Az 2 jelű sorban lévő formátumot szoktuk használni, mert ez akkor is működik, ha a name értéke null. Az 1 jelű esetben hibaüzenetet kapunk, ha a name értéke null.

Ellenőrző kérdések

* Mire valók a kifejezések?
* Írj fel néhány kifejezést!
* Milyen operátorokat ismersz?
* Java nyelven milyen utasításokat ismersz?

Gyakorlati feladat

A videóban szereplő feladatok a demos/src/main/java/statements elérési úton vannak.

A statements csomagba dolgozz!

A feladatok megoldásai a solutions/statements elérési úton vannak.

Kifejezések

A statements.StatementMain osztály main() metódusában definiálj egy int típusú x változót, melynek értéke az 5 és 6 literál összege.

Definiálj egy int típusú y változót, mely a 11 literálból kivont x változó értékét kapja.

Definiálj egy int típusú z változót, mely értéke 8.

Definiálj egy boolean típusú b változót, mely értéke true, ha az x értéke nagyobb, mint az y változó értéke.

Definiálj egy boolean típusú c változót, mely értéke true, ha a b értéke true, vagy z értéke nagyobb, mint 5.

A z értékéhez adj hozzá egyet egy operandusú operátorral.

Időpontok

Készíts egy Time osztályt, amely egy adott időpontot reprezentál egy napon belül. Három attribútuma az óra, perc és másodperc értékét tárolja egész számként. Ezeket a a konstruktorban kapja meg. Készíts el az alábbi metódusokat:

* A getInMinutes() metódus az időpont értékét percekben adja vissza, de a másodperceket figyelmen kívül hagyja.
* A getInSeconds() metódus a teljes időpontot másodpercben adja vissza.
* A earlierThan() metódus paraméterként egy másik Time típusú objektumot kap. Amennyiben az adott objektum által reprezentált időpont korábbi, mint a paraméterül kapott, igazat ad vissza, különben hamisat. Használd a már elkészített metódusokat!
* A toString() metódusa az időpontot óra:perc:másodperc formában szövegként adja vissza.

A TimeMain osztály main() metódusában teszteld az osztályt! Kérj be a felhasználótól két időpontot, és írd ki az elsőt teljesen majd percekben, a másodikat teljesen majd másodpercekben, illetve azt, hogy az első korábbi-e, mint a második!

Egy lehetséges kimenet:

Az első időpont 12:3:43 = 723 perc

A második időpont 4:21:38 = 15698 másodperc

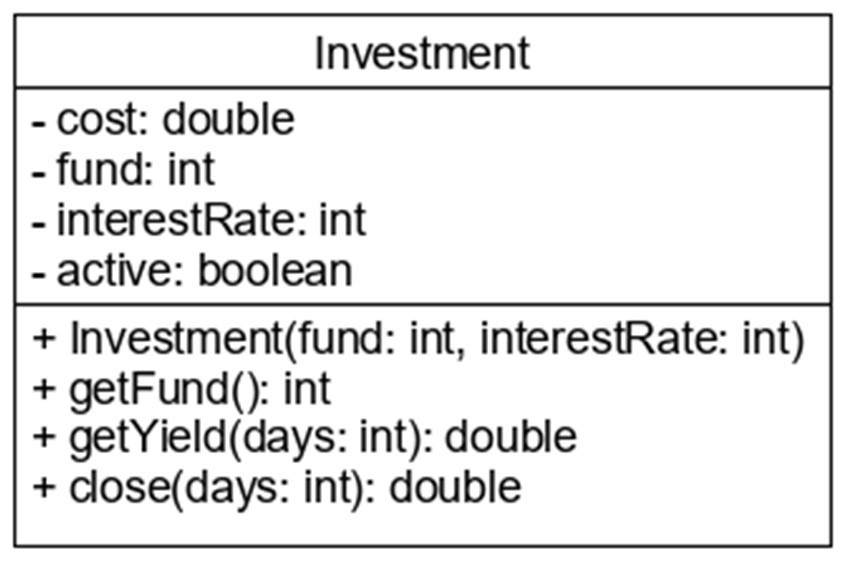
Az első korábbi, mint a második: false

Osztható 3-mal

Írj egy main() metódust a DivByThree, osztályba, mely bekér egy egész számot a felhasználótól, majd kiírja, hogy 3-mal osztható-e!

Befektetések

Egy befektetéskezelő cég legfeljebb egy év időtartamra vesz át összeget befektetésre az ügyfeleitől. Ezután bármikor meg lehet szüntetni a befektetést, és a tőkét az adott napig járó kamattal együtt ki lehet venni. Megszünetéskor a befektető cég kezelési költségként mindig levonja a kivett összeg 0,3%-át. Menet közben megszüntetés nélkül is le lehet kérdezni, hogy mennyi kamat járna az adott napig.



Investment UML

Hozd létre az Investment osztályt! Befektetés létrehozásakor (példányosításkor) az active attribútum értéke mindig igaz. A getYield() metódusa megkapja, hogy hány napra kérik le a hozamot, és visszaadja az adott időszakra kiszámított hozam összegét. A close() metódusa lezárja a befektetést, és ezzel egyidejűleg visszaadja a teljes kifizetett összeget. A lezárást az active attribútum hamisra állításával éri el. A kifizetett összeg tartalmazza a tőkét és a kamatokat csökkentve a kezelési költséggel. Amennyiben már lezárt befektetésre hívják meg a close() metódust, a kifizetett összeg 0 legyen! (Ötlet: használd a három operandusú operátort a kifizetett összeg kiszámításához!)

A metódusok implementálása során törekedj arra, hogy ne írd le kétszer ugyanazt a képletet, hanem használd a már elkészített metódusokat!

Próbáld ki a működését az InvestMain osztály main() metódusában! Kérd be a befektetett összeget és a kamatlábat a felhasználótól, majd írd ki a befektetés adatait!

Próbáld meg kétszer is lezárni a befektetést! Például:

Befektetés összege: 100000 Kamatláb:

8

Tőke: 100000

Hozam 50 napra: 1095.890410958904

Kivett összeg 80 nap után: 101448.16438356164

Kivett összeg 90 nap után: 0.0

Forrás

OCA - Chapter 2/Understanding Java Operators, Working with Binary Arithmetic Operators, Working with Unary Operators, Using Additional Binary Operators Teszt

Melyik kifejezés helyes?

* ☒ person.goHome();
* double perimeter = 2 \* r \* 3,14;
* String message = "Age: " - age;
* boolean isOddNumber = number % 2 = 0; Hogyan hasonlítasz össze két számértéket?
* ☒ a == b
* a ?= b
* a = b
* a === b

Hogyan hasonlítunk össze szövegeket?

* == operátorral
* ☒ equals() metódussal
* == operátorral és az equals()` metódus is jó
* sem a == operátor, sem az equals()` metódus nem jó ### Csomagok (packages)

Csomagok célja

A Java nyelv alapegysége az osztály. Általában egy osztály egy fájlnak felel meg, de ez alól vannak kivételek. Ha nagyon sok fájlunk van, vagy vannak azonos nevűek, akkor azokat különböző könyvtárakba szervezzük azért, hogy könnyebben áttekinthetőek legyenek, könnyen megtaláljuk és differenciáljuk azokat. A Java nyelvben a csomag (package) hasonló jelentőséggel bír, mint az operációs rendszerben a mappa, sőt, tényleges mappaszerkezetet jelent az operációs rendszerben.

Legfontosabb feladatai:

* Struktúrát ad a projektnek: nagyobb projektek több száz, vagy akár több ezer osztályt is tartalmazhatnak. Ezeket pl. alkalmazás rétegek vagy funkciójuk szerint csoportosíthatjuk csomagok használatával.
* Névütközés feloldása: ha több azonos nevű osztály van, akkor ezeket megkülönböztetjük aszerint, hogy melyik csomagban vannak.
* Szabályozza a láthatóságot: a Java nyelvben az osztályok zártak, ami azt jelenti, hogy megadhatjuk, ki mit lásson belőle. Ha nem szabályozzuk, akkor a tagok alapértelmezett láthatósága csomag szintű (un. package private), azaz az ugyanazon csomagban lévő más osztályok hozzáférhetnek az osztályunkban lévő tagokhoz.

Csomagok használata

A Scanner osztály a java.util csomagban van. Hiába deklarálunk egy Scanner típusú változót, a fordító nem fogja megtalálni, mert nem tudja, hol keresse. Ehhez meg kell mondanunk azt is, hogy melyik csomagban van. Ezt többféleképpen is megtehetjük.

Az első, hogy az osztály neve előtt megadjuk a csomagot is ponttal elválasztva:

java.util.Scanner. Persze akárhányszor hivatkozunk a Scanner osztályra, mindig újra és újra így, teljes minősített névvel kell azt tennünk, ami hosszú és áttekinthetetlen kódhoz vezetne.

public class NameReader {

public static void main(String[] args) {

java.util.Scanner scanner = new java.util.Scanner(System.in);

System.out.println("What's your name?");

String name = scanner.nextLine();

System.out.println(name);

}

}

A második, hogy beimportáljuk az osztályt a fájlunkban közvetlen az osztály deklarációja fölé. Így ebben a fájlban bárhol használhatjuk pusztán osztálynévvel hivatkozva rá.

import java.util.Scanner;

public class NameReader {

public static void main(String[] args) { Scanner scanner = new Scanner(System.in);

System.out.println("What's your name?");

String name = scanner.nextLine();

System.out.println(name);

}

}

A harmadik, hogy beimportáljuk az adott csomagban lévő összes osztályt, így a Scanner osztályt is. Ebben az esetben minden, az importált csomagban lévő osztályra hivatkozhatunk kizárólag az osztály nevével. Ha egy csomagban lévő minden osztályt be szeretnénk importálni, akkor azt az osztálynév helyére tett \* karakterrel jelölhetjük (wildcard).

import java.util.\*;

public class NameReader {

public static void main(String[] args) { Scanner scanner = new Scanner(System.in);

System.out.println("What's your name?");

String name = scanner.nextLine();

System.out.println(name);

}

}

Leggyakrabban a második módszert használjuk, a harmadik pedig nem javasolt.

A Clean Code könyv szerint azonban inkább a wildcard használata javasolt, ugyanis ekkor nincs konkrét hivatkozás az osztályra, csak a csomagra, így a függőség lazább. Amennyiben sok osztály van, miért is akarnánk a kódot felesleges importokkal terhelni.

Természetesen ha az egyik osztály a saját csomagjában lévő osztályt akarja használni, akkor nincs szükség importra.

Van azonban egy csomag, amelynek az osztályait importálás nélkül is elérjük, a java.lang csomag. Ebben található például a System és a String osztály is. Mi van akkor, ha két ugyanolyan nevű osztályt szeretnénk használni? Az biztos, hogy ezek két különböző csomagban vannak, de ha mindkettőt importáljuk, akkor vajon melyikre gondolunk a kódban?

import java.util.Date; import java.sql.Date; // HIBA!!!

public class DateCalculator {

public static void main(String[] args) {

Date date = new Date(); // Ez most melyik Date?

// ...

}

}

Ezt a Java fordító nem engedi, jelzi, hogy a kódunk hibás. Két választásunk van:

* az egyiket importáljuk, a másiknál minősített nevet használunk:

import java.util.Date;

public class DateCalculator {

public static void main(String[] args) { Date date; java.sql.Date sqlDate; // ...

}

}

* mindkettő esetén teljes minősített nevet használunk:

public class DateCalculator {

public static void main(String[] args) { java.util.Date date; java.sql.Date sqlDate; // ...

}

}

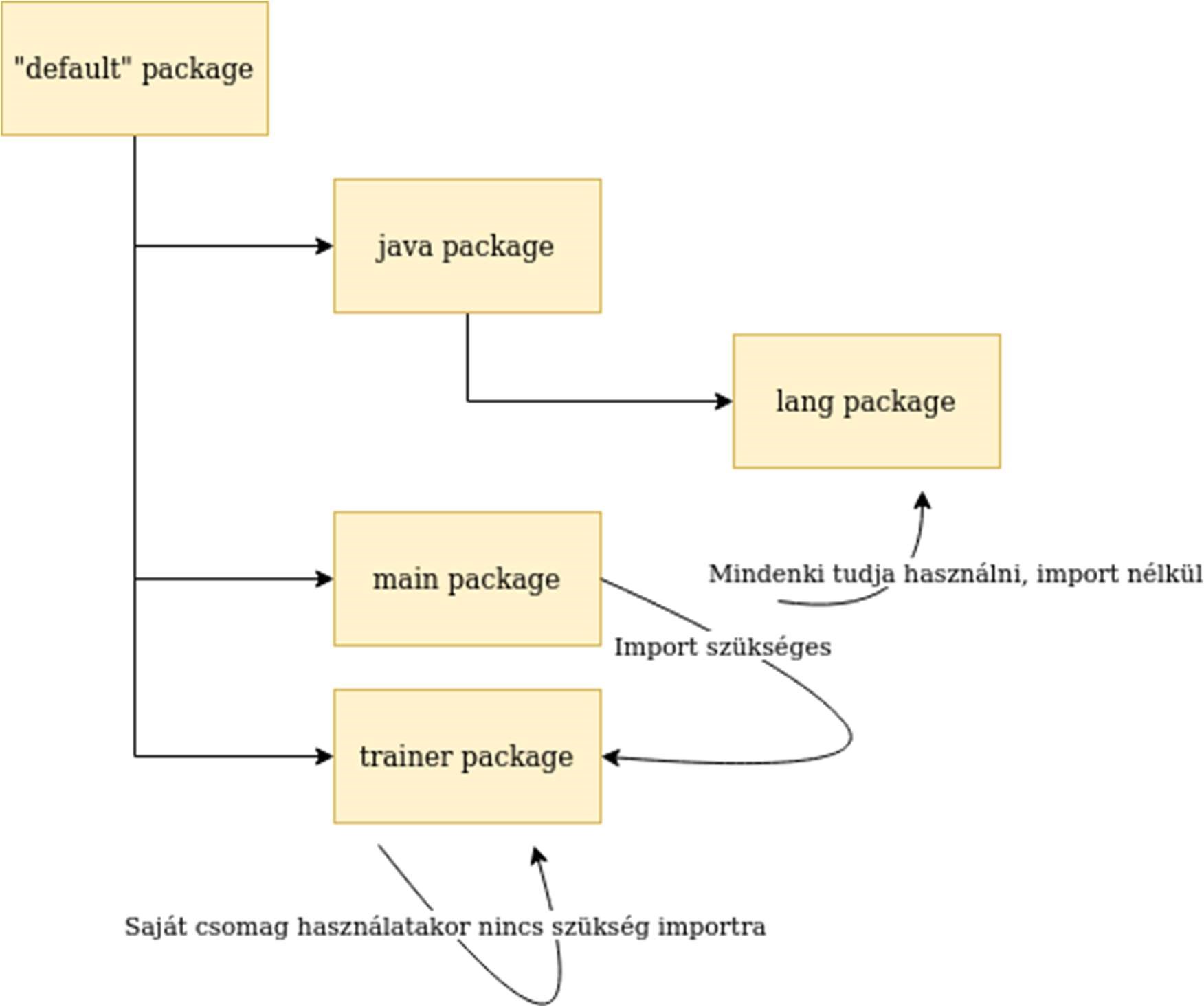
Hogy tudjuk megadni, hogy a mi osztályunk melyik csomagban van? A fájl első sora mindig a csomagdeklarációt tartalmazza.

package business;

public class BusinessLogic { // ...

}

Ha ez elmarad, akkor az osztályunk egy ún. default package-be kerül, ami a projektünk gyökérmappája. Ez nem javasolt, ezért mindig adjunk meg csomagnevet! Mivel nagyon sok cég fejleszt Java nyelven, és gyakran előfordul, hogy azonos osztályneveket használnak, ezért a javasolt csomagnév tartalmazza a domain nevet visszafelé. Például a Training360 domain neve training360.com, ezért használható lenne a com.training360 csomagnévként.



Csomagok

A csomagok valójában fizikailag könyvtárak.

A csomagokban újabb csomagokat is létre lehet hozni, így megvalósítva egy fa struktúrát. Azonban míg a könyvtárakat Windows esetén backslash (\) karakterrel választjuk el, Linux esetén perjellel (/), Java csomagokat pont karakterrel (.). Ezért a java.util csomag valójában a java csomagban lévő util csomag.

Ha importálunk egy csomagot, akkor csak az abban lévő osztályokat látjuk, az alcsomagokban lévő osztályokat nem.

Amennyiben beírjuk az osztály nevét, és az egy másik csomagban van, az IDEA segít nekünk az importban. Kattintsunk az osztály nevére, amit nem talál (ezért piros), és amint aláhúzásra kerül, nyomjuk meg az Alt + Enter billentyűzetkombinációt.

Ellenőrző kérdések

* Mire használatosak a Java csomagok?
* Mi a csomag fizikai fájlrendszerbeli megfelelője?
* Hogyan hozunk létre új csomagokat?
* Hogyan használunk más csomagban lévő osztályokat?
* Hogyan kezeljünk a névütközést?

Feladatok

A videóban szereplő feladatok a demos/src/main/java/packages elérési úton vannak.

A packages csomagba dolgozz!

A feladatok megoldásai a solutions/packages elérési úton vannak.

Köszöntés

Hozz létre a packages.greetings csomagban egy Greeter osztályt, melynek legyen egy public void sayHello() metódusa, mely kiírja, hogy Hello World!.

Hozz létre a packages.main csomagban egy MainProgram osztályt, melynek legyen egy main() metódusa. Ez példányosítsa a Greeter osztályt, majd hívja meg annak sayHello() metódusát!

Forrás

OCA - Chapter 1/Understanding Package Declarations and Imports

Teszt

Milyen módosítással NEM lehet elérni, hogy az alábbi program leforduljon? Account.java

package account;

// 1 public class Account { //...

}

AccountManager.java

package manager;

// 2 public class AccountManager {

public static void main(String[] args) { Account account = new Account(500); // 3 //...

}

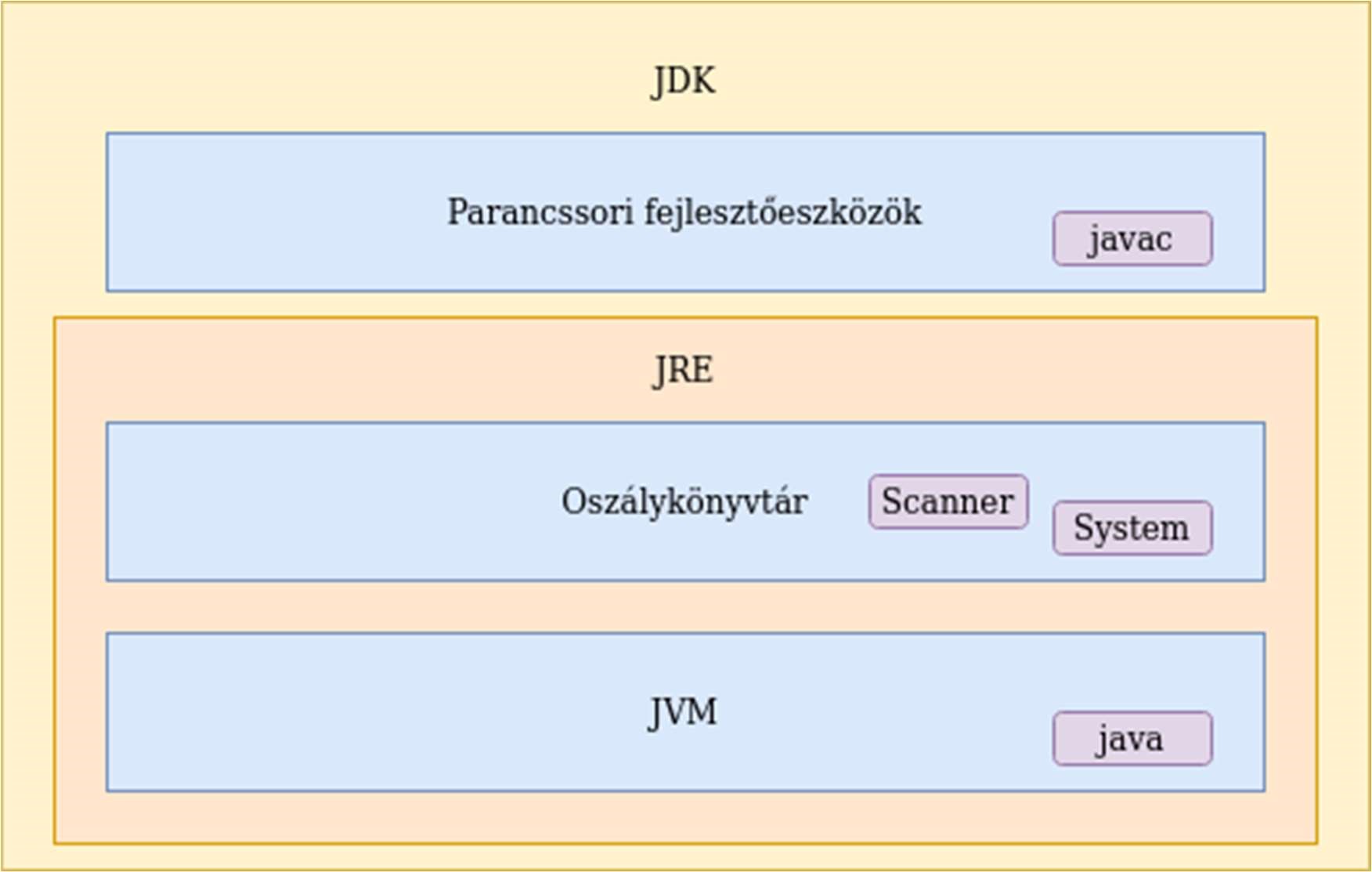
}

* A program lefordul, ha az Account osztályt a manager csomagba mozgatjuk.
* ☒ A program lefordul, ha a // 3 sort lecseréljük az account.Account account = new Account(500); szövegre.
* A program lefordul, ha // 2-es sorba elhelyezzük az import account.\*; szöveget.
* A program lefordul, ha a // 2-es sorba elhelyezzük az import account.Account; szöveget.

Amennyiben beimportálunk egy csomagot, látjuk az abban lévő alcsomagok osztályait is?

* Igen
* ☒ Nem

Java API (javaapi)



JDK és JRE

A Java 11 már nem tartalmaz külön JRE-t, csak a JDK-t lehet letölteni.

A Java nyelv nagyon gazdag osztálykönyvtárral rendelkezik csomagokba szervezve, melyet verzióról verzióra újabbakkal egészítenek ki, így nem kell minden feladatra saját osztályt gyártanunk. Legyen az párhuzamosság kezelése, naplózás, dátum- és időkezelés, reguláris kifejezések, XML feldolgozás, adatbázis-kezelés, fájlkezelés, felhasználói felületek és még sorolhatnánk, a Java alapkönyvtáraiban ezekre mind találunk megoldást. Teljes dokumentációt találhatsz a Java® Platform, Standard Edition & Java Development Kit Version 15 API Specification oldalon. Ez egy un. JavaDoc eszközzel kerül kigenerálásra. Sőt, ha kíváncsi vagy az osztályok forráskódjára, akkor azt is megtalálod a JDK telepítési könyvtárában a lib\src.zip állományban.

Az API dokumentáció jelentősen megváltozott a frissebb verziókban. A 9-es verzióban lett kereshető, és a 11-es verzióban eltűnt a több ablakos nézet.

Ellenőrző kérdések

* Mi az az osztálykönyvtár?
* Hogyan van az osztálykönyvtár szervezve?
* Hogyan kell elemeket felhasználni az osztálykönyvtárból?
* Milyen esetekben nem kell import kulcsszót használni?
* Hol található az osztálykönyvtár dokumentációja?

Feladatok

A javaapi csomagba dolgozz!

A feladatok megoldásai a solutions/javaapi elérési úton vannak.

Navigáció a dokumentációban

Keresd ki a Scanner osztályt az API dokumentációban, majd annak a nextLine() és nextInt() metódusát!

String osztály

Keresd ki a JDK API dokumentációból, a String osztálynál, hogyan lehet egy karakterláncot nagybetűssé tenni!

Írj egy Upper osztályt, ami a Hello World! szöveget nagybetűssé alakítja!

Teszt

Melyik osztály NEM a java.lang csomagban van?

* ☒ Scanner
* String
* System

Melyik állítás igaz?

* A Java osztálykönyvtárat külön le kell tölteni.
* A Java osztálykönyvtár zárt, nem lehet hozzáférni a dokumentációjához.
* A Java osztálykönyvtár nem csomag alapú.
* ☒ A Java osztálykönyvtár újrafelhasználható osztályokat tartalmaz csomagokba rendezve.

JAR állomány (distjar)

Egy Java alkalmazás több száz, vagy akár több ezer osztályból is állhat. Az alkalmazás terjesztése sokkal könnyebb, ha ezek valahogyan egységbe vannak foglalva. A JAR állomány tulajdonképpen a Java class fájlokat és az alkalmazáshoz tartozó egyéb erőforrás állományokat tartalmazza ZIP formátumba összecsomagolva, megkönnyítve ezzel a teljes alkalmazás hordozását. JAR állomány készítését a Maven is támogatja. mvn clean package

A parancs hatására a Maven kitörli az előző fordítás eredményét (clean), azaz törli a target könyvtárat, majd újra fordítja az osztályokat és elkészíti a JAR állományt alapértelmezetten a target könyvtárba. A JAR neve az artifactId és a version összefűzve. Az összecsomagolt osztályok használatához a JVM-nek tudnia kell, hogy ezek hol vannak, azaz a JAR állományt hozzá kell adni a classpathhoz, és meg kell adnunk a futtatandó osztályt benne teljes minősített névvel.

java -classpath target\distjar-1.0-SNAPSHOT.jar distjar.HelloWorld

Futtatható JAR állomány

Amennyiben az összecsomagolt osztályok között van main() metódust tartalmazó, akkor a JAR állomány önállóan futtatható is lehet. Ehhez szükséges, hogy maga a JAR állomány tartalmazzon hivatkozást a main() metódust tartalmazó osztályra, melyet a pom.xml állományban tudunk konfigurálni.

<build>

<plugins>

<plugin>

<artifactId>maven-jar-plugin</artifactId>

<version>3.2.0</version>

<configuration>

<archive>

<manifest>

<mainClass>distjar.HelloWorld</mainClass>

</manifest>

</archive>

</configuration>

</plugin>

</plugins>

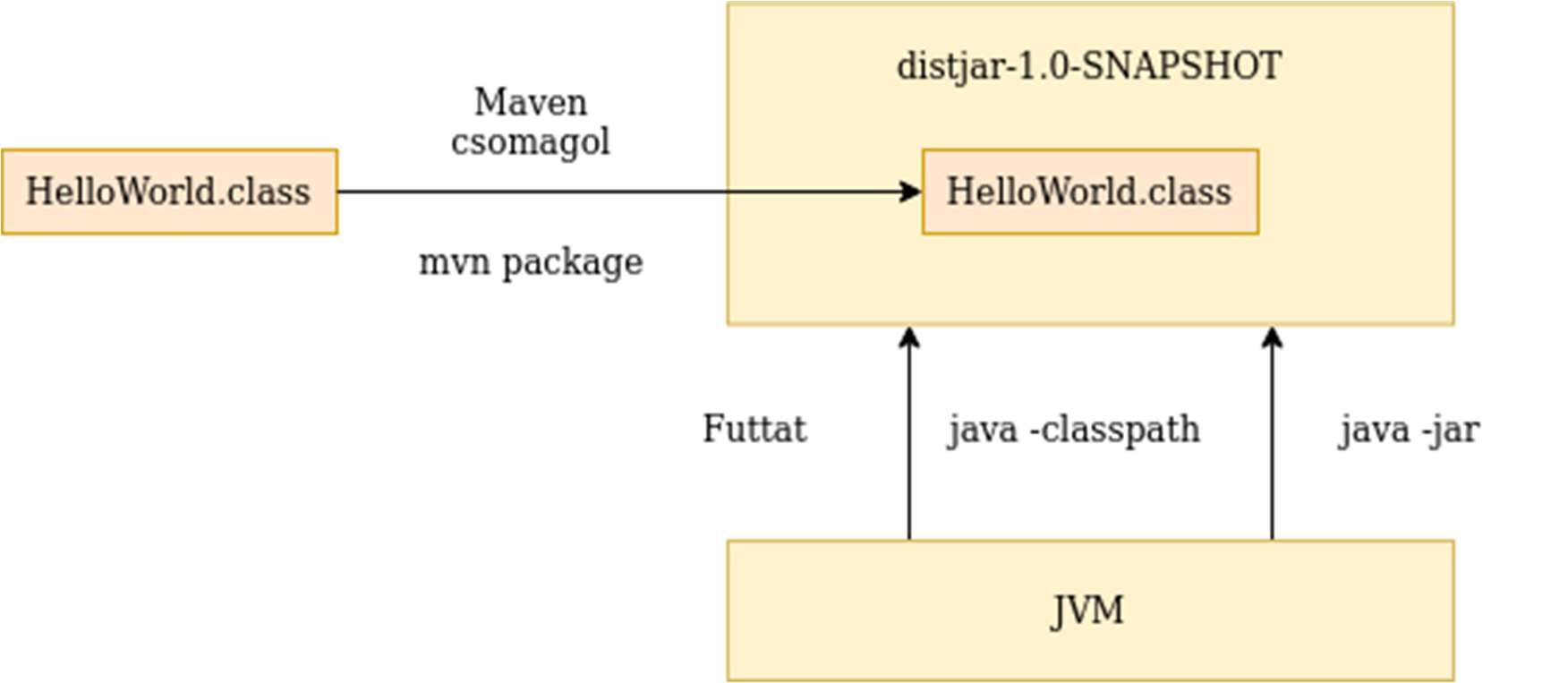
</build>

Nem kötelező a <version> tag használata, azonban erősen javasolt. Elhagyása esetén a Maven WARNING-okat fog kiírni.

A futtatáshoz szükséges információkat JAR állományon belül a META-INF/MANIFEST.MF fájl tartalmazza. Ebben az esetben az alkalmazás a -jar kapcsolóval indítható. java -jar target\distjar-1.0-SNAPSHOT.jar

Amennyiben a teljes alkalmazás több JAR állományból áll, akkor az összes JAR-t a classpathra kell tenni úgy, hogy a -classpath kapcsolónak paraméterül több JAR állományt adunk át.

A JAR állományt a következő paranccsal lehet kitömöríteni: jar xvf distjar-1.0-SNAPSHOT.jar



JAR

Ellenőrző kérdések

* Mire valók a JAR állományok?
* Hogyan épülnek fel a JAR állományok?
* Hogyan lehet futtatni egy JAR állományban lévő main() metódusban lévő osztályt?
* Hogyan lehet futtathatóvá tenni egy JAR állományt?

Feladat

A distjar csomagba dolgozz!

A feladatok megoldásai a solutions/distjar elérési úton vannak.

JAR állomány készítése

Hozz létre egy teljesen új projektet (pl. a C:\training\distjar könyvtárba distjar néven), ami egy distjar.Main osztályt tartalmaz main() metódussal. Ez a Hello User! szöveggel üdvözli a felhasználót!

Készíts JAR-t az alkalmazásból az mvn clean package parancs kiadásával!

Futtatható JAR állomány készítése

A pom.xml állományt egészítsd ki a következővel:

<build>

<plugins>

<plugin>

<artifactId>maven-jar-plugin</artifactId>

<configuration>

<archive>

<manifest>

<mainClass>distjar.Main</mainClass>

</manifest>

</archive>

</configuration>

</plugin>

</plugins> </build>

Készíts JAR-t az alkalmazásból az mvn clean package parancs kiadásával!

A JAR állomány futtatható a java -jar target\distjar-1.0-SNAPSHOT.jar paranccsal.

Teszt

Mire való a JAR állomány?

* ☒ Egy állományba lehet csomagolni az alkalmazáshoz tartozó class és erőforrás állományokat.
* Csak így futtatható egy Java alkalmazás.
* Egy olyan állomány, amit úgy futtathatunk, hogy nem is kell hozzá JVM.
* Csak az alkalmazáshoz tartozó erőforrás állományokat tartalmazza.

Hogyan készíthetünk futtatható JAR állományt Mavennel?

* Parancssorból kiadjuk az mvn clean package runnable distjar.GentlemanMain parancsot, ahol a distjar a projekt neve, a GentlemanMain pedig a main() metódust tartalmazó osztály neve.
* Parancssorból kiadjuk a java -jar target\distjar-1.0-SNAPSHOT.jar parancsot
* ☒ Kiegészítjük a pom.xml állományt egy build elemmel, mely a konfigurációt tartalmazza, majd parancssorból kiadjuk az mvn clean package parancsot.
* Kiegészítjük a pom.xml állományt egy build elemmel, mely a konfigurációt tartalmazza, majd parancssorból kiadjuk a java -classpath target\distjar1.0-SNAPSHOT.jar parancsot.

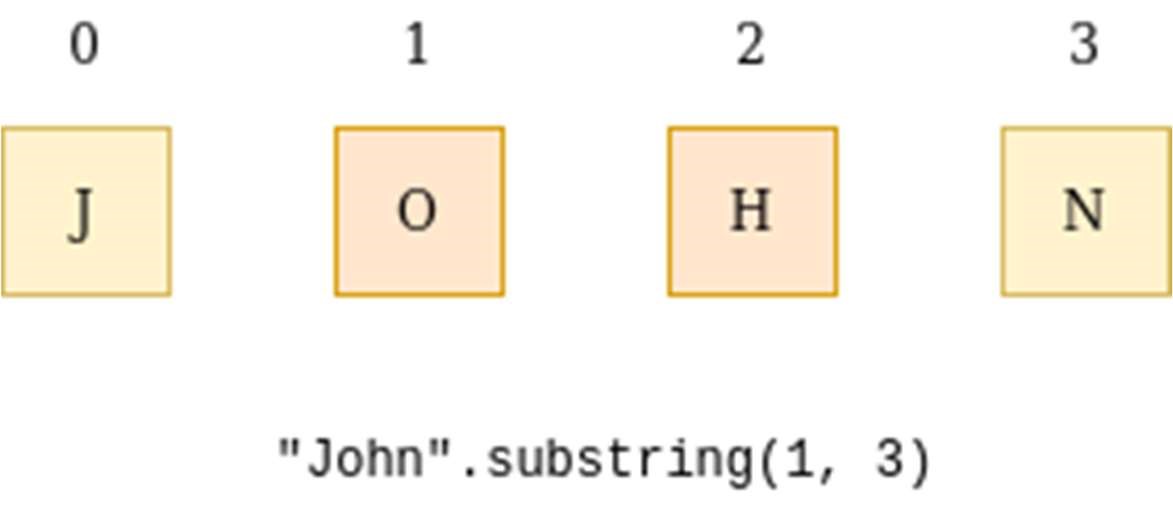
Szöveges típus (stringtype)

Javaban a karakterláncokat, szövegeket a String osztály reprezentálja. Ugyanúgy példányosítható, mint más osztályok, de a String literált a JVM példányosítja helyettünk. A szöveget idézőjelek közé kell tennünk, például "alma". Az üres szöveg literálja a "", ami nem ugyanaz, mint a null. Szövegek összefűzésére (konkatenálására) használható a + operátor. Mivel a String típusú változó referenciát tárol, ezért nem használható két szöveg összehasonlítására a == operátor, ezeket a String equals() metódusával tudjuk vizsgálni.

Metódusok

* boolean equals(String str): a szöveg betűről betűre megegyezik-e a paraméterként átadott másik szöveggel.
* int indexOf(String substring): a paraméterként átadott szöveg hol kezdődik az adott szövegben. Ha nem található, akkor -1-gyel tér vissza. A karakterek indexelése, sorszámozása 0-ról indul.
* int indexOf(String substring, int startIndex): a paraméterként átadott szöveg hol kezdődik az adott szövegben, de a keresést csak a startIndex-től kezdi.
* int length(): a szöveg hosszát adja vissza.
* String substring(int beginIndex): visszaadja a szöveg egy részét a megadott indextől kezdve a végéig.
* String substring(int beginIndex, int endIndex): visszaadja a megadott indexek közé eső szövegrészt. A bal oldali indexű karaktert még tartalmazni fogja, a jobb oldalit nem

Az indexof() metódus negatív értéket ad vissza, ha nem találja a keresendő szöveget.



substring metódus működése

Vigyázni kell a műveletek sorrendjére is!

A "John" + 4 + 4 kifejezés értéke John44, mert összefűzi a John és 4 értékeket, és utána fűz még egy 4 értéket. a 4 + 4 + "John" kifejezés először elvégzi a 4 + 4 műveletet, aminek eredménye 8, és csak utána fűzi hozzá a John szöveget.

Ellenőrző kérdések • Milyen típus a String?

* Hogyan definiálható String literál?
* Hogyan lehet két String-et összekapcsolni?
* Hogyan lehet egy String-et és egy primitív típust összekapcsolni?
* Hogyan lehet két String-et összehasonlítani?

Feladat

A videóban szereplő feladatok a demos/src/main/java/stringtype elérési úton vannak.

A stringtype csomagba dolgozz!

A feladatok megoldásai a solutions/stringtype elérési úton vannak.

String műveletek

A stringtype.StringTypeMain osztály main metódusában definiálj prefix néven egy String típusú változót, és add értékül a "Hello " literál értékét.

Definiálj name néven egy String típusú változót, és add értékül a John Doe literált.

Definiálj egy message változót, mely az előző két változó, összefűzve.

A message változó értékét írd felül a message változó értékével úgy, hogy hozzákapcsolod még a 444 int literál értékét.

A b logikai változó tartalmazza, hogy a message értéke megegyezik-e a "Hello John Doe" literál értékével.

A c logikai változó tartalmazza, hogy a message értéke megegyezik-e a "Hello John Doe444" értékkel.

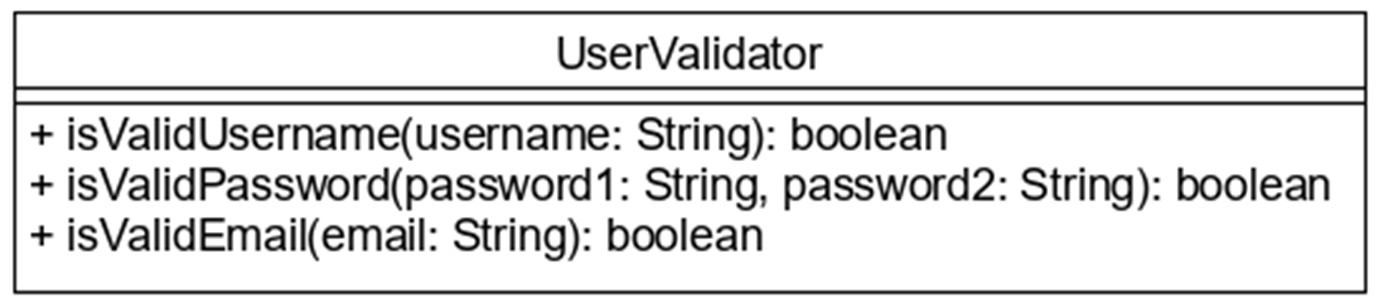
Konkatenálj össze két üres String-et, és írd ki az értékét! Hány karakter hosszú lesz?

Írd ki külön sorban, külön utasításokban a következőket:

* Az Abcde String hossza
* Az első és harmadik karaktere (0-tól indexelünk) vesszővel elválasztva
* Az elsőtől a harmadik karakterig tartó részlete

Regisztrációs adatok vizsgálata

Amikor egy weboldalon regisztrációkor megadjuk az adatainkat, gyakran kapunk olyan visszajelzést, hogy a jelszavunk nem elég erős, vagy nem valid email címet adtunk meg. Készíts egy UserValidator osztályt a stringtype.registration csomagba, mely a regisztrációkor megadott adatokat validálja.



UserValidator UML

Regisztrációkor meg kell adnunk a felhasználónevet, a jelszót kétszer és az email címet. A három metódus ezeket ellenőrzi:

* A felhasználónév megadása kötelező.
* A jelszó legalább 8 karakter hosszú kell legyen, és a két megadott jelszónak egyeznie kell.
* Az email címben kell lennie @ karakternek és valamikor utána (de nem közvetlenül) pontnak. A @ karakter nem lehet az első, az őt követő pont pedig az utolsó.

Tételezzük fel, hogy egyik érték sem lehet null, mivel konzolról kerül beolvasásra, ezért maximum üres string ("").

Készíts ugyanoda egy Registration osztályt, ahol a main() metódusban kérd be az adatokat! Írd ki a felhasználónak egyenként, hogy a megadott adat helyes vagy helytelen! Használd a háromoperandusú operátort!

Teszt

Mi lesz az alábbi műveletsor eredménye?

String message = "Hello Java"; int index = message.indexOf("J"); String word = message.substring(index);

System.out.println(word.length());

* ☒ 4
* 5
* 6
* “Hello”
* “Java”

Hogyan lehet összehasonlítani két karakterláncot?

* Az == operátorral
* ☒ Az equals() metódussal
* A = operátorral
* Az indexOf() metódussal

Mi ad vissza a "abcdxyvz".substring(2, 5)"?

* ☒ cdx
* bcd
* bd
* cx ### Dátum- és időkezelés alapok (introdate)

A Java fejlődése során több jelentős változáson ment át a dátum-idő kezelés. Az első eszköz erre a java.util.Date osztály volt, ami mellé később a java.util.Calendar osztály társult. A Date egy időpontot reprezentált, míg a Calendar szállította a dátumok kezeléséhez szükséges elemeket, mint például a hét napjainak neveit és még sok minden mást. Ez azonban a gyakorlatban túl bonyolult lett, és ezért a Java 8 kiadásakor megjelentek a dátum és időkezelést együtt végző java.time csomag osztályai:

* LocalTime, mely csak időpontot tárol, ezredmásodperces pontossággal
* LocalDate, mely csak dátumot tárol
* LocalDateTime mely dátumot és időpontot is tárol, ezredmásodperces pontossággal

Objektumai nem a szokásos new operátorral hozhatók létre, hanem a különböző paraméterezésű of() metódusokkal. Látható, hogy akár a hónap nevét is meg lehet adni.

LocalDate start = LocalDate.of(2015, 1, 20);

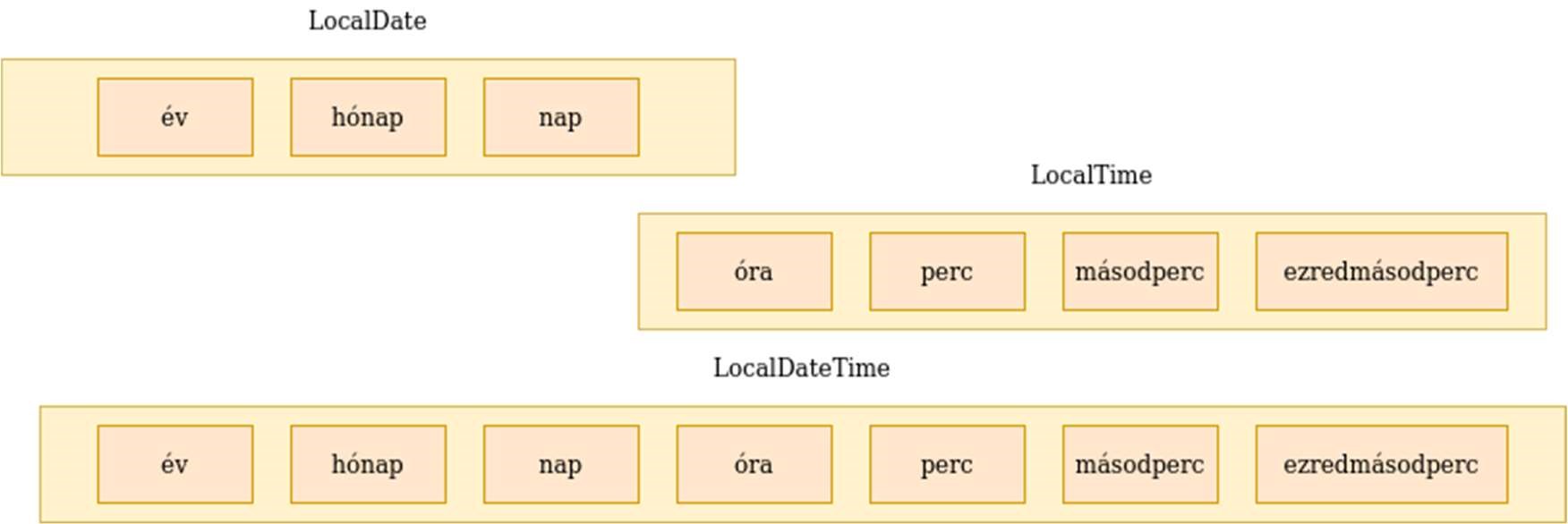
LocalDate stop = LocalDate.of(2015, Month.JANUARY, 30);

LocalDateTime from = LocalDateTime.of(2015, 1, 20, 10, 15);

LocalDateTime to = LocalDateTime.of(2015, Month.JANUARY, 30, 10, 15);

LocalDateTime now = LocalDateTime.of(2015, Month.JANUARY, 20, 10, 15, 30);

LocalTime begin = LocalTime.of(10, 15);



Dátum és időkezelés

Mindhárom osztály esetében rendelkezésünkre áll a now() metódus, amely a számítógép rendszeridejének megfelelő dátumot és/vagy időt adja vissza.

A Java képes a dátumok és idők olvasható megjelenítésére az ISO-8601 szabványnak megfelelően.

LocalDateTime localDateTime = LocalDateTime.now(); System.out.println(localDateTime); // 2015-01-20T10:15:30 Egyik osztály sem alkalmas időzónák kezelésére.

Ellenőrző kérdések

* Mit tárol a LocalDateTime objektum?
* Hogyan lehet egy LocalDate vagy LocalDateTime objektumot létrehozni?
* Mire való a statikus now() metódus?
* Milyen pontossággal lehet az időt megadni?

Feladat

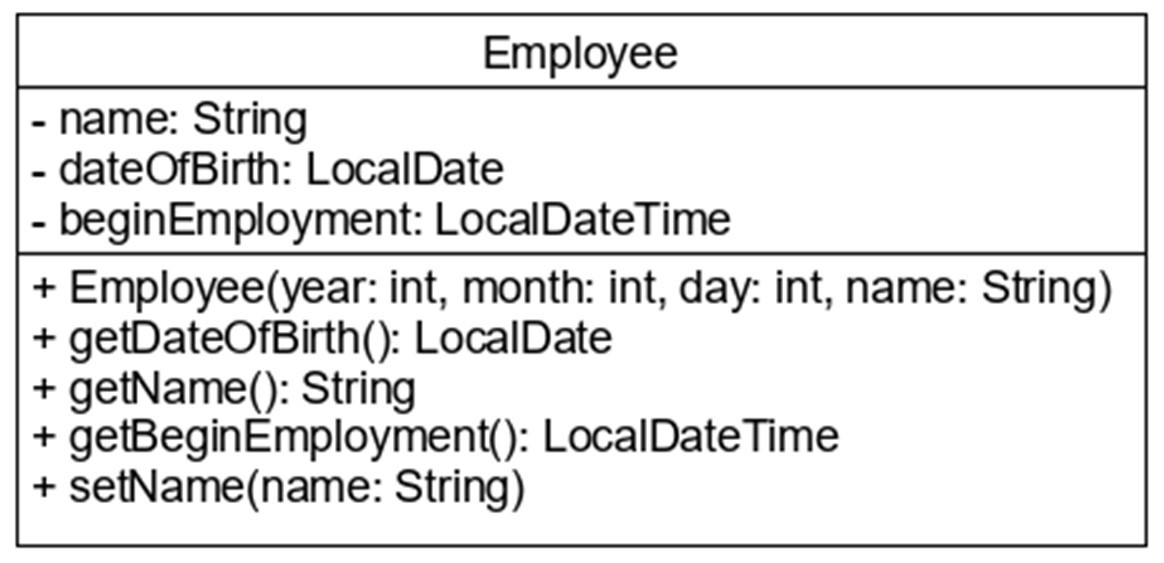
A videóban szereplő feladatok a demos/src/main/java/introdate elérési úton vannak.

A introdate csomagba dolgozz!

A feladatok megoldásai a solutions/introdate elérési úton vannak.

Employee

Egy olyan osztályt - Employee - akarunk létrehozni, amely egy alkalmazott felvételekor rögzíti a munkába álló főbb adatait. Rögziti a nevét name, születési dátumát dateOfBirth és a belépés pillanatát beginEmployment (amikor az objektumot létrehozzuk). Az osztály konstruktorában adjuk meg a felvett adatokat, és ezekből a konstruktor létrehozza a tárolandó objektumokat. Mivel a felvétel után csak a dolgozó neve módosítható, getter metódus mindegyik attribútumra kell, de setter metódus csak a nevére szükséges.



Employee UML

Az osztály tesztelését az EmployeeTest osztály main() metódusában végezd el! Az alkalmazott belépéséhez szükséges adatokat olvasd be, majd az objektum létrehozása után írd ki annak minden adatát!

Performance

Adott előadó fellépéseit szeretnénk nyilvántartani egy Java alkalmazásban, ehhez viszont szükségünk van egy, az adatokat rögzítő Performance osztályra. A fellépés dátumát (date), előadót (artist), és kezdési és befejezési idejét (startTime és endTime) szeretnénk tárolni. Az egyes attribútumok természetesen lekérdezhetők, így getter metódus mindegyikre kell, de az adatok rögzítése után azok változtatására már ne legyen lehetőség, azaz nincs szükség setter metódusokra.

Az osztály tesztelését a PerformanceTest osztály main() metódusában System.out kiírások formájában végezzük. Az objektum létrehozásához szükséges adatokat most nem kell beolvasnod a felhasználótól.

Írj egy getInfo() metódust, ami a következő formátumban adja vissza az előadás adatait:

“Queen: 1989-06-02 18:00 - 20:00”

Teszt

Mely dátum- és időkezelő osztályok jelentek meg a Java 8-ban? (3 helyes válasz)

* Date
* ☒ LocalDate
* ☒ LocalTime
* ☒ LocalDateTime
* Calendar

Hogyan NEM lehet egy LocalDate példányt létrehozni?

* LocalDate.of(2017, 4, 17)
* LocalDate.now()
* ☒ new LocalDate(2017, 3, 17)
* LocalDate.of(2017, Month.APRIL, 17)

Bevezetés a vezérlési szerkezetekbe (introcontrol)

Elágazás

Ha egy utasítást egy feltételtől függően szeretnénk végrehajtani, akkor elágazást kell használnunk. Az elágazás egy fejből és egy törzsből áll. A fejben kell adnunk egy feltételt, amely mindig egy logikai értéket adó kifejezés. Ha a kiértékelése igaz, akkor végrehajtja a törzset, ami egy blokk, mely utasításokat tartalmazhat.

System.out.println("Check");

if ((x % 2) == 0) {

System.out.println("Even");

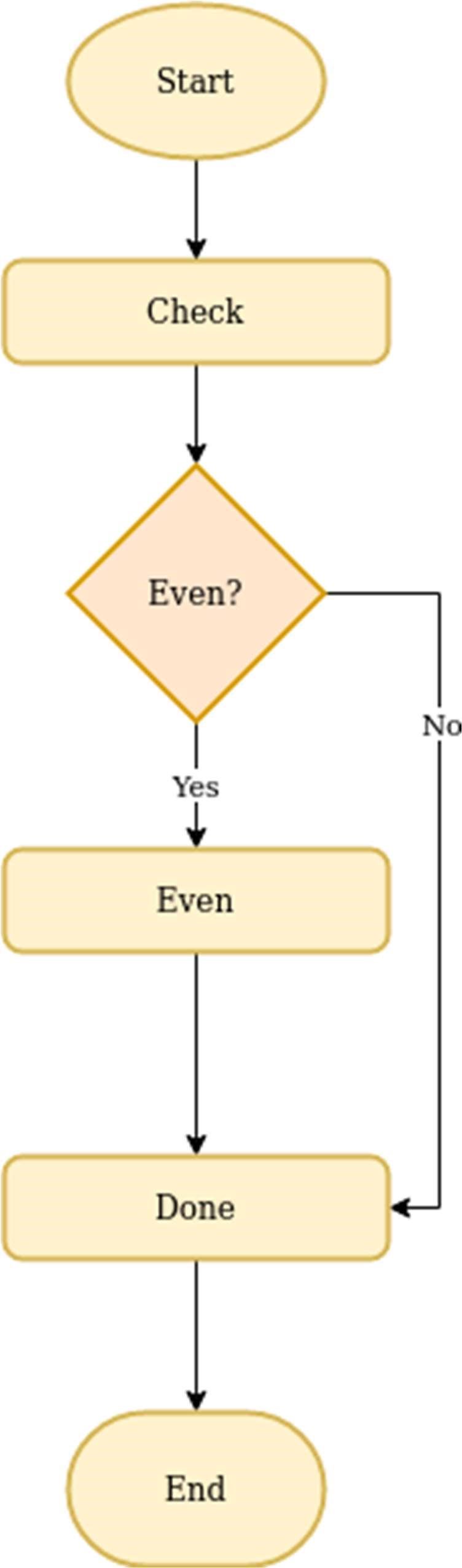
}

System.out.println("Done"); // 1

A példában, ha az x osztható 2-vel, akkor kiírja, hogy Even, majd az 1-essel jelölt sorban történik a végrehajtás, és kiírja, hogy Done.

Ha nem osztható kettővel, átugorja a végrehajtás a feltétel törzsét, és kiírja, hogy Done.

Az algoritmusokat, azaz a lépések sorozatát un. folyamatábrával (flowchart) lehet grafikusan ábrázolni. Téglalapban találhatóak a lépések. A nyilak a lépések egymásutániságát jelölik. Az elágazást rombusz jelöli.



Elágazás

Meg lehet adni egy un. else ágat, ekkor ha a feltétel, igaz, akkor az első blokkot, ha hamis, akkor a második (else ágon lévő) blokkot hajtja végre.

System.out.println("Check");

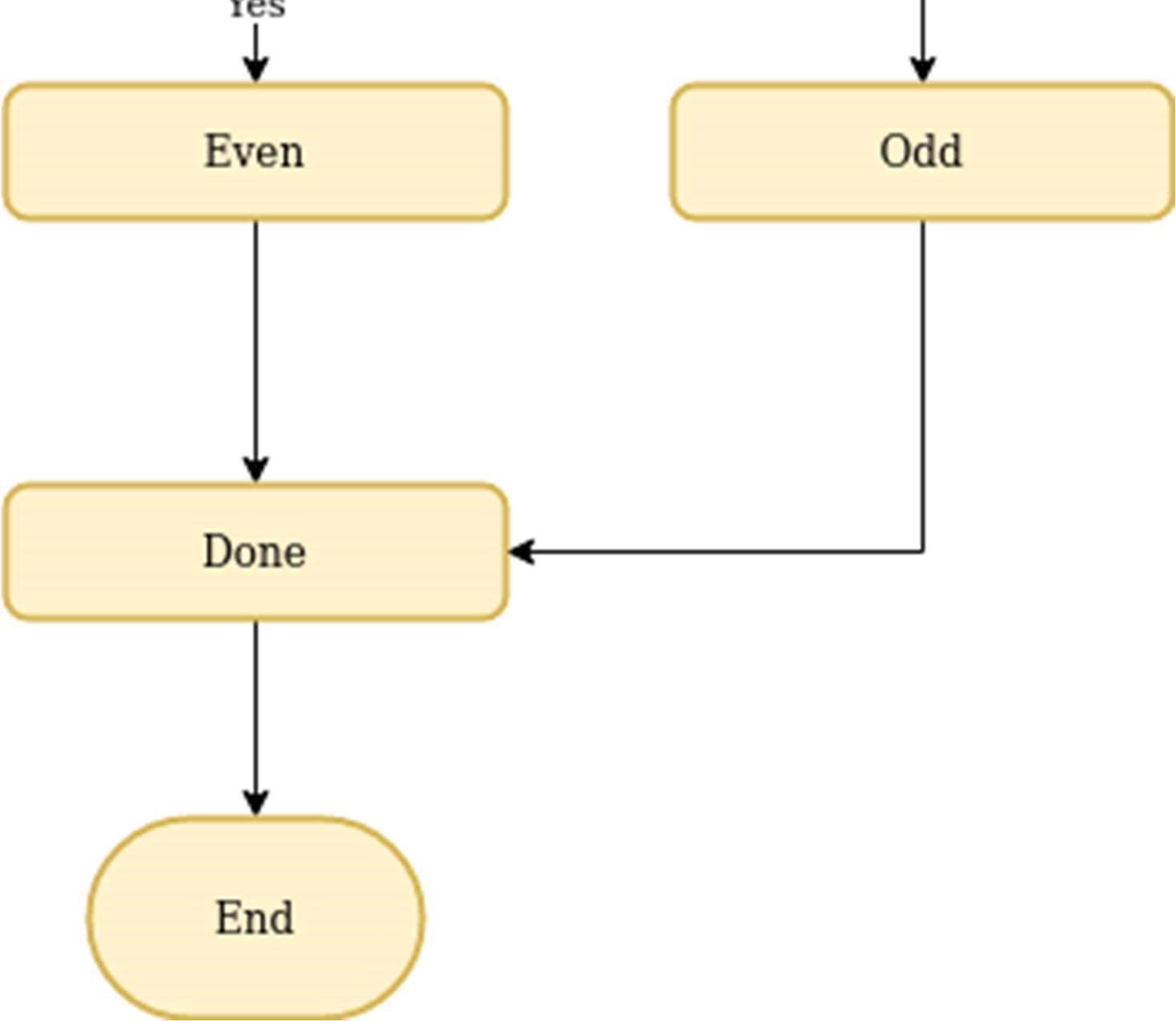
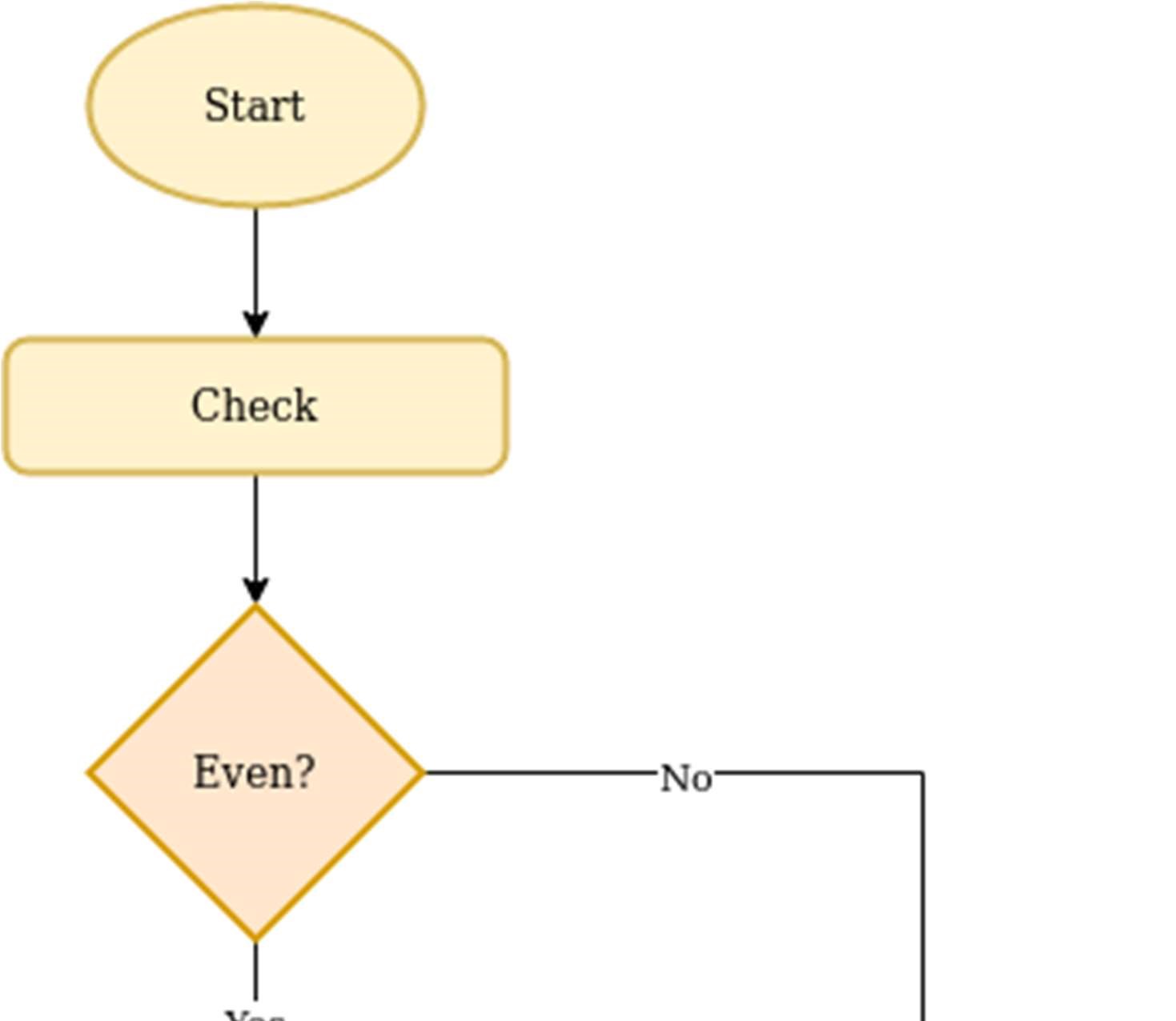
if ((x % 2) == 0) { System.out.println("Even");

} else {

System.out.println("Odd");

}

System.out.println("Done");



Else ág

Ciklus

Ha egy utasításblokkot többször szeretnénk végrehajtani, akkor nem kell az utasításokat sokszor kiadnunk egymás után, hanem elég a programban megadnunk, hogy mely utasításokat és meddig szeretnénk végrehajtani. Ezt a szerkezetet ciklusnak nevezzük. A Java nyelv többféle megoldást nyújt erre. Ezek közül az egyik igen gyakran használt a for kulcsszóval jelölt ciklus. Fejből és törzsből áll. A fejrésze három részből áll: inicializációs rész, feltétel és léptetés. Törzsében azok az utasítások szerepelnek, amelyeket többször is végre szeretnénk hajtani.

for (int i = 0; i < 10; i++) {

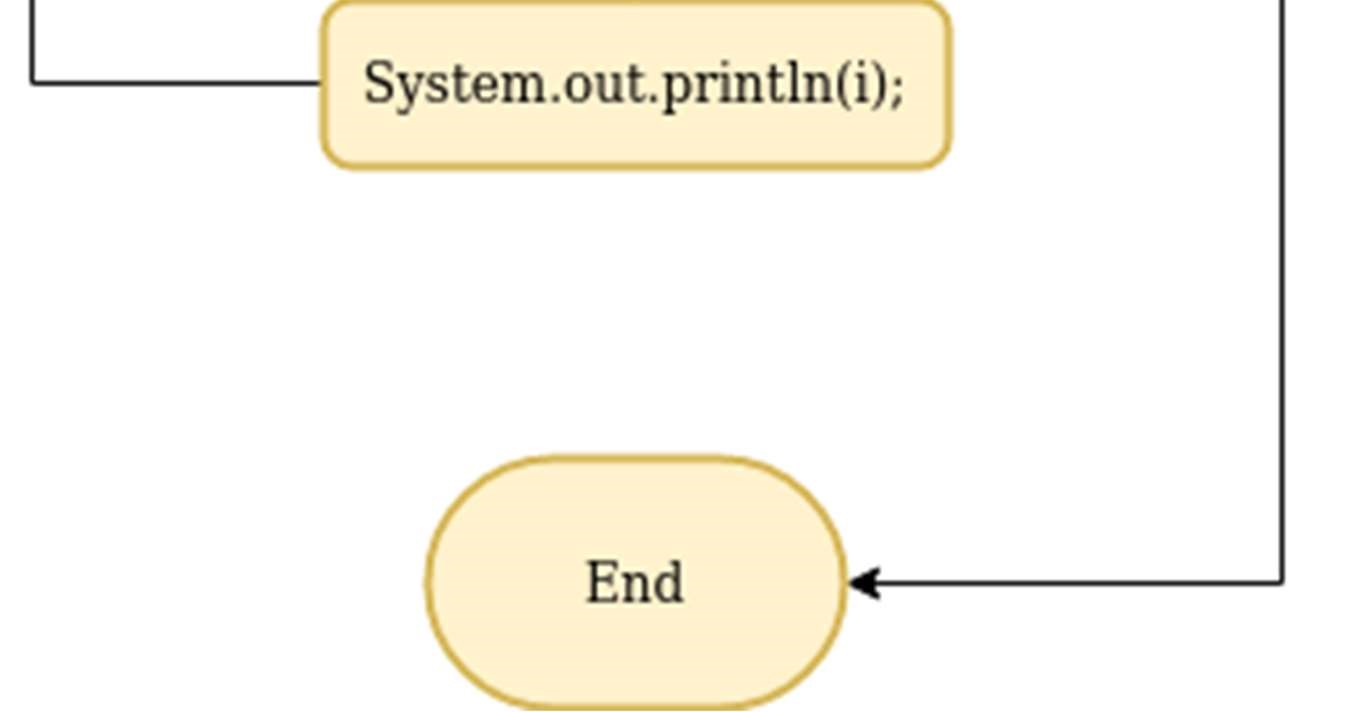
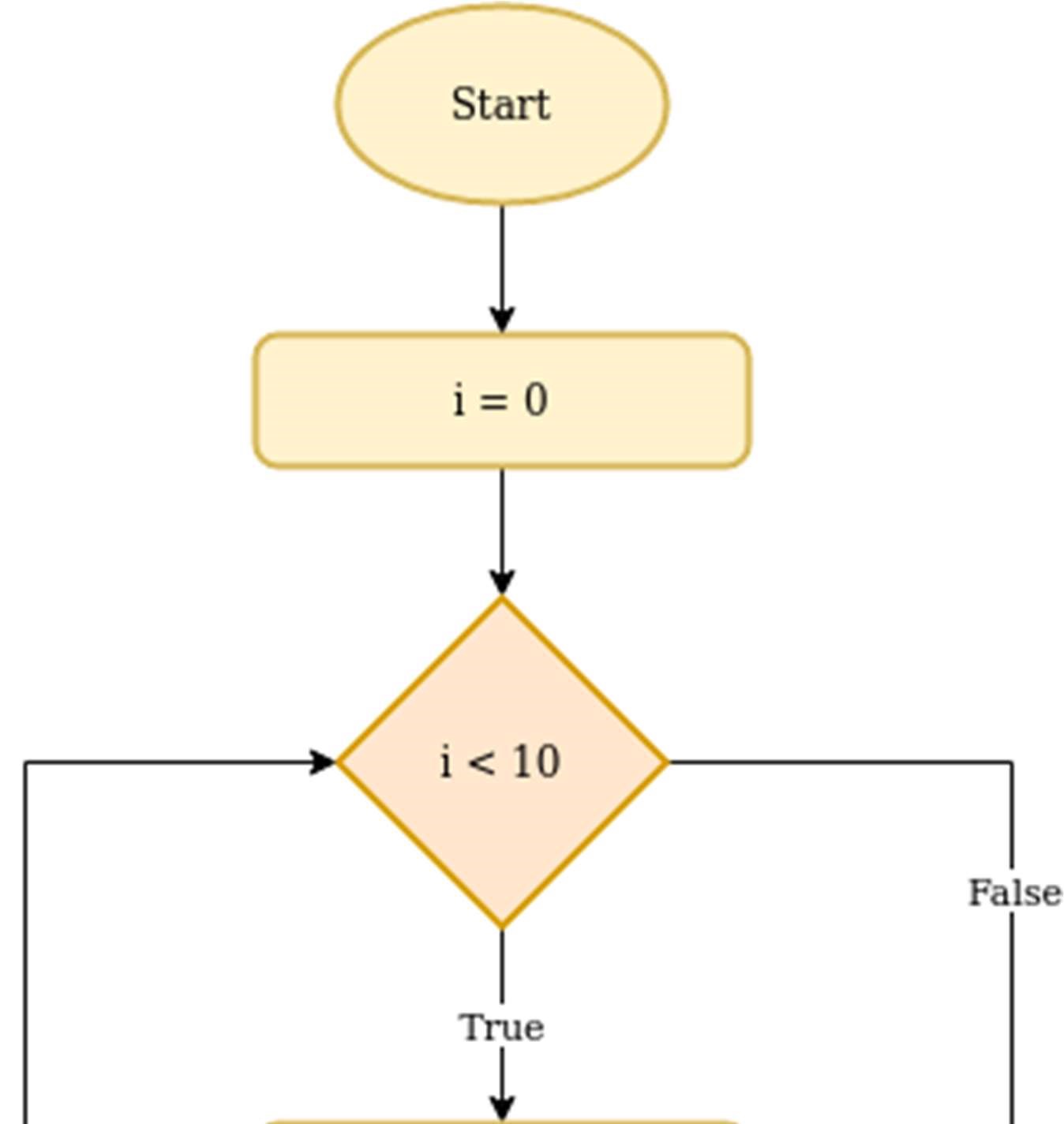
System.out.println(i);

}

Az inicializációs részben a ciklusváltozó kezdőérték adása szerepel. Nagyon gyakran itt is deklaráljuk, ebben az esetben ez a cikluson kívülről nem érhető el. Először ez a rész fut le, méghozzá csak egyetlen egyszer. Ezután megvizsgálja a feltételt, és amennyiben igaz a kiértékelése, akkor végrehajtja a ciklusmagban megadott utasításokat, majd elvégzi a ciklusfejben megadott léptetést, és újra megvizsgálja a feltételt. Mindezt addig ismétli, amíg a feltétel hamissá nem válik, vagy valamilyen módon ki nem ugrunk a ciklusból.

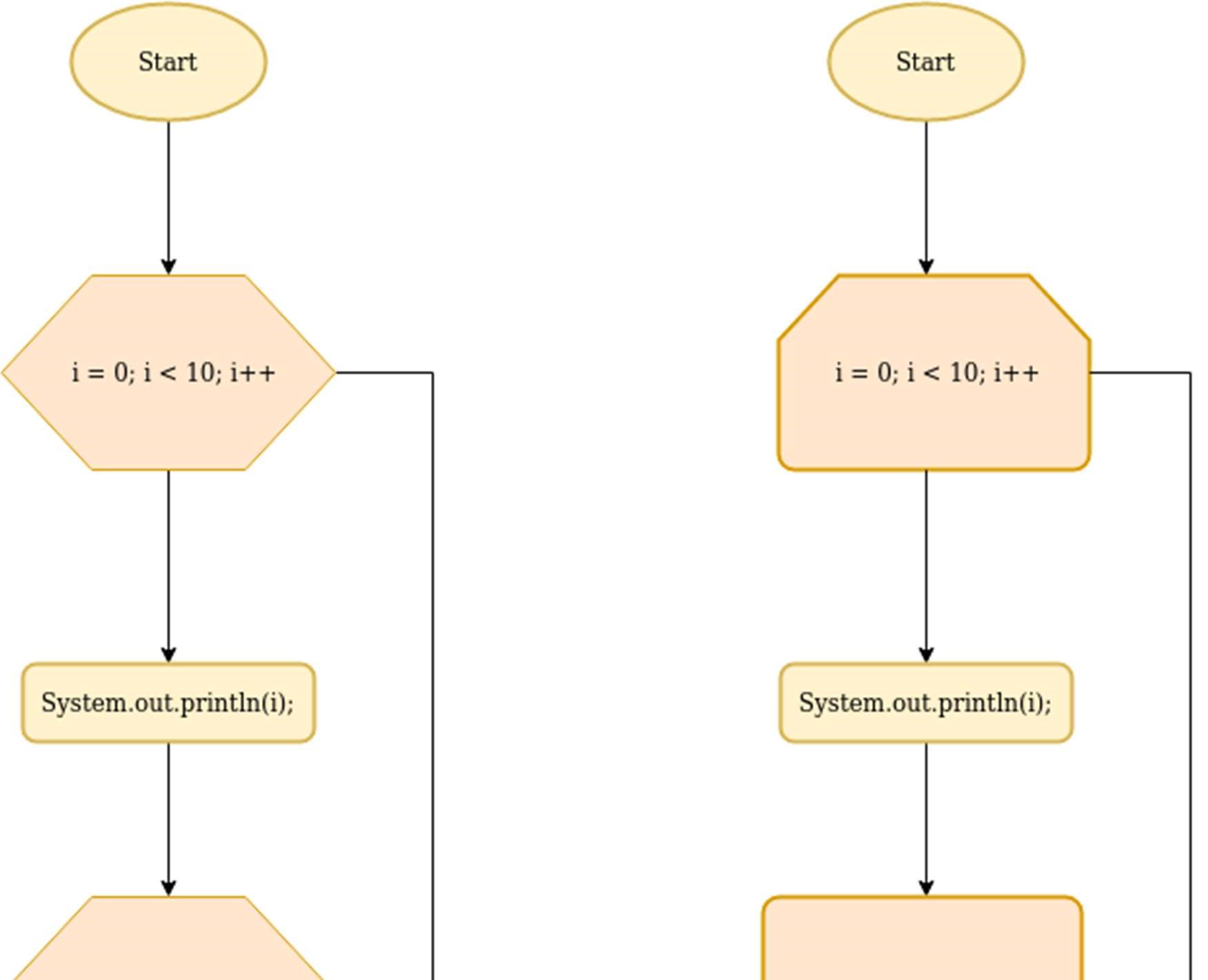
A ciklus fejében deklarált változó nem látszik a ciklus után A fenti példa valójában elszámol 0-tól 9-ig.

A folyamatábrában nincs szabványos jelölés a ciklusra, hiszen a feltétellel és nyilakkal lehet ábrázolni.



Else ág

Az egyszerűség kedvéért azonban vannak rá elterjedt jelölések.



Else ág

Ellenőrző kérdések

* Hogyan lehet feltételes utasítást Javában létrehozni?
* Hogyan lehet ciklust használni Javában?

Feladat

A videóban szereplő feladatok a demos/src/main/java/introcontrol elérési úton vannak.

A introcontrol csomagba dolgozz!

A feladatok megoldásai a solutions/introcontrol elérési úton vannak.

Bevezetés a vezérlési szerkezetek használatába

A introcontrol.IntroControl osztály public int

substractTenIfGreaterThanTen(int number) metódusában add vissza a paraméterként átadott értéket, ha az kisebb vagy egyenlő, mint 10, ellenkező esetben csökkentsd 10-zel, és azt add vissza!

Az introcontrol.IntroControlMain osztály main() metódusában teszteld a megírt metódusokat!

A public String describeNumber(int number) metódusában adj vissza "zero" értéket, ha a paraméterként átadott érték 0, és "not zero" értéket, ha nem 0!

A public String greetingToJoe(String name) metódusban adj vissza Hello Joe értéket, ha a paraméterként átadott név "Joe", és üres Stringet, ha nem!

Az értékesítők 10% jutalékot kapnak az eladások alapján, de csak abban az esetben, ha a havi eladás értéke legalább 1 000 000 Ft. A public int calculateBonus(int sale) metódusban számold ki a jutalékot az eladási összeg alapján, és a jutalék összegét add vissza!

A public int calculateConsumption(int prev, int next) metódusban számold ki a paraméterként megadott mérőóraállások közötti különbséget. Ha a villanyóra eléri a 9999 értéket, átfordul, és újraindul 0 értéktől. Tételezzük fel, hogy csak egyszer fordulhat át, és nem érheti el az előző értéket. Tételezzük fel, hogy 9999 értéknél nagyobbat nem kap paraméterül.

A public void printNumbers(int max) metódussal írd ki a pozitív egész számokat (nullával kezdve) egészen a paraméterként megadott számig (az is legyen kiírva).

A public void printNumbersBetween(int min, int max) metódussal írd ki a pozitív egész számokat a két paraméterként megadott érték között. Feltételezzük, hogy mindkét paraméterként kapott szám nagyobb vagy egyenlő nullával.

A public void printNumbersBetweenAnyDirection(int a, int b) metódussal írd ki a pozitív egész számokat a két paraméterként megadott érték között. Ha az a értéke nagyobb, mint a b értéke, akkor csökkenő sorrendben történjen a kiíratás.

A public void printOddNumbers(int max) metódussal írd ki a páratlan pozitív egész számokat (egytől indítva) egészen a paraméterként megadott számig (az is legyen kiírva, ha páratlan)!

Minősítő

A Qualifier osztály main() metódusába dolgozz! Kérj be a felhasználótól egy számot, és ha az nagyobb, mint 100, akkor írd ki, hogy nagyobb, mint száz, ellenkező esetben száz, vagy kisebb szöveget írd ki!

Menürendszer

Készíts egy konzolos menürendszert egy felhasználókat karbantartó alkalmazáshoz! Az alkalmazásban indulásakor ki kell írni a következőt:

1. Felhasználók listázása
2. Felhasználó felvétele Többi: Kilépés

Majd be kell írni egy számot. Egyes estén ki kell írni, hogy Felhasználók listázása, kettes esetén Felhasználó felvétele. Egyéb esetben nem kell kiírni semmit.

A UserMenu osztályba dolgozz!

Összegszámítás

Kérj be a felhasználótól öt számot, és számold ki az összegüket. A Sum osztályba dolgozz!

Csónakok

Egy csónakkölcsönzőben van 3 csónak. Az elsőben 5-en, a másodikban 3-an, a harmadikban 2-en férnek el. Amikor jön egy csoport, és szeretne csónakot bérelni, akkor úgy kell kiadni nekik a csónakokat, hogy miután kihajóznak, a lehető legtöbb hely és csónak maradjon bent egy következő csoportnak.

Például ha 6-an jönnek, akkor az öt- és kétszemélyes csónakot kell kiadni nekik, mert így még akár egy 3 fős csapat is ki tud hajózni.

Ha 5-en jönnek, akkor az 5 személyes csónakot kell kiadni nekik, mert így 2 csónak összesen 5 hellyel marad bent.

Készíts egy BoatRental osztályt, ahol a main() metódusban bekéred az érkező csapat létszámát, majd írd ki, hogy melyik csónakokat vitték el és még hány fő mehet utánuk! Ha többen voltak, mint 10, akkor jelezd, hogy maradtak még a parton!

Forrás

OCA - Chapter 2/Understanding Java Statements

Teszt

Mit ír ki az alábbi kódrészlet?

for (int i = 4; i <= 10; i++) { if (i >= 7) {

System.out.println(i / 2);

} else {

System.out.println(i \* 2);

}

}

* 8 10 12 14 4 4 5
* ☒ 8 10 12 3 4 4 5
* 8 10 12 3 4 4
* 8 10 12 14 4 4

Mit ír ki az alábbi kódrészlet?

for (int i = 0; i < 2; i++) {

}

System.out.println(i);

* 0 • 1
* 2
* ☒ Nem fordul le

## Bonyolultabb típusok

Tömb (array)

Eddig olyan adattípusokkal ismerkedtünk meg, amelyek csak egy egyszerű értéket tárolhatnak. A tömb már sok ugyanolyan típusú elem tárolására képes, amelyeket a sorszámukkal (index) közvetlenül elérhetünk.

Deklarációkor meg kell adnunk, hogy milyen típusú elemeket szeretnénk tárolni benne. Létrehozni a new kulcsszóval lehet, és meg kell adnunk a tömb méretét is. int[] arrayOfNumbers = new int[10];

A tömb elemei mindig kapnak kezdőértéket: egész típusú elemek esetén ez 0, lebegőpontos szám esetén 0.0, logikai érték esetén false, osztály esetén null lesz.

Az elemek elérése szögletes zárójellel történik, amelybe az elem indexét kell írnunk. Az indexelés 0-tól kezdődik, és olyan indexre nem hivatkozhatunk, amely túlmutat bármelyik irányban az indexelés határain. Azaz egy 10 elemű tömb esetén az index csak 0 és 9 közötti értéket vehet fel. Például a fenti tömb 5. elemét az arrayOfNumbers[4] hivatkozással érhetjük el. Az arrayOfNumbers[10] esetén már ArrayIndexOutOfBoundsException kivételt kapunk.

A tömb hosszát a length tulajdonságán át kérdezhetjük le: arrayOfNumbers.length.

Létrehozhatunk tömb literált, amelyben a tömb elemeit tudjuk megadni kapcsos zárójelek között vesszővel elválasztva:

String[] fruits = {"apple", "peach", "plum", "orange"};

Egy metódus paramétereként is használhatunk tömb literált a tömb létrehozására, de ebben az esetben eltérő a szintaktika:

countArrayElements(new int[] {1, 2, 3, 4})

A tömbök elemeit bejárhatjuk a már megismert for ciklussal, de létezik egy hatékonyabb módja is annak, hogy minden egyes elemet elérjünk: a for-each ciklus. Foreach ciklus használata esetén a tömb elemeit csak kiolvasni tudjuk, módosítani nem, illetve az elem sorszáma sem áll rendelkezésre. Amennyiben ezekre szükségünk van, használjuk a hagyományos for ciklust!

Tömbök között adatok átmásolásának azonban annál jobb módja is van, minthogy ciklussal bejárjuk, és egyenként átmásoljuk az elemeket: a System.arraycopy() metódus.

for(int i = 0; i < fruits.length; i++){

System.out.println(fruits[i]);

}

for(String fruit: fruits){ System.out.println(fruit);

}

String[] favoriteFruits = new String[2];

System.arraycopy(fruits, 1, favoriteFruits, 0, 2); // favoriteFruits --> {"peach", "plum"}

Ellenőrző kérdések

* Hogyan definiálunk Javában tömböt?
* Hogyan férünk hozzá egy tömb eleméhez?
* Hogyan kérjük le a tömb hosszát?
* Hogyan definiálunk tömb literált?
* Hogy lehet a tömb elemeit kiírni?
* Hogyan járhatjuk be egy tömb elemeit?

Feladat

Tömbök kezelése

Az array.ArrayMain osztály main() metódusába dolgozz!

Definiálj egy String tömböt a hét napjaival! Írd ki a második elemét (kedd)! Írd ki a tömb hosszát is!

Definiálj egy öt elem hosszú int tömböt, és tárold le benne (ciklussal) a kettő hatványait (1, 2, 4, 8 stb.)! Ciklusban töltsd fel értékekkel is. Az algoritmus az, hogy mindig az előző elem értékét szorozd meg kettővel! Ciklusban írd ki az értékeit!

Definiálj egy hat elemű boolean tömböt, és felváltva írj bele true vagy false értéket, 0. index esetén legyen false! Ciklusban töltsd fel. Az algoritmus az, hogy mindig az előző elemnek veszed a logikai negáltját. Ciklusban írd ki az elemeit!

Keresés

Hozz létre egy array.ArrayHandler osztályt, és implementálj benne egy boolean contains(int[] source, int itemToFind) metódust, mely visszaadja, hogy a paraméterként megadott érték benne van-e a szintén paramérként átadott tömbben!

A fenti array.ArrayHandler osztályba implementálj egy újabb int find(int[] source, int itemToFind) metódust, mely visszaadja a paraméterként megadott érték indexét, ha benne van a tömbben, és -1-et, ha nincs benne!

Forrás

OCA - Chapter 3/Understanding Java Arrays

Teszt

Melyik a helyes egész számokból álló tömb deklaráció és létrehozás?

* int[10] numbers = new int[];
* int[] numbers = new int[];
* ☒ int[] numbers = new int[10];
* int numbers = new int[10];

Hogyan lehet megtudni a numbers tömb hosszát?

* ☒ a numbers.length attribútummal
* a numbers.length() metódussal
* a numbers.size() metódussal
* a numbers[] kifejezéssel Igaz-e az alábbi állítás?

A tömbök elemeinek mindig van alapértelmezett kezdőértéke.

* ☒ igaz
* hamis

Parancssori paraméterek (cmdarguments)

Elmélet

A main() metódus paramétere egy String[]. A parancssorban átadott paraméterekhez ezen a tömbön keresztül lehet hozzáférni.

public static void main(String[] args) {

System.out.println(args.length);

System.out.println(args[0]);

System.out.println(args[1]);

}

Parancssorban a programot a java TrainerMain John 1970 paranccsal lehet elindítani, ami ekkor kírja a paraméterek számát (2), valamint az első és második paramétert (a paramétereket nullától számozza).

A parancssorban a paramétereket egymástól a space karakter választja el. Amennyiben olyan paramétereket akarunk megadni, melyek tartalmaznak space karaktert, ezeket idézőjelek között kell megadni.

Amennyiben az alkalmazás jar állományba van csomagolva, akkor is ugyanazon módon adhatjuk át a paramétereket, azaz pl. java -jar trainers.jar John 1970.

Ellenőrző kérdések

* Mire való a parancssori paraméterek?
* Hogyan kérhetjük le a parancssori paramétereket?
* Mire kell vigyázni a parancssori paraméterek használatakor? ### Tömbök tömbje (arrayofarrays)

A Java nyelvben nem létezik többdimenziós tömb, azaz olyan tömb, amely egy táblázatra vagy táblázatok sorozatára hasonlít. Ellenben egy tömb elemei lehetnek tömbök is, így elérhető hasonló adatszerkezet. Ekkor arra kell figyelnünk, hogy a tömbök mérete akár mind különböző is lehet.

Létrehozása, ha például egy 5 soros, 3 oszlopos táblázatot szeretnénk egészekből:

int[][] numbers = new int[5][3]; vagy ha különböző hosszú tömböket szeretnénk:

int[][] numbers = new int[5][]; numbers[0] = new int[3]; numbers[1] = new int[8];

Létrehozás literállal:

int[][] numbers = {{1}, {1, 2}, {1, 2, 3}};

Ellenőrző kérdések

* Hogyan lehet Javában többdimenziós tömbhöz hasonló struktúrát létrehozni?
* Hogyan lehet ezt literálként feltölteni?

Feladat

Szorzótábla

A arrayofarrays.ArrayOfArraysMain osztályba dolgozz.

Hozz létre egy int[][] multiplicationTable(int size) metódust, mely a paraméterként megadott méretű szorzótáblát adja vissza! A szorzótábla alakja (csak 4x4 esetén):

1. 2 3 4
2. 4 6 8
3. 6 9 12
4. 8 12 16

Tömbök tömbjének kiírása

Hozz létre egy printArrayOfArrays(int[][] a) metódust, mely kiír egy tömbök tömbjét! A beágyazott tömbök elemeit egymás mellé, a külső tömb elemeit egymás alá.

A main() metódusában teszteld le a működést!

Háromszögmátrix

Hozz létre egy int[][] triangularMatrix(int size) metódust, mely a paraméterként megadott méretű háromszögmátrixot hozza létre, és minden sora a sor számának értékeit tartalmazza! Ilyen kiírást várunk:

0

1. 1
2. 2 2
3. 3 3 3

Napi mért értékek

A int[][] getValues() metódusban hozz létre egy 12 elemű tömböt, és mindegyik elem egy olyan hosszú tömböt tartalmazzon, amennyi nap van az adott hónapban (nem vagyunk szökőévben)! Minden elem kezdőértéke 0.

Bónusz feladat 1.

A tömb kiírásánál figyelj arra, hogy minden egyes számértéknek három karakter legyen fenntartva, azaz ha egy számjegyű, akkor ki kell egészíteni két szóközzel, ha két számjegyű, akkor egy szóközzel (előtte)!

Például:

\_\_1\_\_\_2\_\_\_3

\_10\_\_20\_\_30

100\_200\_300

Teszt

Melyik a helyes deklaráció, amelyik egy 3 soros és 5 oszlopos táblázatot (tömbök tömbjét) hoz létre?

* int[][] table = new int[3];
* int[5] table = new int[3];
* int[3][5] table = new int[][];
* ☒ int[][] table = new int[3][5];

Adott az alábbi deklaráció: String[][] words = new String[3]; Hogyan lehet azt elérni, hogy az első eleme egy 2 hosszú, a második eleme pedig egy 5 hosszú tömb legyen?

words[1] = new String[2]; words[2] = new String[5];

* [x]

words[0] = new String[2]; words[1] = new String[5];

words[0][] = new String[2]; words[1][] = new String[5];

words[][0] = new String[][2]; words[][1] = new String[][5];

Tömbök kezelése (arrays)

A tömbök kezelése nem is olyan egyszerű, sok művelethez van szükség ciklusra. A Java Arrays osztálya ebben könnyíti meg a dolgunk. Statikus metódusai egyszerűvé teszik a tömb kiírását, másolását, rendezését, tömbök összehasonlítását.

Metódusok:

String toString(): A tömb elemeit adja vissza szöveges formában, paraméterként a tömböt kell átadni.

String deepToString(): Tömbök tömbjét adja vissza szöveges formában teljes mélységben olvashatóan, paraméterként a tömböt kell átadni.

boolean equals(): Két azonos típusú elemekből álló tömböt hasonlít össze, és igazzal tér vissza, ha az elemek páronként egyenlőek.

boolean deepEquals(): Két tömbök tömbjét hasonlítja össze, és igazzal tér vissza, ha az elemek minden szinten páronként megegyeznek.

void sort(): Berendezi egy tömb elemeit növekvő sorrendbe, amennyiben az elemek típusának van természetes rendezettsége. Paraméterként a rendezendő tömböt kell megadni.

T[] copyOf(): A paraméterként átadott tömb megadott hosszúságú másolatát adja vissza. Ha az eredeti tömb rövidebb, akkor a maradék helyeket alapértelmezett értékekkel tölti fel (lást a tömb adattípusnál), ha hosszabb, akkor levágja a végét.

T[] copyOfRange(): A paraméterként átadott tömb megadott indexek közötti szakaszát adja vissza. Az esetleges üres helyeket az alapértelmezett értékkel tölti fel.

Ellenőrző kérdések

* Mire való az Arrays osztály?
* Milyen metódusai vannak?
* Hogyan kell meghívni ezeket?

Feladat

Dolgozz az arrays.ArraysMain osztályban!

numberOfDaysAsString metódus

A String numberOfDaysAsString() metódusban definiálj egy numberOfDays nevű változót, mely a hónapok napjainak számát tartalmazza, és add vissza String-ként egy utasítással az értékeit. daysOfWeek metódus

A List<String> daysOfWeek() metódus adja vissza a napok neveit!

multiplicationTableAsString metódus

A multiplicationTableAsString(int size) metódus definiáljon egy size méretű szorzótáblát, és adja vissza az értékeket String-ként egy sorban.

sameTempValues metódus

A sameTempValues(double[] day, double[] anotherDay) hőmérsékleti értékeket vár, órai mérésekkel, két napra. Vizsgáld meg, hogy a paraméterként megadott két nap azonos méréseket tartalmazott-e!

wonLottery metódus

Döntsd el a boolean wonLottery(int[], int[]) metódusban, hogy a megtett számok, és a kihúzott számok megegyeznek-e! Nem biztos, hogy növekvő sorrendben vannak a számok. Azaz ellenőrizd, hogy ugyanazokat az értékeket tartalmazza-e a két paraméter, sorrendtől függetlenül!

Válaszd azt a megoldást, hogy mindkét tömböt rendezed, és úgy hasonlítod őket össze!

Miután a main() metódusban meghívtad a wonLottery() metódust, vizsgáld meg, hogy a paraméterként átadott tömb rendezett lett-e, azaz a rendezésnek lett-e visszahatása a paraméterként átadott tömbre! (Ehhez az kell, hogy a paraméterek változók legyenek, melyek értékét a hívás után vizsgálni lehet.)

Ha igen, próbáld meg valahogy kikerülni a problémát, azaz a metódusnak ne legyen mellékhatása.

Bónusz feladat 1

A sameTempValuesDaylight(double[] day, double[] anotherDay) metódus ugyanúgy hasonlítsa össze az értékeket, de vegye figyelembe a 23 és 25 órás napokat is. Összehasonlítás során mindig a kisebb óraszámot vegye figyelembe alapként, és úgy hasonlítson össze (mindkettőn hívd meg a copyOfRange() metódust)!

Implementálhatsz egy min(int, int) segédmetódust, mely a két óraszám közül a kisebbet adja vissza.

Teszt

Melyik paranccsal írnád ki a String[][] words változóban tárolt szavakat?

* System.out.print(words.toString());
* System.out.print(words.deepToString());
* System.out.print(Arrays.toString(words));
* ☒ System.out.print(Arrays.deepToString(words));

Adott az int[] numbers = {2, 6, 3, 9, 10, 4} egész számokból álló tömb. Hogyan másolnád át a második 3 elemét a int[] secondPart tömbbe a lehető legegyszerűbben?

* Arrays.copyOf(numbers, 3, secondpart, 0, 3);
* ☒ secondpart = Arrays.copyOfRange(numbers, 3, 6);
* secondpart = Arrays.copyOfRange(numbers, 3, 3);
* secondpart = numbers.copyOfRange(3, 6);

Lista (arraylist)

A lista annyiban hasonlít a már megismert tömb típusra, hogy ez is azonos típusú elemek tárolására van, amelyek az indexükkel elérhetőek, de mérete nem fix, hanem dinamikusan változik a tartalommal. Lista típusú változót mindig a List interfésszel deklarálunk, így bármilyen listát megvalósító osztályt bele tudunk tenni. Ezekről később részletesebben is lesz szó, de most nézzük a leggyakoribbat, az ArrayList osztályt.

Megszoktuk már a tömböknél, hogy meg kell adnunk az elemek típusát. Lista esetén is így van, de itt <>-k között kell ezt megtennünk (generikus). Listában nem tárolhatunk primitív típusú elemeket, de szerencsére minden primitív típusnak van megfelelő, ún. burkoló osztálya. Ha a listába primitív értéket teszük, az automatikusan becsomagolódik a burkoló osztályába, a kivett elemet pedig kezelhetjük primitívként, mert az automatikusan kicsomagolódik.

List<String> fruits = new ArrayList<>();

Az ArrayList példányosításakor már nem kell megismételnünk az elemek típusát, elég üres <>-t írni. Ezt hívjuk diamond operátornak.

A lista szöveges reprezentációjának előállítására itt is a toString() metódust használjuk, de itt a példányváltozón kell meghívni.

System.out.println(fruits.toString());

Tömbökből is készíthetünk listát az Arrays.asList() metódussal azzal a megkötéssel, hogy elemeket utólagosan sem hozzáadni, sem törölni nem tudunk.

String[] names = {"Adam", "Eve", "Jonathan"};

List<String> nameList = Arrays.asList(names); Metódusok

boolean add(): új elem hozzáadása a listához. Paraméterként a beszúrandó elemet kell megadni, illetve ha nem a végére akarjuk tenni, akkor az indexet is meg kell adnunk. Ebben az esetben az utána következő elemek eggyel hátrébb kerülnek.

E get(int index): a megadott indexen lévő elemet adja vissza. void clear(): törli a lista összes elemét

boolean contains(Object o): igazzal tér vissza, ha a megadott elem benne van a listában. Ennek eldöntésére az elem equals() metódusát használja.

int indexOf(Object o): megadja, hogy a paraméterként átadott elem milyen indexen szerepel először a listában. Ha nincs a listában, -1-et ad vissza.

boolean remove(Object o): törli a paraméterként átadott elem első előfordulását. Ha az elem nem szerepelt a listában, hamissal tér vissza.

int size(): a lista elemeinek száma

Bejárás

A lista bejárható a hagyományos for ciklussal

List<Integer> numbers= Arrays.asList(23, 41, 2, 7); for(int i = 0; i < numbers.size(); i++){

System.out.println(numbers.get(i));

}

vagy foreach ciklussal

for(int item: numbers){ System.out.println(item);

}

Ellenőrző kérdések

* Mire való az ArrayList?
* Hogyan kell definiálni?
* Mire használjuk a generikust?
* Mi történik akkor, ha nem használunk generikust?
* Milyen metódusait ismered?
* Hogyan tudod bejárni az elemeit?

Feladat

Capsules osztály

Készíts egy Capsules osztályt, mely segít olyan műalkotás megtervezésében, amely újrahasznosított kávékapszulákból áll. Egy hajlítható műanyag csőbe tudjuk helyezni a különböző színű kapszulákat egymás mellé. Így alakul ki az alkotás, amit aztán különböző formára hajlíthatunk.

A Capsules osztály egy ArrayList attribútumban tárolja a betett kapszulákat, méghozzá a színüket String-ként.

A Capsules osztálynak legyen egy addLast(String), addFirst(String), removeFirst(), removeLast() metódusa, mely betesz, illetve kivesz kapszulákat a csőből. Legyen egy List<String> getColors() metódusa, mely visszaadja a kapszulákat tartalmazó listát, hogy ki lehessen írni.

Írj egy main() metódust, mely teszteli a metódusok működését.

Books osztály

Készíts egy Books osztályt, melyben egy ArrayList<String> tárolja a könyvek címeit. Írj egy add(String) metódust, mely felveszi a könyvet. Legyen egy List<String> findAllByPrefix(String prefix) metódusa, mely az összes olyan könyvet visszaadja, mely címének eleje megegyezik a paraméterként átadott szöveggel. Legyen egy List<String> getTitles() metódus, mely visszaadja a könyvek címeit.

Írj egy main() metódust, mely teszteli a metódusok működését.

Bónusz feladat 1.

A Capsules osztály getColors() metódusával kérjük le a kapszulák színeit, majd az eredményt tároljuk le egy változóba. A letárolt változón hívjuk meg a clear() metódust. Majd ismét kérjük le a kapszulák színeit a getColors() metódussal , és nézzük meg, hogy az előző clear() hívásnak volt-e hatása erre?

Bónusz feladat 2.

A Books osztálynak legyen egy removeByPrefix(String prefix) metódusa mely kiveszi a könyvet a címének első pár karaktere alapján (az összes előfordulást).

Mi van akkor, ha egy ciklusban mész végig az elemeken, és ha a feltételnek megfelel az elem azonnal törölni próbálod? Hogyan lehet ezt kikerülni? Használd a removeAll() metódust!

Forrás

OCA - Chapter 3/Understanding an ArrayList

Teszt

Hogyan lehet egész számokból álló listát létrehozni?

* List<int> heights = new ArrayList<>();
* ☒ List<Integer> heights = new ArrayList<>();
* List<Integer> heights = new ArrayList<>;
* List<int> heights = Arrays.asList(1, 5, 7);

Debug (debug)

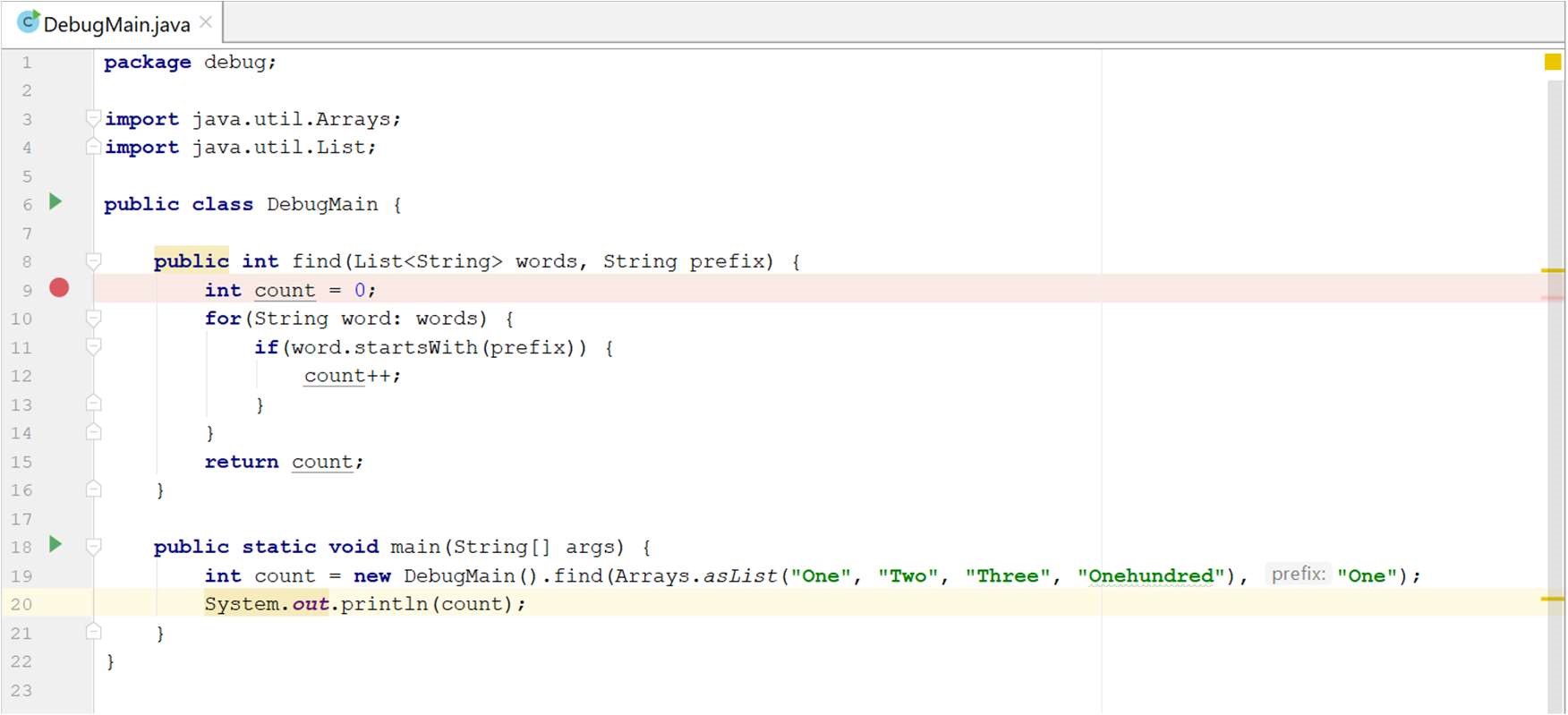
A Debugger a fejlesztőeszköz által biztosított eszköz a hibakeresésre. Meg tudjuk állítani vele az alkalmazást, meg tudjuk vizsgálni annak belső állapotát sőt akár módosítani is tudjuk, valamint alkalmazásunkat utasításonként tudjuk léptetni.

Az alkalmazás megállításához breakpointot tudunk elhelyezni a forráskódban bármely utasítás mellett. Itt a JVM megáll, és a debuggertől várja a további utasításokat. Lehetőség van megállás után a változók, attribútumok vizsgálatára, valamint olyan utasításokat is kiadhatunk, amivel nem csak lekérdezhetjük, de akár meg is változtatjuk az objektum belső állapotát. Megállás után léptethetünk akár utasításonként is, hagyhatjuk tovább futni az alkalmazást, vagy újra leállíthatjuk breakpointtól függetlenül.

A debugger a teszteset futtatásakor is hasznos.

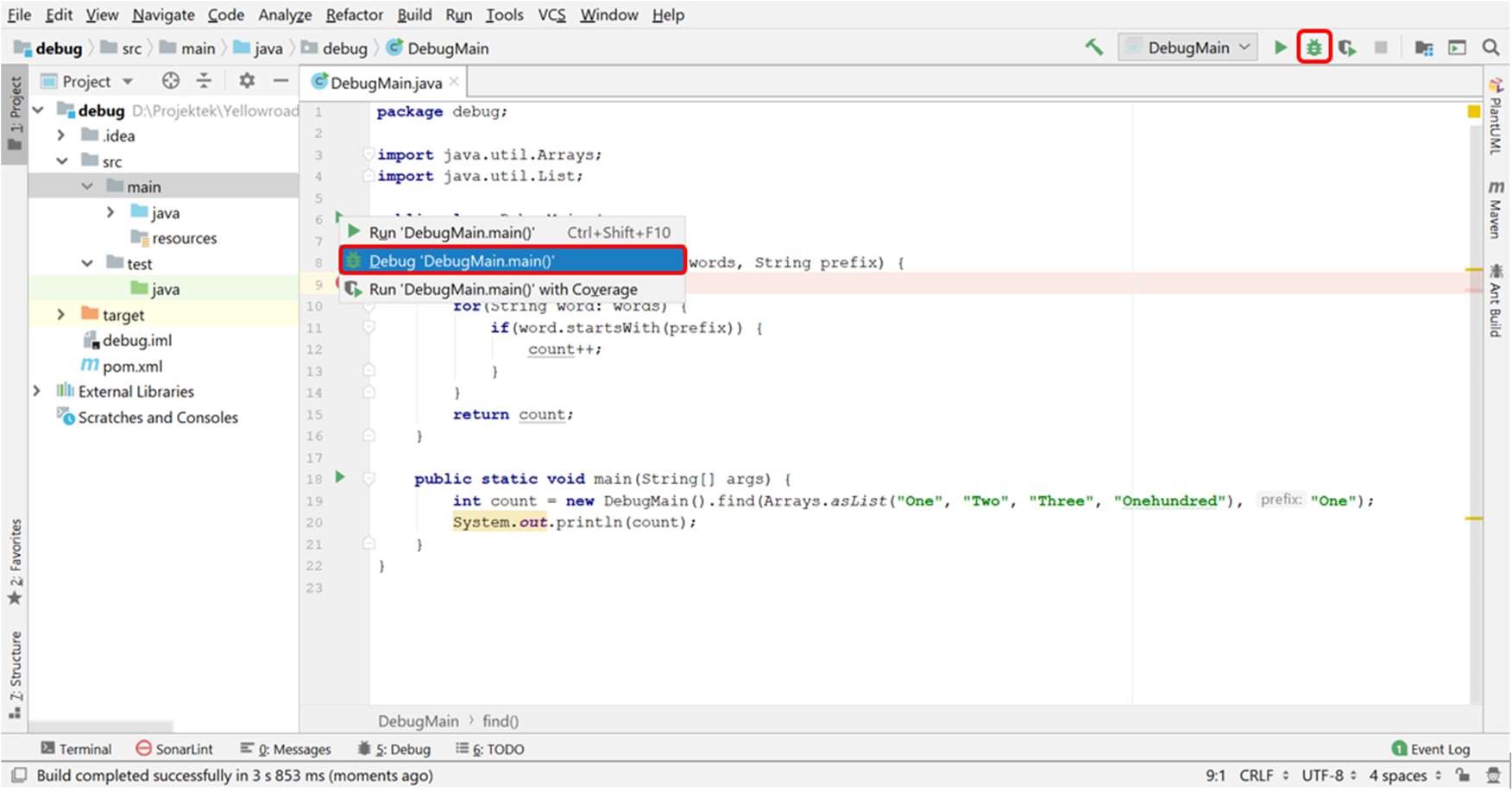
Debugger az IntelliJ IDEÁ-ban

Az IDEA fejlesztőeszközben breakpointot a sor száma mellé kattintva tudunk elhelyezni. A megjelölt sor mellett egy piros pont látható, a háttere pedig halványpirosra vált.



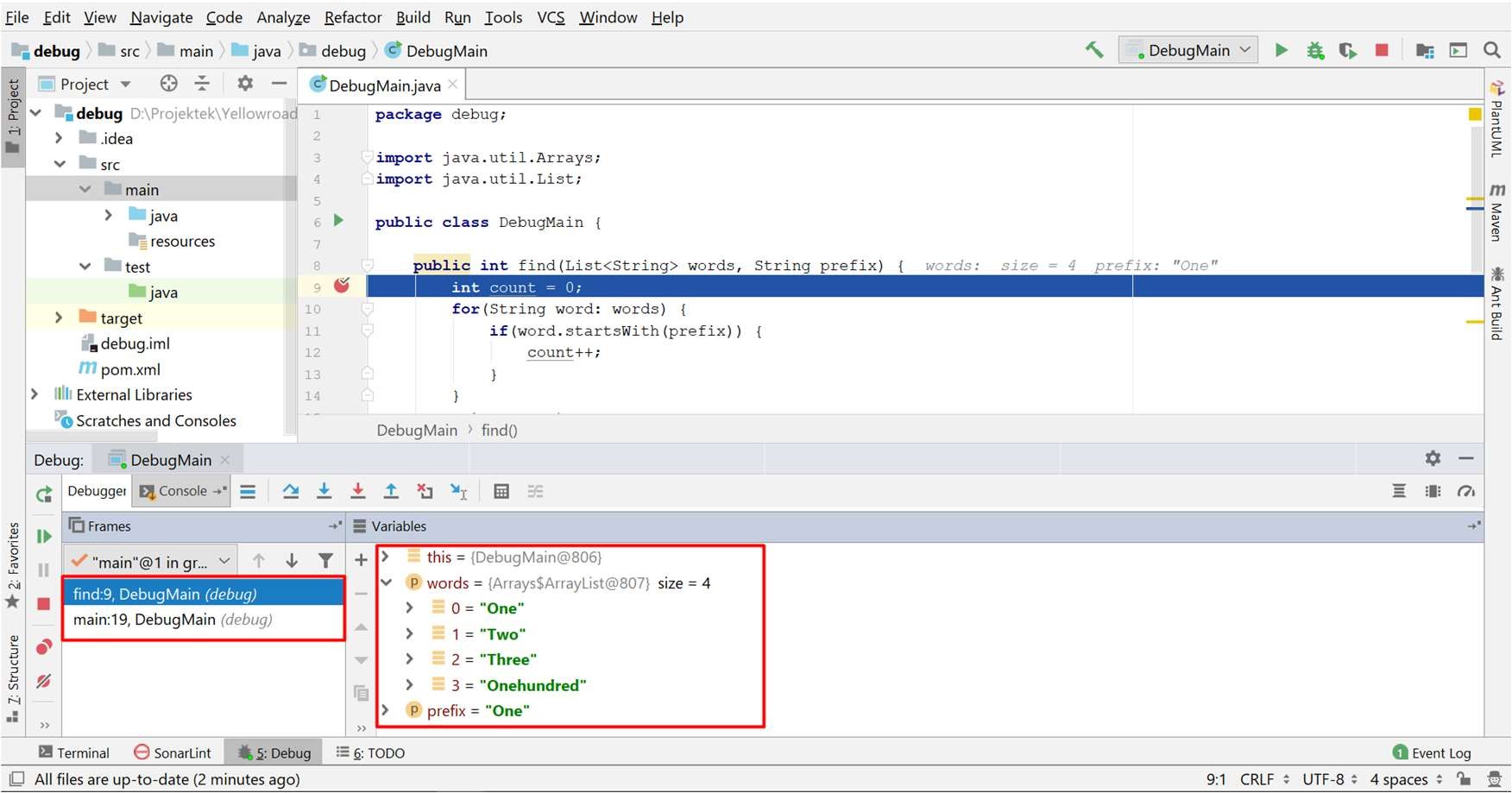
Breakpoint az editorban

Ahhoz, hogy a program felfüggessze a futását a megadott breakpointnál, debug módban kell elindítani. Ezt mindenhonnan, ahonnan futtatni is tudjuk megtehetjük: a main() vagy a tesztmetódus melletti zöld nyíl alól, eszköztárról és a Run menüpontból is a bogár ikonra kattintva.



Debugger elindítása

Elindítás után az editorban kékkel kijelölve láthatjuk a sort, amelyiken állunk, az IDE alsó részén részén pedig megjelenik a Debugger ablak, ahol a call stacket és az aktuálisan elérhető változókat nézhetjük meg. A + jelre kattintva mi is adhatunk hozzá kiértékelendő kifejezéseket. Amennyiben egy változó létező objektumra mutat, akkor lenyitva annak állapotát is követhetjük.



Debugger ablak

A Debugger ablak felső részén található gombokkal illetve a nekik megfelelő funkcióbillentyűkkel léptethetjük az alkalmazást, de mindig csak előre.



Léptetések

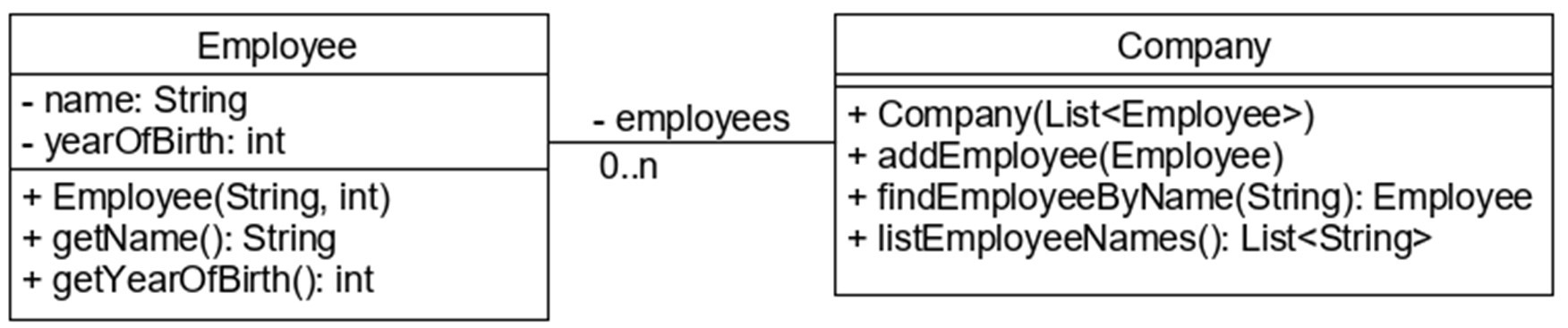
Ellenőrző kérdések

* Mire való a fejlesztőeszköz debug funkciója?
* Mire való a breakpoint?
* Milyen lehetőségek vannak debug közben?

Feladat

Alkalmazottak

Készítsd el az UML alapján az Employee és a Company osztályt! Teszteld a Company metódusait a CompanyMain osztály main() metódusából!



Company osztály diagram

Helyezz el breakpointot a main() metódus első utasítása mellett, és futtasd debug módban! Kísérletezz:

* Melyik léptetés mit eredményez?
* Mikor jelennek meg a lokális változók a debugger ablakban?
* El tudsz helyezni újabb breakpointokat a debugger futtatása közben is? Hogyan lehet azonnal a következő breakpointra ugrani?

Csak pozitívan!

Hozz létre egy NumberStatistics osztályt egy numbers egész számokból álló lista attribútummal a debug.numbers csomagba! Az attribútumot konstruktorban kapja meg, és feladata a kapott listán mindenféle számítások elvégzése. Ezeket a számításokat már megírták, de sajnos valahol hiba csúszott mindegyikbe, mert nem adnak helyes eredményt.

Másold be az elkészült osztályba ezeket a metódusokat, majd írj egy main() metódust ezek tesztelésére! A debugger segítségével keresd meg és javítsd ki a hibákat!

public int sumPositives() { int sum = 0; for(int n: numbers) { if(n != 0) { sum += n;

} }

return sum;

}

public int minDifferenceBetweenNeighbours() { int minDifference = numbers.get(0) - numbers.get(1) >= 0 ? numbers.get(0) - numbers.get(1) : numbers.get(1) - numbers.get(0); for(int i = 1; i < numbers.size() - 1; i++) {

int actDifference = numbers.get(i) - numbers.get(i + 1); if(actDifference < minDifference) { minDifference = actDifference;

} } return minDifference;

}

Elvárt eredmények:

* new NumberStatistics(Arrays.asList(4, 8, -1, -2, 4, 5, 3)).sumPositives() –> 24
* new NumberStatistics(Arrays.asList(4, 8, -1, -2, 4, 5,

3)).minDifferenceBetweenNeighbours() –> 1

* new NumberStatistics(Arrays.asList(-3, -4)).sumPositives() –> 0
* new NumberStatistics(Arrays.asList(1)).minDifferenceBetweenNeighbours() –> IllegalStateException

Teszt

Melyik állítás HAMIS az alábbiak közül?

* A debugger futtatható main() metódusból és bármely teszt metódusból is.
* A debugger ablakban az IntelliJ IDEÁ-ban megtekinthető a call stack.
* A debugger ablakban az IntelliJ IDEÁ-ban megtekinthető az összes lokális változó aktuális értéke.
* ☒ Az alkalmazás futása debug alatt csak a breakpointokon áll meg.

Konstans értékek használata (finalmodifier)

Konstansnak nevezzük azokat a névvel ellátott értékeket, amelyek a program futása alatt nem változtathatóak meg. A Java nyelvben nincs ennek megfelelő elem, de nagyon hasonló van.

final módosító szó

Ha egy változót deklaráláskor final módosítóval látunk el, akkor annak csak egyszer adható érték, és az később nem változtatható meg. Attribútumnál, paraméternél és lokális változónál is használható. A final módosítóval deklarált attribútumnak legkésőbb a konstruktorban értéket kell kapnia.

public double calculateGrossWeight(double netWeight){ final double packageWeight = 2.4; return netWeight + packageWeight;

}

Kvázi konstans

Azokat az attribútumokat, amelyeket static final módosítóval látunk el, az egész program futása alatt elérhetjük, de soha nem változtathatjuk meg. Ezeknek már az osztály betöltődésekor értéket kell kapniuk, és konvenció szerint csupa nagybetűvel írjuk őket, a több szóból állóknál pedig alulvonással segítjük az olvashatóságot.

public static final int NUMBER\_OF\_SEASONS = 4;

Ezeket a kvázi konstansokat általában abban az osztályban deklaráljuk, ahol használni szeretnénk, így elérésükhöz elég a nevükkel hivatkozni rájuk. A public static final módosítóval ellátott attribútumokat más osztályból is elérhetjük, ha az osztálynéven át hivatkozunk rájuk vagy statikus importot használunk. Lássunk mindkettőre példát!

public class Lion {

public static final String SOUND = "roar";

public static final String FOOD\_TYPE = "meat";

public void speak(){

System.out.println(SOUND);

}

}

import static Lion.FOOD\_TYPE;

public class Cub {

private int weight;

public void learnToSpeak(){ System.out.println(Lion.SOUND);

}

public void eat(String food){ if(food.equals(FOOD\_TYPE)){ weight++;

}

}

}

Ellenőrző kérdések

* Mire való a final módosító szó?
* Hogyan definiálunk Javaban konstans-szerű értékeket?
* Hogyan használjuk?

Feladat

Gentleman osztály

Definiálj egy finalmodifier.Gentleman osztályt, melyben definiáld kvázi konstansként a MESSAGE\_PREFIX változót, mely a köszönés elejét tartalmazza! Írj egy String sayHello(String name) metódust, mely a MESSAGE\_PREFIX értékét összefűzi a name paraméter értékével, és az eredményt visszaadja!

CircleCalculator és CylinderCalculator osztály

Definiálj egy finalmodifier.CircleCalculator osztályt, melyben definiáld a Pi-t! Írj egy double calculatePerimeter(double r) metódust, ami a kerületet számolja ki, és egy double calculateArea(double r) metódust, ami a területet!

Írj egy finalmodifier.CylinderCalculator osztályt, melyben legyen egy calculateVolume(r, h) metódus, és egy calculateSurfaceArea(r, h) metódus, és használja a Pi értékét a CircleCalculator osztályból!

Teszteld a finalmodifier.PiMain osztály main() metódusából az elkészült metódusokat!

Írd ki itt a Pi értékét is!

TaxCalculator osztály

Írj egy finalmodifier.TaxCalculator osztályt, mely tartalmazza az ÁFA értékét, ami 27%. Írj egy double tax(double price) metódust, mely a paraméterként megadott érték ÁFA értékét számolja, és egy double priceWithTax(double) metódust, mely az árat adja vissza az ÁFÁ-val együtt!

Bónusz feladat 1.

Definiálj egy finalmodifier.Week osztályt, mely List<String> típusú változóban tartalmazza a hét napjait! Használd az Arrays.asList() metódust!

Próbáld meg a keddet lecserélni a List-ben szerdára! Fog sikerülni?

Próbálj értékül adni a változónak egy példányosított listát! Fog sikerülni?

Bónusz feladat 2.

Mi történik, ha nem adsz értéket egy attribútumnak, és final-ként deklarálod?

Lehet-e lokális változót final módosító szóval úgy deklarálni, hogy nem adsz neki értéket?

Bónusz feladat 3.

A finalmodifier.CylinderCalculatorBasedOnCircle felépítése egyezzen meg a CylinderCalculator osztállyal, de metódusai ne a CircleCalculator PI értékét használják, hanem a metódusait!

Bónusz feladat 4.

Miért ad circleCalculator.calculatePerimeter(10) hívás különleges értéket vissza?

Teszt

Milyen változtatással lehet elérni, hogy az alábbi osztály leforduljon?

public class Coffee { private final String type; private final int weight; private final String taste;

public Coffee(String type, int weight) { this.type = type; this.weight = weight;

}

public void setTaste(String taste) { this.taste = taste;

}

}

* ☒ A taste attribútum elől töröljük a final módosítót.
* A taste attribútumnak deklarációkor kezdőértéket adunk.
* Töröljük a setTaste() metódust.
* A taste attribútumot is inicializálni kell a konstruktorban.
* Nem kell módosítani, az osztály úgy jó, ahogy van.

Konvenció szerint melyik helyes deklaráció?

* ☒ public static final int NUMBER\_OF\_SEMESTERS = 7;
* private final int NUMBER\_OF\_SEMESTERS = 7;
* public static final int numberOfSemesters = 7;
* private final int number\_of\_semesters = 7;

Math és Random osztály (math)

Math osztály

A Math osztály matematikai konstansokat és függvényeket tartalmaz. Mind statikus, ezért mind az osztálynév minősítővel használjuk.

Tartalma:

* E, PI
* Szögfüggvények
* Kerekítő függvények
* Hatvány, exponenciális és logaritmikus függvények • Abszolútérték, minimum, maximum double squareRoot = Math.sqrt(5); double perimeterOfCicle = Math.PI \* r \* 2;

Ötletelj!

A Math.round() metódus a paraméterként kapott lebegőpontos számot mindig a legközelebbi egészre kerekíti. Hogyan kerekítenél egy lebegőpontos számot pontosan 3 tizedes jegyre?

Random osztály

Véletlenszámok generálására használható a Random osztály. Mivel a számítógép nem képes kockát dobálni, csak az általunk kijelölt műveleteket tudja végrehajtani, ezért a kapott számok ténylegesen nem véletlenszerűek, hanem ún. pseudorandom számok. Ez azt jelenti, hogy a véletlenszámot előállító függvény kap egy bemeneti értéket, és valamilyen műveletsorozat végeredményeként visszaad egy másikat. Ha nem tudjuk, pontosan milyen érték ment be, akkor azt sem tudjuk megmondani, milyet kapunk vissza, azonban ha tudjuk, akkor pontosan kiszámítható a visszatérési értéke. Éppen ezért a Random osztályt kétféleképpen is példányosíthatjuk: vagy paraméter nélkül, és akkor véletlennek tűnő számot kapunk vissza, vagy egy kezdőértékkel, amelyet seed változónak nevezünk. Ekkor a működése determinisztikussá válik, így tökéletes teszteléshez.

Példányosítás után nemnegatív egész véletlenszámot a nextInt() metódussal generálhatunk. Ha a felső határt is meg akarjuk kötni, azt paraméterként kell átadnunk. 0 és 1 közötti lebegőpontos számot a nextDouble() metódussal generálhatunk.

Random rnd = new Random();

int randomNumber = rnd.nextInt(); //0, 1, ... Integer.MAX\_VALUE közül egy egész

int randomIntTo10 = rnd.nextInt(10); //0, 1, 2, ... 9 közül egy egész double randomPossibility = rnd.nextDouble();

Ötletelj!

* Hogyan generálnál -10 és 20 közötti egész véletlenszámot?
* És -3 és 3 közötti lebegőpontosat?

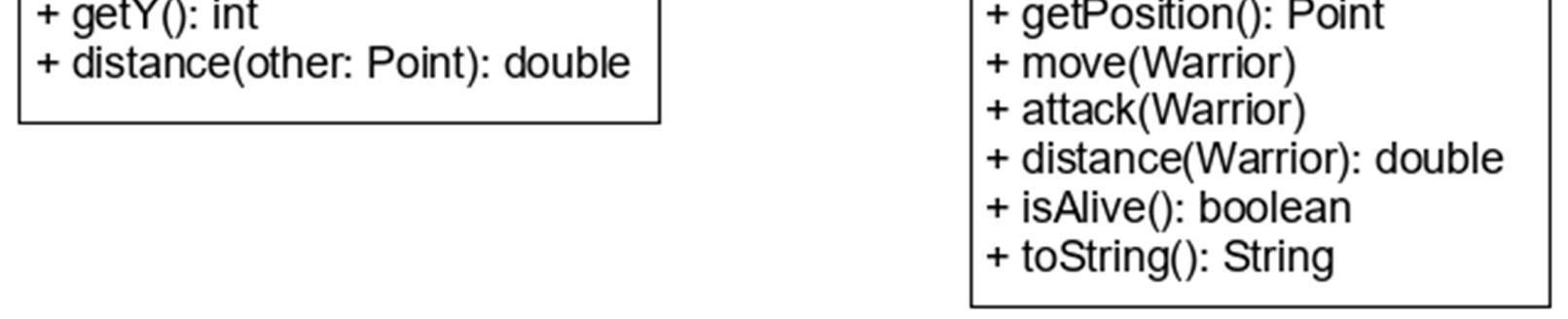
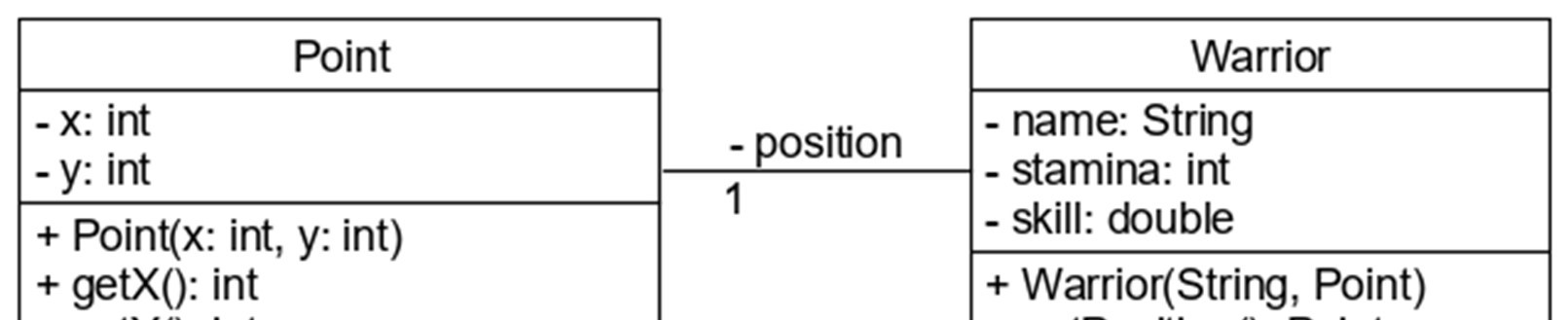
Ellenőrző kérdések

* Milyen attribútumokat és metódusokat tartalmaz a Math osztály?
* Mi a pseudorandom és seed fogalma?
* Milyen metódusokat tartalmaz a Random osztály?

Feladat

Ha harc, hát legyen harc!

Készítsd el egy játék szereplőit! Minden szereplőnek van neve, életereje, ügyessége és pozíciója. A konstruktorban csak a nevét és a pozícióját kapja meg, az életereje 20 és 100 között generálódik véletlenszerűen, az ügyessége pedig egy lebegőpontos szám 0 és 1 között szintén véletlenszerű induló értékkel. Az ügyessége annak a valószínűsége, hogy egy harc során eltalálja az ellenfelét.



Warrior Game UML Metódusai:

* move(): a harcos pozíciója a kapott Point irányába elmozdul eggyel. Csak szomszédos cellába léphet, tehát csak fel, le, jobbra, balra vagy átlósan 1-et.
* distance(): visszaadja, hogy a paraméterül kapott karaktertől milyen messze van. A tényleges számítást a Point osztály distance() metódusában készítsd el, itt csak delegáld a feladatot!
* attack(): generálj egy véletlenszámot, és ha az kisebb, mint az ügyessége, akkor egy 1-3 közötti egész véletlenszámmal csökkentsd az ellenfél életerejét!
* isAlive(): igazat ad vissza, ha a harcos még életben van, különben hamisat.
* toString(): a harcos adatait az alábbi formában adja vissza: `név: (pozíció) életerő

A játék menetét a Game osztály main() metódusában szimuláld! Először hozz létre két harcost, akik harcolni fognak. A harchoz azonos pozíción kell lenniük, ezért egymás felé mozognak, míg el nem érik egymást. Amikor ez megtörténik, egymásra támadnak felváltva, míg valamelyikük meg nem hal.

Minden fordulóban mindkét harcos mozoghat, illetve támadhat. Jelenítsd meg a játék menetét, azaz minden forduló után írd ki a harcosok állapotát a képernyőre, a játék végén pedig hirdess győztest!

Példa kimenet:

1. round

Joachim: (5,7) 47

Kahles: (1,9) 58

1. round

Joachim: (4,8) 47 Kahles: (2,8) 58 ...

Winner: Kahles: (3,8) 51

Kerekítési pontatlanságok

Generálj ezer lebegőpontos véletlenszámot öt tizedesjegyig egy tömbbe, értékük legyen maximum egymillió. Add össze őket, majd kerekítsd a matematika szabályai szerint, valamint kerekítsd le egyesével őket, és add össze a kerekített értékeket. Hasonlítsd össze a két eredményt. Futtasd le ciklusban 100-szor, és vedd a kerekítési különbségek átlagát.

A math.RoundingAnomaly osztályba dolgozz.

Külön metódusba szervezd:

* Számok generálását egy tömbbe: double[] randomNumbers(int size, double max, int scale)
* Összeadást majd kerekítést double roundAfterSum(double[] numbers)
* Kerekítést, majd összeadást: double sumAfterRound(double[] numbers)
* Különbség számolást: double difference(int size, double max, int scale). Ezt a metódust kell a main metódusból 100-szor meghívni, és a visszaadott értékeket átlagolni. Ez a metódus hívja az előző hármat úgy, hogy először legenerál egy tömböt, majd ugyanazzal a tömbbel hívd meg a roundAfterSum, majd a sumAfterRound metódusokat. Majd a két metódus által visszaadott érték különbségével kell visszatérni.

A ciklusban futtatást (100-szor) implementáld a main() metódusban.

Teszt

Hogyan lehet 1 és 5 közötti egész véletleszámot generálni a határokat is beleértve?

* Random.nextInt(5) + 1;
* Random.nextInt(6);
* ☒ new Random().nextInt(5) + 1;
* new Random().nextInt(6);

Hogyan lehet egy x egész szám négyzetgyökét kiszámolni?

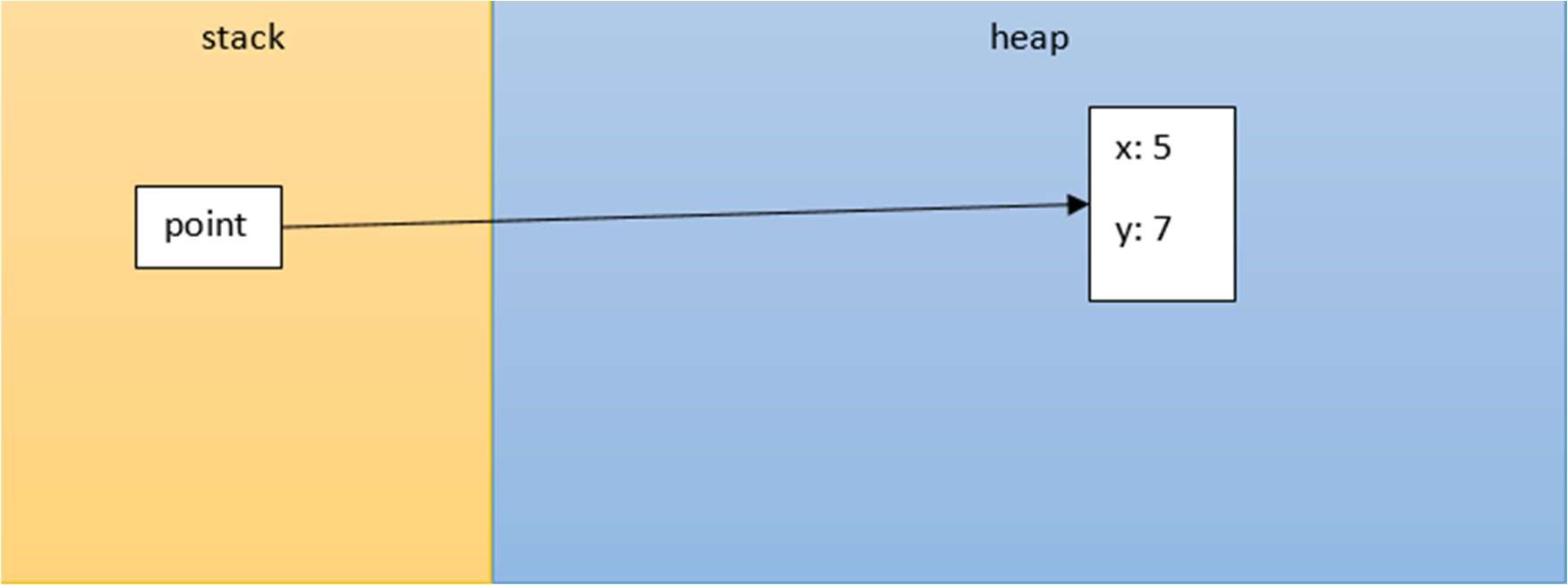
* ☒ double y = Math.sqrt(x);
* double y = new Math.sqrt(x);
* int y = Math.sqrt(x);
* int y = new Math.sqrt(x);

## Bevezetés az osztályok és objektumok világába

Objektumok (objects)

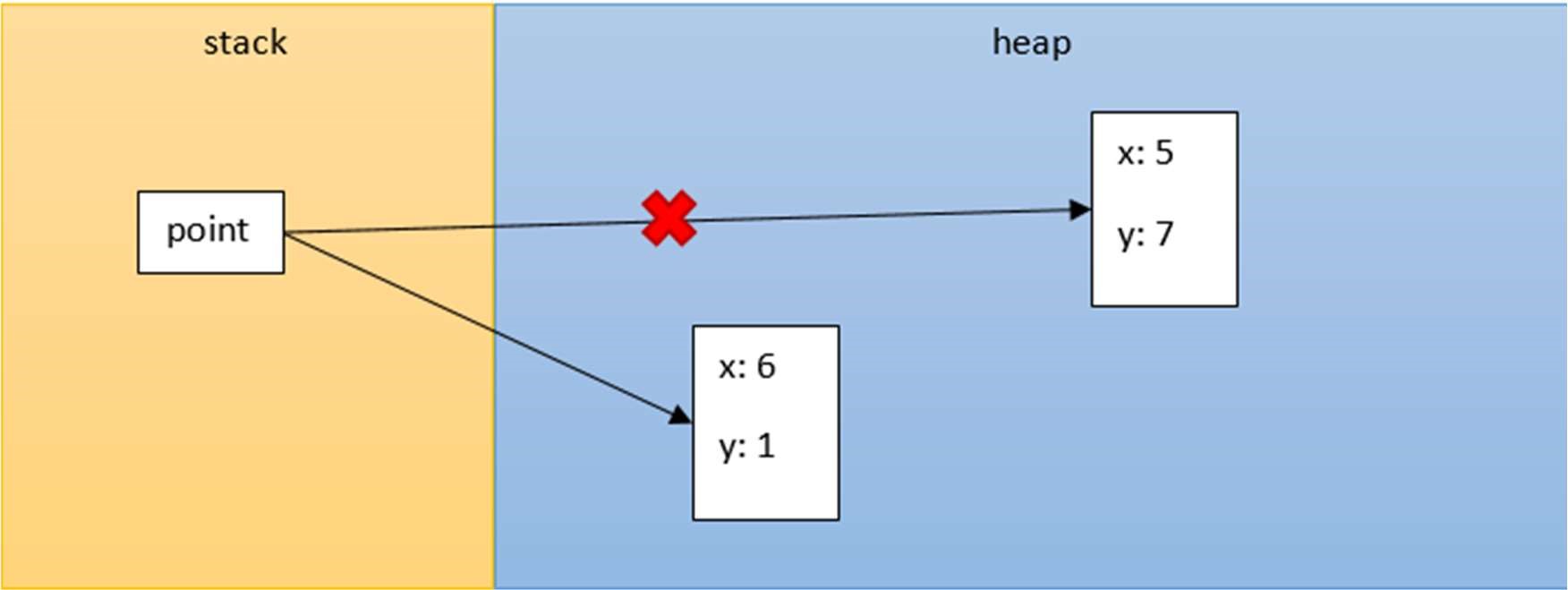
Az osztályok általában csak “tervrajzok”, a tényleges megvalósításukat objektumnak vagy példánynak nevezzük. Az attribútumok az objektum állapotát tárolják, ezért minden egyes objektumnak külön szelet jut a dinamikus memória (heap) területéből. Objektum az osztály példányosításával keletkezik, amelyet a new operátor hívásával érünk el. Ez alól kivétel a String típusú objektum, mely String típusú literál használatakor is létrejön. Ezeket az objektumokat mindig egy rájuk mutató referencián át érhetjük el, amelyet az általunk deklarált változó tartalmaz. Azaz meg kell különböztetnünk az objektumra mutató referenciát magától az objektumtól.

Point point = new Point(5, 7);



memory1

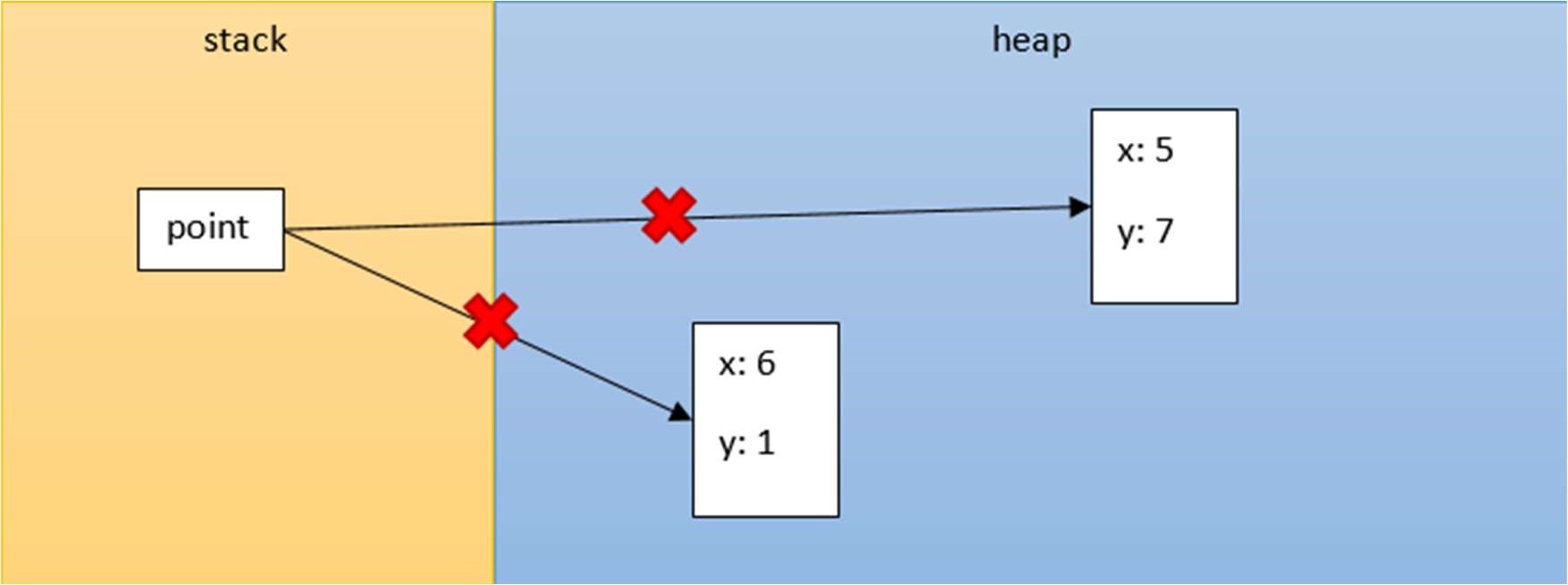
point = new Point(6, 1);



memory2

Amikor egy referencia változó nem mutat egyetlen objektumra sem, akkor azt egy speciális értékkel jelezzük, és ez a null. Mivel attribútumai és metódusai az objektumnak vannak és nem a változónak, ezért az ilyen változón nem hívhatunk egyetlen metódust sem.

point = null;



memory3

Osztály típusú változó lehet egy metóduson belül lokális, de lehet más osztályban attribútum. Objektumok lehetnek tömb vagy kollekció elemei is. Sőt, kollekcióba kizárólag objektumok kerülhetnek, primitív típusok nem.

Ellenőrző kérdések

* Mi a különbség az osztály és a példány között?
* Mit jelent a példányosítás?
* Mi a különbség az objektum és példány között?
* Milyen értéke lehet egy osztály típusú változónak?
* Mit jelent osztály típusú változó esetén az értékadás?

Feladat

Ojektumok

A objects.ObjectsMain osztály main metódusába dolgozz!

Hozz létre egy Book osztályt, melynek ne legyen sem attribútuma, sem metódusa!

Példányosíts egy Book objektumot, de ne add értékül semminek! Meg tudod ezt tenni? Hozzá tudsz férni később?

A System.out.println() metódus paramétereként adj át egy, a paraméterben most példányosított objektumot! Mit ír ki?

Deklarálj egy Book típusú emptyBook változót, de ne adj neki értéket, hanem azonnal írd ki az értékét! Fog sikerülni?

Adj neki értéket, méghozzá a null literált! Írasd ki az értékét!

Vizsgáld meg, hogy az emptyBook változó értéke null érték-e! Írd ki a vizsgálat eredményét!

Definiálj egy book nevű változót, és add értékül neki a definíciós utasításban példányosított Book objektumot!

Írasd ki a book változó értékét!

Adj értéket neki, null literált, majd írd ki az értékét!

Adj neki értékül egy új Book példányt, majd írd ki!

Hozz létre egy anotherBook változót, és adj értékül neki egy új Book példányt!

Hasonlítsd össze egyenlőség operátorral (==) a book és az anotherBook változó értékét, és írd ki az eredményt!

A anotherBook változó értékéül add a book változó értékét! Írd ki! Hasonlítsd össze egyenlőség operátorral (==) a book és az anotherBook változó értékét, és írd ki az eredményt!

Vizsgáld meg, hogy az anotherBook változó értéke Book típusú-e! Ehhez az instanceof operátort kell használni, azaz System.out.println(anotherBook instanceof Book);.

Objektumok száma

A következő kód hány objektumot hoz létre? A futás végére mennyi objektumhoz lehet hozzáférni?

Book book1 = new Book();

Book book2 = new Book();

Book book3 = new Book();

Book book4 = book1;

Book book5 = book1;

Book book6 = book3; Book book7 = null; book4 = book5; book5 = new Book(); book6 = null;

Tömbök és kollekciók

Definiálj egy Book tömböt, és adj értékül neki egy tömb literált, három példányosított Book objektummal!

Definiálj egy List<Book> kollekciót, és az Arrays.asList() metódust használva tegyél bele három példányosított elemet!

Definiálj egy List<Book> üres kollekciót, majd adj hozzá három példányosított objektumot!

Teszt

Melyik a helyes állítás?

* A referencia mindig a stacken jön létre.
* ☒ Az objektum mindig a heapen jön létre.
* Az objektum mindig a stacken jön létre.
* A referencia mindig a heapen jön létre.

Vizsgáld meg az alábbi kódrészletet! Melyik állítás igaz?

public class Main {

public static void main(String[] args) {

Book book1 = new Book("Gárdonyi Géza", "Egri csillagok");

Book book2 = new Book("Arany János", "Balladák");

Book book3 = book1;

}

}

* A main() metódusban 2 referencia és 3 objektum van. • A main() metódusban 3 referencia és 3 objektum van.
* A main() metódusban 2 referencia és 2 objektum van.
* ☒ A main() metódusban 3 referencia és 2 objektum van.

Bevezetés az attribútumok használatába (attributes)

Az osztály attribútumai tárolják az objektum állapotát. Mivel a Java szigorúan típusos nyelv, az attribútumok deklarációjában meg kell adnunk azok típusát és nevét is. Ezen kívül adhatunk meg láthatóság módosítót és egyéb módosítókat is, mint például a final, amellyel már találkoztunk korábban.

private final String name; //[láthatóság módosító] [egyéb módosító] típus azonosító;

Referencia típusú attribútumokkal kapcsolatot építhetünk objektumok között, ez a kapcsolat lehet szorosabb és lazább is, ez csak értelmezés kérdése. Például egy diák és a matematika dolgozatra kapott jegye között szoros a kapcsolat, hiszen a jegy önmagában nem értelmezhető, míg a diák és az osztálya között gyenge a kapcsolat, hiszen a diák nem szűnik meg létezni, ha kikerül az osztályból, legfeljebb átkerül egy másikba.

Láthatósági módosítók

Egy objektum állapota más objektumok számára lehet rejtett és látható is, ez attól függ, hogy milyen láthatóság módosítóval deklaráljuk.

* private: senki számára nem látható és nem módosítható
* default vagy package private: csak az azonos csomagban lévő osztályok számára látható
* protected: csak az azonos csomagban lévő vagy a leszármazott osztályokból látható (az öröklésről később még lesz szó)
* public: mindenki számára látható és módosítható

Leggyakrabban private módosítóval látjuk el őket, mert az information hiding alapelv szerint az attribútumokat elrejtjük a külvilág elől, azokhoz csak metódusokon át lehet hozzáférni.

Getter és setter metódusok

Mivel az objektum attribútumai legtöbbször rejtettek, értéküket kiolvasni és beállítani csak a konstruktorban vagy metódus segítségével lehet. Az értéket lekérdező metódusokat getternek, az értékadó metódusokat setternek nevezzük és konvenció szerintem az metódus neve megegyezik az attribútum nevével get, illetve set előtaggal. Ez alól kivétel a boolean típusú attribútum, amely gettere is vagy has előtagot kap.

private String fontName; private boolean bold;

public String getFontName () {...}

public void setFontName (String fontName) {...}

public boolean isBold () {...}

public void setBold (boolean bold) {...}

Alapértelmezett értékek

Az attribútumok kezdőérték adása nélkül is rendelkeznek valamilyen értékkel, amely az attribútum típusától függ, szemben a lokális változókkal, amelyek nem.

Típus Alapértelmezett érték

byte, short, int, long 0

|  |  |
| --- | --- |
| double, float | 0.0 |
| char | ‘’ |
| boolean | false |
| osztály | null |

Élettartam

Az attribútum addig érhető el, míg az őt tartalmazó objektum. Egy objektum, ha már a program nem használja, egy darabig még létezik a memóriában, vagyis helyet foglal.

Előbb-utóbb azonban elfogyna a memória, ezért egy szemétgyűjtő (garbage collector) időnként felkutatja a referencia nélküli objektumokat, és felszabadítja az általuk elfoglalt területet.

Ellenőrző kérdések

* Mire valók az attribútumok?
* Hogyan deklarálod az attribútumokat?
* Milyen láthatósági módosítókal rendelkezhet?
* Mi az alapértelmezett értékük?
* Meddig lehet hozzáférni?

Feladat

Book osztály

Hozz létre egy attributes.book.Book osztályt, és legyen egy String title attribútuma! Hozz létre egy konstruktort, mely egy paraméteres és értéket ad a title attribútumnak! Hozz létre egy setTitle() metódust, mely értéket ad a title attribútumnak! Hozz létre egy getTitle() metódust, mely lekéri az értékét!

Teszteld a BookMain main() metódusában.

Person és Address osztály

Hozz létre egy attributes.person.Person osztályt, String name, String identificationCard attribútumokkal! Az osztályban hozz létre egy String personToString() metódust, mely szövegként adja vissza a Person adatait!

Hozz létre egy Address osztályt, String country, String city, String streetAndNumber, String zipCode attribútumokkal! Az osztályban hozz létre egy String addressToString() metódust, mely szövegként adja vissza az Address adatait!

Az attribútumok konstruktorban is megadhatóak legyenek, és legyenek getter metódusok. Legyen egy correctData() metódus mindkét osztályban, mellyel át lehet írni az összes paraméter értékét!

A Person osztály tartalmazzon egy hivatkozást az Address osztályra! (Azaz legyen a Person osztálynak egy Address típusú attribútuma! Legyen egy moveTo(Address) metódus, mely beállítja a címet, és egy getAddress(), mellyel lekérdezhetővé válik!

Teszteld az osztályokat a PersonMain main() metódusában!

Bill és Item osztály

Legyen egy attributes.bill.Item osztály, melynek legyen String product, int quantity és egy double price attribútuma! Legyen konstruktor, valamint legyenek getter metódusok!

Legyen egy Bill osztály, melynek legyen egy List<Item> items attribútuma! Legyen egy addItem(Item) metódus, és egy getter az items attribútumhoz!

A Bill osztályban legyen egy calculateTotalPrice() metódus, mely végigmegy a számla tételein, beszorozza a quantity és price értékeket, és összeadja őket!

Teszteld a BillMain main() metódusával!

Forrás

OCA - Chapter 1/Declaring and Initializing Variables

Teszt

Melyik állítás HAMIS?

* Minden attribútum automatikus kezdőértékkel rendelkezik.
* Minden referencia típusú attribútum automatikus kezdőértéke null.
* ☒ Minden lokális változó automatikus kezdőértékkel rendelkezik.
* Minden primitív típusú lokális változónak használat előtt értéket kell adnunk, mert nincs automatikus kezdőértékük.

Bevezetés a konstruktorok használatába (introconstructors)

Az osztály példányosításakor egy speciális metódus, a konstruktor fut le, ezért az attribútumok inicializálását ebben végezzük. A konstruktornak speciális szignatúrája van: nincs visszatérési értéke, és a neve meg kell egyezzen az osztály nevével. Minden osztálynak van konstruktora, akkor is, ha nem írunk. Ebben az esetben a fordító generál egy paraméter nélküli alapértelmezett (default) konstruktort. Az objektum kezdő állapotát a konstruktor paraméterein át tudjuk beállítani. A formális paraméterek neve nagyon gyakran megegyezik az attribútum nevével, ezzel eltakarva azt. Ekkor az attribútumokat this kulcsszóval minősítve érhetjük el.

public class Person {

private String name;

private int age;

public Person(String name, int age){ this.name = name; this.age = age;

}

}

Az IntelliJ IDEA támogatja a konstruktor létrehozását. Miután megadtuk az osztály attribútumait, nyomjuk le az ALT + Insert billentyűkombinációt és válasszuk a Constructor menüpontot! A megjelenő ablakban kiválaszthatjuk, hogy mely attribútumokat szeretnénk a konstruktorban beállítani, és az IDE létrehozza azt nekünk. A létrejött konstruktort később módosíthatjuk, ha szükséges.

Ellenőrző kérdések

* Mire való a konstruktor?
* Hogyan definiáljuk a konstruktort?
* Mikor kerül meghívásra?

Feladat

Feladatok

Hozz létre egy Task osztályt, mely az elvégzendő feladatokról tartalmaz információkat. A feladatnak van címe (title), leírása (description), elkezdésének időpontja (startDateTime), időtartama (duration).

Fordítsd le az osztályt, és nyisd meg az editorban a Task.class fájlt! Van benne konstruktor? Ha van, mi a tartalma?

Task példányt a feladat címének és leírásának megadásával lehet létrehozni. Ennek megfelelően készítsd el az osztály konstruktorát! Fordítás után újra nézd meg a a Task.class fájl tartalmát! Milyen és hány konstruktor van benne?

Készíts minden attribútumhoz gettert, a duration attribútumhoz settert, és egy start() metódust, mely a startDateTime attribútumot az aktuális dátumra és időpontra állítja be!

A main() metódusban teszteld az osztályt!

Étterem

Hozz létre egy introconstructors.Restaurant osztályt, melyben van egy List<String> menu, egy String name és egy int capacity attribútum!

Hozz létre egy Restaurant(String name, int numberOfTables) konstruktort, mely beállítja az étterem nevét, a kapacitást feltölti az asztalok számának négyszeresével (csak négyszemélyes asztalok vannak) és feltölti a menüt pár étellel (ez utóbbit szervezd ki külön metódusba)!

Legyenek az osztálynak getter metódusai!

A RestaurantMain osztály main() metódusában példányosítsd a Restaurant osztályt, majd írd ki az állapotát!

Forrás

OCA - Chapter 1/Creating objects

Teszt

Igaz-e? Default konstruktora minden osztálynak van.

* Igaz
* ☒ Hamis

Mikor fut le a konstruktor?

* Az osztály betöltődésekor.
* ☒ Az osztály példányosításakor.
* Létező objektumon hívhatjuk meg bármikor.

Bevezetés a metódusok használatába (intromethods)

A metódusok az objektum attribútumain dolgoznak. Az alapján, hogy segítségével adatokat nyerünk ki vagy módosítunk megkülönböztetünk lekérdező és állapot módosító metódusokat. A getter metódusok mind lekérdezők, míg a setter metódusok mind állapot módosítók.

Metódusok felépítése

A metódusoknak van feje (láthatósága, visszatérési típusa, neve, formális paraméterlistája) és törzse. Az imperatív programozás eszköze, azaz törzse lokális változó deklarációkat és utasításokat tartalmaz. Egy metódus mindig hozzáfér az őt tartalmazó osztály attribútumaihoz. Konvenció szerint egy metódus neve olyan igét tartalmaz, amely arra utal, hogy mit csinál. Ez lekérdező metódusok esetén leggyakrabban get, find, query, állapot módosító metódusok esetén set, change, modify.

Egy metódus hívhat más metódusokat, amelyek lehetnek saját példányon belül, (objektum) attribútum vagy lokális (objektum) változó látható metódusai.

private List<Double> numbers;

public double getSumOfElements(){ // Fej: láthatóság, visszatérési típus, név, (paraméterlista)

double sum = 0; // Lokális változó for (double a: numbers) { sum = sum + a;

}

return sum;

}

public double getAverageOfElements(){

if (numbers.isEmpty()) { // Attribútum metódusa return 0.0;

}

return getSumOfElements() / numbers.size(); // Saját metódus és attribútum metódusa

}

A lokális változók a deklarációtól kezdve azon blokk végéig érhetőek el, amelyikben deklaráltuk őket. A fenti példában a sum változó a getSumOfElements() metódus végéig, de az a változó csak a ciklus végéig létezik.

Paraméterek

Metódus deklarációjakor egy formális paraméterlistát adunk meg, ami azt jelenti, hogy ezek tényleges értéke csak futási időben derül ki, de rájuk a metóduson belül az itt adott névvel hivatkozhatunk. A paraméterlista üres is lehet, de a metódusfejbe ekkor is ki kell tenni a kerek zárójeleket. Metódus hívása a nevével és az aktuális paraméterek megadásával lehet. Ha a metódus ad vissza valamilyen értéket, azt felhasználhatjuk kifejezésben. Az aktuális paraméterek megadása pontosan olyan sorrendben történik, mint amilyen a formális paraméterek sorrendje.

Deklaráció:

public String sayHappyBirthdayTo(String firstName, String lastName, int age) {

return "Boldog " + age + ". születésnapot " + lastName + " " + firstName + "!";

}

Hívás:

public void otherMethod(){

System.out.println(sayHappyBirthdayTo("Margit", "Balogh", 23)); // OK

System.out.println(sayHappyBirthdayTo("Ferenc", "Tercsik", 24.1)); // Nem jó a 3. paraméter típusa

System.out.println(sayHappyBirthdayTo(28, "Anna", "Tóth")); // Nem jó a sorrend

}

Névütközés van, ha a formális paraméter neve egyezik egy attribútum vagy lokális változó nevével. Az attribútumra hivatkozhatunk a this minősítővel, de a lokális változó sajnos elfedi a paramétert, ezért az nem lesz elérhető.

private String name = "John";

public void sayHello(String name){

String name = "Anonymous"; // Elfedi a paramétert

System.out.println("Hello " + name); // Hello Anonymous

System.out.println("Hello " + this.name); // Hello John

}

Mi történik, ha a paraméteren változtatunk?

A Java nyelvben minden paraméter érték szerint adódik át. Ez azt jelenti, hogy híváskor az aktuális paraméter értéke átmásolódik a formális paraméterként megadott változóba. Ha változtatunk az értékén, akkor ez a formális paraméter változó módosul, az eredeti változót nem érinti. DE VIGYÁZZ! Ha osztály típusú a paraméter, akkor a referenciáját másolja át, vagyis a metódus az eredeti objektumhoz fér hozzá. A legjobb, ha a paramétereken sosem változtatsz.

Speciális metódusok

String toString(): az objektum szöveges reprezentációját adja vissza. Minden osztályban van, akkor is, ha nem írjuk bele, csak akkor alapértelmezett megjelenítése (osztálynév@furcsa karaktersorozat) lesz, amely nem túl informatív. Ha bármi mást szeretnénk megjeleníteni, akkor ezt a metódust újra kell írnunk. Ebben segítségünkre van a fejlesztőkörnyezet, amely képes legenerálni egy olyan toString() metódust, amely az objektum állapotát jeleníti meg olvasható formában, de akár egyedit is írhatunk. Mivel már létező metódust szeretnénk felülírni, ezért fölé tegyük ki a @Override szót (annotációt). A System.out.println() metódus mindig ezt a metódust hívja anélkül, hogy azt explicit módon meghívnánk.

@Override

public String toString(){

return "Nevem " + this.name;

}

Az IntelliJ IDEA a leggyakrabban használt metódusok elkészítésében nagy segítséget nyújt. Nem csak a konstruktor, hanem a getter és setter metódusok automatikus generálására is képes, valamint az olyan speciális metódusokat, mint a toString() is el tudja készíteni egy alapértelmezett implementációval. Utólag természetesen bármelyik legenerált metódust átírhatjuk.

A Generate menüt az ALT + Ins billenytűkombinációval érhetjük el.

Ellenőrző kérdések

* Tipikusan hogyan épül fel egy metódus?
* Mit tartalmazhat egy metódus törzse?
* Milyen metódusokat különböztetünk meg?
* Hogyan tudjuk egy objektum állapotát kiírni legegyszerűbb módon újrafelhasználhatóan?

Feladat

Alkalmazottak

Készítsd el az Employee osztályt, amelyben az alkalmazott nevét (name), belépés évét (hiringYear) és az egész értékű fizetését (salary) tárolod attribútumként!

Az osztály példányosításakor mindhárom adatot meg kell adni, konstruktort ennek megfelelően készítsd el!

Minden attribútumához készíts gettert, valamint a name attribútumhoz settert is! A fizetése utólag emelhető, ezért készíts egy raiseSalary() metódust, amely paraméterként megkapja az emelés mértékét forintban, és ennek megfelelően módosítja a fizetést!

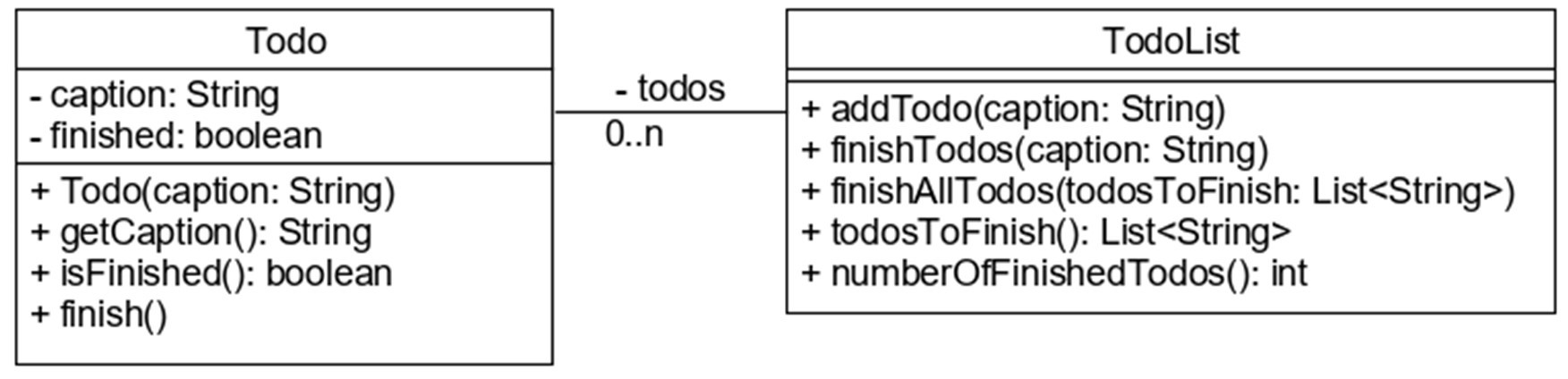
Generáld le az IDE segítségével a toString() metódust! Milyen alakban jelenik meg egy objektum állapota? Teszteld az osztály metódusait az EmployeeMain osztály main() metódusában!

TodoList

Készíts egy intromethods.TodoList osztályt, mely egy tennivaló listát kezel!

Legyen egy intromethods.Todo osztály, melynek finish() metódusa a finished attribútum értékét true értékre állítja!

A TodoList osztály egy List<Todo> típusú attribútumként tárolja a tennivalókat.



Todos UML

A TodoList metódusai:

* addTodo() - felvesz egy új tennivalót a listába
* finishTodos() - az összes olyan tennivalót befejez, melynek a neve megegyezik a paraméterként átadott névvel
* finishAllTodos() - egyszerre több tennivalót lehet befejezni
* todosToFinish() - visszaadja a befejezendő tennivalók neveit • numberOfFinishedTodos() - visszaadja a befejezett tennivalók számát

Teszteld a intromethods.TodoListMain main() metódusából.

Todo toString() metódus

Készíts toString() metódust a Todo osztályban, mely visszaadja a tennivaló nevét, és zárójelben megjeleníti, hogy be van-e fejezve.

Készíts toString() metódust a TodoList osztályban, mely visszaadja a tennivalókat szövegesen. Delegáld a hívást az ArrayList toString() metódusának.

Teszt

Mely főbb részekből áll egy metódus?

* fej és láb
* ☒ fej és törzs
* törzs és láb
* fej, törzs és láb

Az Employee osztálynak az int year attribútuma a belépés évét tartalmazza. A int getWorkLength(int year) metódusának az a feladata, hogy visszaadja, hogy a paraméterül kapott évig összesen hány évet dolgozott az alkalmazott. Hogyan lehet ezt kiszámítani?

* Nem lehet, mert a formális paraméter és az attribútum neve sosem lehet ugyanaz.
* this.year - year
* object.year - year
* ☒ year - this.year

Hogyan lehet a Student osztály move() metódusát meghívni?

public class Student { private String name; private String city; private String street; private int numberOfHouse; //Constructor, getters

public void move(String street, int numberOfHouse, String city) { //Method body

}

}

* A paramétereket az attribútum sorrendjében kell megadni:

student.move("Debrecen", "Xantus utca", 5)

* A paramétereket a formális paraméterlista sorrendjében kell megadni, de amelyik nem változik, azt nem kötelező: student.move("Xantus utca", 5)
* ☒ A paramétereket a formális paraméterlista sorrendjében kell megadni, kötelezően mind: student.move("Xantus utca", 5, "Debrecen")
* A paraméterek tetszőleges sorrendben megadhatók, csak jelezni kell, hogy melyik paraméterbe kerüljön: student.move(city: "Debrecen", street: "Xantus utca", numberOfHouse: 5)

Referenciák (references)

A JVM a memóriát két fő területre bontja: stack és heap. A stackben minden metódus külön területet kap, amelyben a paramétereit és a lokális változóit tárolja. Ez olyan, mintha egy saját fiókos szekrénye lenne címkézhető fiókokkal. Amikor deklarálunk egy változót, akkor a változó neve egy ilyen “fiók” címkéje lesz, a változó értéke pedig bekerül a fiókba. Innen később ki tudjuk olvasni, illetve le is tudjuk cserélni. Primitív típusú változó esetén a valódi értéke kerül ide, osztály típusú esetén pedig egy referencia a létrejött objektumra vagy null. Egy referencia típusú változóban tehát ténylegesen nem maga az objektum, hanem csak egy rá mutató referencia van. Az objektumot szintén egy kis fiókos szekrényként képzelhetjük el egy hatalmas raktárban a heapen. Az objektum attribútumai ezért már a heapre kerülnek. Ha ezek primitív típusúak, akkor közvetlen az objektum területére, ha referencia típusúak, akkor megint csak referenciát tartalmaznak a heapen egy másik objektumra. Ez egy hatalmas irányított gráfként képzelhető el, ahol a nyilak kiindulásánál egy referencia változó, a végpontjában egy objektum áll.

Lássunk egy példát!

public class Trainer { private String name; private int yearOfBirth;

public Person(String name, int yearOfBirth) { this.name = name;

this.yearOfBirth = yearOfBirth;

}

//Getter, setter metódusok

}

public class Main {

public static void main(String[] args) { int yearOfBirth = 1980;

String employeeName = "John Doe";

Employee jack = new Employee("Jack Doe", 1970);

}

}

Ekkor a yearOfBirth változó és annak értéke, az 1980 a stacken helyezkedik el. A John Doe szöveg, mivel az egy String objektum, a heapen kerül eltárolásra, de a rá mutató employeeName változó értéke, azaz a referencia a stacken kerül eltárolásra.

Az Employee objektum a heapen kerül letárolásra, míg a rá mutató jack változó, azaz a referencia a stacken.

Ami még érdekes, hogy az Employee egyik attribútuma, a name maga is referencia típusú, azaz egy másik, String objektumra mutat.

Amikor két változót a == operátorral hasonlítunk össze, akkor azok értékértékeit hasonlítjuk össze. Ez primitív típusoknál maga az érték, míg referencia típus esetén maga a referencia az érték. Ezért referencia típusok összehasonlítása azt vizsgálja, hogy a két referencia ugynarra az objektumra mutat-e, és nem azt, hogy a kettő állapota megegyezik-e.

Egy osztály metódusán belül a példányra a this kulcsszóval lehet hivatkozni. Ezt főleg akkor használjuk, ha a példány egy attribútumát elfedi egy lokális változó, pl. paraméter.

Ellenőrző kérdések

* Egy objektum példányosításkor annak állapota hol kerül tárolásra?
* Mit tartalmaz egy változó primitív típus és osztály típus esetén?
* Hol kerülnek letárolásra a lokális változók?

Feladat

Referenciák

Készíts egy Person osztályt a references.parameters csomagba, melyben eltárolod a nevét (name) és az életkorát (age)! A konstruktor mindkét attribútumot megkapja, és mindenhez generálj gettert és settert!

Deklarálj a ReferencesMain osztály main() metódusában két Person típusú változót! Példányosíts egy új Person objektumot, és add értékül az első változónak! A második változónak add értékül az elsőt! Módosítsd a második változón át az objektum name attribútumát! Írd ki mindkét változó tartalmát a képernyőre! Mit tapasztalsz? Vajon mi történt?

Deklarálj két egész típusú változót! Az elsőnek add értékül a 24-et! A második változónak add értékül az elsőt, majd növeld meg a második változó értékét 1-gyel! Írd ki mindkét változót a képernyőre! Mit tapasztalsz? Miért?

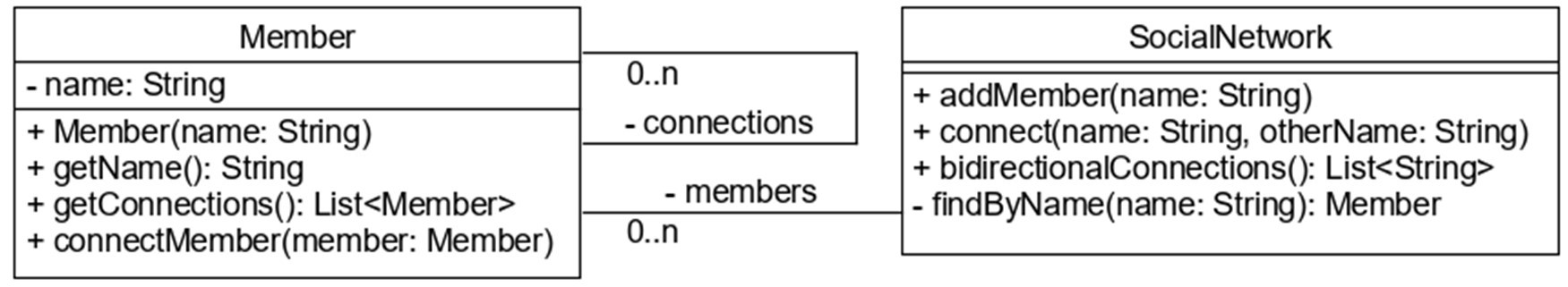
Készíts egy új Person objektumot és add értékül az egyik Person típusú változónak! Írd ki mindkét változó tartalmát a képernyőre! Mit tapasztalsz? Miért?

Próbáld követni, hogy mi történik a memóriában! Segítségedre lesz a debugger.

Közösségi hálózat

Javaslat, hogy a feladat megoldása előtt próbáld meg lerajzolni az objektumokat, és a közöttük lévő referenciákat.

Implementálj egy közösségi hálózatot, ami tagokból áll, és mindegyik tag ismerősnek jelölhet egy másik tagot! Ki kell keresni azon kapcsolatokat, ahol a tagok egymást jelölték be.



Social network UML

Egy tagot reprezentáljon a references.socialnetwork.Member osztály. A connections attribútuma a bejelölt tagokat tartalmazza.

Figyeld meg, hogy az osztály egy saját típusú attribútumot is tartalmaz!

A connectMember() metódusa a listába beteszi a paraméterként átadott elemet.

Hozd létre a references.socialnetwork.SocialNetwork osztályt, mely List<Member> típusú attribútuma az összes tagot tartalmazza!

A addMember() metódusa példányosítson a paraméterként megadott névvel egy Member osztályt, és adja hozzá a listához!

A connect() metódusa kikeresi az első tagot név szerint, majd kikeresi a második tagot név szerint, és az első connectMember() metódusát kell meghívni a második taggal mint paraméterrel.

A kikereséshez implementálj egy privát findByName() segédmetódust a SocialNetwork osztályba, ami kikeresi a members listából a tagot név szerint.

A bidirectionalConnections() metódusa keresse ki azokat a tagokat, melyek egymást bejelölték. Egy ciklusban végig kell menni a members listán, majd azon belül egy másik ciklusban a kapcsolatain (getConnections()). Amennyiben a második tag is bejelölte az első tagot (azaz az második tag benne van a kapcsolatai listájában - használd a lista contains() metódusát) -, a kettő tag nevét fűzd össze egy Stringbe, és tedd egy List<String> típusú változóba!

A következő kódot kell majd megírni a references.SocialNetworkMain main() függvényben:

SocialNetwork socialNetwork = new SocialNetwork(); socialNetwork.addMember("Joe"); socialNetwork.addMember("John"); socialNetwork.addMember("Jane"); socialNetwork.addMember("Richard");

socialNetwork.connect("Joe", "John"); socialNetwork.connect("John", "Joe");

System.out.println(socialNetwork.bidirectionalConnections()); Az utolsó sornak a következőt kell kiírnia:

[Joe - John, John - Joe]

Közösségi hálózat szövegekben

Implementáld a SocialNetwork toString() metódusát a members változó toString() metódusának hívásával!

Implementáld a Member toString() metódusát, hogy írja ki a tag nevét, és azon tagok nevét, akiket bejelölt! Segítségként implementáld a Member osztályban a List<String> connectedNames() metódust, mely egy listaként visszaadja a bejelölt tagok nevét!

Bónusz feladat 1

Miért nem működik a következő metódus a Member osztályban? Próbáld ki!

@Override

public String toString() {

return name + " " + connections.toString();

}

Forrás

OCA - Chapter 1/Distinguishing Between Object References and Primitives Teszt

Melyik állítás IGAZ a Java memóriakezelésére?

* Az objektumok a heapen jönnek létre, míg az attribútumaik a stacken.
* A primitív típusú adatok a stacken, míg az objektumok a heapen tárolódnak.
* ☒ A lokális változók mind a stacken jönnek létre, de a változó értéke primitív típus esetén maga az adat, objektum esetén pedig egy referencia a heapen létrejött objektumra.
* Mivel Javában minden adat objektum, ezért ugyan a változók a stacken jönnek létre, de a tartalmuk mindig a heapen található objektum referenciája.

Mi igaz az alábbi kódrészletre?

int a = 6; int b = a;

* ☒ Mindkét változó tartalma 6, és egyik változtatása sem hat ki a másik értékére.
* Mindkét változó tartalma 6, és bármelyik változtatása kihat a másik értékére.
* Mindkét változó tartalma 6, és a b változó változtatása kihat az a értékére, de ez visszafelé nem igaz.
* Mindkét változó értéke 6, és az a változó változtatása kihat a b változó értékére, de ez visszafelé nem igaz.

# A Java nyelv részletes megismerése

## Típusok és operátorok

Literálok (literals)

A literál a program kódba direktben beírt, “beégetett” érték, melynek önmagában is jelentése van. Fontos, hogy ezt futás közben nem tudjuk megváltoztatni.

Objektumliterál

Java nyelvben egyetlen objektumliterál létezik, ez nem más, mint a null. Ezt az értéket akkor használjuk, ha azt akarjuk definiálni, hogy a változó nem mutat egyetlen objektumra sem, azaz nincs referenciája.

String s = null;

Logikai literál

Két értéke lehet true és false.

Egész számok

Egész számot igen sok féle képpen le tudunk írni, különböző számrendszerekben.

* Bináris, pl. 0b0011
* Oktális, pl. 0377
* Hexadecimális, pl. 0xff
* Decimális, pl. 12

Vigyázzunk, a 0 előtagú számokat oktális és nem decimális számrendszerben értjük, így egészen más értéket kaphatunk, mint szerettük volna.

Alapértelmezett típusa az egész számoknak int , ez átkonvertálható long típusba, csupán a szám mögé kell egy l vagy L betűt írnunk. Javasolt az L használata, mert a l könnyen összetéveszthető az 1 számjeggyel. Pl. 012L

Olvashatóság javítására használhatjuk az aláhúzás (\_) karaktert. pl.: 0b0011\_1100 , 100\_000

Lebegőpontos számok

A lebegőpontos számokat kétféle képpen tudjuk megadni. Fontos, hogy a pont (.) a tizedes elválasztó karakter!

* Decimális megadási mód, pl. -12.3
* Exponens használata, pl. -12.3e4 (-12.3 \* 10^4 értéket képviseli )

Alapértelmezett típusa double. Ha azt szeretnénk, hogy float legyen, akkor az f, illetve az F suffixet kell használnunk, pl. 1.0F.

Az egész értékű számokat kétféleképpen is megadhatjuk, hogy az double típusú literál legyen. Az egyik módszer, hogy kiírunk egy tizedes jegyet, azaz a 2 helyett 2.0-t írunk le.

A másik módszer, hogy a szám után d vagy D postfixet írunk, például 2d.

Karakteres literálok

Karakter literálként meg lehet adni egyetlen egy karaktert pl. 'a'. Figyelnünk kell a speciális karakterekre, például az ékezetes betűkre, ugyanis a Java virtuális gép és az operációs rendszer kódolása eltérhet. Javasolt a fejlesztőezközt úgy beállítani, hogy minden állomány UTF-8 karakterkódolással legyen elmentve. A kódolást a Maven pom.xml fájlban kell megadni.

Speciális karakterek lehetnek például a sortörés, az idézőjel, vagy a visszaperjel, ilyenkor úgynevezett escape karaktert kell használnunk, ami a visszaperjel (\) , így ezek rendre :

'\n', '\”' , '\\' . Karakteres literálokat is megadhatunk oktális, illetve hexadecimális számrendszerben, ekkor a prefix \0 és \u. Például '\u0067'.

Ha egy szöveget szeretnénk megadni, azt idézőjelek ("") között tehetjük meg. Ilyenkor egy String objektum jön létre, amit a Java virtuális gép fog példányosítani.

Osztályliterál

Még egy speciális literál a típust reprezentáló osztály objektum. Ebben az esetben a .class végződést kell használnunk. Például String esetében String.class.

Ellenőrző kérdések

* Milyen objektumliterált ismersz?
* Amennyiben leírsz egy egész számot, annak mi a típusa? Hogyan lehet ezt módosítani?
* Milyen számrendszerben lehet megadni egész számokat?
* Hogyan lehet olvashatóbbá tenni?
* Lebegőpontos számoknál hogyan lehet exponenst megadni?
* Ha leírsz egy lebegőpontos számot, mi a típusa? Hogyan lehet módosítani?
* Hogyan ábrázolja a JVM a karaktereket?
* Hogyan szerepelnek a karakterek a forráskódban? Hogyan lehet ezt Maven pom.xml állományban állítani?
* Milyen speciális karaktereket ismersz?
* Hogyan adsz meg osztályt reprezentáló literált?

Feladat

A literals.LiteralsMain osztályba dolgozz!

Összefűzés

Fűzd össze szövegként az 1 és 2 literált! Milyen megoldásokat ismersz?

Osztás

Vedd a 3 és a 4 hányadosát, és tárold el a double quotient változóban, majd írd ki! Mi lesz az eredmény?

Miért van ez így?

Hogyan lehet ezt pontosítani kizárólag literálok használatával?

Nagy szám

Definiáld a 3\_244\_444\_444 literált, és add értékül a long big változónak!

Karakterkódolás

Definiálj egy String s változót, melynek legyen az értéke árvíztűrőtükörfúrógép! Fordítsd le úgy, hogy a pom.xml állományban megjegyzésbe teszed a karakterkódolásra vonatkozó sorokat! Futtasd az alkalmazást parancssorból! Megjegyzés: Nem várunk különbséget, mert az elmúlt egy évben a Maven “megokosodott” és a karakterkódolás megadása nélkül is korrekt kimenetet biztosít.

String mint objektum

Definiálj egy String word változót, melynek az értéke legyen a TITLE szöveg nagybetűkkel! A szövegliterál kisbetűkből álljon, és hajtsuk végre rajta a toUpperCase() metódust az értékadás előtt!

Szám bináris stringként

Írasd ki az 1 és a -2 értéket bináris formájában! Keresgélj az Integer osztály metódusai között!

Teszt

Melyik NEM karakter literál a Javában?

* 'a'
* ☒ "a"
* '\065'
* '\u0061'
* '\n'

Milyen típusú a 0b0011 literál?

* byte
* ☒ int
* long
* char
* String

Egyszerű típusok (primitivetypes)

Nézzük meg részletesen a primitív típusokat, először azonban értelmezzük a csomagolóosztály fogalmát.

Míg a primitív típusok egyszerű adatokat tartalmaznak, addig a csomagoló osztályok ezen kívül az adaton dolgozó metódusokat is. Tulajdonképpen a primitív típusú adatot burkolják be, és ruházzák fel egyszerű műveletekkel.

A primitív típusok és a nekik megfelelő csomagoló osztályok sorra a következők:

Primitív típus Csomagoló osztály Mit ábrázol?

|  |  |  |
| --- | --- | --- |
| boolean | Boolean | logikai (8 bit) |
| char | Character | 16 bites Unicode karakter (UTF-16) |
| byte | Byte | 8 bites előjeles egész szám |
| short | Short | 16 bites előjeles egész szám |
| int | Integer | 32 bites előjeles egész szám |
| long | Long | 64 bites előjeles egész szám |
| float | Float | 32 bites lebegőpontos racionális szám |
| double | Double | 64 bites lebegőpontos racionális szám |

Autoboxing

A primitív típus és a csomagoló osztálya között a fordító automatikusan tud be- és kicsomagolni. Ezt nevezzük autoboxingnak, illetve unboxingnak.

int number = 5;

Integer numberObj = number; //Autoboxing int number2 = numberObj; //Autounboxing

Számrendszerek

Különböző értékek, különböző számrendszerekben való kiíratása. Az adott burkolóosztálynak van erre megfelelő toString() metódusa.

* Integer.toString(100, 8) oktális számrendszerben
* Integer.toString(100, 2) bináris számrendszerben
* Integer.toString(100, 16) hexadecimális számrendszerben

Ez a metódus a negatív számokat ugyanúgy jeleníti meg, mint a pozitívakat, csak elé teszi az előjelet.

Integer.toString(5, 2) –> 101

Integer.toString(-5, 2) –> -101

Fontos megjegyezni, hogy a negatív számokat a Java virtuális gép úgynevezett kettes komplementer kódban tárolja. Ez azt jelenti, hogy nem előjel bitet alkalmaz a negatív számok esetén, hanem a kivonást vezeti vissza összeadás műveletre. Ez alapján a -5 az a szám, amihez 5-öt adva 0-t kapunk.

5: 00000000000000000000000000000101

-5: 11111111111111111111111111111011 //A -5 kettes komplementer ábrázolása

0: 00000000000000000000000000000000

A kettes komplementer szerinti karaktersorozatot az Integer.toBinaryString() metódussal kapjuk meg.

Integer.toBinaryString(-5) –> 11111111111111111111111111111011

Szövegből átalakítás

Lehetőségünk van szöveget számmá alakítani, illetve fordítva. Ha szövegből a burkoló osztály egy példányát szeretnénk létrehozni, akkor a szöveget a konstruktorban kell átadni. Ha primitív típust szeretnénk visszakapni, akkor használjuk a csomagolóosztály parse prefixű metódusát.

* Integer i = new Integer("123")
* int i = Integer.parseInt("123")

Szövegből logikai értéket a Boolean.parseBoolean(String str) metódussal tudunk készíteni. Amennyiben a paraméterként átadott szöveg a "true" szöveget tartalmazza bármilyen kis-nagybetű kombinációban, a konvertált érték true lesz, bármilyen más esetben false.

boolean first = Boolean.parseBoolean("TruE"); // true boolean second = Boolean.parseBoolean("yes"); //false

“Szélsőséges” eredmények

A csomagoló osztályok konstansokat tartalmaznak a „szélsőséges” eredményekre. Például tárolják az értelmezési tartomány két végpontját, pl. Integer.MIN\_VALUE , Integer.MAX\_VALUE.

A különböző matematikai műveletek eredményét az IEEE szabvány definiálja. Példák:

* 1.0 / 0 eredménye Double.POSITIVE\_INFINITY
* -1.0 / 0 eredménye Double.NEGATIVE\_INFINITY
* Double.POSITIVE\_INFINITY / Double.NEGATIVE\_INFINITY eredménye Double.NaN (Not a Number)

Ellenőrző kérdések

* Milyen primitív típusokat ismersz?
* Hány biten vannak ábrázolva?
* Mit jelent a csomagoló típus?
* Mi az a bináris számrendszer?
* Mit jelent a kettes komplemens számábrázolás?

Feladat

Átváltás kettes számrendszerbe

A primitivetypes.PrimitiveTypes osztályba írj egy String toBinaryString(int n) metódust, mely az adott pozitív egész számot kettes számrendszerbe váltja át!

Ellenőrizd a PrimitiveTypesMain main() metódusban, hogy értéke megegyezik-e a Integer.toBinaryString() metódus által visszaadott értékkel!

A mi metódusunk annyiban legyen más, hogy a szám mindig 32 bites legyen, azaz 32 karakter hosszú szöveget adjunk vissza, és az elején legyen kiegészítve nullákkal!

A 32 legyen külön változóba kiemelve!

Az algoritmus a következő: amíg a szám nagyobb, mint nulla, a számot osztani kell kettővel, és a maradékát is képezni kell. A maradék lesz a bináris számjegy. Fontos, hogy hátulról előre kell a számjegyeket leírni. A maradékos osztás Javaban a % operátorral történik.

Bónusz feladat

Van operator overloading Javában, azaz egyszerű operátorokkal, mint a + operátor, lehet két Integer objektumot összadni?

Mit ír ki a new Integer(1) + new Integer(2) kifejezés? Miért?

Teszt

Melyik szövegből vagy szöveggé alakító utasítás nem jó?

* String s = Integer.toString(23);
* ☒ long n = String.toLong("1\_000\_004");
* int a = Integer.parseInt("23");
* boolean b = Boolean.parseBoolean("ajaj");

Felsorolásos típus (enumtype)

A felsorolásos típus valójában egy osztály, rendelkezik attribútumokkal, konstruktorral és metódusokkal. A különbség, hogy nem a class, hanem az enum kulcsszóval hozzuk létre, és csak a felsorolásos típuson belül definiált elemeket lehet neki értékül adni. Ezeket az elemeket konvenció szerint csupa nagybetűvel írjuk. Az elemek között sorrendiség definiált, így index alapján is elérhetőek, for ciklussal bejárható és switch szerkezetben is használhatóak. Gyakran használjuk logikai értékek helyett is, mert beszédes nevű elemek esetén jobban olvasható.

public enum Coin {

TWOHUNDRED, HUNDRED, TWENTY, TEN, FIVE

}

Használata:

Coin c = Coin.TWOHUNDRED;

Mint látható, a Coin olyan, mint egy osztály, a benne definiált értékek pedig ennek az osztálynak a példányai. Az enum values() metódusa az összes lehetséges értéket visszaadja egy tömbben ugyanolyan sorrendben, mint ahogy azt definiáltuk. Így az egyes enum értékek indexszel elérhetőek, illetve ciklussal bejárhatóak.

Bejárás:

for(Coin i : Coin.values()){

System.out.println(i);

}

Amennyiben az enumot attribútumokkal szeretnénk ellátni, abban az esetben ezt a felsorolás után, attól ;-vel elválasztva tehetjük meg. Konstruktort és metódusokat is hasonlóan készíthetünk bele, mint bármilyen osztály esetén, azonban egy enum konstruktora sohasem lehet publikus. Hogy miért? Mert az enum a fejlesztő által nem példányosítható. Az egyes példányok a konstansként definiált enum értékekbe automatikusan kerülnek bele, ezért a konstruktor hívásához szükséges konkrét paramétereket a definiált értékeknél kell megadnunk.

public enum Coin {

TWOHUNDRED(200), HUNDRED(100), TWENTY(20), TEN(10), FIVE(5);

private final int value;

Coin(int value){ this.value=value;

}

public int getValue(){ return value;

}

}

Így már minden elemhez hozzárendeltünk egy értéket, amit a getteren át le tudunk kérdezni.

Coin coin = Coin.HUNDRED; int coinValue = coin.getValue(); //100

Hasznos metódusok

A values() metódusról már volt szó, ez az összes enum értéket tartalmazó tömböt ad vissza.

Coin[] coins = Coin.values();

A valueOf() metódussal String alapján lehet lekérni a felsorolásos típus egy elemét, ahol a String maga a definiált konstans neve.

Coin c = Coin.valueOf("HUNDRED"); //Coin.HUNDRED

A name() metódus az ellenkező irány, amikor az enum értékből szeretnénk String-et kapni.

Coin c = Coin.HUNDRED;

String nameOfCoin = c.name(); //"HUNDRED"

Az ordinal() visszaadja az adott elem sorszámát. Ugyanúgy, mint a tömb indexelése, ez is 0-val kezdődik.

Coin c = Coin.HUNDRED; int index = c.ordinal(); //1

Ellenőrző kérdések

* Mire használjuk a felsorolásos típusokat?
* Hogyan lehet definiálni a felsorolásos típusokat?
* Milyen hasznos metódusokat ismersz velük kapcsolatban?

Feladat

A hét napjai

Vegyél fel egy enumtype.week.DayType enumot, melynek két értéke a WORKDAY, HOLIDAY! Vegyél fel egy Day enumot, mely a hét napjait tartalmazza, és a szombat és vasárnap legyen megjelölve szünnapnak! A WorkdayCalculator osztályban legyen egy

List<DayType> dayTypes(Day firstDay, int numberOfDays) metódus, melynek meg kell mondani az első napot, majd az utána következő napok számát, és visszaad egy listát, mely azt tartalmazza, hogy a i. nap milyen típusú!

Használj egy private Day nextDay(Day day) segédmetódust, mely megmondja a paraméterként megadott nap után következő napot! Vasárnap után hétfő következik.

Teszteld a WorkdayCalculatorMain osztály main() metódusában!

Mértékegységek

Legyen egy enumtype.unit.LengthUnit enum, mely tartalmazza a milliméter, centiméter, méter, yard, foot és inch mértékegységeket. Mindegyik tartalmazza, hogy SI mértékegység-e, valamint hogy egy egység mennyi milliméterre átváltva.

Írj a UnitConverter osztályban egy BigDecimal convert(BigDecimal length, LengthUnit source, LengthUnit target) metódust, mely átváltja a paraméterként megkapott értéket, melynek meg van adva a mértékegysége a cél mértékegységre! Először váltsd át milliméterre, majd vissza a cél mértékegységre! Négy tizedesjegyre kell kerekíteni.

A List<LengthUnit> siUnits() metódus adja vissza az SI mértékegységeket.

A UnitConverterMain main() metódusában próbáld ki a convert() metódust, majd írd ki az összes mértékegységet, valamint csak az SI mértékegységeket!

Bónusz feladat

Hol lenne a nextDay() valamint a siUnits() metódus helye? Hogy lehet ezt ott definiálni, ha nem példányhoz, hanem osztályhoz tartozik? Hogy lehet meghívni?

Forrás

OCP - Chapter 1/Working with Enums

Teszt

Hogyan lehet szövegből enum példányt előállítani?

* Coin c = Coin.values("TEN");
* Coin c = new Coin("TEN");
* Coin c = Coin.parse("TEN");
* ☒ Coin c = Coin.valueOf("TEN");

Hogyan lehet a felsorolásos típus második elemét lekérdezni?

* Coin c = Coin.ordinal(1);

☒ Coin c = Coin.values()[1];

* Coin c = Coin.values(1);
* Coin c = Coin.valueOf(1);

Operátorok (operators)

Az operátorok kifejezésekben szerepelnek. Például:

int num = 3; int result = num + 2;

Itt a num és 2 az operandus a + és a = pedig az operátor. Egy kifejezésben több operátor is szerepelhet. Ilyenkor nem mindegy, hogy milyen sorrendben értékeljük ki őket, ezért először nézzük meg ezt.

Operátorok kiértékelési sorrendje

* Először a belső zárójel tartalma
* Ha nincs zárójel, akkor a nagyobb precedenciájú operátor
* Egyenlő precedencia esetén balról jobbra, értékadás esetén jobbról balra

Precedenciatáblázat

* Postfix operátor (kifejezés++, kifejezés--)
* Prefix operátor (++kifejezés, --kifejezés)
* További egyoperandusú operátorok (+,-, !)
* Multiplikatív operátorok (\*, /, %)
* Additív operátorok (+, -)
* Léptető műveletek (<<, >>, >>>)
* Összehasonlítás (<,<=,>, >=)
* Egyenlőségvizsgálat (==, !=)
* Bitenkénti (&, ^, |)
* Logikai (&&, ||)
* Feltételes kifejezés (? :) - Figyelem! Három operandusú!
* Értékadások (=, +=, -=, \*=, =, >>=, <<=, >>>=, &=, ^=, |=)

A precedencia az operátorok erősségét jelzi. Nem érdemes megjegyezni a precendencia táblázatot, inkább használjunk helyette megfelelő zárójelezést.

Léptető műveletek

A léptető műveletek bináris műveletek, tehát érdemes ismerni az adott szám bináris reprezentációját. Nézzünk ehhez néhány metódust!

* Integer.toBinaryString() a számot átváltja kettes számrendszerbe.
* Integer.parseInt(String, int) a megadott számrendszerben ábrázolt reprezentációból adja vissza az adott számot. Ha a második paraméter 2, akkor a bináris reprezentációból alakít számmá.

Léptető operátorok:

* a >> b: az a bitjeit jobbra lépteti b-szer, balról ugyanolyan bitekkel tölti fel, amilyen az eredeti bal szélső bit volt.

a << b: az a bitjeit balra lépteti b-szer, jobbról 0 bitekkel tölti fel a helyeket.

* a >>> b: az a bitjeit jobbra lépteti b-szer, balról 0 bitekkel tölti fel a helyeket.

Például a 9 binárisan 1001.

String number = Integer.toBinaryString(9); // Eredmény "1001" number = Integer.toBinaryString(9 >> 1); //// Eredmény "100" number = Integer.toBinaryString(9 << 1); //// Eredmény "10010" number = Integer.toBinaryString(Integer.MIN\_VALUE); //// Eredmény

"10000000000000000000000000000000" number = Integer.toBinaryString(Integer.MIN\_VALUE >> 1); //// Eredmény

"11000000000000000000000000000000" number = Integer.toBinaryString(9 >>> 1); //// Eredmény "01000000000000000000000000000000"

Bitenkénti operátorok

Boolean értékek esetén logikai műveletek, egész számok esetén a bináris reprezentáció minden bitjére végrehajtja a műveletet.

* Az ÉS (&) eredménye csak akkor 1, ha minden operandus 1
* A VAGY (|) eredménye csak akkor 0, ha minden operandus 0
* A KIZÁRÓ VAGY (^) eredménye csak akkor 1, ha az operandusok eltérnek Például:

1110010011110101

& 0011111001111011

----------------

0010010001110001

1110010011110101

| 0011111001111011

---------------- 1111111011111111

1110010011110101

^ 0011111001111011

----------------

1101101010001110

Logikai operátorok

* Az ÉS (&&) eredménye csak akkor true, ha minden operandus true
* A VAGY (||) eredménye csak akkor false, ha minden operandus false

Rövidzár kifejezés

Tudjuk például a VAGY műveletnél, hogy ha a baloldali operandus értéke true, akkor az egész kifejezés értéke is az, így a jobb oldali operandust nem kell kiértékelni. Ha a baloldali kifejezés értéke false, akkor természetesen zajlik tovább a kiértékelés. Nézzük meg példán keresztül a || és a | operátorok közötti különbséget!

* true || (1/0 == 0) kifejezés értéke true

true | (1/0 == 0) kifejezés kiértékelése közben

java.lang.ArithmeticException keletkezik “/ by zero” üzenettel

Természetesen ugyanez elmondható a && és az & műveletekre. Első esetben ha a kifejezés baloldala false, akkor az egész kifejezés hamis, míg a bitenkénti operátornál az egész kifejezés kiértékelésre kerül.

Értékadás

Az értékadásnak is van eredménye, a bal oldali változóba kerülő érték, így ez is használható operandusként.

Ellenőrző kérdések

* Nézd át egyesével az operátorokat! Melyik mire való? Hogyan működik?
* Mi a különbség a bitenkénti és logikai operátorok között?
* Mit jelent a rövidzár kifejezés?

Feladat

Páros szám

A operators.Operators osztályba dolgozz, a teszteléseket viszont az operators.OperatorsMain osztály main() metódusában végezd!

Hozz létre egy boolean isEven(int n) metódust, mely visszaadja, hogy a paraméterként átadott egész szám páros-e!

Léptető operátorok

Milyen matematikai műveletnek felel meg a jobbra vagy balra léptetés? Próbáld ki, hogy mi történik, ha a 16-ot lépteted jobbra vagy balra! És ha a 13-at?

Hogyan lehetne léptetésekkel megvalósítani a szorzást? Készíts egy multiplyByPowerOfTwo() metódust az Operators osztályba, amely az első paraméterként kapott számot megszorozza 2-vel annyiszor, amennyi a második paraméter. (Ne használj ciklust, csak léptető operátort!)

Bónusz feladat

Miért ad a következő kódrészlet kivételt?

int i = -1;

String s = Integer.toBinaryString(i); System.out.println(s); int j = Integer.parseInt(s, 2); System.out.println(j);

Miért megoldás erre a Long.valueOf(s, 2).intValue() kifejezés használata?

Bónusz feladat

Mit ír ki a következő kifejezés, és miért?

System.out.println(0333);

Teszt

Mit ír ki az alábbi kódrészlet?

int x = 5; int y = 2; int z = x++ + (x - 4) \* y - 2; System.out.println(x + " " + z);

* ☒ 6 7
* 5 7 • 6 8 • 5 8 • 6 6
* 5 5

Típuskonverzió (typeconversion)

A Java erősen típusos nyelv. Ha egy kifejezésben az operandusok különböző típusúak, akkor típuskonverzióra van szükség, tehát a típusokat össze kell egyeztetni. Ez általában fordítási időben ellenőrizhető. Van néhány eset, amikor futás közben derül ki, hogy nem konvertálható az érték, így kivétel keletkezik.

Automatikus típuskonverzió

Az automatikus konverzió akkor működik, ha a bővebb ábrázolási tartomány felé kell konvertálni. Kivétel, hogy a float és double változónak adható long érték, de ez adatvesztéssel járhat.

int number = 54; long longNumber = number; double doubleNumber = longNumber;

A byte, short, char típusnak értékül adható megfelelő int literál, ha az értékre belefér.

byte byteNumber = -6; short shortNumber = 12\_398; char charNumber = 6;

Számokkal való műveletvégzéshez mindig a bővebb ábrázolási tartományú típusra konvertálja az operandusokat, de egész számokat legalább int-re. Ezt azért fontos tudnunk, mert két byte típusú szám összege még akkor is int típusú, ha egyébként beleférne az eredmény a byte-ba.

byte a = 4; byte b = 5; int i = 12;

byte c = a + b; //Nem jó, mert a jobb oldal int típusú.

int x = a \* i; // int \* byte --> int

Explicit típuskonverzió

Explicit konverziót akkor kell használni, amikor a szűkebb ábrázolási tartomány felé szeretnénk konvertálni. Ebben az esetben információvesztés történhet. Egész számok esetén elvesznek a felső bitek, míg lebegőpontos számok egészre való konvertálásakor nem kerekítés történik, hanem elvesznek a tizedes jegyek.

public static void main(String[] args){ int i = 5; long l = 500; float f = 1;

double d = 10.1;

i = (int) l; i = (int) d;

}

Ellenőrző kérdések

* Mire való a típuskonverzió?
* Milyen fajta típuskonverziókat ismersz?
* Milyen furcsa esetet ismersz, amikor az automatikus konverzió furcsán működik?
* Hogyan adható meg explicit konverzió?
* Hogyan működik az explicit konverzió egész számok esetén?
* Hogyan működik az explicit konverzió, amennyiben lebegőpontos számot konvertálunk egész számmá?

Feladat

Adatvesztés

A typeconversion.dataloss.DataLoss osztályba dolgozz! Írj egy dataLoss() metódust, mely kiírja az első három olyan pozitív egész long értéket, melyet float, majd vissza long értékké konvertálva adatvesztés történik! Írd ki binárisan is!

Az eredeti és a konvertált érték között mennyi a különbség? Hány bináris számjegynél jelenik meg a probléma?

Teszteld a main() metódusból!

Melyik típusba való?

A typeconversion.whichtype.WhichType osztályba írj egy List<Type> whichType(String s) metódust, mely visszaadja, hogy a paraméterben String-ként megadott long-on biztosan ábrázolható szám milyen más adattípusokba férhet még bele (byte, short, int). A Type egy enum, mely tartalmazza a típusokat, és mindegyikhez külön attribútumban meg lehet adni long-ként a minimális és maximális értéket.

Teszteld a typeconversion.whichtype.WhichTypeMain osztály main() metódusában!

Tipp: A paraméterként átadott értéket long értékké kell alakítani (Long.parseLong()), majd ciklusban végigmenni az enum értékein, és megnézni, hogy belefér-e a tartományba.

Teszt

Melyik értékadás helytelen, ha az x változó int típusú?

* long l = x;
* double d = x;
* ☒ short s = x;
* char c = (char) x;

Egész és lebegőpontos számok (numbers)

A / operátor furcsaságai

A Java nyelvben a / operátornak a + operátorhoz hasonlóan két funkciója is van. Az egyik az egész osztás, a másik a valós osztás. Azt, hogy éppen melyiket kell elvégezni, az operandusok típusa határozza meg. Amennyiben mindkét operandus valamelyik egész típusból való, abban az esetben az eredmény is egész szám lesz, hiába tesszük lebegőpontos változóba.

double quotient = 10 / 4; // az eredmény 2.0

Ha legalább az egyik operandus lebegőpontos szám, akkor az eredmény is lebegőpontos szám lesz.

double fraction = 10 / 4.0; // az eredmény minden esetben 2.5 double fraction = 10 / 4D; double fraction = (double) 10 / 4;

Lebegőpontos számok összehasonlítása

Gyakran nevezzük a tizedes törteket valós számoknak, azonban a számítógép végtelen törteket nem tud ábrázolni, valahol mindig vége lesz a tizedesjegyek sorának. Ráadásul a bináris ábrázolás miatt a véges tizedes törtek sokszor csak végtelen kettedes törtként írhatóak le, vagyis csak pontatlanul ábrázolhatók. Ezek egyik mellékhatása, hogy két lebegőpontos szám nem lesz biztosan egyenlő még akkor sem, ha egyébként annak kellene lennie.

System.out.println(0.1 \* 3 == 0.3); // false

System.out.println(0.1 \* 3); // 0.30000000000000004

A legjobb megoldás, ha két lebegőpontos számot nagyon kicsi eltéréssel már egyenlőnek tekintünk. De hogyan lehet ezt leírni? Ha az összehasonlítandó számok a és b, a megengedett eltérés pedig delta:

boolean equals = Math.abs(a - b) < delta; Például:

System.out.println(Math.abs(0.1 \* 3 - 0.3) < 0.005); // true

System.out.println(Math.abs(0.1 \* 3 - 0.3) < 1.0e-15); // true

Ellenőrző kérdések

* Mennyi lesz az 5/6 eredménye Javában? És az 5.0/6.0 eredménye? Miért?
* Hogyan lehet két lebegőpontos szám egyenlőségét megvizsgálni?

Feladat

Kör

Készíts egy Circle osztályt, amelyben eltárolod annak egész értékű átmérőjét (diameter) és a Pi értékét két tizedesjegy pontossággal! Az átmérő konstruktorban kap értéket. Készíts két metódust: az egyik a kör kerületét adja vissza (perimeter()), a másik a területét (area())! Ezek visszatérési típusa lebegőpontos legyen!

Próbáld ki az osztály működését a CircleMain main() metódusában! Készíts két kört és írd ki mindkét kör kerületét és területét! A körök átmérőjét konzolról olvasd be!

Matematikai feladatok

Készíts egy MathOperations osztályt! A main() metódusában írj ki a felhasználónak egy négy alapműveletet és zárójeleket tartalmazó számítási feladatot, majd kérd be tőle az eredményt! Ellenőrizd a kapott értéket, és jelezd vissza, hogy helyesen oldotta-e meg a feladatot. A megengedett eltérés 0.0001 legyen

A Scanner osztályt használhatod double típusú adatok bekérésére is. A nextDouble() metódusa a futtató operációs rendszer alapértelmezett formátumában értelmezi a beírt szöveget, azaz magyar környezet esetén a választ tizedesvesszőt használva kell megadni, például 5,342.

Forrás

OCA - Chapter 1/Understanding Default Initialization of Variables, Understanding Variable Scope

Teszt

Mit ír ki az alábbi kódrészlet?

int a = 10;

double b = (15 - a) / 2 \* (double) 3; System.out.println(b);

* ☒ 6.0
* 7.5
* 0.8333333333333334
* 0.0

Mit ír ki?

System.out.println(5 + 6 + "0");

* 11
* 56
* ☒ 110
* 560

## Vezérlési szerkezetek

Unit tesztelés JUnittal (introjunit)

A unit tesztelés célja az alkalmazás legkisebb egységének tesztelése. Ezért Javában unit tesztelni osztályokat szoktunk.

Törekedjünk arra, hogy a unit teszt legyen automatikus, megismételhető. Az alapkoncepció, hogy bízunk benne, hogyha a darabok hibátlanok, akkor az egész is az. Fontos, hogy a hibákat minél előbb megtaláljuk, és gondoljunk a szélsőséges esetekre is, így biztonságosabban tudjuk változtatni a kódot, hiszen amíg a unit teszt lefut, addig helyesen működik a program.

A unit teszt dokumentálja is a kódunkat, azaz kitalálható belőle egy osztály működése.

JUnit

A JUnit egy keretrendszer Javában implementálva Java osztályok unit teszteléséhez. Egy JUnit teszteset általában három részből áll:

* Given - adott állapot (általában példányosítunk)
* When - meghívunk rajta egy metódust (metódus hívás)
* Then – az történik-e, amit vártunk (objektum állapotának, vagy a metódus visszatérési értékének vizsgálata)

Utóbbit úgynevezett assert-ekkel tudjuk megtenni. Ezek csupán annyit vizsgálnak, hogy a visszaadott és érték megfelel-e az elvártnak. A Hamcrest keretrendszer segít abban, hogy az objektumok vizsgálatát meg tudjuk valósítani, illetve az esetleges különbségeket emberibb formában jeleníti meg. A JUnitot támogatja a Maven, illetve az IDE-k is. Mavenben a tesztek az src/test/java mappában találhatóak és ezen belül a tesztelendő osztálynak megfelelő csomagban.

Példa teszt:

import org.junit.Test; import static org.hamcrest.CoreMatchers.equalTo; import static org.junit.Assert.assertThat;

public class TrainerTest {

@Test

public void testCreate() {

//Given

Trainer trainer = new Trainer("John Doe");

//When

String name = trainer.getName();

//Then

assertThat(name, equalTo("John Doe"));

}

}

A példa teszt leellenőrzi, hogy a Trainer objektum megfelelően jött-e létre, vagyis az-e a neve, amit konstruktorban átadtunk.

Tesztlefedettség

A tesztlefedettség azt méri, hogy mely kódsorok futottak le a tesztek futtatása közben. Ezt az IDE-k is támogatják. Segít nekünk abban, hogy további teszteket írjunk az esetlegesen nem tesztelt kódrészletekre. Cél a minél nagyobb tesztlefedettség (kb. 80%).

Mikor jó egy unit teszt?

Ahhoz, hogy igazán jó unit teszteket írjunk néhány szabályt be kell tartanunk.

* A tesztesetek legyenek egymástól függetlenek.

Ha az egyik teszteset függ egy másik teszteset eredményétől, akkor nem tudhatjuk, hogy az adott funkció önmagában használva jól működik-e, illetve ha mindkét teszteset elbukik, akkor csak az egyik funkció működik rosszul vagy mindkettő. A tesztesetek függetlenségét biztosítja a JUnit azzal, hogy a különböző teszteseteket véletlenszerű sorrendben futtatja.

* Egység teszt ne tartalmazzon külső függőséget

Ez nem azt jelenti, hogy a tesztelendő osztályon kívül semmilyen más osztályt nem tartalmazhat. Amiről már biztosan tudjuk, hogy jól működik, mert vagy a Java SE osztálykönyvtár vagy harmadik féltől származó library része, azt korlátlanul használhatjuk. Azonban a saját magunk által készített osztályokat helyettesítsük valamilyen test double-val, ami az eredeti interfészével rendelkezik, de annak valamilyen módon leegyszerűsített változata.

* Minden esetre kell teszt, de ne vidd túlzásba

Az összes lehetséges esetre lehetetlen tesztet írni, de nem is szükséges.

Partícionáljuk az eseteket a bemenet és/vagy az elvárt eredmény szerint! Ezután minden partícióból csak egy elemre írunk tesztet. Ha valamilyen szempont szerint rendezhetők az egy partícióba tartozó elemek, akkor a szélsőséges esetekre külön tesztet írunk.

Ellenőrző kérdések

* Mire való a JUnit keretrendszer?
* Tipikusan hogyan épül fel egy teszteset?
* Hogyan támogatja a JUnit-ot a Maven?
* Hogyan támogatja a JUnit-ot az IDE?

Feladat

Első teszt implementálása

A pom.xml állományba vedd fel függőségként a JUnit keretrendszert test scope-pal!

<dependencies>

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

<version>4.12</version>

<scope>test</scope>

</dependency>

</dependencies>

Hozz létre egy introjunit.Gentleman osztályt, melyben van egy public String sayHello(String name) metódus, mely visszaad egy String-et, mely egy üdvözlő szöveg (Hello), hozzáfűzve a paraméterként átadott név!

Létrehoztunk egy introjunit.GentlemanTest osztályt a teszt ágon, mely azt ellenőrzi, hogy John Doe nevet átadva a visszaadott szöveg valóban Hello John Doe.

Hibás teszt

Rontsd el először a programot, hogy hibás üzenetet adjon vissza! Hogy jelzi a JUnit ezt? Rontsd el a tesztesetet, hogy hibás legyen, amire ellenőriz! Hogyan jelzi ezt a JUnit a futtatáskor?

Tesztlefedettség mérése

Implementáld, hogy ha a sayHello() metódus null paramétert kap, a visszaadott szöveg Hello Anonymous legyen! Futtasd le a tesztlefedettség mérést, és nézd meg, hogy hogyan jelzi a fejlesztőeszköz, hogy az új ág nem lett lefedve! Implementáld a megfelelő tesztesetet a GentlemanTest osztályban, és futtasd le újra a lefedettség mérést!

Bónusz feladat 1.

Hova fordítja le a teszt fájlokat a Maven? Hova teszi a tesztek futtatásáról a Maven a riportokat? És mit tartalmaznak ezek?

Bónusz feladat 2.

Hogyan lehet megoldani, hogy mvn clean package parancs kiadása esetén ugorja át a teszt esetek futtatását?

Bónusz feladat 3.

Kik a JUnit fő fejlesztői? Milyen könyveket írtak, melyek hasznosak lehetnek a Java programozáshoz?

Bónusz feladat 4. Teszt eset futtatása parancssorból

Telepítsd fel a Maven legújabb verzióját a gépedre, és futtasd le a tesztesetet Mavennel parancssorból! Rontsd el a tesztelendő metódust, hogy más üzenetet adjon vissza! Hogy jelzi ezt a Maven futáskor?

Teszt

Milyen rész nincs egy JUnit tesztesetben?

* Given
* ☒ Apply
* When
* Then

Vezérlési szerkezetek és az elágazás (controlselection)

Minden algoritmus felépíthető három vezérlési szerkezet használatával:

* szekvencia
* szelekció
* iteráció

A szekvencia jelenti az utasítások egymásutániságát, a szelekció az elágazást, az iteráció pedig a ciklus használatát. Ez a tétel egyben azt is jelenti, hogy nincs szükség ugró utasításra, a fenti három szerkezettel minden algoritmus felépíthető.

Feltételes elágazás

A feltételes elágazáshoz az if kulcsszót kell használni, és két részből áll: fejből és törzsből. A fejben egy logikai kifejezés szerepel, ami ha igaz, akkor hajtódik végre a törzsben szereplő utasítássorozat. A kifejezéshez else ág is fűzhető. Ha a fejben szereplő logikai kifejezés hamis, akkor ide ugrik a vezérlés. Az else ághoz további if-eket fűzhetünk (else if).

if ((x % 2) == 0) {

System.out.println("Even");

} else {

System.out.println("Odd");

}

A fenti példában, a fejben szereplő feltétel megvizsgálja, hogy x páros vagy páratlan szám-e, majd kiírja az eredményt.

A switch utasítás

Ez is több részből áll. Szükség van egy kifejezésre, ami a switch fejében található. Ez lehet:

* bármi, ami int típussá automatikusan konvertálható
* felsorolásos típus
* String

Az úgynevezett case ágakban, fordítási időben ismert konstansok szerepelhetnek. Amikor a JVM a switch utasításhoz ér, kiértékeli a kifejezést, majd egyszerre a case ágakat. Amennyiben egyezést talál, elkezdi a végrehajtást, ami a következő break utasításig vagy a switch végéig tart. Ha nem talál egyezést, akkor a default ágra ugrik. Mivel a default ág megadása nem kötelező, elképzelhető, hogy egyetlen ág sem hajtódik végre.

public String getTypeOfDayWithSwitchStatement(String dayOfWeekArg){ String typeOfDay; switch(dayOfWeekArg){ case "Monday":

typeOfDay="Start of the work week"; break; case "Tuesday": case "Wednesday": case "Thursday":

typeOfDay="Midweek"; break; case "Friday":

typeOfDay="End of work week" break; case "Saturday": case "Sunday": typeOfDay="Weekend"; break; default:

throw new IllegalArgumentException("Unknown day"); }

return typeOfDay;

}

}

A példában jól látható, hogy ha például kedd, szerda vagy csütörtök a beérkező paraméter, akkor a nap típusa mindegyiknél Midweek lesz, és csak az utána lévő break utasításra ugrunk ki a switch-ből. Ha olyan napot kapunk, amit nem ismerünk, akkor a default ágban kivételt dobunk.

Ellenőrző kérdések

* Milyen vezérlési szerkezeteket ismersz elágazásra?
* Hogyan lehet az else ágakat összefűzni?
* Mikor kell használni a break utasítást?

Feladatok

Napszaktól függő köszönés

Írj egy metódust, mely paraméterként megkapja az órát és a percet, és amennyiben 5:00 után van, köszönjön jó reggelttel, 9:00 és 18:30 között jó napottal, 20:00-ig jó estéttel, majd jó éjttel.

A controlselection.greetings.Greetings osztályba dolgozz!

Hónap napjainak visszaadása

Írj egy olyan metódust, mely az év és a hónap magyar neve alapján visszaadja, hogy az hány napos! Használj switch szerkezetet! Figyelj arra, hogy ne számítson a kis- és nagybetű különbség!

A controlselection.month.DayInMonth osztályba dolgozz!

Ha nem ismert a hónap, dobj kivételt a következő módon:

throw new IllegalArgumentException("Invalid month: " + month);

Figyelj a szökőévre (év osztható néggyel, de nem osztható százzal, kivéve, ha osztható 400-zal)!

Hét napjai

Írj egy metódust, mely várja a hét neveit, és hétfő esetén azt adja vissza, hogy “hét eleje” van, kedd, szerda és csütörtök esetén, hogy “hét közepe” van, pénteken “majdnem hétvége”, és szombat és vasárnap esetén “hét vége”!

Figyelj arra, hogy ne számítson a kis- és nagybetű különbség!

Ha nem ismert a nap, dobj kivételt a következő módon:

throw new IllegalArgumentException("Invalid day: " + day);

A controlselection.week.DayOfWeeks osztályba dolgozz!

Magánhangzó

Írj egy metódust, mely kap egy karakter paramétert! Amennyiben magánhangzót kap, a következő mássalhangzót adja vissza! Ha mássalhangzót kap, akkor a mássalhangzót adja vissza! Elég, ha az angol ábécé karaktereivel működik.

A controlselection.consonant.ToConsonant osztályba dolgozz!

Ékezetek

Írj egy metódust, mely magyar ékezetes karakter esetén annak ékezet nélküli párját adja vissza! Ha a karakter nem ékezetes, akkor magát a karaktert adja vissza!

A controlselection.accents.WithoutAccents osztályba dolgozz!

Teszt

Mi lesz a spouse változó értéke?

String name = "Joe"; String spouse = "Jean"; switch(name) { case "John":

spouse = "Eve"; case "Joe":

spouse = "Sarah"; case "Jake":

spouse = "Mary";

}

* Jean
* Eve
* Sarah
* ☒ Mary

Mennyi lesz a power változó értéke?

int stamina = 30; int power;

if(stamina <= 10) { power = 1;

} else if(stamina <= 30) { power = 2;

} else if(stamina <= 80) { power = 3; } else {

power = 4;

}

* 1
* ☒ 2
* 3
* 4

Ciklusok (controliteration)

A while utasítás

A while utasítás két részből áll: fejléc és törzs. A fejlécben egy logikai kifejezést kell definiálni. Addig hajtja végre a törzset, amíg a fejlécben lévő kifejezés igaz. Minden végrehajtás előtt kiértékelődik a feltétel, és amint hamis lesz, kiugrik az utasításból.

Vigyázzunk, nagyon könnyű végtelen ciklust implementálni. Például ha a lenti példában elfelejtjük növelni a ciklusváltozót.

int count = 1; while(count < 11){

System.out.println("Count is: " + count); count++;

}

do-while utasítás

A feltétel a ciklus végén értékelődik ki, így a törzs egyszer mindenképpen lefut.

int count = 1;

do {

System.out.println("Count is: " + count); count++; } while(count < 11);

A fenti példában amennyiben a count értékét 20-ra állítjuk kezdetben, akkor while esetén nem írna ki semmit, do-while esetén viszont kiírja egyszer, hogy 20.

for utasítás

Szintén fejből és törzsből áll. A fej a következőket tartalmazza:

* Inicializációs utasítás
* Feltétel
* Léptető utasítás

Az inicializációs utasításban deklarált változó(k) csak a ciklus törzsében látható(ak). A for ciklus gyakran használjuk 0 és n-1 közötti értékek bejárására.

for (int i = 0; i < 10; i++) {

System.out.println(i);

}

A fejben található három részből bármelyik elhagyható, sőt, végtelen ciklust is implementálhatunk vele:

for ( ; ; ) {}

for-each utasítás

Tömb vagy kollekció bejárására használjuk.

String[] numbers = {"one","two","three"};

//List<String> numbers = new ArrayList<>();

for (String number: numbers){ System.out.println(number);

}

A példában a number változó végigmegy a tömb minden egyes elemén. A for-each ciklust pontosan ugyanígy kell használni, ha nem tömböt, hanem egy lista elemeit szeretnénk bejárni.

Ezt a ciklust tipikusan az elemek elérésére használjuk, azokat lecserélni nem lehet a number változón át. Ugyanakkor gyorsabb, mint a for ciklus, ezért ha nincs szükség az elem lecserélésére vagy az indexére, mindig ezt használjuk.

Ellenőrző kérdések

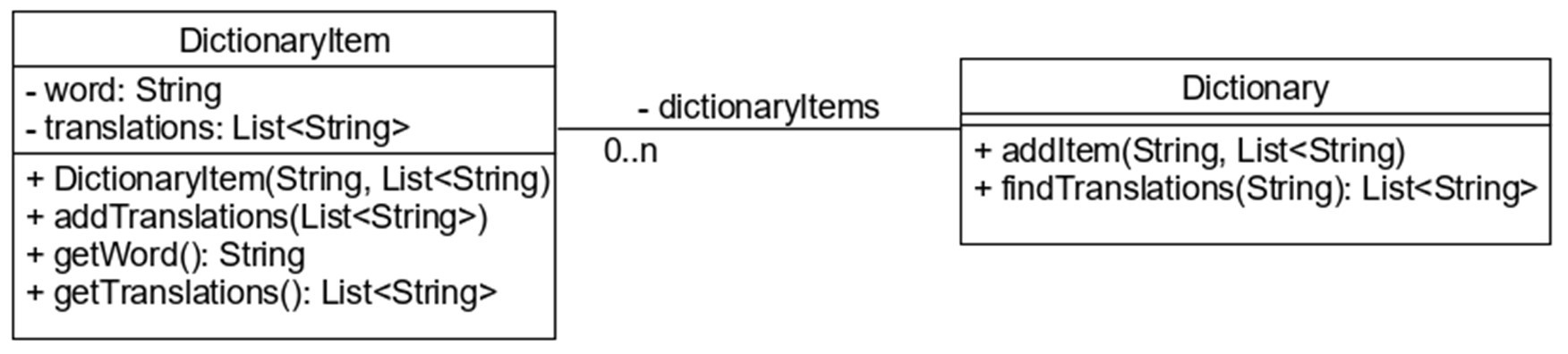
* Milyen ciklusszervező utasításokat ismersz Javában?
* Mi ezek között a különbség?
* Melyik ciklusszervező utasítással hogyan lehet végtelen ciklust képezni?
* Milyen részekből áll a for ciklus feje?
* A for ciklus fejében deklarált változó látszik-e a ciklus utáni utasításban?
* Mely ciklusszervező utasítást használjuk tömb vagy kollekció elemeinek bejárására?

Feladat

Szótár

Írj egy magyar-angol szótárt, mellyel csak ebben az irányban lehet fordítani, és az adott szóhoz több fordítást is tartalmazhat! Az adott szóhoz ki lehet keresni a hozzá tartozó fordításokat.

Hozzá lehet adni új szót, több fordítással. Amennyiben már létező szót adunk meg, a fordítások listáját bővíteni kell, de csak olyan elemekkel bővíthetjük, melyek nem szerepelnek a fordítások között.



Dictionary UML

A controliteration.dictionary csomagba dolgozz! A DictionaryItem osztály tartalmazza attribútumként a szót a fordításait.

Az addTranslations() metódus a paraméterként átadott szavakat csak akkor teszi be a fordítások listájába, ha abban még nem szerepel (használhatod a List contains() metódusát is).

A konstruktor mindkét attribútumot várja, de a fordításokat az addTranslations() metódussal adja hozzá.

A Dictionary osztály addItem() metódusával új fordításokat lehet felvenni. Figyelni kell, hogy ha az adott szó már szerepel, akkor nem kell felvenni még egyszer.

A findTranslations() metódusa visszaadja a paraméterül kapott szó fordításait. Ha nem találja a szót, üres listával tér vissza.

Pi értéke

Szász Pál matematikus verse alapján írd ki a Pi első harminc tizedesjegyét! Minden számjegy rendre megegyezik a szó hosszával a versben.

Definiálj egy számlálót, ami az adott számjegy értékét számlálja! Iterálj végig a karaktereken! Ha a karakter egy betű, akkor növeld a számláló értékét! Ha a karakter nem betű, és a számláló értéke nem nulla, fűzd hozzá a számláló értékét egy Stringhez! A második karakter legyen egy tizedespont!

Bónusz feladat

Mi történik a tesztesetek futtatásakor, ha a DictionaryItem konstruktor második List<String> paraméterét értékül adjuk az attribútumnak. Miért?

Forrás

OCA - Chapter 2/Understanding Java Statements

Teszt

Mi lesz a number változó értéke?

int number = 5; while(number < 20) {

number += 3;

}

* 5
* 17
* ☒ 20
* 23

Hányszor írja ki az “in” szót?

for(int i = 2; i < 5; i++) { System.out.println("in");

}

* 2
* ☒ 3
* 4
* 5

Haladó vezérlési szerkezetek (controladvanced)

Címkék alkalmazásai

A feltételes és ciklusképző utasításokat meg lehet címkézni. Ez különösen hasznos, ha egymásba ágyazott utasításokat használunk, és a break és a continue utasításokkal szeretnénk vezérelni ezeket.

OUTER: for (int i = 0; i < table.length; i++) { INNER: for(int j = 0; j < table[i].length; j++) { table[i][j] = i + j;

}

}

break utasítás

Ciklusokban használható. Ha címke nélkül használjuk, akkor kilép a legbelső éppen végrehajtott ciklusból. Ha címkével használjuk, akkor a címkézett utasításból lép ki.

A példában egy String tömbről szeretnénk eldönteni, hogy tartalmazza-e a searchFor szót.

boolean foundIt; for (int i=0; i<words.length; i++){ if (words[i].equals(searchFor)) { foundIt = true; break;

}

}

Jól látható, hogy ha a ciklusban beléptünk az if törzsébe, akkor a foundIt változót truera állítjuk, és kiugrunk a ciklusból a break utasítás segítségével.

continue utasítás

Szintén ciklusban használható. Ha nem használunk címkét, akkor az éppen zajló legbelső ciklus törzsének többi részét kihagyja, és újra a feltétel kerül kiértékelésre. Amennyiben címkét használunk, akkor az adott címkéjű ciklus feltétel kiértékelésével folytatódik a végrehajtás.

String words = "peter piper picked a ..."; int numPs = 0;

for (int i = 0; i < words.length(); i++) { if (words.charAt(i) != 'p') { continue;

}

numPs++;

}

A break és contiune utasításokat lehetőleg kerüljük, csak akkor használjuk, ha jobban átlátható, olvashatóbb kódot tudunk írni.

Ellenőrző kérdések

* Mire való a continue utasítás?
* Mire való a break utasítás? Feladat findDuplicates() metódus

Szűrd ki egy List<Integer> listában a többször szereplő elemeket, és add vissza.

Több megoldás elképzelhető, egyik (nem hatékony) megoldás, hogy egy ciklusban végigmész az elemeken, majd egy másik ciklusban pedig végigmész az összes elem előtt lévő elemen. Ha egyezőséget találsz, átteszed az elemet egy másik listába, és szakítsd meg a belső ciklust, különben ha egy elem háromszor ismétlődik, rosszul fog működni.

CSV validálás

Egy használt autó kereskedés az autók adatait CSV fájlban tárolja. Minden sor az alábbi szerkezetű kell legyen: rendszám;gyártási év;márka;szín.

Például:

"ABC-123;2007;Volvo;red" További szabályok:

* A rendszám mindig 7 karakterből áll és van benne - karakter.
* A gyártási év korábbi, mint 2019, de későbbi, mint 1970.

A feladat egy olyan metódus írása, mely a valid sorokat kigyűjti. A feladatot már megoldották, de sajnos a kód egyes sorai megsérültek. Ennyit sikerült megmenteni belőle:

public List<String> filterLines(List<String> lines) { List<String> validLines = new ArrayList<>(); for (String line: lines) {

String[] parts = line.split(";"); if(parts.length != 4) {

//Innen kezdve hiányzik jópár sor

} validLines.add(line);

}

return validLines;

}

Forrás

OCA - Chapter 2/Understanding Advanced Flow Control

Teszt

Az alábbi metódust 5 paraméterrel hívva mit ad vissza?

public String findPerfectMatch(int number) {

List<String> words = List.of("ninetyeight", "five", "eight", "ten", "thirteen");

String perfect = null;

for(String word: words) {

if(Math.abs(word.length() - number) > number) { continue;

}

if(word.length() == number) { perfect = word; break; }

int newLength = word.length() > number ? number : word.length(); perfect = word.substring(0, newLength);

}

return perfect;

}

* ninet
* five
* ☒ eight
* ten
* thirt

Bevezetés a kivételkezelésbe (introexception)

Kivételkezelést akkor használunk, ha fel akarunk készülni olyan esetekre, melyek normál működés esetén nem szoktak bekövetkezni. Fontos, hogy a kivételkezelést ne használjuk vezérlésre. Példák a kivételkezelésre:

* Hálózati kapcsolat megszűnik
* Hiba a fájlműveletek közben
* Nem értelmezhető paraméterek átadása metódushíváskor

Figyeljünk arra, hogy metódusokban ne használjunk speciális visszatérési értéket (pl.: 1, 0, null) kivételes esetekben, hanem használjunk kivételkezelést.

Teendők kivételes esetekben

A kivétel figyelmen kívül hagyható (nagyon ritkán). Például, ha le szeretnénk zárni egy erőforrást, ami már le van zárva. Ekkor a keletkezett kivétel figyelmen kívül hagyható, hiszen a cél az volt, hogy le legyen zárva az erőforrás.

Tovább dobhatjuk, delegálhatjuk a hibát, ha nem tudunk vele mit kezdeni, majd az alkalmazásunk egy másik pontján kezeljük.

Kezelhetjük, illetve javíthatjuk a hibát. Fontos, hogy a felhasználót értesítsük a keletkezett kivételről.

Kivételek

A kivétel nem más, mint egy osztály. Vannak az osztálykönyvtárban előre megírt kivételek, de mi magunk is írhatunk saját kivétel osztályt. A leggyakrabban használt kivételek:

* NullPointerException
* IllegalArgumentException
* IllegalStateException

Kivételeket dobhatók a throw kulcsszóval, de le is kell példányosítani a kivételosztályt.

throw new IllegalArgumentException("Wrong args!");

Kétféle kivételt különböztetünk meg: nem kezelendő illetve kötelezően kezelendő kivételeket.

Nem kezelendő kivételek

Ezek az úgynevezett unchecked kivételek. Alapvetően nem látható, csak tesztek futtatása közben vagy a dokumentációból derülhet ki, hogy egy ilyen kivétel kiváltódhat.

Kötelezően kezelendő kivételek

Kétféleképpen kezelhetjük a kötelezően kezelendő, azaz checked kivételeket. Tovább dobhatjuk, ezt a metódus fejlécében a throws kulcsszóval tehetjük meg, jelezvén, hogy a metódus közben kivétel keletkezhet.

public void readFromFile(String fileName) throws IOException, FileNotFoundException{ .

.

.

}

Másik megoldás, hogy úgynevezett try-catch szerkezettel kezeljük a kivételt, mely befoglalja a kiváltó utasítás(ok)at. Csak akkor használható, ha a körbezárt blokk dobhatja az adott kivételt.

public void divTwoNumbers(int a, int b) { try{

int result = a / b;

System.out.println("A hányados alsó egészrésze: " + result);

} catch (ArithmeticException ae) {

System.out.println("Aritmetikai hiba!");

} }

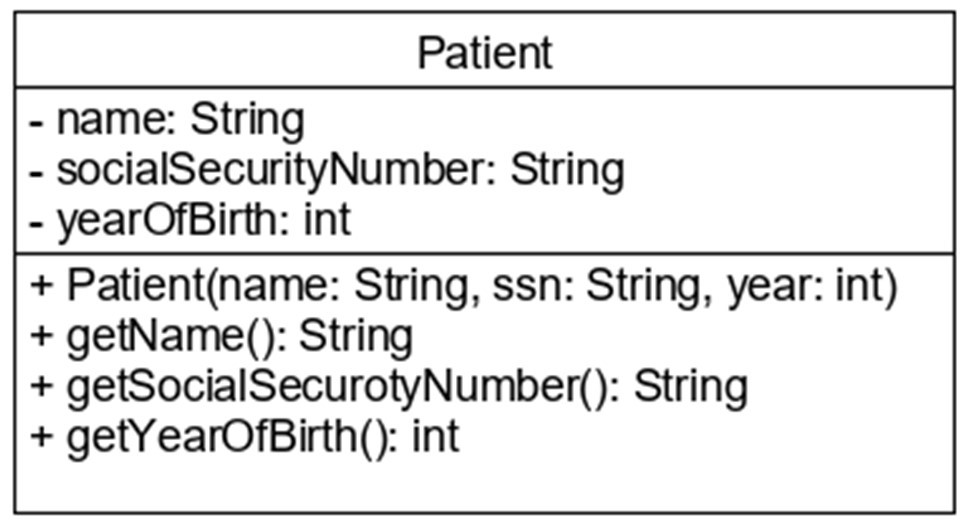
Ellenőrző kérdések

* Mi az a kivétel? Sorolj fel néhány példát!
* Mit tehetsz, ha egy kivétel dobódott?
* Hogyan tudsz kivételt dobni?
* Készíthetsz-e saját kivételosztályt?

Feladat

Rendelő 1

Készítsd el a Patient osztályt, mely tárolja a beteg nevét, TAJ számát és születési évét!



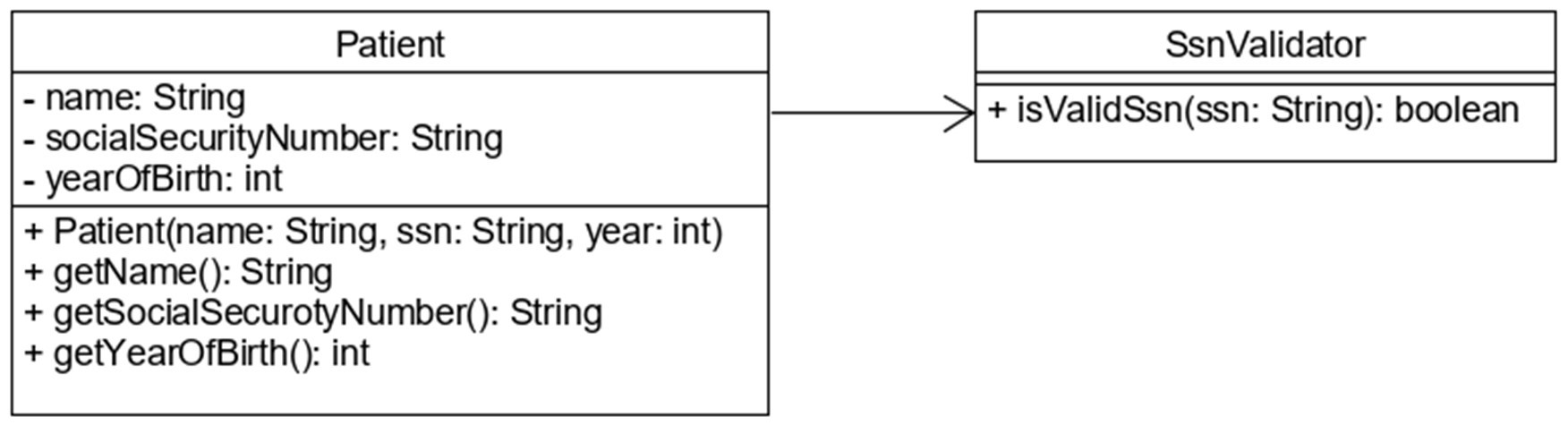
Patient UML

Konstruktorban ellenőrizd a paramétereket, és IllegalArgumentException dobásával jelezd a nem megfelelő értékeket!

A name nem lehet üres, a yearOfBirth pedig legalább 1900 kell legyen.

Rendelő 2

Egészítsd ki a Patient osztályt, hogy a TAJ számot is ellenőrizze! Ehhez hozd létre az SsnValidator osztályt!



Patient és SsnValidator UML

A TAJ számot a következő algoritmussal ellenőrizd: A TAJ szám egy kilenc számjegyből álló szám. A TAJ szám első nyolc számjegyéből a páratlan helyen állókat hárommal, a páros helyen állókat héttel szorozzuk, és a szorzatokat összeadjuk. Az összeget tízzel elosztva a maradékot tekintjük a kilencedik, azaz ellenőrző kódnak. Ha az ellenőrző kód nem egyezik, akkor ne fogadd el a TAJ számot!

Ne felejtsd el, hogy a számmá konvertálás kivételt dobhat!

Teszt

Milyen két csoportra bonthatók a kivételek?

* ☒ checked, unchecked
* error, exception
* try, catch

## String, StringBuilder

String alapok (stringbasic)

A String osztály mögött valójában egy karaktertömb van. A String immutable (módosíthatatlan), vannak ugyan metódusai, de ezek nem a belső állapotot változtatják, hanem mindig egy új objektummal térnek vissza.

A Java virtuális gépen belül létezik egy String pool, ami egy memóriaterület. A String literálok itt jönnek létre, nem pedig a heapen. Ha mégis a heapen szeretnénk létrehozni, akkor használjuk a new operátort.

A String osztályban taláható egy intern() metódus, ami azt nézi, hogy a paraméterként átadott String benne van-e a poolban, ha igen, arra ad vissza egy referenciát, ha nincs, akkor belerakja.

Fontos, hogy String objektumokat mindig az equals() metódussal hasonlítunk össze.

String s = "abc";

String s2 = "abc";

s.equals(s2); // Igaz lesz

Az == operátor az objektumok referenciájának egyezőségét vizsgálja, ezért String-ek esetén csak akkor működik helyesen, ha tényleg ugyanaz az objetum van mögötte.

Például két változó ugyanazt a literált kapja értékül.

Mikor és hogyan használjunk String-eket?

A String típusú adat a legszélesebb körben használható, de pár jó tanácsot érdemes megfogadni:

* Amennyiben bájttömbből hozzuk létre, adjuk meg a karakterkódolást. Nem mindegy, hogy az adott karakterkódhoz milyen karakterkép társul.
* Ne használjuk arra, hogy több adatot összefűzve tároljunk benne.
* Ha csak bizonyos értékeket tartalmazhat, akkor használjunk inkább enumot.
* Vannak esetek, amikor számok helyett megfelelőbb a String használata. Amikor az adatnak nem a nagysága a fontos, nem akarunk vele aritmetikai műveleteket végezni, akkor praktikusabb String-ként tárolni. Ilyen például az irányítószám, az adóazonosító jel és a telefonszám.

Ellenőrző kérdések

* Hol tárolódhatnak a String objektumok?
* Mitől függ a tárolás helye?
* Mikor true az intern() metódussal kapott ún. canonical representation alapú összehasonlítás visszatérési értéke?
* Melyik az előnyös megoldás String objektumok létrehozására és miért?

Feladat

String-ek

Azonos tartalmú String objektumokat hozunk létre, amelyeket vagy a heapre, vagy a poolba szánunk.

publikus metódusok:

public String createStringForHeap() public String createStringForPool()

Kis kedvenceink

Tervezz meg egy házi kedvencek adatainak nyilvántartására szolgáló Pet osztályt! Minden kedvencnek van neve, születési ideje, neme és regisztrációs száma. A neme csak hím, nőstény vagy ismeretlen lehet. A regisztrációs száma mindig egy 6 jegyű szám, például 000147. Melyik attribútuma milyen típusú legyen?

Készíts gettereket minden attribútumhoz!

A Vet osztály tartalmazza egy adott orvoshoz tartozó kisállatok listáját. Készíts egy add() metódust, mely az újonnan érkező kisállatot adja a listához, de csak akkor, ha még nincs ott! Ennek vizsgálatára készíts egy privát areEquals() metódust, mely a paraméterül kapott két kisállat egyezőségét adja vissza! Két kedvenc akkor tekinthető ugyanannak, ha ugyanaz a regisztrációs számuk. A listához készíts gettert!

Teszt

Mit ír ki az alábbi kódrészlet?

String password1 = "John123";

String password2 = "John123";

System.out.println(password1 == password2);

System.out.println(password1.toUpperCase() == password2.toUpperCase());

* true true
* ☒ true false
* false true
* false false

Mire való a String pool?

* A memória ezen részén tárolódnak a String-ek.
* ☒ A memória ezen részén tárolódnak a literálként létrehozott String-ek, mégpedig mindegyik csak egyszer.
* A memória ezen részén tárolódnak a new operátorral létrehozott String-ek.
* A memória ezen részén tárolódnak a String metódusok által visszaadott String-ek, mégpedig mindegyik csak egyszer.

Konkatenáció (stringconcat)

A konkatenáció összefűzést jelent. A String tartalmaz egy concat() metódust, amely egy másik String-et vár paraméterül, de figyeljünk, hogy az objektum immutable, ezért egy új String objektum fog létrejönni. Összefűzésre használható még a + operátor.

String s1 = "abc"; s1.concat("def"); // s1 értéke még mindig "abc" String s2 = s1.concat("def"); // s2 értéke "abcdef"

s1 += "def"; //s1 értéke is "abcdef"

Figyelni kell a műveletek sorrendjére. Például, ha két számot és egy szöveget szeretnénk összekonkatenálni, akkor ha először a két szám szerepel, akkor ezek összeadódnak és ehhez fűződik hozzá a szöveg. Viszont, ha előbb van a szöveg, akkor a balról jobbra történő kiértékelés miatt először a szöveghez hozzáfűződik az első szám, és így már egyben egy String lesz, majd ugyanez történik a másodikkal. Nézzünk erre példát:

System.out.println("Szöveg" + 1 + 5); // Eredmény: "Szöveg15"

System.out.println(1 + 5 + "Szöveg"); // Eredmény: "6Szöveg"

Egy lehetséges megoldás utóbbi kiküszöbölésére, ha a számok elé egy üres String-et írunk:

System.out.println("" + 1 + 5 + "Szöveg"); // Eredmény: "15Szöveg" Ennél jobb megoldás, ha a csomagoló osztályok toString() metódusát hívjuk.

System.out.println(Integer.toString(1) + 5 + "Szöveg");

Ellenőrző kérdések

* Mit jelent az, hogy a String immutable?
* Hogyan fűzhetők össze String objektumok?
* Mi történik konkatenálás esetén az objektumok szintjén? Feladat toString()

Készítsd el az Employee osztályt és annak a toString() metódusát! Az osztály tartalmazza az alkalmazott nevét, foglalkozását és fizetését, melyeket konstruktorban kap meg. A toString() metódus az alkalmazott adatait az alábbi formában adja vissza:

Kis Géza - minőségellenőr - 520000 Ft

Hibakezelés:

Minden adat megadása kötelező, és a fizetés csak 1000-rel osztható pozitív szám lehet.

Bármilyen hiba esetén dobj IllegalArgumentException kivételt!

Név összefűzése

Készítsünk olyan osztályt, amely egy név részelemeinek konstruktorban történő megadásával magyar vagy nyugati stílusú név összefűzést csinál, kezelve az opcionális elemek hiányát is. A névelemek a következők: családi név, köztes név, keresztnév (givenName) és titulus (Mr, Ms, Dr).

Hibakezelés:

A családi név és az adott (kereszt) név kötelező attribútumok. Hiányuk esetén (null vagy üres String) dobjon IllegalArgumentException-t!

A titulus legyen enum. Az egyik metódusban használj += operátorokat, a másikban concat() metódust. publikus metódusok:

public Name(String familyName, String middleName, String givenName, Title title)

public Name(String familyName, String middleName, String givenName) public String concatNameWesternStyle() public String concatNameHungarianStyle()

Tippek:

Vezessünk be egy isEmpty(String) metódust, amelynek visszatérési értéke true ha a paraméter String null vagy üres String!

Teszt

Melyik műveletsornak nem 12Degree az eredménye?

* ☒ 1 + 2 + "Degree"
* Integer.toString(1) + 2 + "Degree"
* "" + 1 + 2 + "Degree"
* 1 + "" + 2 + "Degree"

Főbb String metódusok (stringmethods)

A String immutable, így a módosítónak tűnő metódusai nem az adott objektum állapotán dolgoznak, hanem egy új String-et adnak vissza.

String name = "John Doe";

String upperName = name.toUpperCase(); // name értéke nem változik A metódusok meghívhatóak literálokon is.

String upperName = "John Doe".toUpperCase();

Amikor egy változó értékét akarjuk összehasonlítani egy literállal, akkor mindig a literált tegyük előre, mert ez nem lehet null (szemben a változó értékével), és ezért nem keletkezhet NullPointerException.

String s = "john doe"; boolean b = "John Doe".equalsIgnoreCase(s); A metódushívások egymás után láncolhatók.

String upperForename = "John Doe".toUpperCase().substring(0, 4);

Ebben a leckében csak a legfontosabb String metódusokat vizsgáljuk meg. A többi metódusért érdemes megnézni a Java API dokumentációt.

Tulajdonságokat lekérdező metódusok

Ezek a metódusok a String-et különböző szempontok szerint vizsgálják meg, ezért a legtöbb logikai értékkel tér vissza.

A szöveg karakterben mért hosszát adja vissza a length() metódus. Minden olyan esetben hasznos, ahol a karaktereket be szeretnénk járni, vagy indexszel el szeretnénk érni. Ezek a metódusok ugyanis nem létező karakterre való hivatkozáskor

ArrayIndexOutOfBoundsException kivételt dobnak. A charAt() metódusa például a paraméterként átadott indexű karaktert adja vissza.

String word = "Hello World"; for(int i = 0; i < word.length(); i++) { System.out.print(word.charAt(i) + ",");

} //H,e,l,l,o, ,W,o,r,l,d,

Gyakran szükség van arra, hogy megvizsgáljuk, kaptunk-e számunkra értelmes tartalmú szöveget. Az isEmpty() metódus azt vizsgálja, hogy üres-e a String. Ha nem csak az üres, hanem a csupán whitespace karaktereket tartalmazó szöveget is szeretnénk kiszűrni, használjuk a Java 11-ben megjelent isBlank() metódust. Ez akkor is igazzal tér vissza, ha a szöveg ugyan 5 karakterből áll, de egyik sem nyomtatható.

private boolean hasContent(String s) { // hasContent(" ") --> true return s != null && !s.isEmpty();

}

private boolean hasReadableContent(String s) { // hasReadableContent("

") --> false return s != null && !s.isBlank();

}

Két String egyezőségének vizsgálatára használható az equals() metódus. Amennyiben nem számítanak a kis-nagybetű különbségek, használd az equalsIgnoreCase() metódust!

String word1 = "soap";

String word2 = "SoaP";

System.out.println(word1.equals(word2)); // false

System.out.println(word1.equalsIgnoreCase(word2)); // true

A szöveg nagyon egyszerű mintákra való illeszkedését vizsgálja a startsWith(), endsWith() és a contains() metódus. Mindhárom a keresett szövegrészt kapja paraméterül, és logikai értékkel tér vissza. Csak kis-nagybetű érzékeny verziójuk létezik, de a toLowerCase() vagy a toUpperCase() metódusokkal kombinálva jól használhatóak akkor is, ha nem számítanak a kis-nagybetű különbségek.

Ezek a metódusok nem a mintát várják bemenetként, hanem csak konkrét szövegrészt!

"Hello World".startsWith("Hell") // true

"Hello World".endsWith("World") // true

"Hello World".contains(" w") // false

"Hello World".toLowerCase().contains(" w".toLowerCase()) //true

Ha nem csak az a kérdés, hogy egy szövegrész megtalálható-e egy String-ben, hanem az is, hogy hol, akkor használjuk az indexOf() metódust. Ez a paraméterül kapott szövegrész első előfordulásának kezdő indexét adja vissza. Amennyiben a paraméter nem található meg a szövegben, a visszatérési érték -1.

int firstO = "Hello World".indexOf("o") // 4 int firstOne = "Hello World".indexOf("one") // -1

Ha nem a legelső karaktertől kezdve szeretnénk keresni, akkor második paraméterként az indulási indexet is megadhatjuk.

int secondO = "Hello World".indexOf("o", 5) // 7

Szöveget gyártó metódusok

Név alapján azt gondolhatnánk, hogy ezek a metódusok módosítják a String-et, de mivel az immutable, ezért a művelet eredménye mindig egy új String, melyet a metódusok visszatérési értéke tartalmaz.

A szövegből kiemelhetünk egy rész a substring() metódussal. Bemenetként a kiemelendő rész kezdő és záró indexét várja. Ha a szöveg végére van szükségünk, a záró index elhagyható.

String part = "This is the full sentence.".substring(5, 7); // "is" String end = "It's another sentence.".substring(5); // "another sentence."

Vigyázzunk, hogy mind a nyitó, mind a záró index létező legyen, különben kivétel keletkezik.

Ha a cél az, hogy a szöveg elején és végén lévő láthatatlan karaktereket levágjuk, akkor szerencsére nem kell nekünk megkeresni, hogy mely része kell a szövegnek, a trim() metódus megteszi ezt helyettünk. A metódus nem foglalkozik a szöveg közepén lévő többszörös szóközökkel vagy sorvége jelekkel, kizárólag a String elején és végén lévőkkel.

String content = " \t readable characters \n".trim(); // "readable characters"

A szöveg egy részének cseréjére alkalmazhatjuk a replace() metódust. Ezzel az első paraméterként átadott szövegrész minden előfordulását lecserélhetjük a második paraméterként átadottra.

String resultString = "apple pear plum".replace(" p", " g"); // "apple gear glum"

A Java 11 óta ha ugyanazt a szöveget szeretnénk sokszor összefűzni, akkor nem kell a concat() metódust meghívnunk sokszor, hanem elég a repeat() metódust egyszer. Paraméterként az ismétlések számát várja. Ha a paraméter értéke 0, akkor üres szöveget ad vissza, de ha negatív, akkor kivételt dob.

String empty = "ho".repeat(0); // ""

String santa = "ho".repeat(3); // "hohoho"

String oops = "ho".repeat(-2); // IllegalArgumentException

Sokszor egyetlen szövegként kapunk meg több adatot, és ilyenkor az egyes részeket valamilyen elválasztó karakter vagy karaktersorozat szeparálja el egymástól. A split() metódus a kapott elválasztó karaktersorozat mentén szétvágja a szöveget és a keletkezett darabokat tömbként adja vissza. Bemenetként nem csak konkrét, hanem reguláris kifejezést tartalmazó szöveget is elfogad, így akár több karakter mentén is vághatunk. Ha paraméterként üres szöveget adunk át, akkor minden egyes karaktert külön szövegbe tesz.

String[] words = "apple pear plum".split(" "); // ["apple", "pear", "plum"]

String[] characters = "apple".split(""); // ["a", "p", "p", "l", "e"]

A Java 11 óta nem csak szétvágni, de összerakni is tudunk részekből szöveget. A join() metódus első paraméterként a használandó elválasztó karaktert várja, utána pedig sorban az összefűzendő String-eket. Az összefűzendő részeket akár tömbként vagy listaként is odaadhatjuk. Ezt a metódust nem egy String objektumon, hanem magán az osztályon kell meghívni.

String message = String.join("-", "Java", "is", "cool"); // "Java-is-cool"

List<String> words = List.of("Java", "is", "cool");

String message2 = String.join("-", words); // "Java-is-cool"

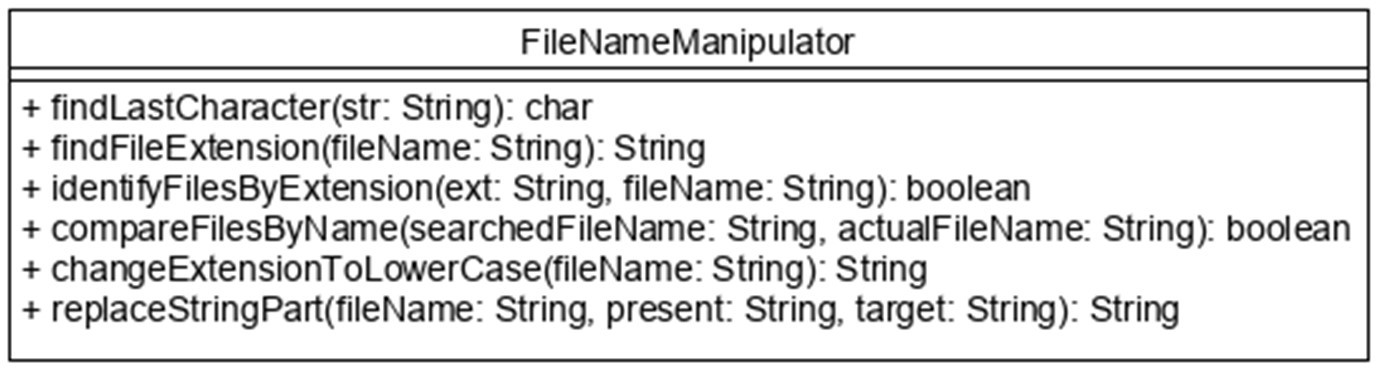
Ellenőrző kérdések

* A length() metódus mit ad vissza a következő string esetében: " a p p l e" ?
* A charAt() metódus mit ad vissza: "index".charAt(2) hívás esetén?
* Az indexOf() metódus mit ad vissza "index".indexOf('x') hívás esetén?
* A substring() metódus hogyan értelmezi a paraméterként átadott indexeket?
* Az equals() és equalsIgnoreCase() metódusoknak mi a jelentősége?
* A contains() metódusnak mi a visszatérési értéke?
* A replace() metódus char vagy CharSequence paramétereket fogad. Mit jelent a CharSequence?
* A trim() metódus mit eredményez a következő string esetében: “ apple” ?

Feladat

Fájlnevek kezelése

Készítsünk egy FileNameManipulator osztályt, amely fájlnevek ellenőrzésére, illetve ehhez kapcsolódó String műveletekre alkalmas metódusokat tartalmaz.



FileNameManipulator UML

Hibakezelés

Az egyes funkcióknál a feldolgozhatatlan paraméterek és paraméter kombinációk esetén dobjon IllegalArgumentException-t!

Tippek

Ha igény van rá, alkalmazzuk a metódusok láncolását! Figyeljünk a vezető és követő whitespace karakterekre! URL feldolgozás

https://earthquake.usgs.gov/fdsnws/event/1/query?format=geojson&starttime=201401-01&endtime=2014-01-02

Mi mindent tudhatunk meg ebből az URL-ből?

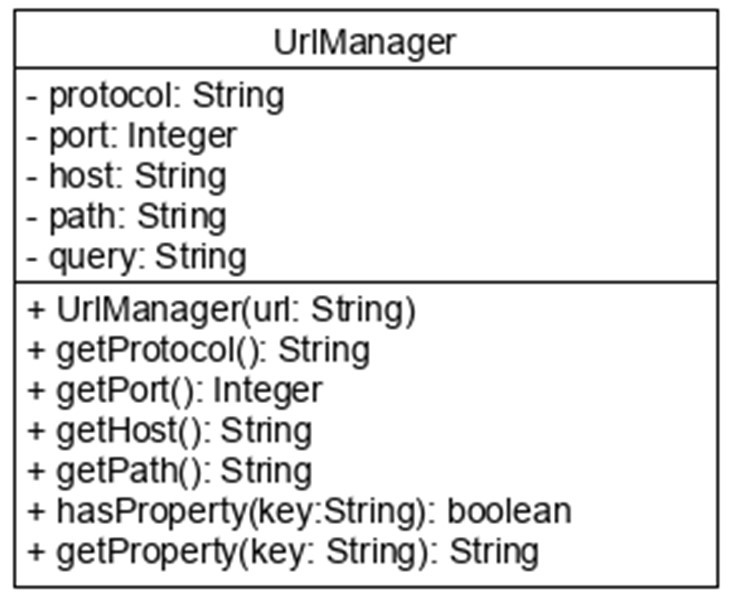
Egy URL általános alakja:

protocol://host:port/path?query-string, ahol a path több / jellel elválasztott részből állhat, a query-string pedig key=value párok (property) & jellel elválasztott sorozata. Port megadása csak akkor kötelező, ha nem a protokoll által alapértelmezetten használt porton történik a kommunikáció.

A fenti URL részei eszerint:

* protocol: https (kötelező)
* host: earthquake.usgs.gov (kötelező)
* port: (nincs megadva)
* path: fdsnws/event/1/query • query-stringként átadott adatok: – format = geojson
  + starttime = 2014-01-01
  + endtime = 2014-01-02

Készíts egy URL feldolgozót, mely a teljes URL-t konstruktorban kapja meg, és képes azonosítani az egyes részeit! A részek leválasztásához készíts privát segédmetódusokat! A protocol és a host szabvány szerint nem kis-nagybetű érzékenyek, ezért ezeket mindig csupa kisbetűvel tárold, míg a többi részt úgy, ahogy van. A path és a query legyen üres String, ha nincsenek az URL-ben.



UrlManager UML

A getProperty() metódus a property értékét adja vissza, amennyiben megtalálható a kapott key a query attribútumban kulcsként.

Hibakezelés

A paraméterként kapott szöveget mindig validáld! Ha nincsenek meg az URL kötelező részei, dobj IllegalArgumentException kivételt! Amennyiben a hasProperty() vagy a getProperty() metódus null-t vagy csak whitespace-eket tartalmazó String-et kap, szintén IllegalArgumentException kivétellel jelezd!

Teszt

Mit ír ki az alábbi kódrészlet?

System.out.print(" Hexo World ".trim().replace("x", "ll").substring(2, 10));

* ☒ “llo Wo”
* “lo Wor”
* “llo Worl”
* “llo World”

StringBuilder (stringbuilder)

A legfontosabb, hogy míg a String immutable, addig a StringBuilder osztály módosítható. Ha egy szövegen nagyon sok műveletet szeretnénk végrehajtani, akkor használjunk StringBuilder-t.

Konstruktorral hozható létre. A konstruktor lehet üres (ekkor alapértelmezetten egy üres karakterláncot ábrázol), megadhatunk egy String-et, akkor az lesz az értéke. Megadhatunk neki capacity-t is, ugyanis a háttérben egy karaktertömb áll, aminek meg lehet adni a méretét. Hasznos, hiszen egyből akkora tömb jön létre, amekkorát szeretnénk.

A StringBuilder-en kívűl használható még az úgynevezett StringBuffer osztály is, ez szálbiztos.

Főbb StringBuilder metódusok

* append() – újabb érték hozzáfűzése
* toString() – karakterlánccá konvertálás
* charAt(), indexOf(), length(), substring() – ugyanúgy működnek, mint a String-ben
* insert() – új karakterláncot illeszt be
* delete() / deleteCharAt() – szövegrészt/karaktert töröl
* reverse() – karakterek sorrendjét megfordítja

Ellenőrző kérdések

* Mi a lényeges különbség a StringBuilder append() metódusa és a String concat() metódusa között?
* Mi a különbség a StringBuilder és a StringBuffer között?
* Melyek a StringBuilder főbb metódusai?
* Hogyan konvertálható String objektum StringBuilder objektumba és fordítva?

Gyakorlati feladat - Név összefűzés

Készítsünk olyan osztályt, amely egy név részelemeinek megadásával magyar vagy nyugati stílusú név összefűzést csinál, kezelve az opcionális elemek hiányát is. A névelemek a következők: családi név, köztes név, keresztnév (givenName) és titulus (Mr, Ms, Dr, Prof). A neveket tovább lehet módosítani, lehet beszúrni például titulust, törölni belőle részeket.

Hibakezelés

A családi név és az adott (kereszt) név kötelező paraméterek. Hiányuk esetén (null vagy üres String) dobjon IllegalArgumentException-t.

Megvalósítás

A titulus legyen enum. publikus metódusok:

public String concatNameWesternStyle(String familyName, String middleName, String givenName, Title title)

public String concatNameHungarianStyle(String familyName, String middleName, String givenName, Title title)

public String insertTitle(String name, Title title, String where) public String deleteNamePart(String name, String delete)

Tippek

Vezessünk be egy isEmpty(String) metódust, amelynek visszatérési értéke true ha a paraméter String null vagy üres String!

Gyakorlati feladat - Palindróma

Készítsünk olyan osztályt, amelynek metódusa egy szóról, szövegrészletről el tudja dönteni, hogy az palindróm (visszafelé is ugyanaz).

Hibakezelés

A metódus null paraméter esetén dobjon IllegalArgumentException-t.

Megvalósítás publikus metódusok:

public boolean isPalindrome(String word)

Tippek

Eltérő case-t (kisbetű, nagybetű) ne vegye figyelembe!

Scanner (stringscanner)

A Scanner osztályt arra használjuk, hogy különböző primitív vagy String tipusú értékeket olvassunk be különböző forrásokból, pl.String, fájl, adatfolyam. A konstruktorban megadhatjuk a forrást, jól használható például akkor is, ha billentyűzetről szeretnénk beolvasni:

Scanner scanner = new Scanner (System.in);

Metódusok

* nextXXX() – különböző típusokhoz, eltérő, de mindig next prefixű metódusok beolvasásra

int number = scanner.nextInt(); //Csak szám beolvasására jó!

* hasNextXXX() – boolean visszatérési értékű, megadja, hogy a következő érték adott típusú-e.
* next() / hasNext() - a következő elválasztó karakterig adja vissza a szöveget. Az elválasztó karakter alapértelmezetten whitespace (Charater.isWhiteSpace()). Az elválasztó karakter a useDelimiter() metódussal változtatható meg.
* nextLine() / hasNextLine() – akkor használjuk, ha egy szöveget soronként szeretnénk feldolgozni.

Ellenőrző kérdések

* Mi a Scanner szerepe String feldolgozásoknál?
* Mi a delimiter szerepe, mi az alapértelmezett delimiter?
* Hogyan lehet az alapértelmezett delimitert visszaállítani?
* Hogyan lehet számokat beolvasni Scanner segítségével?

Gyakorlati feladat - String beolvasás

Készítsünk egy StringScanner osztályt. Ennek metódusaival számok olvashatók be és összegezhetők a delimiter megadásával vagy anélkül, illetve többsoros szövegből kiszűrhetők azok a sorok, amelyek adott szót tartalmaznak.

Hibakezelés

Az egyes funkcióknál a feldolgozhatatlan paraméterek és paraméter kombinációk esetén dobjon IllegalArgumentException-t Megvalósítás publikus metódusok:

public int readAndSumValues(String intString, String delimiter) public int readAndSumValues(String intString) public String filterLinesWithWordOccurrences(String text, String word) Tippek

Vezessünk be egy isEmpty(String) metódust, amelynek visszatérési értéke true ha a paraméter String null vagy üres String! A túlterheléses metódusok esetén lehetőség van egymás hívására!

Fájl olvasása Scannerrel (filescanner)

A Scanner osztály nem csak konzolról való olvasást tesz lehetővé, hanem szöveges fájlok feldolgozására is alkalmas. Ebben az esetben a Scanner konstruktorának a fájlt kell átadnunk, melyet a Path objektum reprezentál. Ezt létre tudjuk hozni a fájl elérési útvonalával:

Scanner scanner = new Scanner(Path.of("books.txt"));

A fájl megnyitása közben több hiba is történhet, melyet kezelnünk kell. Például nem létezik a fájl, nincs jogosultságunk olvasni. Azaz a Scanner példányosítása IOException kivételt dobhat, melyet valahogy kezelnünk kell. A fájlokat és a Scanner példányt is használat után le kell zárni a close() metódussal. A Java képes ezeket automatikusan lezárni, amennyiben a kivételeket try-with-resources szerkezettel kezeljük, és a lezárandó objektumokat ennek a fejrészében példányosítjuk.

try (Scanner scanner = new Scanner(Path.of("books.txt"))) {

while(scanner.hasNextLine()) { String line = scanner.nextLine();

System.out.println(line);

}

}

catch (IOException ioe) {

throw new IllegalStateException("Cannot read file", ioe);

}

A try blokk végén a Scanner példányon akkor is meghívódik a close() metódus, ha minden rendben lefutott, és akkor is, ha hiba történt. A Scanner lezárásával automatikusan lezáródik a megnyitott fájl is.

A fájl útvonalát mind relatív, mind abszolút elérési úttal megadhatjuk. Ha a fájl az alkalmazás részét képezi, akkor Maven használata esetén azt az erőforrásoknak fenntartott srckönyvtáron belül kell elhelyeznünk, és Path helyett a Class getResourceAsStream() metódusával tudjuk a fájlt megnyitni. Az itt elhelyezett fájlok ún. gyökérútvonallal is megadhatók, melyet mindig / jellel kezdünk. A /books.txt azt jelenti, hogy a projekt classpath gyökerében található books.txt fájl. Relatív útvonal esetén a viszonyítási pont az osztályt tartalmazó csomag lesz. Ha a FileScannerMain osztályunk a training package-ben van, akkor a “books.txt” útvonal az src.txt fájlra mutat.

try (Scanner scanner = new

Scanner(FileScannerMain.class.getResourceAsStream("/books.txt"))) { while(scanner.hasNextLine()) { String line = scanner.nextLine();

System.out.println(line);

}

}

A getResourceAsStream() metódust az aktuális osztály neve + .class előtaggal kell hívni. Az útvonal megadása ebben az esetben rendszerfüggetlen, azaz a könyvtárak elválasztására mindig / jelet használunk. A metódus nem dob IOException kivételt, ezért a catch ág ebben az esetben nem kell, de a Scanner objektum automatikus lezárásához még mindig try-with-resources szerkezetben példányosítjuk.

Ellenőrző kérdések

* Hogyan lehet szöveges fájlok olvasásához példányosítani a Scanner osztályt?
* Milyen kivétel keletkezhet, ha a fájl eléréséhez a Path.of() metódust használjuk, és ezt hogyan kezeljük?
* Hová kell elhelyezni a fájlt, ha az az alkalmazásunk része Maven használata esetén? Hogyan tudjuk ebben az esetben megnyitni?
* Hogyan zárjuk le a megnyitott fájlt olvasás után?

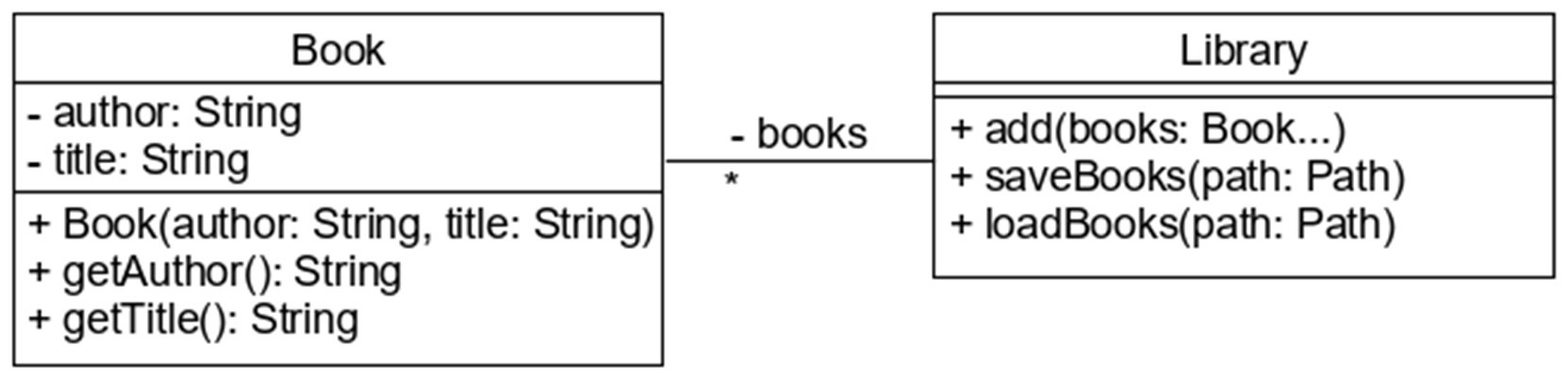
Feladat

Bakancslista

A bakancslistánkat egy szöveges fájlba írtuk, minden pontot külön-külön sorba. A bucketlist.txt a projekt gyökérkönyvtárában található. Listázd ki a fájl tartalmát a képernyőre a BucketList osztály main() metódusában! Amennyiben hiba történik a fájl megnyitásakor vagy olvasása közben, jelezd a hibát a felhasználónak!

Könyvtár

Egy könyvtár nyilvántartása a könyvek leltári számát, szerzőjét, címét és a kiadás évét tartalmazza. Készíts egy Book osztályt, mely konstruktorában megkapja ezeket az adatokat! Minden adata lekérdezhető, de egyik sem módosítható. A Library osztály attribútumként egy List<Book>-ot tartalmaz. A loadFromFile() metódusa a classpathon található books.csv fájlból tölti be a könyvek adatait. A fájl minden sora egy könyv adatait tartalmazza pontosvesszővel elválasztva.



UML osztály diagram

Tipp: Könnyebb a fájl olvasása, ha mind a ;-t, mind a sorvége karaktert beállítod a Scanner elválasztójaként. Ehhez használd delimiterként a ";|(\r\n)" kifejezést! printf (formatlocaleprintf)

A C/C++ nyelv egy igen széles körben elterjedt funkciója mely paraméteres szövegek kiírására jól használható. Ez a metódus Java-ban a PrintWriter osztályban található. Nézünk egy példát.:

System.out.printf("The result is %d", 500);

A %d az egész számot jelöli. A paraméterek varargs formátumban adhatók meg.

Konverziós karakterek

Nézzünk még egy példát:

System.out.printf(%8.2f, 1000.0/3.0);

Ez azt, jelenti, hogy 8 karakteren írunk ki, két tizedesjegy pontossággal egy lebegőpontos számot. További placeholder karakterek:

* %s-szöveg
* %d-egész szám
* %f-lebegőpontos szám

Valójában Java-ban a Formatter osztály implementálja ezeket a műveleteket.

Dátumok használata

Dátum placeholder: %tc . Itt is az oprendszer alap locale-je van használva, ha ezen változtatni szeretnénk:

System.out.printf(Locale.US, ”%tc”,new Date());

Ellenőrző kérdések

* Hogyan lehet formátum string alapján szövegeket kiírni?
* Hol használható? Milyen alternatív neve van?
* Milyen formázási karaktereket ismersz? Hogyan lehet egész számot, lebegőpontos számot, dátumot formázni?
* Alapesetben milyen locale-lal dolgozik? Hogyan lehet más locale-t megadni?
* Milyen szerkezettel valósították meg a változók átadását?
* Milyen szerepe van az autoboxingnak?

Gyakorlat - PrintFormat

Készítsünk egy PrintFormat osztályt, amelynek paraméterezhető metódusai különböző String formázásokat biztosítanak a String.format() metódus segítségével.

Hibakezelés

Amennyiben a format stringben felsorolt paraméterek száma több, mint a híváskor átadott, dobjon IllegalArgumentException-t. Megvalósítás publikus metódusok:

public String checkException(String formatString, Integer i, Integer j) public String printFormattedText(Double number) public String printFormattedText(int count, String fruit) public String printFormattedText(int number) public String printFormattedText(Integer i, Integer j, Integer k)

Megjegyzés

A hibakezelés nem általános megoldás, csupán egy tipikus kivétel bemutatását szolgálja.

## Programozási tételek és egyéb algoritmusok

Összegzés tétele (algorithmssum)

Nevezetes algoritmusok

Vannak olyan problémák amik gyakran előfordulnak a programozás során, így az ezen algoritmusoknak külön névvel látták el, programozási tételeknek vagy nevezetes algoritmusoknak hívják. Ezek általában valamilyen kollekción dolgoznak. Nézzük ezeket sorban:

* Összegzés tétele
* Számlálás tétele
* Szélsőérték keresés
* Eldöntés tétele

Összegzés tétele

Az algoritmus bemenete egy n elemű lista. Egyszerűbb esetben számokat tartalmazó lista, de akár objektumokat tartalmazó lista is lehet valamilyen szám értékkel. Ezeket a számokat illetve számértékeket akarjuk összeadni.

Elméleti megvalósítás

* Változó deklarálása
* Ciklusban iterálás
* Ha szükséges feltétel az elemre
* Ha szükséges az elem konvertálása számmá
* Összeghez hozzáadni a számot
* Összeget visszaadni

Gyakorlati megvalósítás

public int sum(List<Integer> numbers) { int sum = 0;

for (Integer n: numbers) { sum += n;

}

return sum;

}

public int ageSumCalculator(List<Trainer> trainers){ int sum = 0;

for(Trainer trainer: trainers){ sum += trainer.getAge();

}

return sum; }

Ellenőrző kérdések

* Mi a bemenete és a kimenete az összegzés algoritmusának?
* Mi legyen a kezdőértéke a majdani visszatérési értéket tároló változónak?

Értékesítők számai

Hozz létre egy Salesperson osztályt, a szükséges attribútumokkal:

* name, az értékesítő kolléga neve
* amount, az üzletkötéseiből származó árbevétel

Feladat egy metódus megírása a megfelelő osztályban, ami összegzi a cég öszes értékesítőjének árbevételét.

Összes jóváírás

Hozz létre egy Transaction osztályt, a szükséges attribútumokkal:

* accountNumber, számlaszám
* transactionOperation (TransactionOperation enum, CREDIT vagy DEBIT)
* amount, a tranzakció összege

Hozz létre egy TransactionSumCalculator osztályt, amelyben van egy int sumAmountOfCreditEntries(List<Transaction> transactions) metódus, amely összegzi a credit tranzakciók összegét.

Számlálás tétele (algorithmscount)

Az algoritmus bemenete egy n elemű lista. A feladat az, hogy számoljuk meg azokat az elemeket amelyekre igaz egy feltétel. Például számoljuk meg a 15-nél nagyobb számokat egy listában.

Elméleti megvalósítás

* Változó deklarálása számlálónak
* Ciklusban iterálás
* Feltétel teljesülése esetén számláló növelése
* Számláló visszaadása

Gyakorlati megvalósítás

public int countLetters(String s, char c) { int count = 0;

for (int i = 0; i < s.length(); i++) { if (s.charAt(i) == c) { count++;

} }

return count;

}

public int countElderly(List<Trainer> trainers, int minAge ) { int count = 0; for (Trainer trainer: trainers) { if (trainer.getAge() >= minAge) { count++;

} }

return count; }

Ellenőrző kérdések

* Mi a bemenete és a kimenete a számlálás algoritmusának?
* Mi legyen a kezdőértéke a majdani visszatérési értéket tároló változónak?

Nagy összegű bankszámlák

Hozz létre egy BankAccount osztályt, a szükséges attribútumokkal:

* nameOfOwner, a számla tulajdonosának neve
* accountNumber, a számlaszám
* balance, egyenleg

Feladat egy metódus megírása a BankAccountConditionCounter osztályban, ami megszámlálja, hány olyan számla van, amelynek az aktuális egyenlege meghaladja a paraméterként kapott alsó határt.

Kis összegű tranzakciók

Hozz létre egy Transaction osztályt, a szükséges attribútumokkal:

* accountNumber, számlaszám
* transactionType (credit vagy debit, egy külön TransactionType enum)
* amount, a tranzakció összege

Feladat egy metódus megírása a TransactionCounter osztályban, ami megszámlálja hány olyan tranzakció van, amely a paraméterként kapott összeghatárnál kisebb értékű. Szélsőérték keresés tétele (algorithmsmax)

Az algoritmus bemenete egy n elemű lista. A feladat, hogy visszaadjuk azt az elemet, ami a legnagyobb vagy a legkisebb, az elemeknek ezért összehasonlíthatónak kell lenniük. Figyelnünk kell az egyenlőségre. Ilyen esetben általában az első vagy az utolsó elemet szokás visszaadni, de lehet, hogy az összeset.

Elméleti megvalósítás

* Változó deklarálása szélsőértéknek
* Ciklusban iterálni
* Amennyiben a ciklusváltozó nagyobb, kisebb, a szélsőértéket le kell cserélni a ciklusváltozó értékére
* Szélsőérték visszaadása

Gyakorlati megvalósítás

public int max(List<Integer> numbers) { int max = Integer.MIN\_VALUE; for (Integer n: numbers) { if (n > max) { max = n;

} } return max;

}

public Trainer trainerWithMaxAge(List<Trainer> trainers) { Trainer trainerWithMaxAge = null; for (Trainer trainer: trainers) {

if (trainerWithMaxAge == null || trainer.getAge() > trainerWithMaxAge.getAge()) {

trainerWithMaxAge = trainer;

} } return trainerWithMaxAge;

}

Ellenőrző kérdések

* Mi a bemenete és a kimenete a szélsőérték kiválasztás algoritmusának?
* Mi legyen a kezdőértéke a majdani visszatérési értéket tároló változónak?

Legjobb értékesítő

Hozz létre egy Salesperson osztályt, a szükséges attribútumokkal:

* name, az értékesítő kolléga neve
* amount, az üzletkötéseiből származó árbevétel
* target, a cél árbevétel, amit az adott étékesítő számára előírt az értékesítési igazgató

Feladat a következő metódusok megírása a megfelelő osztályokban:

* kiválasztja a legnagyobb árbevételt elért értékesítőt
* kiválasztja azt az értékesítőt, aki a célt a legnagyobb összeggel meghaladta
* kiválasztja azt az értékesítőt, aki a legnagyobb összeggel alulmúlta a célt

Legidősebb trainer

Hozz létre egy Trainer osztályt a következő attribútumokkal:

* name, a trainer neve
* age , az életkora A feladat:
* Egy MaxAgeCalculator osztályban hozz létre egy Trainer trainerWithMaxAge(List<Trainer> trainers) metódust, amely kikeresi a legidősebb trainert.

Legnagyobb szám

Hozz létre egy IntegerMaxCalculator osztályt, valamint benne egy metódust, ami a kapott egész számok listájából kiválasztja a legnagyobbat.

Eldöntés tétele (algorithmsdecision)

Az algoritmus bemenet egy n elemű lista. A feladat az, hogy döntsük el, hogy van-e olyan elem a listában, amelyre igaz egy feltétel.

Elméleti megvalósítás

* Változó deklarációja a találat tényének (hamis kezdőértékkel)
* Ciklusban iterálni addig, amíg van elem, és nincs találat
* Feltétel teljesülésének esetén a találat tényét igazra állítani, és kilépni a ciklusból
* Találat tényének visszaadása

Gyakorlati megvalósítás

public boolean containsGreaterThan(List<Integer> numbers, int min) { for (Integer i : numbers) { if (i > min) { return true;

} }

return false;

}

public boolean containsLowerThan(List<Integer> numbers, int max) { boolean contains = false; int i = 0;

while (i < numbers.size() && !contains) { if (numbers.get(i) < max) { contains = true;

} i++; } return contains; }

Ellenőrző kérdések

* Mi a bemenete és a kimenete az eldöntés algoritmusának?
* Mi legyen a kezdőértéke a majdani visszatérési értéket tároló változónak?
* Meddig iteráljunk a ciklusban?

Nagy összegű bankszámlák

Hozz létre egy BankAccount osztályt, a szükséges attribútumokkal:

* nameOfOwner, a számla tulajdonosának neve
* accountNumber, a számlaszám
* balance, egyenleg

Legyen az osztálynak withdraw és deposit metódusa paraméterként kapott összeg levételére ill. betételére a számlára.

Feladat egy metódus megírása, ami eldönti van-e olyan számla, amelynek az aktuális egyenlege meghaladja a paraméterként kapott alsó határt.

Nagy összegű terhelés

Hozz létre egy Transaction osztályt, a szükséges attribútumokkal:

* accountNumber, számlaszám
* transactionOperation TransactionOperation enum, a tranzakció típusa
* amount, a tranzakció összege
* dateOfTransaction, a tranzakció dátuma
* status, a tranzakció státusza (CREATED, SUCCEEDED, PENDING)

Feladat egy metódus megírása, ami eldönti van-e olyan terhelés (debit) tranzakció egy adott dátum intervallumon belül, amely egy adott összeghatárnál nagyobb.

Tipp

A dateOfTransaction attribútum típusa legyen LocalDateTime típusú, mely tárolja a dátumot és az időt. Ennek van egy isAfter() és isBefore() metódusa, mellyel eldönthető, hogy a paraméterként átadott másik dátum előtte vagy utána van-e.

Pl.:

DateTimeFormatter formatter = DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm");

LocalDateTime today = LocalDateTime.parse("2017-03-08 10:00", formatter); LocalDateTime tomorrow = LocalDateTime.parse("2017-03-09 10:00", formatter);

assertThat(today.isBefore(tomorrow), is(true));

Tranzakciók átvezetése a számlákra

Készíts egy metódust, amelyik megkap egy tranzakció listát és egy számlák listát paraméterként és végrehajtja az összes tranzakciót, azaz minden tranzakcióhoz megkeresi az érintett számlát és ha megtalálja, akkor a creditOrDebit értéke alapján a számla megfelelő metódusának (withdraw vagy deposit) meghívásával módosítja a számla egyenlegét. Sikeres végrehajtás esetén a tranzakció státuszát állítsd SUCCEEDEDre, különben tedd PENDING-re. Feltételezzük természetesen, hogy a számlaszámok egyediek a számlák listában.

Megjegyzés

Ahol nincs külön megadva osztálynév, ott a tesztesetek alapján meghatározható osztályokba dolgozz. Célszerű ezeket az elején létrehozni, és utána megírni a metódusokat.

Rekurzió (recursion)

Az önmagát hívó metódusokat rekurziónak nevezzük. A metódus hívhatja magát közvetlenül, vagy akár más metódusokon keresztül is. Minden rekurzió ciklussá formálható. Vigyázzunk, mert könnyen implementálhatunk végtelen rekurziót.

Rekurzív feladat

Rekurziót főleg akkor alkalmazunk, mikor egy feladat visszavezethető egy hasonló, egyszerűbb esetre. Létezik legegyszerűbb eset, melyben a megoldás már magától értetődő. Létezik, egy olyan egyszerűsítési folyamat, melyet alkalmazva véges sok lépésben eljutunk a legegyszerűbb esethez. Minden lépésben feltételezzük, hogy a következő egyszerűbb esetnek már van megoldása.

Rekurzió részei

A rekurzió tartalmaz egy állapotot mely elérhet egy küszöböt, egy utasítást mely az állapotot a küszöb felé viszi, illetve egy leállító feltétel, amely azt vizsgálja, hogy az állapot elérte-e a küszöböt.

Leggyakoribb példa a rekurzióra, a faktoriális számítás:

public long getFactorial(int n) { if(n > 1) {

long solution = getFactorial(n - 1); return n \* solution;

} else { return 1;

}

}

Ellenőrző kérdések

* Mit jelent a rekurzió?
* Hogyan biztosítható, hogy véges lépésben befejeződjön a rekurzív algoritmus?

Faktoriális számítás rekurzívan

A matematikában egy n nemnegatív egész szám faktoriálisának az n-nél kisebb vagy egyenlő pozitív egész számok szorzatát nevezzük. n! = n \* (n-1) \* … \* 2 \* 1 Írd meg a faktoriális számítás algoritmusát rekurzívan.

# Objektumorientáltság

## Attribútumok és metódusok

Immutable objektumok (immutable)

Az immutable objektumok állandó állapotúak, azaz létrehozásuk után az állapotuk már nem módosítható. Az attribútumok csak a deklarációnál vagy konstruktorban kaphatnak értéket, később már nem. Ezt úgy biztosíthatjuk, hogy az attribútumokat a final kulcsszóval látjuk el, és nem írunk az osztályba setter metódusokat. Amennyiben az attribútum referenciát tartalmaz más objektumokra, akkor annak az objektumnak az állapota még módosítható marad a getter metóduson át is. Ezt úgy akadályozhatjuk meg, hogy vagy ez az objektum is immutable, vagy nem a tárolt referenciát, hanem egy másolatot adunk vissza a getterrel. (Jó gyakorlat, ha eleve egy másolatot tárolunk el a konstruktorban is.)

import java.util.ArrayList;

public class Trainer {

private final String name; private final List<String> courses;

public Trainer(String name, List<String> courses) { this.name = name;

this.courses = new ArrayList<>(courses);

}

public String getName() { return name;

}

public List<String> getCourses() { return List.copyOf(courses);

}

}

Már találkoztál immutable objektummal, hiszen a String is ilyen. A String metódusai sosem módosítják a benne eltárolt szöveget, hanem mindig egy új példányt adnak vissza, amely a művelet eredményét tartalmazza.

Ellenőrző kérdések

* Miért szerencsés az immutable objektumok használata?
* Mitől lesz egy osztály immutable?
* Milyen buktatók lehetnek immutable objektumok használatakor?

Space agency

Valamikor a távoli jövőben…

A Naprendszert számos űreszköz járja, ezeket különböző szervezetek irányítják, megadják az aktuális céljukat. Az esetleges zavaró hatások miatt a SpaceAgency nyilvántartja ezeket, regisztrációs számuk és a kitűzött céljuk ismeretével. A célt mindig koordinátákkal adjuk meg, ez a Naphoz, mint origohoz rögzített, és egyes kitüntetett csillagok irányában felvett x, y és z irányokban vannak meghatározva. Az úticél módosítása is koordinátákban kerül megadásra, mindig a már megadott célponthoz képest a különbség kerül átadásra. Az űreszköz Satellite navigációs rendszere ennek alapján automatikusan irányítja önmagát.

Megvalósítás

SpaceAgency osztály és feladatai: Regisztrálni lehet az útjukra indított eszközöket és azonosítójuk alapján ki is lehet keresni.

public void registerSatellite(Satellite satellite) public Satellite findSatelliteByRegisterIdent(String registerIdent)

Satellite osztály és feladatai: A CelestialCoordinates immutable, azaz állapota nem módosítható. Az aktuális úticél új különbözeti koordináták megadásával frissíthető, ekkor a Satellite attribútuma új értéket vesz fel.

public Satellite(CelestialCoordinates destinationCoordinates, String registerIdent)

public void modifyDestination(CelestialCoordinates diff) public String toString()

CelestialCoordinates osztály attribútumként a koordinátákat tartalmazza, ezek mind final változók, értéket a konstruktorban kapnak. Getter metódusok segítségével olvashatók.

public String toString()

Hibakezelés

Üres String, mint paraméter nem fogadható el, továbbá a szatellit regisztrációnál nem kaphat null paramétert. Amennyiben a megadott azonosítóval nem található űreszköz, szintén kivételt várunk.

Tippek

Az üres String paraméter vizsgálatára célszerű külön privát metódust írni.

private boolean isEmpty(String str)

JavaBeans objektumok (javabeans)

A Java Bean olyan speciális osztály, amelynek készítésekor be kell tartanunk néhány konvenciót, hogy az általunk írt osztály könnyen beilleszthető legyen egy már kész rendszerbe.

Szabályok:

* Minden attribútum privát, csak publikus getter és setter metódusokon át érhető el, illetve módosítható. (property = attribútum + getter és setter metódusa)
* A getter metódusok a “get” + attribútumnév nagy kezdőbetűvel elnevezési konvenciót követik. Ez alól kivétel a boolean típusú attribútum, mert ez “is” előtagot kap.
* A setter metódusok a “set” + attribútumnév nagy kezdőbetűvel elnevezési konvenciót követik.
* Rendelkeznek üres konstruktorral.

public class Pet { private String name; private String color; private int age; private boolean purebred;

public Pet() {

}

public String getName() { return name;

}

public String getColor() { return color;

}

public int getAge() { return age;

}

public boolean isPurebred() { return purebred;

}

public String setName(String name) { this.name = name;

}

public void setColor(String color) { this.color = color;

}

public void setAge(int age) { this.age = age;

}

public void setPurebred(boolean purebred) { this.purebred = purebred;

}

}

Ellenőrző kérdések

* Mire való a JavaBeans szabvány?
* Hogyan nevezzük el az életkor (age) nevű privát attribútum lekérdező metódusát (int típus)?
* Hogyan nevezzük el az életkor (age) nevű privát attribútum beállító metódusát (int típus)?
* Hogyan nevezzük el az érvényes (valid) nevű privát attribútum lekérdező metódusát (boolean típus)?

Kutya osztály

Hozz létre egy kutya (Dog) osztályt, amely a következő attribútumokat tartalmazza:

* name: szöveges típusú
* age: egész típusú
* pedigree: logikai típusú (igaz, ha fajtiszta)
* weight: valós típusú

Generálj minden attribútumhoz gettert és settert.

Ember osztály

Hozz létre egy ember (Human) osztályt, amely a következő attribútumokat tartalmazza:

* name: szöveges típusú
* weight: valós típusú
* iq: egész típusú

Generálj minden attribútumhoz gettert és settert!

Forrás

OCA - Chapter 4/Encapsulating Data

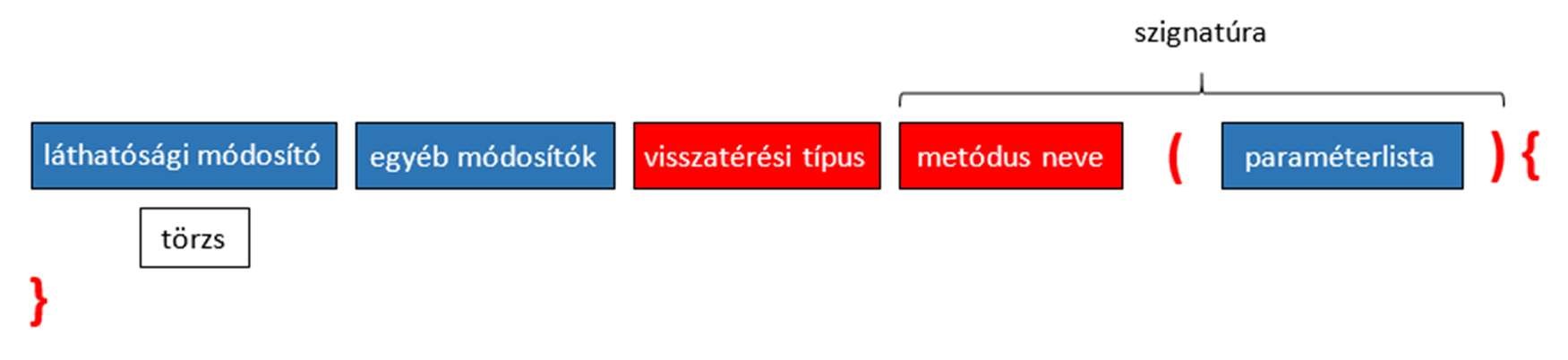
Metódusok (methodstructure)

A Java metódusok két fő részből állnak: fej és törzs.

A fej tartalmazza a láthatósági és egyéb módosítókat, a visszatérési érték típusát, a metódus nevét, a formális paraméterlistát és a metódus által dobható kivételeket. Amennyiben nincs visszatérési érték, akkor ezt a void kulcsszóval jelezzük. A metódus törzse tartalmazza az utasításokat. Ezek mindig a {} jelek közé kerülnek.

Láthatósági módosító a public, protected és private, mely meghatározza, hogy mely más osztályok férhetnek hozzá a metódushoz. A metódus neve lehetőleg ige legyen, és ha több szóból áll, akkor az elsőt kivéve minden szót nagy kezdőbetűvel írunk (camelCase).

A formális paraméterlistában a paraméterek vesszővel elválasztva szerepelnek. Minden paraméternek van típusa és neve, de itt is megadhatunk módosítókat, mint például a final kulcsszó. Híváskor fontos a formális paraméterek száma, típusa és sorrendje, hiszen ez alapján köti össze a fordító a konkrét értéket a paraméter változóval. Ez alól van egy kivétel, a varargs, amelynél sok ugyanolyan típusú érték is átadható a metódusnak, és ezeket tömbként érhetjük el a metódus belsejéből. Mivel előre nem tudhatjuk, mennyi aktuális paraméter tartozik ebbe a tömbbe, ezért varargs kizárólag az utolsó formális paraméter lehet.



Metódus szerkezete

(A piros elemek kötelezőek, a kékek opcionálisak.)

A metódus mindig implicit megkapja az objektum referenciáját is (melyen meg lett hívva), amelyhez a this kulcsszóval férhetünk hozzá. Ezt aztán felhasználhatjuk ahhoz, hogy elérjük az objektum attribútumait, ha esetleg névütközés van valamelyik paraméterrel, illetve vissza is adhatjuk.

private List<String> names;

public void addElementsWithA(String... names) { for (String name: names){ if (name.startsWith("A")) { this.names.add(name);

}

}

}

Visszatérési érték típusát a fejben deklaráltuk, a konkrét értékét pedig a metóduson belül a return utasítás után kell megadnunk. A return utasítás azonnal kiugrik a metódusból, így utána már nem írhatunk további utasításokat (erre a fordító is figyelmeztet). Ha a metódus ad vissza valamilyen értéket, akkor minden ágon kell szerepelnie return utasításnak, de ha nem ad vissza értéket, akkor is elhelyezhetünk a törzsben üres return-t.

Ellenőrző kérdések

* Mi a metódusok felépítése?
* Mi a visszatérési típus megadás olyan metódusnál, ami nem ad vissza értéket?
* Létezik paraméter nélküli metódus? Mi értelme van?
* Hogyan próbálja meg a Java az aktuális és a formális paramétereket megfeleltetni egymásnak?
* Milyen utasítás segítségével ad vissza értéket a metódus?
* Lehet egy olyan metódusban return utasítás, melynek a visszatérési típusa void?
* Lehet-e egy metódusban több return utasítás?
* Mit jelent az implicit paraméter fogalma?

BodyMass osztály

Készíts egy BodyMass osztályt, amely testtömegindexet számol. Adatai: tömeg (kilogramban megadva), magasság (méterben megadva).

Publikus metódusai:

* Getterek (getWeight, getHeight)
* double bodyMassIndex(): visszaadja a testtömegindexet, használja a következő képletet: tömeg osztva a magasság négyzetével.
* BmiCategory body(): visszaadja a testalkatot (BmiCategory legyen egy enum):
  + ha a bmi (bodymassindex) < 18.5, akkor BmiCategory.UNDERWEIGHT
  + ha bmi > 25, akkor BmiCategory.OVERWEIGHT
  + különben BmiCategory.NORMAL
* boolean isThinnerThan(BodyMass): igazat ad, ha a példányom bmi-je kisebb, mint a paraméterként kapott példányé

Pendrives osztály

Készíts egy Pendrives osztályt, amely metódusaival pendrive-ok közül lehet kikeresni a megfelelőt. A feladat részeként készíts egy Pendrive osztályt is.

A Pendrive tagjai:

* attribútumai: name, capacity, price. A kapacitás egész szám Gb-ben megadott érték. Az ár egész szám forintban megadott érték.
* Legyenek getterei az attribútumokra.
* Legyen String toString() metódusa, amely egy Stringbe összefűzve adja vissza a pendrive adatait.
* Legyen void risePrice(int percent) metódusa, amely megadott százalékkal megemeli a pendrive árát.
* Legyen int comparePricePerCapacity(Pendrive) metódusa, amely összehasonlítja a példányt egy paraméterként kapott másik példánnyal az ár/kapacitás alapján. Az eredmény 1 legyen, ha az aktuális példány ár/kapacitása nagyobb, az eredmény -1 legyen ha a paraméterként kapott példányé nagyobb, és az eredmény 0 legyen, ha egyformák.
* Legyen egy boolean cheaperThan(Pendrive) metódusa, amely igazat ad ha a példány ára kisebb, mint a paraméterben kapott példány ára.

A Pendrives részletei: A metódusokat úgy implementáld, hogy nem hívod a Pendrive getPrice() metódusát.

* Legyen Pendrive best(List<Pendrive>) metódusa, amely a legjobb ár/kapacitás értékű pendrive-t adja vissza, azaz amelyiknél ez a legkisebb.
* Legyen Pendrive cheapest(List<Pendrive>) metódusa, amely a legolcsóbbat adja vissza.
* Legyen void risePriceWhereCapacity(List<Pendrive>, int percent, int capacity) metódusa, amely adott százalékkal megemeli azon pendrive-ok árát, amelynek a kapacitása a megadott értékkel megegyezik.

Értékmásolás szerinti paraméterátadás (methodpass)

A Java nyelvben mindig értékmásolás szerinti paraméterátadás van: a híváskor átadott aktuális paraméter értéke (ha referencia változó, akkor ez a referencia) átmásolódik a formális paraméterbe, amely csak a metóduson belül érhető el.

public void tryChangeValues(int yearOfBirth, String name, List<String> courses) { yearOfBirth = 1970; name = "John"; courses.add("Java");

// vagy courses = new ArrayList<>();

}

Hívása:

List<String> courses = new ArrayList<>(); int originalYearOfBirth = 1980; String originalName = "Jack"; tryChangeValues(originalYearOfBirth, originalName, courses);

Visszahatás

Primitív és immutable objektumok esetén a metóduson belül tett változtatások nem hatnak vissza a hívó félre, azaz nem módosítják az eredeti változóban tárolt értéket, de már referenciák átadásakor bizony az eredeti objektumhoz fér hozzá a metódus.

Az originalYearOfBirth változó értéke 1980, az originalName változó értéke “Jack” marad. A courses változóban tárolt referencia sem változik, de a listába belekerült egy új elem.

Amennyiben a teljes listát egy újra cseréljük, szintén nincs visszahatás, hiszen magát a kapott referenciát cseréljük le. Visszahatás csak akkor lehetséges, ha a kapott referencián át elérhető objektum állapotát módosítjuk. Ilyen módosítást tapasztalhatunk az Arrays.sort() metódus esetén, amely a paraméterként kapott tömb elemeinek sorrendjét változtatja meg.

A legjobb gyakorlat, ha a paramétereken sosem módosítunk a metódusban, csak ha kifejezettképp ez a cél.

Mi történik a memóriában?

Egy metódus hívásakor a stacknek az adott metódus számára fenntartott területére a formális paraméterekbe átmásolódnak az aktuális paraméterek értékei, ami - mint láttuk - lehet konkrét érték és referencia is. A metódus ezután csak ezen értékekhez, illetve ezen referenciákkal mutatott objektumokhoz fér hozzá.

Ha adott az alábbi két osztály, tudod követni, mi történik a memóriában?

public class Main {

public static void main(String[] args){ int a = 16; int b = 43;

Person adam = new Person("Adam"); int c = doIncAndSum(a, b); System.out.println(a); //16

System.out.println(b); //43

System.out.println(c); //61

System.out.println(adam.getName()); //"Adam" changeName(adam);

System.out.println(adam.getName()); //"Peter"

}

public static int doIncAndSum(int x, int y){ x++; y++;

int result = x + y; return result;

}

public static void changeName(Person person){

person.setName("Peter");

}

}

class Person {

private String name;

public Person(String name){ this.name = name;

}

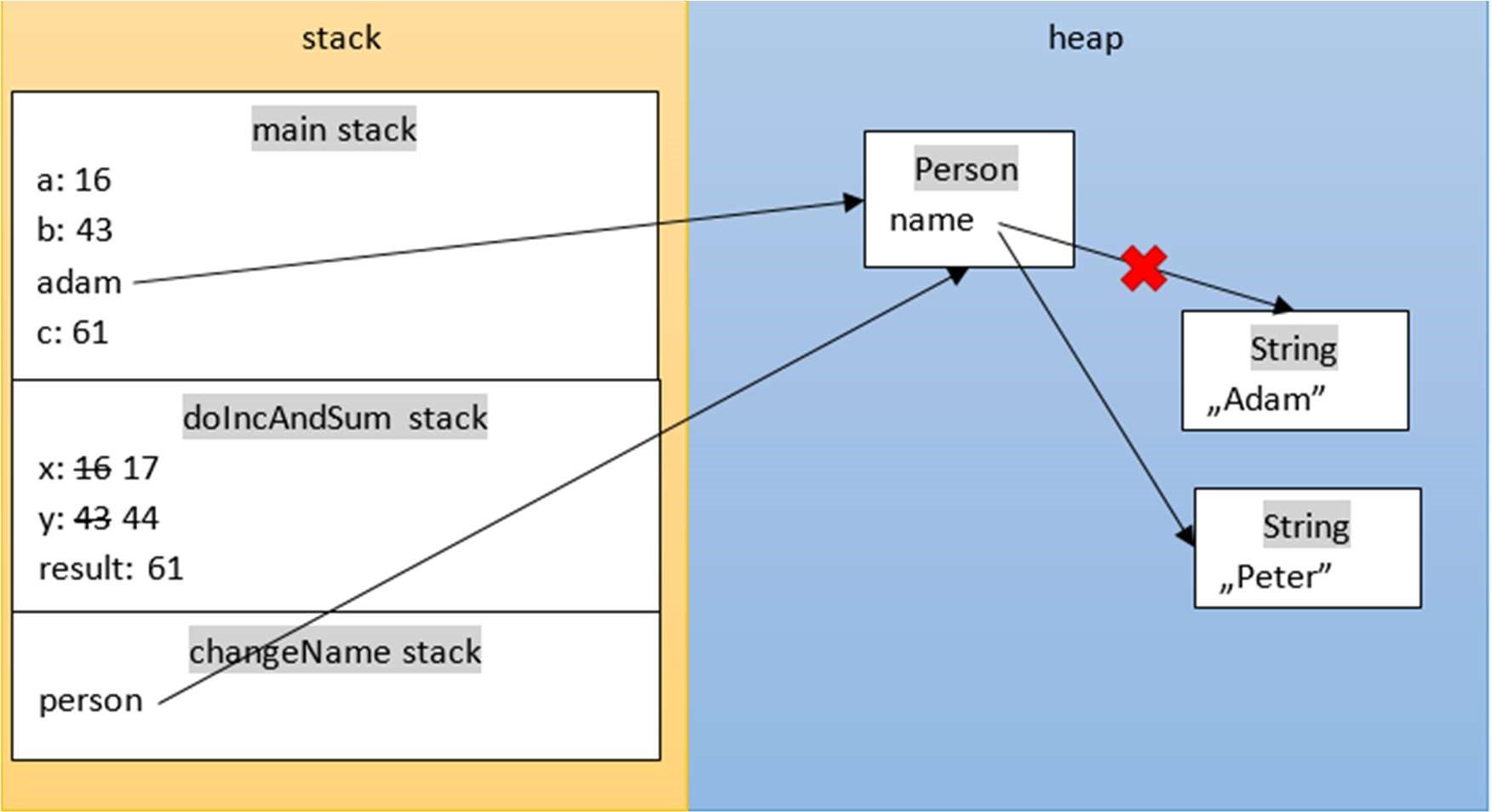
public String getName(){ return name;

}

public void setName(String name){ this.name = name;

}

}



Stack és heap paraméterátadáskor

Ellenőrző kérdések

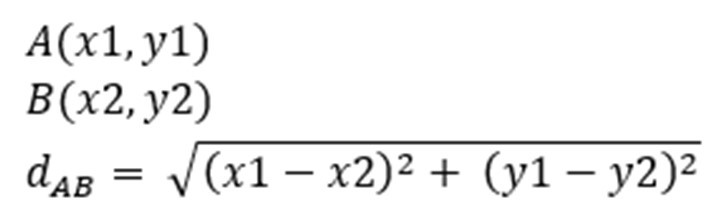
* Mi az az értékmásolás szerinti paraméterátadás?
* Javában hogyan történik a paraméterátadás? Magyarázd el primitív és osztály típusú paraméterek esetén is!
* Képes-e a metódus a paraméterként kapott objektum állapotát módosítani?

Feladat

Katonák

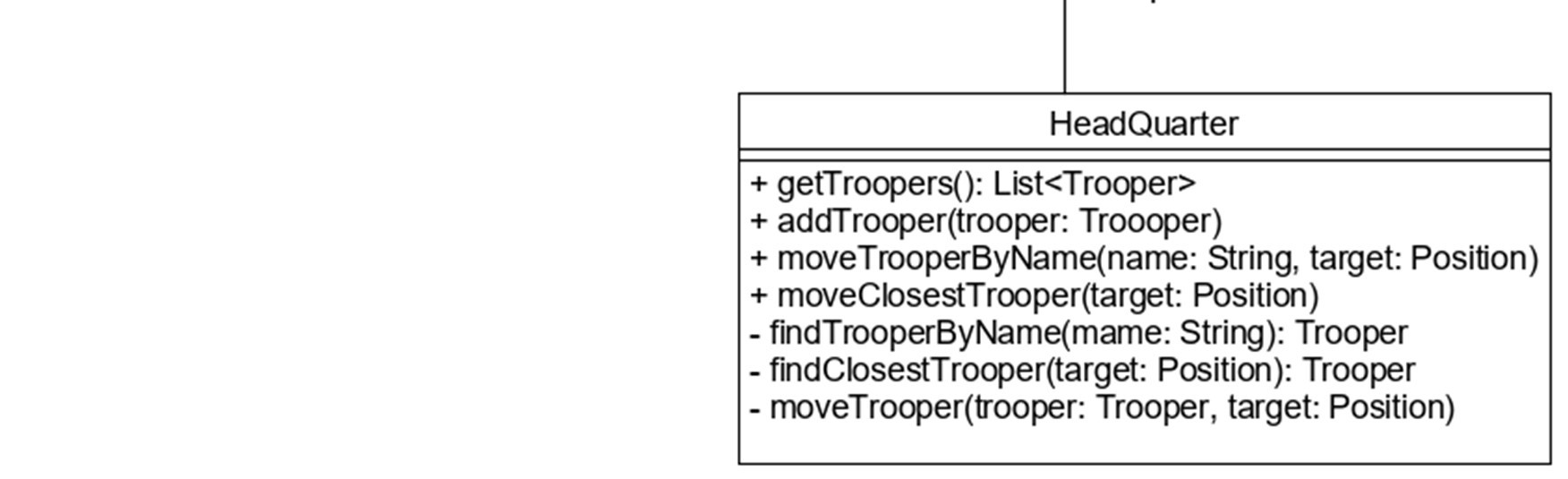
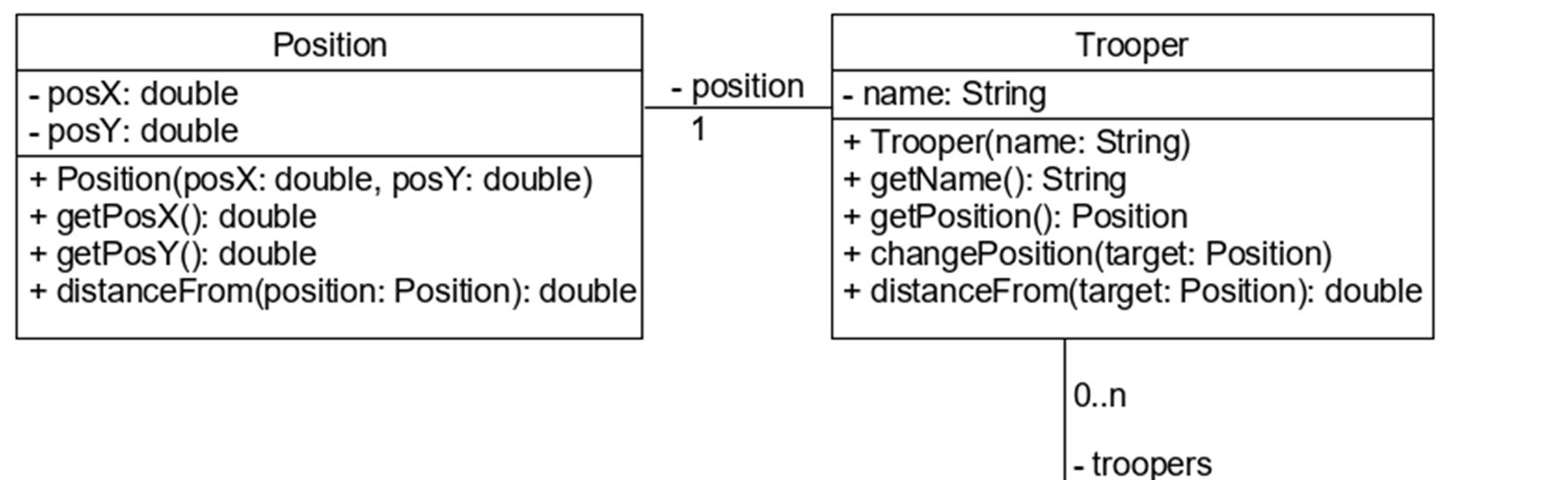
Hozz létre egy Position immutable osztályt, mely egy síkbeli pontot reprezentál! A pont távolságát egy másiktól a distanceFrom() metódusa adja vissza.

Két pont távolságát síkban az alábbi képlettel lehet kiszámítani:



Távolság számítása

A Trooper osztály egy gyalogos adatait tartalmazza. A gyalogos tudja változtatni a pozícióját, illetve meg tudja mondani, hogy egy adott ponttól milyen távol van. A HeadQuarter tartalmazza a gyalogosok listáját. Ezen osztály irányítja őket, valamint információt szolgáltat róluk. Az addTrooper() metódussal egy katonát lehet áthelyezni a panacsnokság alá. A moveClosesTrooper() a katonák közül a legközelebbit küldi a megadott pozícióra, míg a moveTrooperByName() egy adott nevű katonát. A mozgatandó katona megtalálását és áthelyezését privát segédmetódusok végzik.



Troopers UML

Minden publikus metódus ellenőrizze a paraméterként kapott adatokat. Objektum sehol sem lehet null, illetve a Trooper neve nem lehet üres. Hibás paraméter esetén a metódus dobjon IllegalArgumentException kivételt!

Mely metódusoknak van és melyeknek nincs visszahatása?

Metódus paraméterek (methodparam)

A metódus elején érdemes a paraméterek értékét ellenőrizni, mivel csak a számukat és típusukat ellenőrzi a fordító. Különösen vizsgálandó referencia változó átadásakor, hogy null-e az értéke, illetve csak bizonyos értékkészletből vehetnek fel értéket a paraméterek. Kerüljük a boolean típusú, illetve a jelzésre használt szám paramétereket, használjuk helyettük enum-ot, mert az sokkal jobban olvasható. Ha nagyon sok bemenő paraméter van, nehéz követni a sorrendjüket, akkor megfontolandó egy paraméterosztályt létrehozni, és azt átadni a metódusnak.

A metódusok csak egyetlen értéket tudnak visszaadni, ha többet szeretnénk, akkor azokat egy osztályba kell csomagolnunk, vagy esetleg több külön metódusba kell szerveznünk a feladatot. Amennyiben nincs mit visszaadnunk, például keresés esetén nincs találat, akkor a visszatérési érték lehet null, vagy jelezhetjük kivétel dobásával keresett elem hiányát. Amennyiben a metódus kollekcióval tér vissza, az null helyett inkább üres kollekció legyen.

Ellenőrző kérdések

* Milyen ajánlásokat ismersz a metódusok paraméterére vonatkozóan?
* Mit teszel, ha egy metódusnak több értéket is vissza kéne adnia?
* Milyen ajánlásokat ismersz a metódusok visszatérési értékére vonatkozóan?

Feladat

Measurement osztály

Készíts egy Measurement osztályt, amely mérési eredményeket reprezentál.

* adatai: egy valós tömb, amely a mérési eredményeket tárolja.
* a mérési adatokat kapja meg konstruktor paraméterben
* legyen int findFirstIndexInLimit(int lower, int upper) metódusa, amely visszaadja az első olyan mérési adat indexét, amely a megadott határok közé esik. Adjon vissza -1-t, ha nincs ilyen adat.
* legyen double minimum() metódusa, amely a legkisebb mérési eredményt adja vissza.
* legyen double maximum() metódusa, amely a legnagyobb mérési eredményt adja vissza.
* legyen ExtremValues minmax() metódusa, amely a legnagyobb és a legkisebb mérési ereményt adja vissza egy ExtremValues objektumban.

Az ExtremValues osztály egy egyszerű immutable adat transzfer osztály, amely min és max adatokat tárol, konstruktorral állíthatóak be, és getter metódusokkal kérhetők le.

Változó hosszúságú paraméterlista (methodvarargs)

A változó hosszú paraméterlistát a Java 5-ben vezették be. Előtte a több, ugyanolyan típusú paramétert kollekcióba téve adhattuk át a metódusnak, vagy több, ugyanolyan nevű metódust készítettünk különböző számú paraméterrel (metódus túlterhelés). A varargs használata elkerülhetővé tette ezt.

A paraméterlistában a típus után három pont (...) jelzi, hogy itt több, ugyanolyan típusú érték is megadható, akár felsorolással, akár tömbként. A metódusból a varargsként átadott paramétereket tömbként látjuk. Az ugyanolyan típusú paraméterek száma akárhány lehet, még nulla is. Ha van minimális száma a megadható paramétereknek, akkor azokat a varargs előtt külön kell deklarálnunk.

public void putStudentsIntoCourse(String... students) { for (String name: students) { course.add(name);

}

}

Hívni akár tömbbel, akár String-ek sorozatával lehet:

putStudentsIntoCourse("Emma", "Dániel", "Péter", "Ferenc", "Mariann"); vagy

String[] names = {"Emma", "Dániel", "Péter", "Ferenc", "Mariann"}; putStudentsIntoCourse(names);

VIGYÁZZ! Varargs csak egy, a legutolsó paraméter lehet. Sose használd különböző értelmű paraméterek összevonására csak azért, mert a típusuk egyezik!

Ellenőrző kérdések

* Hogyan kell deklarálni a paraméter listában a változó hosszúságú paraméter listát?
* Lehet egy metódusnak vegyes paraméter listája (fix és változó hosszúságú is)?
* Metódus hívásnál hogyan kell megadni a változó hosszúságú paramétert?

Vizsga statisztika

Készíts egy ExamStats osztályt, amely képes változó számú vizsgaeredmény (pontokban megadva) esetén “statisztikát” készíteni. Ehhez az objektum adott vizsga esetén megkapja a max pontszámot konstruktorban, majd az egyik metódusában a küszöbérték (százalék) valamint a vizsgaeredmények (pontok) felsorolásával ki tudja számolni az adott küszöbérték feleltti eredmények számát. Egy másik metódusban az alsó küszöbérték (százalékban) megadása és a vizsgaeredmény felsorolás megadásával meg tudja mondani, bukott-e valaki a vizsgán.

Publikus metódusok:

public ExamStats(int maxPoints) public int getNumberOfTopGrades(int limitInPercent, int... results) public boolean hasAnyFailed(int limitInPercent, int... results)

Tipp

Ha a felsorolást nem adja meg a felhasználó (kihagyja a paramétert), a metódusnál nem mutat hibát az IDE. Vararg esetén ugyanis ilyenkor automatikusan üres tömb lesz a paraméter. Ebben az esetben viszont IllegalArgumentException-t várunk, a megfelelő tájékoztató szöveggel.

Metódus hívások láncolása (methodchain)

Egy kifejezésen belül ugyanazon objektumon több metódust is hívhatunk láncoltan. Ennek feltétele, hogy a metódus azon objektumpéldánnyal térjen vissza, amelyen meghívtuk (this). Gyakran használjuk a Builder tervezési mintánál. (A Builder osztály egy másik osztály inicializálását és példányosítását végzi.) Lássunk erre egy példát!

Készítsünk egy Peasant osztályt, ahol a sakktáblán a pozícióját követhetjük nyomon a posX, posY attribútumokkal. A metódusok a lépéseket imitálják. A pozíció 1 és 8 között változhat, és bármilyen irányban csak egyet léphet, ha tud.

public class Peasant { public static final int MIN\_X = 1; public static final int MAX\_X = 8; public static final int MIN\_Y = 1; public static final int MAX\_Y = 8;

private int posX; private int posY;

public Peasant(int posX, int posY) { this.posX = posX; this.posY = posY;

}

public Peasant forward() { if (posX < MAX\_X){ posX++;

}

return this;

}

public Peasant back() { if (posX > MIN\_X) { posX--;

}

return this;

}

public Peasant left() { if (posY > MIN\_Y) { posY--;

}

return this;

}

public Peasant right() { if (posY < MAX\_Y) { posY++;

}

}

public String toString() {

return "X: " + posX + ", Y: " + posY;

}

}

A mozgását láncolt metódushívással könnyen megadhatjuk:

Peasant peasant = new Peasant(2, 3) // lehelyeztük a tábla (2, 3) mezőjére

.forward() // előre lépett

.forward() // előre lépett

.left() // balra lépett

.backward() // hátra lépett

.left() // balra lépett

.forward() // előre lépett

.right(); // jobbra lépett

System.out.println(peasant); //X: 4 Y: 2

Ellenőrző kérdések

* Mit tudsz a this pszeudóváltozóról?
* Milyen feltételei vannak a láncolható metódusoknak?

Gyakorlat - Robot és mozgatása

Egy Robot objektumot mozgatunk, utasításokat adva. Ezek menj és fordulj lehet. Az utasítások láncban is kiadhatók, azaz a robot egyszerre több utasítást is kaphat, amit sorban végrehajt és ezzel egy adott távot megtéve, adott irányban áll meg.

Hozz létre egy Robot osztályt, amiben legyen két attribútum: distance, amelyben a megtett eddigi összes távolságot, és azimut amiben az aktuális irányszöget (fokban) tárolja.

Publikus metódusok:

public Robot go(int meter) public Robot rotate(int angle)

Bónusz feladat

Egészítsd ki a Robot osztályt azzal, hogy a robot a megfelelő utasításra feljegyzi az aktuális pozícióját egy NavigationPoint objektumban, és ezt az objektumot hozzáadja egy listához.

Kiegészítő publikus metódus:

public Robot registerNavigationPoint()

Így a robot mozgása utólag végigkövethető - feltéve, hogy kapott utasítást az adott pozícióban ennek feljegyzésére a láncolt utasítások között. Ehhez a NavigationPoint objektumot a robot aktuális távolságával és irányával (azimut) hozzuk létre, majd a robot listájához hozzáadjuk. A teszteléshez a NavigationPoint objektumban meg kell írni a megfelelő toString metódust is.

Metódusnév túlterhelés (methoverloading)

Metódus túlterhelésről (method overloading) akkor beszélünk, ha egy osztályon belül több ugyanolyan nevű, de eltérő paraméter szignatúrával rendelkező metódus van. (Azaz a formális paraméterek típusának listája eltérő.) Akkor hasznos, ha több metódusnak is ugyanaz a feladata, de ehhez más és más bemenő adatra van szüksége. Híváskor a fordító onnan tudja, hogy melyik metódust kell futtatnia, hogy megnézi, hogy az aktuálisan kapott paraméterek mely formális paraméter szignatúrának felelnek meg.

Hogyan dönti el a fordító, hogy ez melyik?

1. Típusra pontos egyezést talál.
2. Primitív típus esetén nagyobbat talál.
3. Primitív típus esetén a csomagolóosztálynak megfelelőt talál.
4. Ugyanolyan típusú varargs paramétert talál.

A sorrend nagyon fontos: az első találatnál leáll, és csak egy konverziót végez!

Adva vannak az alábbi metódusok egy osztályban:

public void play(short a) {...} // 1 public void play(long a) {..} // 2 public void play(Integer a) {...} // 3 public void play(String... a) {...} // 4 public void play(String a, String b) {..} // 5 Melyik fut le az egyes esetekben?

short x = 3; play(4); play(x); play("Hello"); play("alma", "körte");

Az első híváskor az aktuális paraméter int típusú. Nincs pontos egyezés, de nagyobb primitív típust talál, ezért a // 2 fut le.

A második híváskor az aktuális paraméter short típusú. Ilyen paraméterű pont van, ezért az // 1 fut le.

A harmadik híváskor a paraméter String, amely csak a varargs-ot váró // 4 metódusnak felel meg.

A negyedik hívásra van pontos egyezés, ezért az // 5 metódus fut le, hiába felel meg a // 4 paraméter szignatúrának is.

Ellenőrző kérdések

* Mi határozza meg egy metódus paraméter szignatúráját?
* Mitől különbözhet két paraméter szignatúra?
* Fordítási vagy futási időben történik a hívás és a definíció összerendelése?

Gyakorlat - Time osztály

Szükségünk van egy időpontot reprezentáló osztályra Time, amely többféle paraméterezéssel példányosítható. Az osztály metódusai segítségével adott objektumát össze tudjuk hasonlítani másik Time objektummal és meg tudjuk mondani a kapott objektumról, hogy azonos időpontot reprezentál vagy az adott objektum korábbi időpontot reprezentál (mindkét esetben egy-egy napon belül vagyunk!).

Konstruktorok:

public Time(int hours, int minutes, int seconds) public Time(int hours, int minutes) public Time(int hours) public Time(Time time) Publikus metódusok:

public boolean isEqual(Time time)

public boolean isEqual(int hours, int minutes, int seconds)

public boolean isEarlier(Time time) public boolean isEarlier(int hours, int minutes, int seconds)

Megjegyzés

A Java természetesen rendelkezik a megfelelő dátum és időkezelő osztályokkal, de azok esetében is hasonló módon lett megoldva a többféle paraméterezés.

[rating feedback=java-methodoverloading-time]

Bónusz feladat

Kocsmatúrát tervezünk, és a maximális időkihasználás érdekében a legjobb kocsma a legkorábban nyitó intézmény. Valósítsuk meg ennek kiválasztását adott listából a Pub (kocsma neve és nyitási időpontja Time osztály használatával), valamint a listát tároló ListOfGoodPubs osztály segítségével.

Pub osztály publikus metódusok:

public Pub(String name, int hours, int minutes) public String toString()

ListOfGoodPubs osztály publikus metódusok:

public ListOfGoodPubs(List<Pub> goodPubs) public Pub findTheBest()

Hibakezelés

A ListOfGoodPubs osztály nem kaphat a konstruktorban null értéket, vagy üres listát!

Kivételkezeléssel (IllegalArgumentException) jelezzük, ha rossz a paraméter.

Statikus attribútumok és metódusok (staticattrmeth)

Statikus attribútumok

A statikus (static) attribútumok az osztályhoz, és nem az objektumhoz tartoznak, de minden az adott osztályú objektum eléri és közösen használhatja azokat.

Osztálybetöltéskor jönnek létre és inicializálódnak. Elérésükhöz osztályon belül csak a nevet kell megadnunk, osztályon kívülről azonban akár az osztálynevet, akár a változónevet használhatjuk minősítőnek. Konvenció szerint az osztály nevét szoktuk, ezzel is jelezve, hogy az attribútum “közös tulajdon”, nem az adott objektumé. Nagyon gyakran tárolunk bennük az osztályban használt konstanst, ilyenkor az attribútum static final módosítóval rendelkezik. A konstansok nevét csupa nagybetűvel írjuk, a több szóból állókat pedig '\_' jellel választjuk el.

class Parcel {

public static final double BOX\_WEIGTH = 2.3;

private double netWeight;

public Parcel(double netWeight){ this.netWeight = netWeight; }

public double grossWeight(){ return netWeight + BOX\_WEIGHT;

}

}

Ellenőrző kérdések

* Miben speciális a statikus attribútum?
* Hogyan lehet hivatkozni a statikus attribútumra?
* Hivatkozhat-e az osztály bármelyik metódusa a statikus attribútumra?

Statikus metódusok

Az osztályban deklarált metódusok csak példányosítás után, a példányon át érhetőek el, és jellemzően a példányváltozókkal dolgoznak. A statikus (static) metódusok példányosítás nélkül is elérhetőek, kívülről az osztálynév minősítővel. Elérik a statikus atribútumokat és más statikus metódusokat hívhatnak. Nem érnek el nem statikus attribútumokat és metódusokat, de a példány attribútumai és metódusai elérhetik a statikus metódusokat.

Használhatjuk konstruktorok helyett, főleg, ha sok túlterhelt konstruktorra van szükség, vagy más osztályt kell példányosítani.

class Sector{ private double degree; private double radius;

public Sector(double degree, double radius){ this.degree = degree; this.radius = radius;

}

public void setDegree(double degree){ this.degree = degree;

}

public void setRadian(double radian){ degree = radianToDegree(radian);

}

public static double radianToDegree(double radian){ return radian / Math.PI \* 180;

}

}

A fenti osztályban a radianToDegree metódust akkor is használhatjuk, ha nincs körcikkünk, csak szeretnénk átváltani radiánból fokra. A példány is használhatja, ha a körcikknek utólag állítjuk be a szögét, de azt csak radiánban tudjuk megadni (setRadian).

Ellenőrző kérdések

* Hogyan kell egy statikus metódust deklarálni?
* Hogyan kell egy statikus metódust meghívni?
* Az osztály milyen attribútumaira és metódusaira hivatkozhat a statikus metódus?
* Meghívhatja-e egy nem statikus metódus a statikus metódust? És fordítva?
* Példányosíthatja-e egy másik osztály objektumát egy statikus metódus?

Statikus import

Egy osztály statikus metódusait és attrobútumait közvetlenül importálhatjuk, ha az import után használjuk a static kulcsszót. Ebben az esetben a metódus és az attribútum minősítő nélkül is használhatóvá válik, ezért sok statikus import használata rontja a kód olvashatóságát.

import static java.lang.Math.PI;

class Circle{

private double radius;

public double area(){

return radius \* radius \* PI;

}

}

Gyakorlat - banki tranzakciók követése

BankTransaction osztály

Az osztály követi a létrehozott példányai által reprezentált banki tranzakciókat. Számolja a tranzakciókat, azok értékét összegzi, és utasításra átlagolja azokat. Minden tranzakció értéket megvizsgál, és a nap elején beállított statikus currentMinValue és currentMaxValue változókban nyilvántartja az adott tranzakcióig előforduló minimális és maximális tranzakció értékeket.

Publikus statikus metódusok:

public static void initTheDay()

public static BigDecimal averageOfTransaction() public static long getCurrentMinValue() public static long getCurrentMaxValue() public static BigDecimal getSumOfTrxs() Publikus metódus:

public BankTransaction(long trxValue)

Hibakezelés

Amennyiben a létrejövő BankTransaction objektum tranzakció értéke kívül esik a

konstansként megadott min és max határokon (1 és 10 000 000),

IllegalArgumentException-t várunk a megfelelő értesítő szöveggel.

Tipp

A megoldás készüljön fel arra az (egyébként nem túl valószínű) esetre is, ha egyetlen érvényes tranzakció sem volt. Ilyenkor az averageOfTransaction, a getCurrentMinValue és a getCurrentMaxValue metódusok nullát adjanak vissza.

Több esetben jól alkalmazható a Java három operandusú művelete!

Statikus import megjelenése

Figyeld meg, hogy a teszt osztályok hogyan használják az import és az import static direktívákat.

Figyeld meg, hogy a fejlesztő környezeted az általad írt példákban mikor generál import és mikor import static direktívákat.

[rating feedback=java-staticattrmeth-banktransaction]

## Konstruktorok és inicializátorok

Default és paraméter nélküli konstruktor (defaultconstructor)

Amennyiben egy osztálynak nem adunk meg konstruktort, a fordító biztosít egy default konstruktort az osztály példányosításához. Ez semmi más utasítást nem tartalmaz, csak egy super() hívást, amely az ős (jobb híján az Object osztály) paraméter nélküli konstruktorának hívása. Amennyiben bármilyen konstruktort implementálunk, akár paraméterrel, akár paraméter nélkül, az osztály nem kap default konstruktort.

A paraméter nélküli konstruktorban is inicializálhatunk attribútumokat, ha van valamilyen értelmes, az alapértelmezettől eltérő kezdőértékük.

####Ellenőrző kérdések

* Van-e olyan Java osztály, amelynek egyáltalán nincs konstruktora?
* Mikor lehet paraméter nélküli példányosítás?
* Mit csinál a default konstruktor?
* Mi a különbség a default konstruktor és a paraméter nélküli konstruktor között?

Gyakorlat 1 SimpleDate osztály

Az osztály év, hó, nap dátumokat reprezentál, de csak 1990. január 1. utáni dátumokat, ezeket year, month, day nevű (int típusú) attribútumokban tárolja, konstruktora nincs. Az értékek beállítása publikus metódusból történik.

publikus metódusok:

public void setDate(int year, int month, int day) public int getYear() public int getMonth() public int getDay()

A dátum részek megadásakor legyen ellenőrzés, csak korrekt év (>= 1900), korrekt hónap (1-12) és a hónapnak megfelelő nap érték fogadható el. Vegyük figyelembe a szökőéveket is! Inkorrekt paraméter értékek esetén a metódus dobjon IllegalArgumentException-t, a megfelelő tájékoztató szöveggel.

Tipp

Az ellenőrzést szervezzük ki privát metódusokba! pl. a következők lehetnek:

private metódusok:

private boolean isCorrect(int year, int month, int day) private boolean isLeapYear(int year) private int calculateMonthLength(int year, int month)

Gyakorlat 2 SimpleDateFormatter osztály

A dátum adatot formázni is kellene, mégpedig országonként eltérő módon. Hozzunk létre egy CountryCode enumot HU, EN, US értékekkel. Az osztály egyik publikus metódusa átvesz egy enumot és annak alapján állítja össze a dátum stringet, a másik az alapértelmezett országkódot használja. Ezt az osztály paraméter nélküli konstruktora állítja be.

publikus metódusok:

public SimpleDateFormatter()

public String formatDateString(SimpleDate simpleDate)

public String formatDateStringByCountryCode(CountryCode countryCode, SimpleDate simpleDate)

Tanulmányi feladat

Hogyan tudjuk ellenőrizni, hogy a konstruktor nélküli osztályunk valóban kapott üres konstruktort?

Decompiler segítségével visszafejthetjük a .class fájlt és annak metódusait megtekinthetjük. Ehhez fordítsuk le a kész SimpleDate.java forráskódot (Maven ablak, build), majd navigáljunk parancssorból a projekt targetkönyvtárába. Ezt egyszerűen megtehetjük TotalCommander használatával, ha megkeressük a classes mappát, a Commands menüben az Open command prompt window pont ezen a mappán nyílik meg.

Adjuk ki a javap defaultconstructor.date.SimpleDate parancsot. A javap a JDK beépített decompiler alkalmazása, és a .class fájl alapján visszaadja annak tartalmát metódus szinten (további paraméterezés is lehetséges). Itt látható lesz az a defaultconstructor.date.SimpleDate() metódus is, amit a javac szerkesztett bele a class fájlba.

Konstruktor túlterhelés (constructoroverloading)

Dönthetünk úgy, hogy többféle bemenő paraméterkombinációval szeretnénk létrehozni az objektumokat. Ehhez több, különböző paraméter szignatúrájú konstruktorra van szükségünk, azaz a konstruktor is túlterhelhető. A paraméter egyeztetés ugyanúgy működik, ahogy a a túlterhelt metódusoknál.

Amennyiben az egyik konstruktor általi műveletek megismétlendők lennének egy másikban, akkor nem kell az utasításokat átmásolnunk, mert a konstruktorok hívhatnak más konstruktorokat a this kulcsszó segítségével. Amennyiben egyre bővülő paraméterlistával hozzuk létre a konstruktorokat, akkor gyakorlatilag mind használni tudja az előzőt. Ezt hívjuk teleszkóp konstruktornak. Fontos megjegyezni, hogy egy konstruktor csak egyetlen másikat hívhat, és a hívásnak a legelső utasításnak kell lennie!

public class Product {

private String name;

private int stock;

private LocalDate bestBefore;

public Product(String name) { this.name = name;

}

public Product(String name, int stock) { this(name); this.stock = stock;

}

public Product(String name, int stock, LocalDate bestBefore) { this(name, stock);

this.bestBefore = bestBefore;

}

}



overloading

Ha nagyon sokféle paraméterezéssel példányosíthatunk egy osztályt, érdemes megfontolni builder osztály létrehozását.

Ellenőrző kérdések

* Hány konstruktora lehet egy osztálynak?
* Milyen szabályok vonatkoznak az egy osztályon belüli konstruktorokra?
* Mi határozza meg, hogy egy példányosítás melyik konstruktort használja?
* Hogyan hívhatja meg egy osztály konstruktora ugyanazon osztály egy másik konstruktorát?
* Milyen szigorú szabály vonatkozik a this() használatára?

Gyakorlat 1 SimpleTime osztály

Az osztály objektuma egy időpontot reprezentál egy napon belül, és többféleképpen hozható létre. Az objektum el tudja dönteni, hogy a paraméterként kapott azonos típusú objektumtól percekben kifejezve mennyire különbözik. konstruktorok:

public SimpleTime(int hours, int minutes) public SimpleTime(int hours) public SimpleTime(SimpleTime time) publikus metódusok:

public int difference(SimpleTime time) public String toString()

Gyakorlat 2 BusTimeTable osztály

Az osztály egy buszmenetrendet reprezentál, ahol az indulási időket SimpleTime objektumok listája tárolja. Ezt többféleképpen létre lehet hozni, lásd a konstruktorokat (óránként indul, generáláskor az első indulás óráját, az utolsó indulás óráját, és az ismétlődő perceket kell megadni). A listából ki lehet keresni a következő indulás idejét.

konstruktorok:

public BusTimeTable(List<SimpleTime> timeTable) public BusTimeTable(int firstHour, int lastHour, int everyMinute) publikus metódusok:

public List<SimpleTime> getTimeTable() public SimpleTime nextBus(SimpleTime actual)

Hibakezelés

Ha az adott napon több busz már nem indul, a metódus dobjon IllegalStateExceptiont a megfelelő információs szöveggel.

Bónusz feladat

Írj egy olyan metódust a BusTimeTable osztályban, ami az aznapi első busz indulást adja vissza!

Gondolj arra, hogy a listában nem feltétlenül az első elem az első busz indulási ideje! A metódus teszteléséhez bővítsd ki a BusTimeTableTest osztályt a megfelelő teszt metódusokkal.

Inicializátorok (initializer)

Inicializátornak hívjuk az osztályba írt név nélküli blokkot, mely ugyanúgy utasításokat tartalmaz, mint a metódusok. Feladatuk az osztály változóinak inicializálása. Jogosan felmerülhet a kérdés, hogy miért van erre szükség, amikor ezeket akár a deklaráció sorában, akár a konstruktorban megtehetjük. Valóban nagyon ritkán használjuk, de mégis van létjogosultsága. A statikus inicializátorban (static kulcsszó előzi meg) található kód az osztály betöltődésekor, a nem statikus inicializátorban található példányosításkor, még a konstruktor előtt lefut. Statikus attribútumok értékének megadására, ha azok bonyolultan állíthatóak csak elő, használhatunk statikus metódusokat. (Például véletlenszámot kell generálnunk, vagy ciklussal kell feltöltenünk egy kollekciót, esetleg közben kivételt is kezelnünk.) Statikus inicializátort akkor használunk, ha egyszerre több statikus attribútum értékét akarjuk kiszámolni. A nem statikus inicializátor anonymous belső osztályok esetén használható jól, mivel ezekhez nem tudunk konstruktort írni, lévén nincs nevük.

Egy osztály akár több inicializátort is tartalmazhat, ebben az esetben a deklaráció sorrendjében futnak le.

public class Lottery {

public static final Set<Integer> numbers;

static {

Set<Integer> draws = new TreeSet<>();

Random rnd = new Random(); while(draws.size() < 5) {

draws.add(rnd.nextInt(90) + 1);

}

numbers = draws;

}

}

Ellenőrző kérdések

* Mikor hajtódik végre a statikus inicializáló blokk?
* Mikor hajtódik végre a példányszintű inicializáló blokk?
* Az osztály mely elemeire hivatkozhat a statikus inicializáló blokk?
* Mondj néhány példát, amikor a statikus inicializáló blokk használata célszerű!

Gyakorlat - Hitelkártya használat

A CreditCard osztály egy hitelkártyát reprezentál, amelyet a létrehozásakor adott összeggel “feltöltenek” és ebből tudunk gazdálkodni. A feltöltés forintban vagy tetszőleges valutában történhet, ezt a feltöltés során forintra konvertálja a rendszer.

A kiadás payment() többféle valutában is lehetséges, ezeket a rendszer egész forintra konvertálja és így terheli meg a kártyát. Ha nem adunk meg valutát, automatikusan forintnak veszi a terhelést.

A hitelkártya “használatba vételekor” (CreditCard osztály betöltése) az aktuális átváltási faktorok (Rate) feltöltésre kerülnek a CreditCard osztályba, egy statikus final listába (statikus inicializálás).

konstruktorok:

public CreditCard(long balance, Currency currency) public CreditCard(long balance) publikus metódusok:

public long getBalance()

public boolean payment(long amount, Currency currency) public boolean payment(long amount)

A Rate osztály az egyes valutákhoz (Currency enum) tartozó átváltási faktorokat tárolja, ezek listájából dolgozik a payment() metódus. Hozzuk létre a Currency enum-ot is HUF, EUR, SFR, GBP, USD értékekkel! konstruktor:

public Rate(Currency currency, double conversionFactor) publikus metódusok:

public Currency getCurrency() public double getConversionFactor() Öröklődés

Öröklődés (introinheritance)

Az osztályok tervezésekor és újra felhasználásakor felmerülhet, hogyan lehetne bővíteni egy már meglévő osztály attribútumait, metódusait anélkül, hogy az eredeti osztályt módosítanánk, illetve hogyan tehetnénk bele közös kollekcióba a különböző típusú objektumokat, ha valamilyen szempontból közösen szeretnénk kezelni őket (például egy rajzon lévő összes vonal, ellipszis, téglalap stb.) Az első esetben egy általános osztályból készítünk speciálisabbat (specializáció), a második esetben a speciális osztályokat általánosítjuk (generalizáció).

Specializáció: Vehicle –> MotorVehicle –> Car –> Taxi

Generalizáció: (Koala, Giraffe, Beaver) –> Mammal –> Animal

Az általánosabb osztályt szülőnek, illetve ősnek, a speciálisabbat gyermeknek vagy leszármazottnak nevezzük.

A leszármazott az ős minden tagjával rendelkezik, amire az ős engedélyt ad, és még többel is rendelkezhet. A leszármazást a Javaban az extends kulcsszó jelöli, amely jól jellemzi, hogy a leszármazott tulajdonképpen az ős kiterjesztése.

class Animal { // ...

}

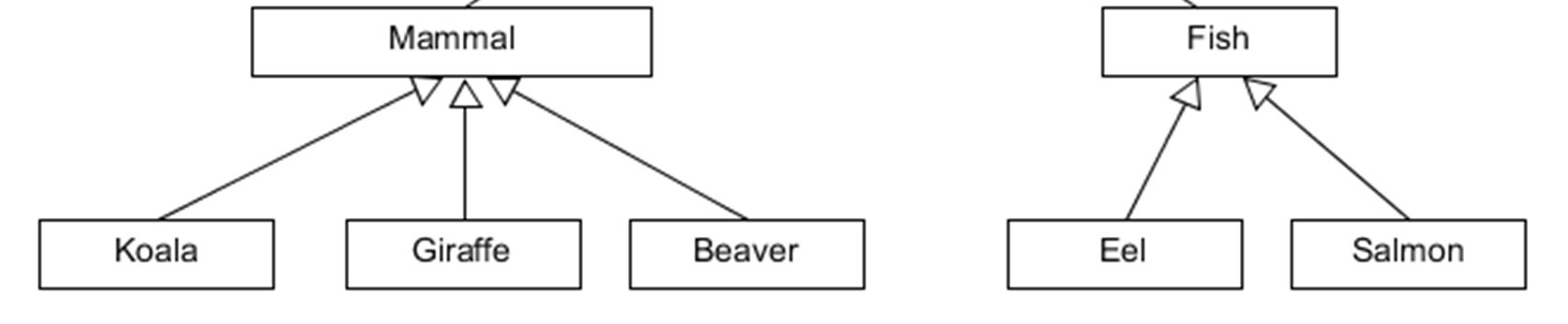
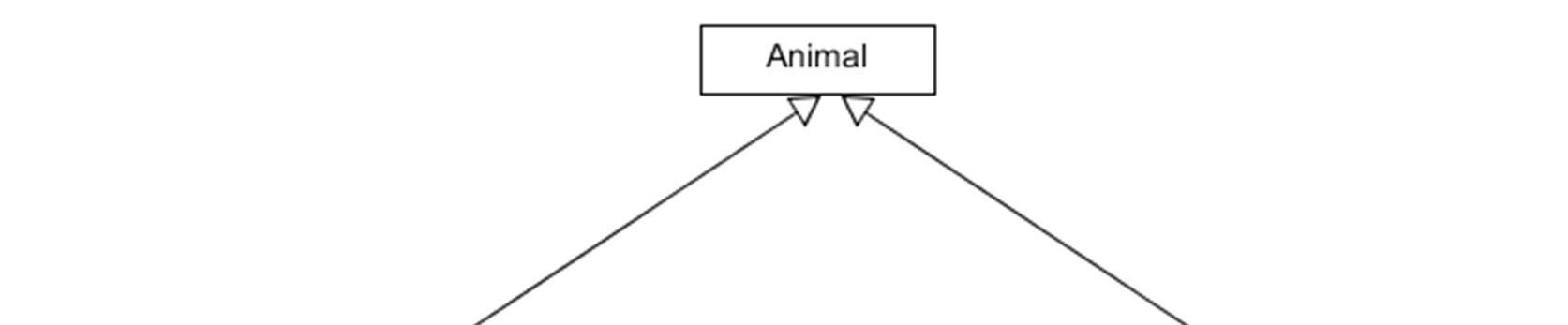
class Mammal extends Animal { // ...

}

class Koala extends Mammal { // ...

}

A Java nyelvben csak egyszeres öröklődés van. Ez azt jelenti, hogy egy osztálynak csak egy közvetlen szülője lehet, de akárhány gyereke. Közvetve vagy közvetlen, de minden osztály az Object osztályból származik. Azért látjuk, hogy egy általunk létrehozott új osztály olyan metódusokkal is rendelkezik, amiket nem írtunk bele, mert azokat az ősétől, az Object-től örökölte. (equals, hashCode, toString). Az osztályok és a köztük lévő öröklődési kapcsolatok egy fa szerkezettel ábrázolható.



class\_hierarchy

Osztályok közötti kapcsolatok

Láttuk, hogy a specializáció azt jelenti, hogy az utód is mindig egy ős. Azaz, a koala egy emlős, az emlős egy állat. Ezt is-a kapcsolatnak nevezzük, és öröklődésre utal. (a Koala is a Mammal)

Amikor az osztály azért tud valamit, mert egy része tud valamit, akkor az nem öröklődés, hanem tartalmazási kapcsolat, azaz kompozíció. Ilyenkor az osztály attribútumként tartalmaz egy másik osztályt. Ezt az angol kifejezés után has-a kapcsolatnak hívjuk. (the Person has a Name)

Amikor csak lehetséges, használjunk kompozíciót öröklődés helyett, mert azt attribútumban tárolt objektum futási időben dinamikusan cserélhető.

Ellenőrző kérdések

* Mire való az öröklődés?
* Amennyiben nem írjuk ki, öröklődnek-e valahonnan a Java osztályok?
* Hogyan kell Javaban öröklődést definiálni?
* Mi a különbség az is-a és has-a kapcsolatok között?
* Mit használjunk inkább öröklődés helyett? Miért?

Gyakorlat 1

#####Person, Employee, Boss és BigBoss osztályok

Ezek egy munkahelyi hierarchiát reprezentálnak, a fenti sorrendben egymásból öröklődő osztályok. Jelen esetben a Person osztályt nem is példányosítjuk, ez az alatta levő osztályok egyfajta absztrakciójának tekinthető. A különböző alkalmazottak fizetését eltérő módon számítjuk. Míg az Employee alapfizetéssel rendelkezik, a Boss esetében az alapfizetéshez hozzáadódik a vezetői pótlék (beosztottak száma \* LEADERSHIP\_FACTOR \* alapfizetés), míg a BigBoss esetében ehhez hozzáadódik egy vezetői prémium is (bonus).

Person osztály String name és String address attribútumokkal Publikus metódusok:

public Person(String name, String address) public void migrate(String newAddress) Employee osztály double salary attribútummal Publikus metódusok:

public Employee(String name, String address, double salary) public double getSalary() public void raiseSalary(int percent)

Boss osztály LEADERSHIP\_FACTOR = 0.1 és int numberOfEmployees attribútummal Publikus metódusok:

public Boss(String name, String address, double salary, int numberOfEmployees) public double getSalary() public int getNumberOfEmployees() BigBoss osztály double bonus attribútummal Publikus metódusok:

public BigBoss(String name, String address, double salary, int numberOfEmployees, double bonus) public int getNumberOfEmployees() public double getBonus() public double getSalary()

Gyakorlat 2

Item, Basket és ShoppingBasket osztályok, öröklődés helyett kompozíció

Az öröklődés mellett/helyett kompozíció is alkalmazható, ahol az alkotó osztályok egymás szolgáltatásait használják ki új funkciók megvalósítására. Itt az alap Basket osztály Item objektumokat tárol, és a ShoppingBasket osztály attribútumként tárol egy Basket objektumot, magasabb szintű és részben másféle funkciók kiszolgálására.

Item osztály String barcode, double nettoPrice, int vatPercent attribútumokkal Publikus metódusok:

public double getTaxAmount() // a nettoPrice és a vatPercent alapján kiszámolja az adó összegét public double getNettoPrice() public String getBarcode() public String toString() // generált string reprezentáció

Basket osztály List<Item> items attribútummal Publikus metódusok:

public void addItem(Item item) public void removeItem(String barcode) public void clearBasket() //a Basket ürítése public List<Item> getItems() // az Item lista másolatát adja vissza!

ShoppingBasket osztály Basket basket attribútummal publikus metódusok:

public void addItem(Item item) public void removeItem(String barcode) public double sumNettoPrice() // az összes tételre public double sumTaxValue() // az összes tételre public double sumBruttoPrice() public void checkout() // befejezzük a vásárlást, a kosár ürítése public void removeMostExpensiveItem() // kikeresi és eltávolítja a kosárból a legdrágább tételt

Bónusz feladat

Jelenleg az alkalmazásban nincsen hibavédelem, minden paramétert elfogadunk ahogy van. Nézd végig a metódusokat, és ahol indokolt, kivételdobásokkal védd meg a hibás adatbeviteltől.

Konstruktorok és az öröklődés viszonya (inheritanceconstructor)

Első és legfontosabb, amit meg kell jegyeznünk, hogy a konstruktorok nem öröklődnek. A második, hogy az utód osztály példányosításakor mindig lefut az ős osztály valamelyik konstruktora, akár tartalmaz explicit hívást az utód konstruktora, akár nem. A hívást a super() metódus valósítja meg.

Hogyan is működik mindez. Tekintsük a következő osztályokat:

public class Person {

private String name;

private int age;

public Person(String name, int age) { this.name = name; this.age = age;

}

}

public class Employee extends Person {

private int salary;

public Employee(String name, int age, int salary) { this.name = name; // nem fordul le this.age = age; // nem fordul le this.salary = salary; // ez lefordul

}

}

Az Employee példányosításakor azt várjuk, hogy legyen neve, kora és fizetése is, ezért ezeket átadjuk a konstruktorának. Csakhogy ő nem éri el a name és age attribútumokat, mert ezek a szülőben vannak deklarálva és a private láthatóság miatt semmilyen másik osztályból nem érhetőek el. Gondolhatnánk, hogy ha van publikus setter metódusa, akkor az öröklődik, tehát használható.

public class Person {

private String name;

private int age;

public Person(String name, int age) { this.name = name; this.age = age;

}

public void setName(String name) {...}

public void setAge(int age) {...}

}

public class Employee extends Person {

private int salary;

public Employee(String name, int age, int salary) { this.setName(name); this.setAge(age); this.salary = salary; } // még mindig nem fordul le

}

Ez teljesen igaz, használhatjuk, azonban a fordító még mindig hibát jelez. A hiba oka az, hogy nem használtuk a szülő konstruktorát sehol. A fordító ilyenkor is elhelyez az Employee konstruktorában egy super() hívást, amely az ős paraméter nélküli vagy default konstruktorát hívná. Csakhogy nincs ilyen a Person-ban, ezért szükséges, hogy mi hívjuk meg az ott lévő két paraméteres konstruktort, átadva neki a szükséges adatokat.

A helyes megoldás ebben az esetben:

public class Person {

private String name;

private int age;

public Person(String name, int age) { this.name = name; this.age = age;

}

}

public class Employee extends Person { private int salary;

public Employee(String name, int age, int salary) { super(name, age); this.salary = salary;

}

}

Két fontos szabályt azonban be kell tartanunk:

* A super() hívás mindig a konstruktor első utasítása kell legyen.
* Egy konstruktor vagy super(), vagy this() hívást tartalmaz, a kettőt egyszerre sosem.

Mivel minden osztály példányosításakor lefut az ős konstruktora is, ezért super hívásnak mindenképp kell lennie valamelyik konstruktorban, akkor is, ha az implicit, azaz nem írjuk ki. (A default konstruktor ezek szerint implicit tartalmaz egy super() hívást és semmi mást.) Az első szabály miatt először mindig az ős konstruktora fut le, és csak utána a gyereké, azaz jelen esetben először az Object, majd a Person és legvégül az Employee kontruktora.

Ellenőrző kérdések

* Mi a konstruktorok végrehajtási sorrendje egy leszármazott osztály példányosításánál?
* Mi határozza meg, hogy az ős osztály melyik konstruktora hajtódik végre?
* Milyen szigorú szabály vonatkozik a super(...) használatára?
* Mi a következménye az ős osztályra vonatkozóan, ha a leszármazott osztálynak csak default konstruktora van?
* Mi a következménye az ős osztályra vonatkozóan, ha a leszármazott osztály konstruktora nem tartalmaz super(...) hívást?

Gyakorlat 1

Car és Jeep osztályok

A Car osztályból öröklődik a Jeep osztály, egyes metódusokat felülírva és használva az ős attribútumait. Mindkét autótípus esetében van drive(int) metódus, ami a vezetést szimulálja, adott km megtételét és közben az elhasznált üzemanyaggal csökkenti annak mennyiségét is, valamint ki tudjuk számítani a tankolható üzemanyagot is

(calculateRefillAmount()). A Jeep abban különbözik ősosztályától, hogy üzemanyagot nem csak a tankban, hanem kannákban is tud szállítani. Felhasználáskor mindig először a kannákat ürítjük ki, utána a tankban levő üzemanyag mennyisége csökken.

Car osztály double fuelRate, double fuel double tankCapacity attribútumokkal Publikus metódusok:

public Car(double fuelRate, double fuel, double tankCapacity) public void modifyFuelAmount(double fuel) public void drive(int km) // csökkenti az üzemanyag mennyiségét, nem fogyhat ki!

public double calculateRefillAmount() //kiszámolja, mennyit lehet tankolni

Jeep osztály double extraCapacity és double extraFuel attribútumokkal

Publikus metódusok:

public Jeep(double fuelRate, double fuel, double tankCapacity, double extraCapacity, double extraFuel) public void modifyFuelAmount(double fuel) public void drive(int km) // csökkenti az üzemanyag mennyiségét, nem fogyhat ki!

public double calculateRefillAmount() //kiszámolja, mennyit lehet tankolni

Tipp

Érdemes egy külön metódusba kiszervezni, hogy adott km megtételéhez van-e elegendő üzemanyagunk!

Gyakorlat 2

Course osztály, enum Facility PROJECTOR, COMPUTERS és CHALKBOARD elemekkel, Room és ClassRoom osztályok

A Room leszármazottja a ClassRoom osztály.

Ezek egy oktatócég termeit és kurzusait reprezentálják, meg kell tudni mondani, hogy adott kurzust annak létszáma és igénye (facility) alapján adott teremben meg lehet-e tartani.

Room osztály String location és int capacity attribútumokkal.

Publikus metódusok:

public Room(String location, int capacity) ClassRoom osztály Facility facility attribútummal Publikus metódusok:

public ClassRoom(String location, int capacity, Facility facility) public boolean isSuitable(Course course)

Course osztály int participants és Facility facilityNeeded attribútumokkal Publikus metódusok:

public Course(int participants, Facility facilityNeeded)

Bónusz feladat

Mi történik, ha próbaképpen egy új osztályt akarunk örökíteni a ClassRoom-ból? Az IDE mit “követel” és miért?

Object ősosztály (objectclass)

Az Object osztály a java.lang csomagban található, és minden olyan osztály közvetlen őse, amelyben nem adunk meg explicit őst. A Java nyelvben egyszeres öröklődés van, ezért ez az osztály az öröklődési hierarchia gyökéreleme. Mivel minden osztály közvetlen vagy közvetve leszármazottja, ezért mindegyik örökli az Object publikus metódusait (pl. equals, hashCode, toString). Mivel minden osztály Object és minden példány Object példány is, ezért az Object típusú referenciaváltozónak bármilyen objektumot értékül adhatunk. Sőt, még a tömbnek is őse, ezért akár tömböt is bele tehetünk. Gyakorlatilag minden adatszerkezet mögött Object áll, a kollekciók mögött is.

Már tapasztalhattuk, hogy minden objektumot át tudunk adni a System.out.print metódusnak, azonban többnyire valamilyen furcsa szöveg jelenik meg a képernyőn. Ez azért van, mert ez az objektum toString() metódusát hívja, amely mindig van neki, hiszen az Object osztálytól örökölte. Lehetőségünk van ezt a metódust felülírni, hogy számunkra értelmezhető szöveg jelenjen meg.

Ellenőrző kérdések

* Van-e őse az Object osztálynak?
* Lehet-e létrehozni olyan Java osztályt, aminek nincs őse?
* Sorold fel az Object osztály legfontosabb metódusait!
* Mi a toString() metódus szerepe?
* Amennyiben egy objektumot kiíratunk, melynek osztályában nincs implementálva a toString() metódus, mi kerül kiírásra?

Gyakorlat - SimpleBag

Book, Beer és SimpleBag osztályok

A SimpleBag osztály egy tetszőleges számú Object típusú objektumot tartalmazó adatszerkezetet reprezentál, a Book és Beer osztályok a kipróbálását segítik. A tartalmazott objektumok között lehetnek azonosak is. A SimpleBag osztály a tartalmazott objektumokat egy List<Object> items attribútumban tárolja, és saját metódusai vannak az adatszerkezet kezelésére. Az adatszerkezet bejárása a “kurzor” szemlélettel lehetséges. A kurzor a létrehozáskor az első elem előtt áll, és bármikor újra ide helyezhető a beforeFirst metódus meghívásával. Azaz a bejárás megismételhető.

publikus metódusok:

public SimpleBag()

public void putItem(Object item) public boolean isEmpty() public int size() public void beforeFirst() public boolean hasNext() public Object next()

public boolean contains(Object item) public int getCursor()

Book osztály String author és String title attribútumokkal publikus metódusok:

public Book(String author, String title) public String toString() public boolean equals(Object o) // generált metódus! Beer osztály String name és int price attribútumokkal publikus metódusok:

public Beer(String name, int price) public boolean equals(Object o) // generált metódus!

Attribútumok öröklődése (inheritanceattributes)

Az Information hiding alapelv szerint egy osztály attribútumai mindig privátok, ahhoz csak a publikus metódusokon át lehet hozzáférni. Vajon mi történik, ha az osztályt kiterjesztjük? Azt láttuk, hogy az utód osztály nem fér közvetlenül hozzá ezekhez az attribútumokhoz, de tartalmazza ezeket is. Láthatósági módosítókkal természetesen szabályozhatjuk ezt, hiszen a protected és public kulcsszóval ellátott attribútumokat elérjük az utódból, azonban nem ez a követendő.

A Java nyelv megengedi, hogy a gyerek osztály ugyanolyan névvel deklaráljon egy attribútumot, amilyen a szülőben már volt. Ebben az esetben az attribútum elfedi a szülőben deklaráltat, de nem írja felül, vagyis az létezik, és elérhető a super minősítőn át. Ne tegyünk ilyet, mert a kód bonyolulttá, áttekinthetetlenné válik.

Láthatósági módosítók

Attribútumokra a láthatósági módosítók ugyanazok, mint a metódusokra.

módosító láthatóság

|  |  |
| --- | --- |
| private | csak a saját osztályból elérhető (illetve ugyanolyan osztályú objektumból) |
| [default] (package private) | ugyanazon csomagban lévő osztályból érhető el |
| protected | ugyanazon csomagban lévő és leszármazott osztályból érhető el |
| public | minden osztály számára elérhető |

Ellenőrző kérdések

* Öröklődnek-e az attribútumok?
* Minden örökölt attribútumot elér közvetlenül a leszármazott?
* Milyen láthatósági módosítók vannak? Ezek közül mely(ek) az ajánlottak?
* Hogyan érheti el a leszármazott osztály az ős osztály private attribútumát?

Gyakorlat - Book és ShippedBook osztály

Készíts egy Book osztályt, pontosan az alábbi előírások alapján!

Attribútumok:

* private String title; • protected int price; publikus metódusok:

public Book(String title, int price) public String getTitle() public int purchase(int pieces)

Készíts egy ShippedBook osztályt a Book osztály leszármazottjaként, egy shippingCost attribútummal.

publikus metódusok:

public ShippedBook(String title, int price, int shippingCost) public int order(int pieces) public String toString()

Tanulmányozandó

Az öröklődés és a konstruktorok hívási lánca alapján értelmezzük, hogy mi történik a gyerekosztály példányosítása során!

Metódusok öröklődése (inheritancemethods)

Hasonlóan az attribútumokhoz, a metódusok is megjelennek a leszármazottban, de a láthatóságukat a szülő osztály korlátozhatja. Csak a public és protected metódusok érhetőek el a leszármazott osztályból biztosan, a láthatósági módosító nélküliek csak akkor, ha a szülő és a gyermek is ugyanazon csomagban található. A gyermek osztály az örökölt és általa elérhető metódusokat felülírhatja (override) az alábbi szabályok szerint:

* A gyermek metódus láthatósága nem lehet szűkebb.
* A visszatérési érték típusa primitív típus esetén megegyezik, osztály esetén csak ugyanolyan, vagy a leszármazottja lehet (kovariáns típus).
* A metódus szignatúrájának meg kell egyeznie.
* Nem dobhat újabb vagy bővebb kivételt, de ezt a részt akár el is hagyhatjuk.
* final kulcsszóval ellátott metódus nem írható felül.

Ha egy metódust deklaráltak a szülőben és a gyermekben is, akkor a gyermekből elérhető a szülő metódusa is a super minősítővel.

Ha eredetileg nem volt hozzáférése a metódushoz, és ugyanolyan névvel létrehoz egyet, akkor az elfedi az örököltet.

public class Bear {

public void hunt() {

System.out.println("Bear is hunting."); eat();

}

public void speak() {

System.out.println("Bear roars.");

}

protected void sleep() {

System.out.println("Bear is sleeping.");

}

private void eat() {

System.out.println("Bear is eating.");

}

}

public class Grizzly extends Bear {

public void speak() { // Override

System.out.println("Grizzly growls.");

}

public void sleep(int length) { // Overload

System.out.println("Grizzly is sleeping for " + length + " hours.");

}

public boolean eat() { // Hide System.out.println("Grizzly is eating.");

}

}

A Bear négy metódusából az eat nem elérhető a Grizzly-ből, ezért a Grizzly eat metódusa nem írja felül a Bear eat metódusát, hanem elrejti azt.

A hunt metódus változtatás nélkül öröklődik, a speak metódust pedig a Grizzly felülírja, mivel a két metódus szignatúrája megegyezik. A két sleep metódus paramétereiben nem egyezik, ezért a Grizzly-ben deklarált sleep az örökölt metódus túlterhelése, azaz itt két különböző szignatúrájú sleep is létezik.

Próbáljuk ki, mi történik, ha minden elérhető metódust meghívunk egy Grizzly objektumból.

public static void main(String[] args) { Grizzly grizzly = new Grizzly(); grizzly.hunt(); // Bear is hunting. Bear is eating.

grizzly.sleep(); // Bear is sleeping.

grizzly.sleep(10); // Grizzly is sleeping for 10 hours.

grizzly.speak(); // Grizzly growls. grizzly.eat(); // Grizzly is eating.

}

A két sleep, a speak és az eat metódus láthatóan úgy működik, ahogy várjuk. A örökölt hunt metódus hívja az eat metódust, ami létezik mindkét osztályban. Láthatóan az ősben lévő fut le. Lássuk mi történik, ha megváltoztatjuk az eat láthatóságát a Bear-ben protectedre, azaz elérhetővé válik a leszármazottban. Ekkor a Grizzly eat metódusa nem elrejti, hanem felülírja azt, azaz többé nem lehet a visszatérési értéke boolean, mert az ellentmondana a fenti 2. szabálynak.

public class Bear {

public void hunt() {

System.out.println("Bear is hunting."); eat();

}

public void speak() {

System.out.println("Bear roars.");

}

protected void sleep() {

System.out.println("Bear is sleeping.");

}

protected void eat() {

System.out.println("Bear is eating.");

}

}

public class Grizzly extends Bear {

public void speak() {

System.out.println("Grizzly growls.");

}

public void sleep(int length) {

System.out.println("Grizzly is sleeping for " + length + " hours.");

}

public void eat() {

System.out.println("Grizzly is eating.");

}

}

Hívjuk meg újra a Grizzly hunt metódusát:

public static void main(String[] args) { Grizzly grizzly = new Grizzly(); grizzly.hunt(); //Bear is hunting. Grizzly is eating.

grizzly.eat(); //Grizzly is eating.

}

A metódusok mindig abból az osztályból hívódnak meg, amilyen objektumban vagyunk (dinamikus kötés), de figyeljünk a rejtett és az override-olt metódusok közötti különbségre!

Ha az utódban a felülíró metódust a @Override annotációval látjuk el, akkor már fordítási időben kiderül, ha az mégsem felülírja, hanem elrejti vagy túlterheli az ősben lévő metódust.

Statikus metódusok öröklődése

A statikus metódusok is öröklődnek, de sosem írhatóak felül, csak elrejthetőek. Az elrejtés szabályai azonban ugyanazok, mint a nem statikus metódusok felülírási szabályai kiegészítve még eggyel:

• A szülőben statikus metódus a gyermekben is statikus kell legyen, és a szülőben nem statikus metódus nem definiálható felül a gyermekben statikus metódussal.

Mindkettő fordítási idejű hibához vezet.

A statikus metódusok statikusan kötődnek az őket hívó metódushoz. Azaz ugyanazon osztálybeli metódus lesz mindig meghívva, mint ahonnan hívják. Képzeljük el, hogy a fenti Bear és Grizzly osztályokban a hunt és az eat metódusok is statikusak.

public class Bear {

public static void hunt() {

System.out.println("Bear is hunting."); eat();

}

protected static void eat() {

System.out.println("Bear is eating.");

}

}

public class Grizzly extends Bear {

public static void eat() {

System.out.println("Grizzly is eating.");

}

}

Ekkor a Grizzly osztályból hívva a hunt metódust, mivel az a szülőben van definiálva, azt tapasztaljuk, hogy a szülő eat metódusát hÍvja. De ezt is vártuk, hiszen a gyerekben lévő statikus eat metódus csak elrejti a szülő eat metódusát, és nem felülírja azt.

public static void main(String[] args) {

Grizzly.hunt(); // Bear is hunting. Bear is eating. Grizzly.eat(); // Grizzly is eating.

}

Ha példányból hívjuk őket, akkor a deklarált típus számít, és nem az, hogy milyen típusú a tényleges objektum.

public static void main(String[] args) { Bear bear = new Grizzly(); Grizzly grizzly = new Grizzly(); bear.eat(); // Bear is eating. grizzly.eat(); // Grizzly is eating.

}

Éppúgy ahogy a statikus változók esetén, a statikus metódusoknál is kerüljük az referencia változón való hívást, és inkább használjuk az osztály nevét minősítőként.

Ellenőrző kérdések

* Öröklődnek-e a metódusok?
* Minden örökölt metódust elér a leszármazott?
* Milyen láthatósági módosítók vannak?
* Hogyan érheti el a leszármazott osztály az ős osztály private metódusát?
* Milyen szabályai vannak a metódus felüldefiniálásnak?
* Lehet-e a felüldefiniáló metódusnak más típusú visszatérési értéke, mint a felüldefiniáltnak?
* Mi a haszna az @Override annotációnak? Kötelező-e használni?
* Hogyan lehet a leszármazott osztályban a felüldefiniált ős osztálybeli metódust elérni?
* Az A osztály leszármazottja a B, annak leszármazottja a C. Az A osztályban van egy m() metódus, amit a B osztály nem definiált felül. Felüldefiniálhatja-e ezt a metódust a C osztály?
* Mi történik, ha a leszármazott osztályban van egy, az ős osztály private metódusával azonos nevű metódus?
* Hogyan lehet megakadályozni, hogy a leszármazott osztály felüldefiniálja az ős osztály egy metódusát? Mikor lehet erre szükség?

Gyakorlat 1

Product és PackedProduct osztályok

Az ős Product osztály adott terméket reprezentál, a PackedProduct osztály ennek becsomagolt specializációja. Egymástól metódusokat örökölnek, de ezeket az osztályra jellemző módon felül kell írni (overwriting).

Product osztály String name, BigDecimal unitWeight és int numberOfDecimals attribútumokkal. Ha a tizedes értékek száma nics megadva, alapértelmezetten két tizedesjeggyel számolunk (egységként kg értendő).

publikus metódusok:

public Product(String name, BigDecimal unitWeigth, int numberOfDecimals) public Product(String name, BigDecimal unitWeigth) public BigDecimal totalWeight(int pieces)

PacketProduct osztály int packingUnit és BigDecimal weightOfBox attribútumokkal. Ezek megadják, hogy a termékből hány darab helyezhető egy dobozba, és annak súlya alapján a csomagolt termék súlya számítható.

publikus metódusok:

public PackedProduct(String name, BigDecimal unitWeight, int numberOfDecimals, int packingUnit, BigDecimal weightOfBox) public BigDecimal totalWeight(int pieces)

Írd felül az örökölt totalWeight() metódust úgy, hogy egy szállítmány (azaz a termékek és a szükséges számú dobozok) összes súlyát adja vissza, szintén numberOfDecimals tizedesre kerekítve. A darabszámtól függően lehet, hogy lesz egy nem tele doboz is! A felüldefiniálás során felhasználhatók örökölt metódusok is!

Gyakorlat 2

DebitAccount és CreditAccount osztályok

Az ős DebitAccount és a leszármazott CreditAccount osztályok egy, a saját számlához csatolt terheléses kártyát és egy kombinált kártyát reprezentálnak. Míg az előbbi csak a számlaegyenleg értékéig használható (debit kártya), a kombinált kártya a számlaegyenleg felett az előre megállapított hitelkeretig felhasználható. Minden tranzakciónak költsége van, ez a megadott konstans értékek és a tranzakció értéke alapján számítódik, és levonásra kerül az egyenlegből.

DebitAccount osztály String accountNumber és long balance attribútumokkal, valamint \* double COST - 3.0-ra inicializálva, és \* long MIN\_COST - 200-ra inicializálva konstans értékekkel. publikus metódusok:

public DebitAccount(String accountNumber, long balance) public final long costOfTransaction(long amount) public boolean transaction(long amount) public void balanceToZero() // az egyenleget nullázza le, túlköltés esetén

CreditAccount osztály a DebitAccount osztály leszármazottjaként, long overdraftLimit attribútummal.

publikus metódusok:

public CreditAccount(String accountNumber, long balance, long overdraftLimit) public boolean transaction(long amount) Absztrakt osztályok és interfészek

Absztrakt osztályok (abstractclass)

Absztrakt az az osztály, amely valamilyen elvont fogalmat, csoportot reprezentál, ezért közvetlenül nincs értelme példányt létrehozni belőle. Arra használjuk, hogy a leszármazott speciális osztályokból összevonjuk a közös tulajdonságokat és viselkedést, még akkor is, ha nem biztos, hogy pontosan tudjuk, mit is kellene csinálnia egy-egy metódusnak. Absztrakt osztályt az osztálynév előtti abstract módosítószóval hozhatunk létre. Amennyiben nem tudjuk, hogy egy elvárt viselkedés pontosan mit is takar, akkor a metódus törzsét elhagyhatjuk. Ebben az esetben a metódust is el kell látnunk abstract módosítóval. Az absztrakt osztály leszármazottainak implementálniuk kell az absztrakt metódusokat, vagy ők is absztrakttá válnak.

public abstract class Animal {

private String name;

public String getName() { return name;

}

public abstract void move();

}

public abstract class Bird extends Animal {

private int eggs;

public void layEggs(int numberOfEggs) { eggs = numberOfEggs;

}

}

public class Duck extends Bird {

public void move() {

System.out.println("Waddle");

}

}

Az Animal osztályban deklaráltunk egy move metódust, de implementáció híján a metódus, és ezért az osztály is absztrakt. A Bird osztályban kiegészítettük az Animal osztályból örökölt metódusokat egy újjal, de mivel nem implementáltuk a move metódust, ez az osztály is absztrakt kell legyen. A Bird osztályból származtatott Duck osztály már nem tartalmaz egyetlen absztrakt metódust sem, hiszen ez már implementálja az örökölt move metódust. Egyikben sem adtunk meg konstruktort, de mindegyikben ott van a default konstruktor. Absztrakt osztályt ugyan nem tudunk példányosítani, de változó statikus típusa lehet. Ebben az esetben bármely leszármazott osztály példánya tárolható benne, de csak az absztrakt osztályban is létező metódusok hívhatók a változón.

public static void main(String[] args) { Animal animal = new Duck(); animal.move(); // Waddle

animal.layEggs(5); // Fordítási hiba: Animalben nincs layEggs metódus

Bird bird = new Duck(); bird.move(); // Waddle

bird.layEggs(5); // OK

}

Ellenőrző kérdések

* Mikor kell ellátni egy osztályt abstract minősítővel?
* Lehet-e absztrakt osztályt példányosítani?
* Lehet-e absztrakt osztály egy referencia dinamikus típusa?

Gyakorlat - Absztrakt osztály létrehozása, használata

Feladat egy egyszerű játék logikájának megvalósítása. Ez a játék két konkrét játék karaktert támogat: a nyilast, és a baltás harcost. Vannak közös viselkedések, azért bevezetünk egy karakter közös őst. Mivel azonban van olyan viselkedés, amely különbözik (de minden karakternek van) a második szintű támadás, ezért a közös ős absztrakt. A megoldások az abstractclass.gamecharacter csomagban legyenek.

Definiálj egy síkpont tárolására képes immutable Point osztályt, x és y long típusú attribútumokkal. Definiálj egy distance metódust, amely megkap egy másik pontot, és visszatér a két pont távolságával (Pitagorasz-tétel).

Definiálj egy karakterek modellezését megvalósító Character absztrakt osztályt.

* szükséges attribútumok
  + position, hol van a karakter (Point típusú).
  + hitPoint, mennyi életereje van még a karakternek (int típus), alapértelmezett értéke 100.
  + random, Random típus, véletlenszámok generálásához.
  + Hozz létre még egy isAlive metódust, amely igazzal tér vissza, ha még él a karakter, azaz a hitPoint nagyobb, mint 0 (egyébként hamis).
* protected getActualPrimaryDamage metódus, amely visszaad egy egy és tíz közötti véletlen értéket (egész).
* private getActualDefence metódus, amely visszaad egy nulla és 5 közötti véletlen értéket (egész).
* protected void hit(Character enemy, int damage) metódus,
  + amely lekérdezi az aktuális védelmet (használva a getActualDefence metódust).
  + Ha gyengébb a védelem, mint a sebzés (damage paraméter), akkor levonja a sebzés értékét az életerőből, hívja a decreaseHitPoint private metódust.
* public void primaryAttack(Character enemy), amely csak továbbhívja a hit metódust a saját enemy paraméterével, és a getActualPrimaryDamage metódus visszatérési értékével.
* private void decreaseHitPoint(int diff), amely levonja a diff paraméter értékét az életerő (hitPoint attribútum) értékéből.
* deklarálj egy abstract public void secondaryAttack(Character enemy) metódust, melyet a leszármazott konkrét osztályok implementálnak majd.

Hozz létre egy Archer osztályt, amely kiterjeszti a Character osztályt a következőképpen:

* szükséges attribútumok
  + numberOfArrow, hány nyíl van még (int típusú).
* hozz létre egy konstruktort, amely megkapja a nyílas pozícióját és erre állítja be a position örökölt attribútumot. Állítsa be a numberOfArrow attribútum értékét 100ra.
* generálj gettert a numberOfArrow attribútumra.
* hozz létre egy privát getActualSecondaryDamage metódust, ami visszaad egy 1 és 5 közötti egész véletlen értéket.
* definiálj egy shootingAnArrow privát metódust, amely megkapja az ellenséges karakter referenciáját (enemy). Csökkenti a nyílak számát numberOfArrow, és meghívja a hit örökölt metódusát a kapott ellenséges karakter referenciával, és a getActualSecondaryDamage metódus visszatérési értékével.
* implementáld az örökölt secondaryAttack metódust úgy, hogy meghívod az előzőleg definiált privát shootingAnArrow metódust, átadva a kapott enemy paramétert.

Hozz létre egy AxeWarrior osztályt, amely kiterjeszti a Character osztályt a következőképpen:

* hozz létre egy konstruktort, amely megkapja a harcos pozícióját és erre állítja be a position örökölt attribútumot.
* hozz létre egy privát getActualSecondaryDamage metódust, ami visszaad egy pozitív véletlen egész értéket amely az elsődleges támadás maximum kétszerese.
* implementáld az örökölt secondaryAttack metódust úgy,
  + hogy ha az ellenség közelebb van mint két egység (használd a Point distance metódust), akkor meghívja az örökölt hit() metódust, átadva a kapott enemy paramétert és a getActualSecondaryDamage() metódus visszatérési értékét.

Hozz létre egy BattleField osztályt, ami használja a karaktereket.

* szükséges attribútumok
  + round, hány kör van még (int típusú).
* generálj egy gettert a round attribútumra.
* hozz létre egy private boolean oneHit(Character attacker, Character defender) metódust, amely igazzal tér vissza, ha mindkét karakter él. Törzsében meghívja a támadó (attacker) primaryAttack() majd secondaryAttack() metódusait, melyeknek paramétere a védekező defender, ha még mindkettő él.
* hozz létre egy public Character fight(Character one, Character other) metódust, amely visszatér az élve maradt karakterrel. Törzsében
  + mindaddig hívja a privát oneHit metódust felváltva a támadó és védekező szerepeket, amíg valamelyik meg nem hal. Minden körben először a one támadja az other-t, majd az other a one-t. – közben növeli a round attribútum értékét.
  + végül visszatért az élő karakter referenciával.

Interfészek (interfaces)

Az interfész fogalmával már más területen is találkozhattál. Jelentése: kapcsolódási felület és leírás. Tulajdonképpen interfésznek nevezzük egy rendszer azon elemeit, amelyen át a rendszerhez kapcsolódni lehet anélkül, hogy pontosan ismernénk annak belső működését. Interfészt képeznek ezért egy osztály publikus tagjai, de a Java nyelvben egy speciális nyelvi elem is.

Az interface olyan osztályhoz hasonló egység, amely ezt a kapcsolódási felületet biztosítja. Előírja az őt implementáló osztály számára, hogy milyen publikus metódussal vagy metódusokkal kell rendelkeznie, hogy kompatibilis legyen más osztályokkal. Természetesen az implementáló osztály eldöntheti, hogyan valósítja meg az előírt metódust, de a metódus szignatúrájának meg kell egyeznie az előírttal. Ezen kívül még konstansokat (public static final) deklarálhat, melyeket kívülről az interfész nevét használva minősítőként érhetünk el.

Egy osztály több interfészt is implementálhat. Az interfészek között lehetséges a többszörös öröklődés, azaz egy interfésznek több őse is lehet. Mivel az interfész csak azt írja elő, hogy milyen metódust kell tartalmaznia az osztálynak, azért csak metódusfejeket tartalmaz, melyek mindig publikusak és absztraktak, ezért ezeket a módosítókat akár el is hagyhatjuk az interfészben.

public interface Writable {

int DEFAULT\_CONTENT\_SIZE = 100;

boolean write(String text);

}

ugyanaz, mint a

public interface Writable {

public static final int DEFAULT\_CONTENT\_SIZE = 100;

public abstract boolean write(String text);

}

Egy interfész üres is lehet, ebben az esetben jelölő (marker) interfésznek nevezzük.

public interface Erasable { void erase();

}

public interface Flat {

}

Az osztály létrehozásakor az implements kulcsszó után vesszővel elválasztva kell felsorolnunk azokat az interfészeket, amelyeket az osztály implementál. A fordító hibát jelez, ha az interfészben lévő metódust elfelejtjük implementálni az osztályban.

public class Paper implements Writable, Erasable, Flat {

private String content = "";

private int maxContentSize = DEFAULT\_CONTENT\_SIZE;

public Paper() {

}

public Paper(int maxContentSize){ this.maxContentSize = maxContentSize; }

public boolean write(String text) { String newContent = content + text; if (newContent.length() <= maxcontentSize) { content = newContent; return true;

}

return false;

}

public void erase(){ content = "";

}

}

Interfész nem példányosítható, hiszen tényleges működéssel nem rendelkezik, de lehet egy változó statikus típusa. Ebben az esetben az objektumból csak az interfészben deklarált metódusok érhetőek el.

public static void main(String[] args) {

Paper paper = new Paper(200); Writable writable = paper;

Erasable erasable = paper;

writable.write("alma"); paper.write("körte");

System.out.println(paper.getContent()); // almakörte erasable.erase();

System.out.println(paper.getContent()); // (üres sor)

}

Mindhárom változó ugyanarra a Paper objektumra mutat, de a writable csak a write, az erasable csak az erase metódust éri el, míg a paper mindegyiket. Ennek igazán akkor látjuk hasznát, ha egy metódusunk például csak írni szeretné az objektum tartalmát anélkül, hogy tudná, milyen objektum is az valójában. Bármilyen jó neki, amelyik implementálja a Writable interfészt, hiszen ekkor biztosan meg tudja hívni rajta a write metódust.

public class Printer {

public boolean addContent(Writable writable, String content){ return writable.write(content);

}

public static void main(String[] args) {

Printer printer = new Printer(); Paper paper = new Paper(200); if (printer.addContent(paper, "Ezt írjuk a papírra.")) {

System.out.println("A papír nyomtatása sikerült");

}

}

}

Az interfészek valamelyest feloldják azt a szabályt, miszerint a Javaban csak egyszeres öröklődés van, mert így egy osztály sok interfésztől “örökölhet” metódusokat.

Mi történik, ha egy osztály két olyan interfészt implementál, mely ugyanolyan nevű metódust ír elő (névütközés)?

1. Ha a két metódus szignatúrája azonos, és ugyanazt a logikát akarjuk társítani hozzá, akkor nincs probléma.
2. Ha a két metódusnak más a paraméter szignatúrája, akkor mindkét metódust implementáljuk. (overload)
3. Ha a két metódusnak ugyanaz a szignatúrája, de más a visszatérési értéke (és ez nem oldható fel), akkor ezek sajnos ütköznek. Nem implementálhatjuk mindkét interfészt.

Ellenőrző kérdések

* Mit értünk interfész alatt? Milyen értelmezéseit ismered az interfész fogalomnak?
* Hogyan kell Javaban interfészt deklarálni?
* Hol használható később egy interfész?
* Mit jelent az, hogy egy osztály implementál egy interfészt?
* Egy osztály implementálhat-e több interfészt?
* Milyen attribútumokat használhatunk interfészekben?
* Ha nem adjuk meg explicit módon, milyen módosítók szerepelnek az attribútumnál implicit módon?
* Ha nem adjuk meg explicit módon, milyen módosítók szerepelnek a metódusoknál implicit módon?
* Hol láttál eddig interfészeket?
* Mit jelent interfészeknél a névütközés? Hogyan lehet kezelni?

Gyakorlat 1 - Létező interfész használata

A következőkben írunk egy olyan osztályt, amely lehetőséget teremt párhuzamos futtatásra. Ehhez implementálni kell a Runnable interfészt.

public interface Runnable {

public abstract void run();

}

Ennek az interfésznek csak egy void run() metódusa van, amit majd nekünk kell implementálnunk. (Ezt a feladatot ellenőrző teszt csak mint osztályt fogja használni, és nem fog tényleges szálat indítani, de valójában az is megtehető a (new Thread(new SimpleThread())).start() utasítással.)

Feladat egy olyan osztály készítése, amely egy szálban képes futni, megkapja a feladatok listáját, majd végrehajtja azokat (most csak rendre kiveszi a listából a feladatokat egyenként).

* Hozz létre egy SimpleThread nevű osztályt a interfaces.simplethread csomagban, amely implementálja a Runnable interfészt.
  + Hozd létre (generáltasd le) a megfelelő metódust, amelyet megkövetel az interfész.
  + Hozz létre egy tasks nevű privát String listát.
  + Hozz létre egy getter metódust a tasks attribútumhoz.
  + Írj egy konstruktort, ami megkapja a tasks listát kívülről, erre állítja be a tasks attribútum értékét
  + A részlépések végrehajtásához hozz létre egy private boolean nextStep() metódust, amely
* kiveszi a legutolsó elemet a tasks listából
* visszatérési értéke igaz, ha még van szöveg a listában, egyébként hamis.

– a run() metódusban mindaddig hívd a nextStep metódust, amíg van feladat (amíg a nextStep igazzal tér vissza)

Gyakorlat 2 - Saját interfész definiálása, implementálása, használata

Létrehozunk egy interfészt, implementáljuk azt több osztályban, majd használjuk anélkül, hogy tudnánk a pontos implementációt. A következő interfészt, és az osztályokat mind a interfaces.animal csomagba tedd.

* Hozz létre egy Animal interfészt, amely két metódust deklarál:
  + int getNumberofLegs(), amely visszaadja az állat lábainak a számát, – String getName(), amely visszaadja az állat nevét.
* Az első állat a kacsa (Duck) lesz, amely implementálja az Animal interfészt úgy, hogy a lábak száma kettő, a név pedig “Duck”.
* A második állat az oroszlán (Lion) lesz, amely implementálja az Animal interfészt úgy, hogy a lábak száma négy, a név pedig “Lion”
* A harmadik állat a féreg (Worm) legyen, amely implementálja az Animal interfészt úgy, hogy a lábak száma nulla, a név pedig “Worm”.
* Használd az előző állat példányokat az Animal interfész segítségével a Zoo osztályban. Miután létrehoztad a Zoo osztályt,
  + szükség lesz egy animals privát attribútumra, amelyben Animal interfészt implementáló példányok vannak. (a lista alap típusa Animal)
  + az állatok listáját kívülről adják majd meg, ezért szükség van egy olyan konstruktorra, melynek 1 paramétere van, amire beállítja az attribútumot.
  + hozz létre egy publikus metódust getNumberOfAnimal névvel, amely visszaadja, hogy hány állat található a területen (a lista mérete).
  + hozz létre egy publikus metódust getNumberOfLegs névvel, amely visszaadja, hogy összesen hány lába van az állatoknak (összegezd az állatok lábát használva a példányok getNumberofLegs metódusát).

Default interfész metódusok (interfacedefaultmethods)

Az interfész utólagos módosításának az a veszélye, hogy az összes őt implementáló osztályt is módosítani kell. A Java 8 előtt ezt úgy lehetett megoldani, hogy új interfészt származtattunk a régiből, amely tartalmazta az új metódusfejet, és ettől kezdve eldönthettük, hogy ezt vagy a régit implementáljuk egy osztály által. A Java 8 bevezette a default metódust, amely nem csak a metódus fejét, de az alapértelmezett működését is tartalmazza. Ezeket a metódusokat el kell látnunk a default módosítószóval, és nem tehetjük mellé a static, a final és az abstract módosítók egyikét sem. Természetesen az implementáló osztály dönthet úgy, hogy felülírja ezt a működést, de ha nem teszi, akkor sincs probléma. Ez által a régebben írt osztályok is működőképesek maradnak. Amennyiben az interfészből újabb interfészt származtatunk, akkor az dönthet úgy, hogy

* az eredeti default metódust meghagyja,
* absztrakttá teszi,
* illetve felül is írhatja másik default implementációval.

public interface HasName {

default String getName() { return "Anonymous";

}

}

public interface HasUniqueName extends HasName { // absztrakttá teszi a getName metódust

String getName();

}

public interface CanGetNewName extends HasName { // meghagyja az eredeti getName metódust

void setName(string newName);

}

public interface HasTwoNames extends HasName { // új implementációval látja el a getName metódust

default String getName() { return "John Doe";

}

}

A Java 9 verzió bevezette a privát metódusokat is az interfészekben. Mivel ezek nem írhatóak felül az implementáló osztály által, ezért ezeket mindig implementálnia kell, és a default módosítót sem használjuk vele.

Ellenőrző kérdések

* Miért vezették be a default interfész metódusokat?
* Milyen szabályok vonatkoznak a default interfész metódusokra?
* Leszármazott interfésznek milyen lehetőségei vannak a default interfész metódussal kapcsolatban?
* Mi problémába futhatunk többszörös öröklődésnél, és hogyan lehet feloldani?

Gyakorlat - Nyomtatható kiadványok

A feladat során egy nyomtatót és különböző nyomtatható anyagokat (újság, mesekönyv) kell implementálni. A nyomtató képes színesben nyomtatni. Az újság csak fekete-fehér lehet, míg a mesekönyv színes (különböző oldalak lehetnek különböző színűek). A nyomtató nyomtatáskor a színeket vezérlőkarakterekkel jelzi.

Írj egy Printable interfészt a getLength(), getPage(), és getColor() metódusokkal. A getColor() metódus default implementációja, hogy mindig feketét ad vissza, ami az interfészben egy konstans #000000 érték.

A Printable interfészt implementálja a NewsPaper és StoryBook osztály is. A NewsPaper osztály az oldalak tartalmát egy List<String> attribútumban tartsa nyilván. A StoryBook használjon egy ColoredPage immutable osztályt, mely az oldal tartalmát (String) és az oldal színét (String) tartalmazza. A StoryBook egy ilyen listát tartalmazzon, így minden oldalra megmondható, hogy milyen színű.

A Printer osztály print() metódusa menjen végig a nyomtatható anyag oldalain, és fűzze össze egy String-be, sortörésekkel elválasztva (használj StringBuilder-t). Amennyiben az oldal színes (azaz nem fekete-fehér), az oldal tartalma elé írja ki vezérlőkarakterekként az oldal színét. Azaz ha az oldal piros (#FF0000), és az oldal tartalma Content, akkor úgy fűzze hozzá a String-hez, hogy [#FF0000]Content.

Static interfész metódusok (interfacestaticmethods)

Az interfészben definiálhatunk statikus metódust is, amelyet mindig implementálnunk kell. Ez csak publikus lehet, nem látható el a default módosítóval, és nem öröklődik. Kizárólag az interfész nevével minősítve hívható.

A cél az, hogy az interfészbe helyezzük el azokat a metódusokat, amelyek az interfész különböző példányain dolgoznak, ne pedig egy különálló osztályba. Ilyen metódus a List.of() mely az átadott elemek listájával tér vissza, függetlenül a konkrét lista implementációtól. Ez gyakorlatilag kiváltja a régóta meglévő Arrays.asList() metódust, mely egy olyan konkrét osztály statikus metódusa, amelynek semmi köze nincs a listákhoz.

Java 9 óta a statikus metódus is lehet privát.

Ellenőrző kérdések

• Hogyan definiálhatunk statikus interfész metódust? Hogyan lehet meghívni?

Feladat

Értékek

Készíts egy Valued interfészt, mely egyetlen absztrakt metódust definiál. A getValue() metódus egy Valued példány értékét adja meg. Hozz létre az interfészben egy sum() metódust, mely paraméterként egy List<Valued> típusú adatot kap, és a benne található elemek összértékét számítja ki!

Interfészek és az öröklődés viszonya (interfaceextends)

Az interfészek között többszörös öröklődés van, azaz egy interfész több interfészt is kiterjeszthet. Ekkor az extends kulcsszó után a szülő interfészeket vesszővel elválasztva soroljuk fel. Az interfészek közötti kapcsolat nem hierarchikus, hanem hálós, de körkörös öröklődés nem lehet benne. Nincs olyan kitüntetett gyökéreleme, mint az osztályhierarchiának.

Egy osztály több interfészt is implementálhat, de dönthet úgy, hogy valamelyik metódust absztraknak hagyja, azonban ekkor az osztály is absztrakt lesz. Az első konkrét osztálynak minden örökölt absztrakt metódust implementálnia kell.

Fontos megjegyeznünk, hogy habár a specifikációk többszörösen öröklődnek, az implementációk csak egyszeresen. Ha két interfészből ugyanolyan szigantúrájú default metódust örököl az osztály, és azt nem írja felül, az futási idejű hibához vezethet, hiszen nem lehet eldönteni, melyiket kell futtatni. Az osztálynak mindenképp felül kell írnia a metódust. Ezt még akkor is meg kell tennie, ha a metódus csak az egyik interfészben rendelkezik default törzzsel, a másikban nem.

Ellenőrző kérdések

* Milyen kulcsszó szükséges az interfész öröklődéshez?
* Egy interfésznek hány ősinterfésze lehet?

Gyakorlat - Robotok implementálása interfész(ek) alapján

Egy játék során robotokat fogunk mozgatni. Alapvetően kétféle mozgást kell megvalósítani, gyaloglást és repülést. A szükséges alapműveleteket, mint funkciókat interfészekben definiáljuk. Az egyes konkrét robotok ezeket implementálják, ennek megfelelően fognak majd mozogni.

Az interfészek nem függetlenek egymástól, a MoveableRobot interfész leszármazottja a repülést megvalósító (esetünkben a felemelkedés funkciót leíró) FlyableRobot interfész. A robotok mozgását a koordinátákat tartalmazó Point osztály objektumai segítségével lehet követni.

MoveableRobot interfész:

definiált metódusok:

void moveTo(Point position); void fastMoveTo(Point position); void rotate(int angle); List<Point> getPath();

A getPath() metódus azoknak a pontoknak a listáját adja vissza, amelyeket a robot mozgása során érintett (a konkrét mozgató utasítások minden esetben bejegyzik a célként kapott Point objektumot).

FlyableRobot extends MoveableRobot interfész:

definiált metódus:

void liftTo(long altitude);

Point osztály final long x, long y és long z attribútumokkal. Csak FlyableRobot esetében kap a z attribútum nullától különböző értéket, a MoveableRobot síkban mozog.

publikus metódus:

public Point(long x, long y, long z)

Két robotot fogunk létrehozni, az egyik csak síkban mozog (C3PO), a másik repülni is tud (AstroBoy).

C3PO implements MovableRobot osztály Point position, int angle és List<Point> path attribútumokkal. publikus metódusok:

public C3PO(Point position) public void moveTo(Point position) public void fastMoveTo(Point position) public void rotate(int angle)

Tipp

Mivel C3PO nem tud sietni, esetében a moveTo() és fastMoveTo() metódusok ugyanúgy működnek. Célszerű egy privát metódust létrehozni (pl. void walkTo(Point position)), amely beállítja a robot új pozícióját és be is jegyzi ezt a path-ba.

AstroBoy implements FlyableRobot osztály Point position, int angle és List<Point> path attribútumokkal, valamint long ALTITUDE = 5 konstans értékkel; publikus metódusok:

public AstroBoy(Point position) public void walkTo(Point position) public void flyTo(Point position) public void liftTo(long altitude)

public void moveTo(Point position) // delegálja a funkciót public void fastMoveTo(Point position) //komplex mozgás, felemelkedik, elrepül a célpont fölé, és leereszkedik public void rotate(int angle)

Mindegyik mozgás bejegyzi, hogy milyen koordináták mentén történt.

Absztrakt metódusok implementálása felsorolásos típusokban (enumabstract)

Absztrakt metódust az enum típus is tartalmazhat. Mivel maga az enum nem maradhat absztrakt, minden egyes példánynál, a felsorolt konstansokban implementálni kell az adott metódust. Ezt a konstans neve utáni kapcsos zárójelek között tehetjük meg.

public enum TransactionState {

SUCCESS { @Override

public boolean isComplete() { return true; }

}, REJECTED { @Override

public boolean isComplete() { return true; }

}, PENDING { @Override

public boolean isComplete() { return false;

}

};

public abstract boolean isComplete();

}

Ugyanezt az eredményt érhetjük el azzal is, ha az isComplete() metódust kiemeljük egy interfészbe, és ezt az interfészt implementálja a TransactionState enum. Természetesen ebben az esetben sem maradhat absztakt a metódus, ezért vagy magában az enumben, vagy minden egyes példányban meg kell adnunk a működést.

public interface HasCompleteState { boolean isComplete();

}

public enum TransactionState implements HasCompleteState {

// ...

}

Ellenőrző kérdések

* Enum esetén hogyan lehetséges absztrakt metódus létrehozása?
* Implementálhat-e interfészeket egy enum? Ha igen, hogyan?

Feladat

Megrendelések

Egy webshop a megrendelések állapotát az OrderState enumben tárolja. Állapotai: \* NEW, ez az állapota minden újonnan leadott megrendelésnek. \* CONFIRMED, miután visszaigazolták a rendelést. \* PREPARED, miután a megrendelést összekészítették. \* ONBOARD, miután átadták a futárnak. \* DELIVERED, miután sikeresen kiszállították. \* RETURNED, sikertelen kiszállítási kísérlet után. \* DELETED, miután bármilyen okból a rendelést törölték.

Egy megrendelést addig lehet visszamondani, amíg még nem adták át a futárnak.

Készíts egy Deletable interfészt, és implementáld az enumban! Az interfész egyetlen absztrakt metódust tartalmazzon: boolean canDelete(). Az OrderState enum NEW, CONFIRMED és PREPARED értékei esetén igazat, minden más esetben hamisat kell visszaadjon.

Állapotgép (statemachine)

Az állapotgép tulajdonképpen egy olyan „szerkezet” amely különböző bemenetek hatására egyik állapotból a másikba lép. Fontos, hogy az állapotok száma véges, továbbá, hogy az állapotgép a saját állapota és a bement ismeretében egy állapotból csakis egy állapotba léphet át. Egy ilyen állapotgép például a jelzőlámpa, ahol a bemenet az idő múlása.

Az állapotgépet UML diagrammon nagyon egyszerűen lehet vizualizálni. Így felépíthető egy gráf ahol a csomópontok éppen az állapotok.

Gyakorlatban

Javaban állapotgép implementációjára nagyon jól használható a felsorolásos típus. Ennek elemi lehetnek az állapotok, valamit deklarálhatunk egy metódust ami ezek között az állapotok között vált.

public enum TrafficLight {

RED {

TrafficLight next() {

return TrafficLight.RED\_YELLOW;

}

},

RED\_YELLOW {

TrafficLight next() {

return TrafficLight.GREEN;

}

},

GREEN {

TrafficLight next() { return TrafficLight.YELLOW;

}

},

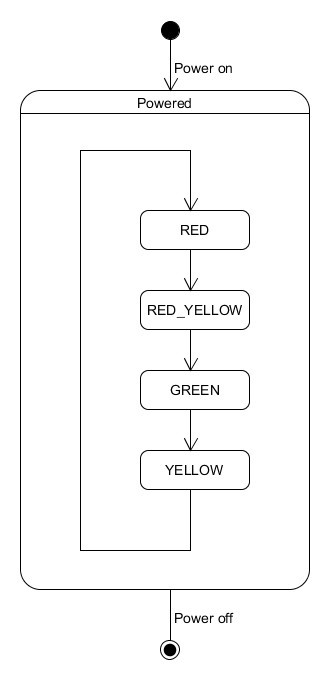
YELLOW { TrafficLight next() { return TrafficLight.RED;

}

};

abstract TrafficLight next();

}



UML állapotdiagram

Ellenőrző kérdések

* Hogyan működik az állapotgép?
* Hogyan váltunk át egyik állapotból a másikba?

Ülésfűtés gomb

Egy gombbal lehet állítani az autóban az ülésfűtést. Alapállapotban ki van kapcsolva. Ha ekkor megnyomjuk, akkor maximális (3-as) fokozatra kapcsol a fűtés, ha mégegyszer megnyomjuk, akkor 2-es fokozatra áll, ha mégegyszer, akkor 1-esre áll, és ha mégegyszer, akkor kikapcsol. Szimuláljuk a gomb működését, hogyan vált a 4 állapot között.

Közlekedési lámpa

A közlekedési lámpa (magyarországi) működését modellezzük. Piros után piros-sárga, utána zöld, majd sárga és ismét piros következik.

Elromlott az írógép

Képzeld el, hogy van egy régi hagyományos írógép, aminek a CAPS LOCK-ja elromlott és minden sor végén vált automatikusan. Ha kisbetűs sort írtak vele, akkor nagybetűre vált, ha nagybetűs sort írtak, akkor kisbetűre vált.

Bónusz feladat

Szimuláljuk egy egyszerű lift működését. Csak a földszint és az emelet között közlekedik. Négy állapota lehet:

* Földszint nyitott ajtóval
* Földszint csukott ajtóval
* Emelet nyitott ajtóval
* Emelet csukott ajtóval

Meg lehet hívni a liftet bármelyik szinten, illetve meg lehet nyomni a liftben a Földszint ill. az Emelet gombot. Ezekkel a lift haladási irányát adjuk meg. Vigyázz, mert lehet, hogy a lift aktuális iránya más, amikor meghívjuk a liftet.

## Haladó OO elvek

Polimorfizmus (polymorphism)

Statikus és dinamikus típus

A statikus típus az a típus, amellyel a változót deklaráljuk, a dinamikus típus pedig az, amellyel az adott objektumot példányosítjuk. A kettő megegyezhet, vagy a statikus típus lehet a dinamikus típusnak bármely ősosztálya, vagy olyan interfész amit a dinamikus típus implementál.

String s = new String("Word"); // Statikus és dinamikus típus is String Object o = new String("Word"); // Statikus típus Object, a dinamikus típus a String

A dinamikus kötés azt mondja meg, hogy mikor kerül eldöntésre, hogy az osztályhierarchiában metódus felülírás esetén melyik metódus kerül meghívásra. Javaban a dinamikus típus dönti el, futásidőben, hogy melyik metódus lesz érvényes.

Polimorfizmus

A szó többalakúságot jelent, azaz egy példányosított objektum több formában is megjelenhet.

Nyilvánvaló, hogy ezek közül az egyik az osztály, melyből példányosításra került (dinamikus típusa), de ezen kívül bármely ősosztályként is megjelenhet, akár Objectként is, sőt bármely implementált interfaceként is. Minden esteben az instanceof operátor igazat fog visszaadni.

Típuskényszerítés

Az osztályhierarchiában az „ős-felé” automatikus a típuskényszerítés, viszont másik irányba explicit módon nekünk kell megadnunk a típuskényszerítést.

Ellenőrző kérdések

* Hol használjuk a statikus és dinamikus típus fogalmakat?
* Mit jelent a statikus és dinamikus típus?
* Mi teheti lehetővé azt, hogy a kettőnek nem kell megfelelnie egymásnak?
* Mit jelent a polimorfizmus?
* Mire és hogyan használható az instanceof operátor?
* Referencia típusok között milyen típuskényszerítés lehetséges?

Dinamikus kötés (virtualmethod)

A dinamikus kötés a polimorfizmus egyik legfőbb jellemzője. Legyen adott egy A osztály és annak egy leszármazottja B osztály. Az A osztályban deklarálunk egy metódust melyet a B osztályban felülírunk. Az, hogy egy adott példány esetén melyik kerül meghívásra az futás időben dől el, és kizárólag az objektum dinamikus típusa határozza meg. Az ilyen metódusokat virtuális metódusoknak hívjuk.

Ha egy metódust “overrideolunk”, akkor annak láthatósága nem szűkíthető, csak bővíthető.

Példa: Adott a következő két osztály, nézzük meg, mikor melyik getFreeTime() metódus hívódik meg.

package virtualmethod.trainer;

public class Human implements HasName {

private static final int DEFAULT\_FREE\_TIME =4;

private String name;

public Human(String name) { this.name = name;

}

public String getName() {

return name;

}

public int getFreeTime() { return DEFAULT\_FREE\_TIME;

}

}

package virtualmethod.trainer;

import java.util.List;

public class Trainer extends Human {

private List<Course> courses;

public Trainer(String name, List<Course> courses){ super(name); this.courses=courses;

}

@Override

public int getFreeTime() {

return Math.max(super.getFreeTime() - courses.size(),0); }

}

És most nézzük az ezekhez tartozó (helyes) teszteket.

@Test public void testFreeTimeByHuman() { Human human= new Human("John Doe");

assertThat(human.getFreeTime(),equalTo(4));

}

@Test public void testFreeTimeByTrainer() {

Trainer trainer = new Trainer("John Doe", Arrays.asList(new Course("Course1")));

assertThat(trainer.getFreeTime(), equalTo(3));

Human human = new Trainer("John Doe", Arrays.asList(new Course("Course1")));

assertThat(trainer.getFreeTime(), equalTo(3));

}

Első két esetben a válasz elég egyértelmű, viszont harmadik esetben amikor az objektum statikus típusa Human, dinamikus típusa pedig Trainer, akkor jól látható a teszteseten, hogy a Trainer osztályban lévő metódus hívódik meg, tehát a dinamikus típusban lévő!

Ellenőrző kérdések

* Mit jelent a dinamikus kötés?
* Mit jelent, hogy virtuális metódus?
* Override esetén a metódus nevére, paramétereire, visszatérési típusára, láthatósági módosítószóira és a dobott kivételekre milyen szabályok vonatkoznak?

Gyakorlat 1 - Személykocsi, teherkocsi modell

Hozz létre egy Vehicle osztályt. Az általános jármű osztálynak van önsúlya, és legalább egy vezető. Ezek adják a teljes súlyát.

* Vezess be egy vehicleWeight privát attribútumot, a jármű súlya.
* PERSON\_AVERAGE\_WEIGHT konstans érték: egy személy átlagos súlyát tartalmazza. Az értéke legyen 75.
* Hozz létre egy konstruktort, amely megkap egy értéket az attribútum számára, és beállítja azt.
* Szükséges metódus a getGrossLoad, amely visszaadja a mozgó jármű súlyát. (A jármű súlyához adjuk hozzá a sofőr súlyát.)

Hozz létre egy Car osztályt, amely az általánosabb jármű osztályból származik (Vehicle) a következők alapján

* tartalmaz egy attribútumot
  + numberPassenger: egész szám, amely az utasok számát jelenti (sofőr nélkül)
* konstruktor, amely megkap két értéket az attribútumok számára, és beállítja azokat (az ős attribútumát a super hívással).
* getGrossLoad visszaadja a mozgó gépkocsi súlyát. Hívd az ős azonos nevű metódusát, és add hozzá az utasok súlyát (Ez lesz a visszatérési érték).
* Definiáld felül a toString metódust, a következő formára:
  + Car{numberOfPassenger=4, vehicleWeight=1700}

Hozz létre egy Van osztályt, amely egy kisteherautót modellez. Ez az osztály az általános autóból származik, kiterjeszti a Car osztályt.

* Tartalmaz egy cargoWeight egész attribútumot a rakomány súlyának tárolására.
* Definiálj egy konstruktort, amely megkapja a rakománysúlyt, és az ős osztály két attribútumához szükséges értékeket is (összesen három egész érték). Hívd a supert az ős attribútumok inicilizálására, és állítsa be az új attribútumot is.
* Definiáld felül a getGrossLoad metódust. Hívd az ős azonos nevű metódusát, és add hozzá a rakomány súlyát (Ez lesz a visszatérési érték).
* Definiáld felül a toString metódust, a következő formára:
  + Van{cargoWeight=1222, numberOfPassenger=4, vehicleWeight=1200}

Virtuális metódusok használata (Mindig az hívódik, amelyikre szükség van.)

Gyakorlat 2 - FerryBoat

A komp FerryBoat képes bármilyen autót tárolni, aminek kisebb a súlya a megengedett összsúlynál. Viszont a komp is egy speciális jármű.

* Definiálj egy állandót MAX\_CARRY\_WEIGHT néven, amely tárolja a maximálisan szállítható autó súlyát.
* Ha szállít autót, akkor azt egy Car típusú car attribútumba tárold. (Ha nem szállít, akkor ez null.)
* Hozz létre egy konstruktort, amely megkap egy egész értéket, amivel meghívod a super-t.
* Definiáld felül a getGrossLoad metódust. Hívd az ős azonos nevű metódusát, és add hozzá a szállított autó súlyát (Ez lesz a visszatérési érték).
* Definiálj egy canCarry metódust, ami igazat ad vissza, ha a paraméterben kapott autót szállíthatja, azaz a szállítandó autó súlya kisebb, mint a MAX\_CARRY\_WEIGHT (egyébként hamis). (Akár autó, akár kisbusz a referencia célja mindig a megfelelő metódus hívódik meg.)
* Definiálj egy load metódust, ami igazat ad vissza, ha a paraméterben kapott autót berakodta, azaz a súlya kisebb, mint a MAX\_CARRY\_WEIGHT (ekkor tárolja el az autót a referenciába).
* Definiáld felül a toString metódust, a következő formára FerryBoat= + a tárolt autó toString eredménye:
  + FerryBoat{car=Van{cargoWeight=200, numberOfPassenger=1, vehicleWeight=1200}}
  + FerryBoat{car=Car{numberOfPassenger=1, vehicleWeight=1200}}

is-a has-a kapcsolatok (isahasa) is-a reláció

Az is-a kapcsolat azt jelenti, hogy egy objektum példánya saját osztályának és az összes ősének, és az összes interfésznek, melyet ezen osztályok implementálnak. Az instanceof operátorral kérdezhető le. Ez a kapcsolat statikus.

has-a reláció

A has-a kapcsolat azt jelenti, hogy egy osztály egy másik osztályra attribútumaként hivatkozik. Ezt kompozíciónak is nevezzük. A kapcsolat dinamikus, tehát futás közben változtatható. Amit szokás megvizsgálni, az a számosság, azaz hogy egy adott osztályból hány példány kapcsolódhat az osztályunkhoz. A kötelezőség annak vizsgálata, hogy lehet-e olyan eset, hogy egyetlen példány sem kapcsolódik. Valamint az irány, hogy mely osztályból példányából tudjuk elérni a másik osztály példányát, azaz merről van hivatkozás a másik példányra.

Öröklődés helyett kompozíció

Manapság a has-a reláció az elterjedtebb, ugyanis az dinamikus. Tehát inkább tartalmazási viszonyt fogalmazunk meg, mint leszármazottit. Ennek oka az újrafelhasználhatóság.

Mi alapján döntünk?

* Meg kell vizsgálni, hogy valóban is-a kapcsolatról van-e szó
* Tényleg bővítettük az osztályt?
* Csak a polimorfizmus ne használjunk is-a kapcsolatot, inkább interfészt használjunk.

Ellenőrző kérdések

* Melyik operátorral tudjuk lekérdezni, hogy a referált objektum altípusa-e az adott típusnak?
* Hogyan implementáljuk az is-a relációt?
* Hogyan implementáljuk a has-a relációt?
* Miért jobb a has-a reláció az is-a relációnál?

Feladat Html dekorátor

Feladat egy html objektum forrássá alakítása osztályokkal. Származtatással a sima szövegből több szöveg is specializálódik vastag, dőlt, aláhúzott. Megnézzük öröklődéssel, majd utána tartalmazással (dekorátor minta használatával)

Deklarálj egy TextSource interfészt az isahasa csomagban, melyben csak egy metódus van: String getPlainText().

Hozz létre egy HtmlText osztályt az isahasa csomagban, ami implementálja a

TextSource interfészt. Egy plainText attribútumban tárolja a kívánt szöveget, melyet a konstruktorban kap meg.

* a getPlainText() ebben az esetben csak visszaadja a tárolt szöveget.

Hozz létre egy Channel interfészt, ami csak egy metódust tartalmaz: int writeByte(byte[] bytes)

Szükség van még egy kliens osztályra, amely kap egy TextSource-ot és ráírja a csatornára a tartalmát.

* A csatornáját egy attribútumban tárolja.
* Konstruktorban megkapja a csatorna referenciát, amit eltárol az attribútumban.
* definiál egy publikus writeToChannel metódust, ami megkap egy TextSource referenciát. A paraméter tartalmát lekéri a getPlainText metódussal, amit átalakít byte tömbbé (String osztálynak van ilyen metódusa), és ezzel hívja a csatorna writeByte metódusát.

Első megoldás származtatással

A BoldHtmlText a vastagon szedett szöveget megvalósító osztály az isa csomagban van, és a HtmlText-ből származik.

* a konstruktorban megkapott szöveget a super-nek adjuk át, így inicializáljuk.
* a getPlainText <b> és </b> közzé fogja az ősben definiált getPlainText eredményét.

Hasonlóan kell megvalósítani a ItalicHtmlText osztályt, de ez a <i> és </i> tageket használja. A UnderlinedHtmlText osztály az <u> és </u> tageket használja. A ItalicAndBoldHtmlText a <i><b> és </b></i> tagek kombinációját használja. A UnderlinedAndItalicAndBoldHtmlText a <u><i><b> és </b></i></u> tagek kombinációját használja.

Vegyük észre, hogy minden variációra külön osztály kell (pl.: vastagbetűs és aláhúzott és dőlt).

Második megoldás tartalmazással

Most oldjuk meg ezt a feladatot tartalmazással, a dekorátor minta segítségével.

Hozzunk létre egy TextDecorator abstract osztályt (a hasa csomagban), amely implementálja a konkrét dekorátoroknak a tartalmazás kapcsolatot.

* A TextSource interfészt implementálja. (az előírt metódust nem definiálja felül, ezért is lesz absztrakt)
* egy védett attribútuma van: TextSource textSource

Bold osztály (ami egy konkrét dekorátor a hasa csomagban) a TextDecorator osztályból származik.

* definiál egy konstruktort, amely megkap egy TextSource referenciát, amire beállítja az örökölt textSource attribútumot.
* implementálja az őstől kapott getPlainText metódust úgy, hogy a tartalmazott textSource referenciája meghívja a getPlainText metódust, és az értéket közrezárja a <b> és </b> tag-ek közé. (Ez még nagyon hasonló a származás megoldásához. Különbség az, hogy ott a super-en hívtuk meg a getPlainText metódusát, itt pedig az attribútumon.)

Italic osztály (ami egy konkrét dekorátor a hasa csomagban) a TextDecorator osztályból származik.

* definiál egy konstruktort, amely megkap egy TextSource referenciát, amire beállítja az örökölt textSource attribútumot.
* implementálja az őstől kapott getPlainText metódust úgy, hogy a tartalmazott textSource referenciája meghívja a getPlainText metódust, és az értéket közrezárja a <i> és </i> tag-ek közé.

Underlined osztály (ami egy konkrét dekorátor a hasa csomagban) a TextDecorator osztályból származik.

* definiál egy konstruktort, amely megkap egy TextSource referenciát, amire beállítja az örökölt textSource attribútumot.
* implementálja az őstől kapott getPlainText metódust úgy, hogy a tartalmazott textSource referenciája meghívja a getPlainText metódust, és az értéket közrezárja a <u> és </u> tag-ek közé.

Az előbbi dekorátorok létrehozásának paramétere lehet az alap HtmlText, de lehet bármelyik dekorátor is, hiszen mindegyik implementálja a TextSource interfészt. Ezért a dőlt vastagbetűs szöveget a dekorátorok láncolásásval megoldhatjuk, nem kell új osztály. Ugyanígy a többi variációra sem kell. Az előbbi dekorátorokkal azok bármilyen kombinációja láncolható.

Feladat - Flotta

A flottában vegyesen vannak teherszállító hajók (cargo ship), személyszállító hajók (liner) és kompok (ferry boat). A kompok személyeket és terhet is szállíthatnak. Ha a flotta behajóz, akkor folyamatosan töltik fel a hajókat, mindaddig, amíg meg nem telnek, el nem fogy az utas, vagy teher.

Hozz létre egy Ship interfészt, mely a hajót jelöli (marker interfész, metódus nélkül), egy CanCarryGoods és CanCarryPassengers interfészt, mely azt jelöli, hogy egy hajó tud-e személyeket, vagy terhet szállítani. A CanCarryGoods interfészben hozz létre egy int loadCargo(int cargoWeight) és int getCargoWeight() metódust. A

CanCarryPassengers interfészben egy int loadPassenger(int passengers) és egy int getPassengers() metódust.

A CanCarryGoodsBehaviour implementálja a CanCarryGoods interfészt. Ennek az osztálynak két attribútuma van: int cargoWeight és int maxCargoWeight.

Implementáld a loadCargo(int weight) metódust, melynek paramétere a betöldendő rakomány súlya, és a weight attribútumba eltárolja a letárolt rakományt, és visszatér a le nem tárolt rakomány súlyával.

Vezessünk be egy CanCarryPassengersBehaviour osztályt, ami implementálja a CanCarryPassengers interfészt. Ennek az osztálynak két attribútuma van: int passengers és int maxPassengers. Implementáld a loadPassenger metódust, melynek paramétere a beszálló utasok száma, és a passengers attribútumba letárolja a beszállt utasok számát, és visszatér a kintmaradó utasok számával.

A Liner osztály implementálja a Ship és a CanCarryPassengers interfészt, és legyen egy CanCarryPassengers típusú attribútuma.

A CargoShip osztály implementálja a Ship és a CanCarryGoods interfészt, és legyen egy CanCarryGoods típusú attribútuma.

A FerryBoat osztály implementálja a Ship, CanCarryGoods, CanCarryPassengers interfészt, és legyen mindkét típusú attribútuma.

Mindhárom osztálynak legyen olyan konstruktora, mely elvárja a CanXxxBehaviour osztályok konstruktorában elvárt adatokat.

Minden szükséges (interfész által kikényszerített) metódust úgy implementálj, hogy delegáld a kérést a megfelelő attribútum megfelelő metódusának.

Azaz pl. a FerryBoat osztály loadCargo() metódusa hívja a CanCarryGoods loadCargo() metódusát.

Készíts egy Fleet osztályt, melynek van egy List<Ship> attribútuma, mely a hajókat tartalmazza. A loadShip() metódusa végigmegy a listán, és sorban feltölti a hajókat a személyekkel és terhekkel. A maradék személyeket és terheket (melyek nem fértek el) a waitingPersons és waitingCargo attribútumokban tárolja el. Ez utóbbiakhoz generálj getter metódusokat is.

# Kivétel- és fájlkezelés

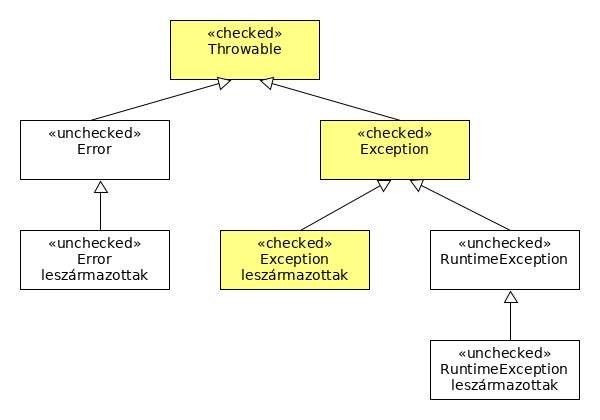
## Kivételkezelés

Kivételkezelés (exceptions)

Kivételek hierarchiája

A kivételek hierarchiájának tetején a java.lang.Throwable osztály áll. Két leszármazottja van:

* java.lang.Error olyan hiba, amiből nem lehet visszaállni (nem kötelező kezelni) java.lang.Exception olyan kivétel, amit kezelni kell vagy lehet - ennek leszármazottja a java.lang.RuntimeException, melyet nem kötelező kezelni



Exception hierarchia

try-catch szerkezet

* A try blokkban szerepelnek azok az utasítások amiket vizsgálni szeretnénk. A catch ág(ak)ban szerepel a kivételkezelés, a kivételt lekezelő utasítások. Itt Exception-öket kell megadni. Ha ezek egymástól függetlenek akkor mindegy a sorrend, ha van hierarchia az Exception-ök között akkor először a legspeciálisabbal kell kezdeni. A finally ág minden esetben végrehajtódik, tipikusan erőforrás lezárására használjuk

Kivétel továbbdobása

A kivételt nem feltétlenül kezeljük, hanem tovább is dobhatjuk egy olyan modulnak, kódrészletnek, ami már tud a kivétellel mit kezdeni. Ehhez használjuk a throw kulcsszót vagy példányosítunk egy új kivételt, és ebbe a kivételbe burkoljuk, csomagoljuk az eredeti kivételt.

Gyakori kivételek

* ArithmeticException – például nullával való osztás esetén ArrayIndexOutOfBoundsException – tömb túlindexelés esetén

ClassCastException – explicit konverziónál NumberFormatException – Stringből számot próbálunk konvertálni, de ezt nem lehet megcsinálni, mert pl. betű szerepel benne StackOverflowError – végtelen számú egymásba ágyazott metódushívás OutOfMemoryError – elfogy a JVM memóriája

Ajánlott gyakorlati megoldások (best practice)

* Ne szerepeljen üres catch ág. Egy kivétellel mindenképpen kezdeni kell valamit, legalább naplózzuk le, ha nem tudjuk lekezelni vagy továbbdobni. Mindenképpen használjunk minél speciálisabb kivételeket. Egy metódusban legfeljebb egy trycatch blokkot használjunk. Fontos, hogy vagy naplózzunk vagy dobjuk tovább a kivételt, egyszerre mind a kettőt ne tegyük, hiszen egy kivétel így többször is naplózásra kerülhet. Manapság a nem kezelendő kivételek az elterjedtebbek.

Ellenőrző kérdések

* Hogyan néz ki a try utasítás?
* Hogyan működik a try utasítás?
* Hogyan lehet továbbdobni egy kivételt?

Gyakorlat 1 - Polinom példa

Készítsen egy Polinom osztályt, amelynek segítségével tetszőleges polynomial értékét ki tudjuk számolni adott x értéknél. A polinomot az együtthatóinak a tömbje írja le. A polinomot lehessen inicializálni az együtthatók tömbjével (double[]), de lehessen inicializálni String[] -el is (pl. beolvasott értékek). Az osztálynak legyen egy public double evaluate(double x) metódusa, amely adott x értéknél visszaadja a polynomial értékét. Valamint legyen getter az együtthatók tömbjére. A double[] paraméterű konstruktor dobjon NullPointerException kivételt, ha a paraméter null. A String[] paraméterű konstruktor dobjon NullPointerException kivételt, ha a paraméter null, valamint dobjon IllegalArgumentException-t ha az egyes String-ek nem alakíthatók double számmá. Mi az a polinom?

https://hu.wikipedia.org/wiki/Polinom

Bemeneti adatok ellenőrzése példa

Egy felhasználóktól nyert adatokat kell feldolgozni és a hibás sorokról jelentést készíteni. Az bemeneti adatok sorai tartalmazhatnak megjegyzésbe tett sorokat is, tehát lehet benne adatsor és lehet megjegyzés sor.

Az adatsorok szerkezete: sorszám, mértérték, mérésdátum. Pl. 12, 34.5, 2014.05.22. A megjegyzésbe tett sorok ugyanúgy , karakterrel elválasztott három részből állnak, csak az első rész nem alakítható számmá. Pl. M12, 12, 2014.01.01. A hiba jelentés tartalmazzon bejegyzést minden olyan sorról, amely nem megjegyzésbe tett és hiba van benne. A jelentés egy List<String> legyen, ahol a String tartalmazza a hibás sor sorszámát és a hiba kódját, sorszam, hibakód alakban. Figyelem, a hibaüzenetben a sorban szereplő sorszámot kell kiírni, és nem azt a számot (indexet), amelyik pozíción szerepel az adott sor.

Hibakódok:

* 2: WORDCOUNT\_ERROR, azaz a sor nem bontható 3 db , karakterrel elválaszható részre.
* 4: VALUE\_ERROR, azaz a második rész nem double szám.
* 8: DATE\_ERROR, azaz a harmadik rész nem yyyy.MM.dd. alakú dátum
* 12: VALUE\_AND\_DATE\_ERROR: azaz egyszerre van VALUE\_ERROR és DATE\_ERROR is.

Megoldáshoz további részletek

* Készíts egy FaultList osztályt, amelynek van egy public List<String> processLines(List<String> lines) metódusa
* A hibakódokhoz célszerű egy enumot definiálni
* Egy sor feldolgozására célszerű egy private metódust készíteni, ami visszaadja a sor feldolgozás eredményét (az eredmény vagy a hibakódok valamelyike vagy NO\_ERROR vagy COMMENT)

Saját kivétel (exceptionclass)

Saját kivétel implementálásánál először el kell tűnődni azon, hogy az adott kivétel nincse még implementálva, valamint azon, hogy biztosan kivételes eset keletkezik-e.

Csak Exception-t írunk Error-t nem, és azon belül is inkább a nem kezelendő kivétel írása az elterjedtebb. Miután ezeket átgondoltuk, jöhet a származtatás például a RuntimeException osztályból.

Konvenciók

Egy kivételnél tipikusan két dolgot adhatunk meg. Az egyik egy üzenet, aminél fontos, hogy minél informatívabb legyen. A másik a kiváltó kivétel megadása.

A kivételek általában módosíthatatlanok, ezért tipikusan konstruktor(ok)ban inicializáljuk. A kiváltó kivétel már deklarálva van az ősosztályban ezért a konstruktorban super hívással tudunk hivatkozni az ősben lévő konstruktorra, így nem kell tárolnunk a kiváltó kivételt és az üzenetet.

Példa:

public class InvalideAgeException extends RuntimeException {

private int parameterAge;

private int minAge;

public InvalideAgeException(String message, int parameterAge, int minAge){

super(message); this.minAge = minAge;

this.parameterAge = parameterAge; }

public int getParameterAge() { return parameterAge;

}

public int getMinAge() { return minAge;

}

}

Ellenőrző kérdések

* Milyen kivételfajták léteznek?
* Hogyan hozhatunk létre saját kivétel osztályokat?
* Miért hozzunk létre saját kivétel osztályokat?

Gyakorlat - SimpleTime példa

Készíts egy SimpleTime osztályt, amely egyszerűsített időpont reprezentáló osztály.

* Lehet létrehozni óra és perc megadásával és lehet időpontot megadni “hh:mm” alakú String-el is.
* Legyen felüldefiniálva a toString úgy, hogy “hh:mm” alakú időt adjon.
* Legyen getHour, és getMinute metódusa is.

Hibakezelés:

Definiálj egy saját InvalidTimeException-t, amely RuntimeException leszármazott. Dobjon InvalidTimeException-t “Hour is invalid (0-23)” szöveggel, ha a konstruktornak nem megfelelő óra értéket adnak meg. Dobjon InvalidTimeException-t “Minute is invalid (0-59)” szöveggel ha a perc hibás. Dobjon InvalidTimeException-t “Time is null” szöveggel ha null String-et adnak meg. Dobjon InvalidTimeException-t “Time is not hh:mm”, ha érvénytelen a String formátuma.

Készíts egy Course osztályt. A kurzusnak van neve (name) és kezdete (begin), ami SimpleTime típusú. Legyen konstruktora, ahol megkapja az adatokat, legyenek getterei, valamint legyen felüldefiniálva a toString-je úgy, hogy hh:mm: kurzusnév alakú legyen.

Bank példa

Készíts egy Bank osztályt. A Bank számlákat (Account-okat) tárol egy listában.

* az Account listát konstruktorban tudja megkapni
* képes adott számlaszámú Account egyenlegét csökkenteni egy megadott összeggel,
* képes adott számlaszámú Account egyenlegét növelni

Egy Account attribútumai:

* számlaszám (accountNumber, String),
* egyenleg (balance, double) (a valóságban inkább BigDecimal, de most az egyszerűség kedvéért legyen double).
* max levonható összeg (maxSubtract, double) Egy Account műveletei:
* példányosítás: számlaszám és egyenleg megadásával, a maxSubtract legyen 100000
* getterek
* setter a maxSubtract attribútumra
* subtract: egyenleg csökkentése egy megadott értékkel
* deposit: egyenleg növelése egy megadott értékkel

Hibakezelés

Legyen egy InvalidBankOperationException, amely RuntimeException leszármazott és van egy ErrorCode attribútuma, amely egy enum LOW\_BALANCE, INVALID\_AMOUNT, INVALID\_ACCOUNTNUMBER konstansokkal. Az Account és a Bank osztály dobjon ilyen kivételt a megfelelő értékkel inicializálva, ha

* nincs elegendő egyenleg a csökkentéshez,
* valamelyik metódusnak érvénytelen összeget adnak meg,
* a keresett számlaszám nincs meg

Dobj IllegalArgumentException kivételt, ha a Bank konstruktora null listát kap vagy az Account null számlaszámot.

Bank példa más hibakezeléssel

A feladat ugyanaz, mint az előbb, de a hibakezelés eltér. Legyen most minden hibafajtára

külön hiba osztály, amely egy közös osztály leszármazottja. Azaz legyen

InvalidBankOperationException (a közös ős), valamint

InvalidAccountNumberBankOperationException,

InvalidAmountBankOperationException, LowBalanceBankOperationException.

(Mik ezen megoldások előnyei, hátrányai?)

Multi-catch (exceptionmulticatch)

Gyakran előfordul, hogy több kivételt is le szeretnénk kezelni egy blokkban, de ezek a kivételek semmilyen kapcsolatban nem állnak egymással, csupán mindegyik az Exception osztály leszármazottja. Mit tehetünk ilyenkor?

* Duplikálni nem célszerű
* Multi catch, amikor egy catch ágban több kivételt is le tudunk kezelni, úgynevezett pipe (|) karakterrel elválasztva, egy névvel.

Nézzük hogyan:

public class TrainerParser {

public static final String SEPARATOR =";";

public Trainer parse(String line) {

try {

String[] fields = line.split(SEPARATOR);

Trainer trainer = new Trainer(fields[0],

Integer.parseInt(field[1]));

} catch (NullPointerException | IllegalArgumentException | IndexOutOfBoundsException e) {

throw new ParseException("Invalid line = " + line, 0);

}

}

}

Ellenőrző kérdések

* Mit szokás csinálni az elkapott kivételekkel?
* Mikor kapjunk el egy catch ágban többféle kivételt?

Gyakorlat - Converter példa

Adatbázisban kódolva tárolunk több igaz, hamis értéket egyetlen szöveges adatban, ahol ‘0’ karakter a hamis és ‘1’ karakter az igaz. Készíteni kell tehát konvertert, amely átalakítja a szöveges adatot boolean tömb adattá. A konvertáláshoz két osztályt is kell készíteni. Legyen egy BinaryStringConverter osztály, amelynek van public boolean[] binaryStringToBooleanArray(String) és egy public String booleanArrayToBinaryString(boolean[]) metódusa. A String-ből konvertáló IllegalArgumentException-t dob, ha a String-ben nem csupa 0 és 1 van. A boolean[]ből konvertáló pedig IllegalArgumentException-t dob, ha a tömb üres.

Majd pedig kell egy AnswerStat osztály, amely az igaz/hamis adatokon számol statisztikát, jelen esetben az igaz értékek százalékos arányát.

* legyen egy convert metódus, amely a binaryStringToBooleanArray metódus hívását csomagolja be. Azaz elkapja a konvertálás során előforduló

NullPointerException-t vagy IllegalArgumentException-t és logolja a hibát, majd tovább dobja becsomagolva egy InvalidBinaryStringException-be, amely egy saját RuntimeException-ból származó kivételosztály. Mivel mindkét hibafajtára ugyanazt kell csinálni, használja a multi-catch-et.

* Legyen egy int answerTruePercent(String answers) metódusa, amely a paramétert boolean tömbbé alakítja, majd meghatározza és visszaadja az igaz értékek százalékos arányát egészre kerekítve.
* A konstruktor paraméterben kapja meg a BinaryStringConverter-t.

Try-With-Resources szerkezet (exceptionresource)

Sokszor előfordul, hogy egy kivétel keletkezése után valamilyen erőforrást le kell zárnunk, ezt megoldhatjuk a finally ágban, de előfordulhat az is, hogy maga az erőforrás lezárása során keletkezik kivétel.

Megoldás a Try-With-Resources használata. Amennyiben a try-nak a fejlécében erőforrásokat deklarálunk, akkor JVM ezeket automatikusan lezárja. Ezt automatic resource menagementnek is nevezzük. Ekkor nem kell sem catch sem finally ágakat írnunk, hiszen ezt úgy képzeljük el, mint egy implicit finally blokk.

Ha változókat deklarálunk (akár többet is) a try fejlécében, akkor ezek a változók csak a try blokkban érhetőek el.

AutoCloseable interfész

Minden erőforrásnak implementálnia kell az AutoClosable interfészt. Ebben az interfészben található a close() metódus, amit a Java virtuális gép hív meg. Ez a metódus dobhat kivételeket.

Mi történik, ha kivétel keletkezik? A catch ágban le tudjuk kezelni, vagy tovább dobni.

Mi történik, ha a try blokkban is kivétel keletkezik? Ekkor a JVM az eredeti kivételt dobja, és hozzá fűzi a close() által dobott kivételeket. Ez az úgy nevezett supressed exception.

Példa a Try-With-Resources szerkezetre:

try (BufferedReader reader = new BufferedReader(new StringReader(values)))

{

while ((line = reader.readLine()) != null) { Trainer trainer = parseLine(line); trainers.add(trainer);

}

catch (IOException ioe) {

throw new IllegalStateException("Error by parsing, general io", ioe); }

}

Ellenőrző kérdések

* Mi a finally rész szerepe erőforrás kezeléskor?
* Hogyan működik a try paraméteres alakja?

## Fájlkezelés

Szöveges állomány beolvasása (ioreadstring)

A Path interfész

A Java 7-ben megjelent Path interfész egy könyvtárat vagy fájlt reprezentál. Egy Path típusú objektum a Java 11 óta a Path.of() statikus metódussal hozható létre, mely egy elérési útvonalat tartalmazó szöveget vár paraméterként. Ez lehet abszolút vagy relatív útvonal is.

Path fileInRootDirectory = Path.of("C:\\employees.txt");

Path fileInActualDirectory = Path.of("employees.txt");

Az útvonal elválasztó karaktere rendszerfüggő, ezért nem a legjobb megoldás, ha az beleégetjük a kódba. Ez a karakter a FileSystems.getDefault().getSeparator() metódussal lekérdezhető, de maga a Path is tartalmaz olyan metódusokat, amellyel elkerülhető az elválasztó explicit használata.

Path path = Path.of("employees", "john-doe.txt")

Path file = Path.of("employees").resolve("a.dat");

Az első esetben a path változó egy relatív útvonalat tart az employees mappában lévő john-doe.txt fájlra. A második esetben az employees könyvtárra mutató Path objektumot kombináljuk a resolve() metódusnak átadott útvonallal, amely így végeredményben az employees könyvtárban lévő a.dat nevű fájlra mutat.

Fájl tartalmának beolvasása

A Files osztály az állományok kezelésével kapcsolatos metódusokat tartalmaz. Ezek közül több is a szöveges fájlok tartalmának beolvasását végzi el nekünk. A readString() statikus metódusa a paraméterként átadott Path objektum által hivatkozott fájlt egyetlen komplett szövegként olvassa be, és Stringként adja vissza.

Path file = Path.of("employees.txt"); try {

String employees = Files.readString(file);

System.out.println(employees);

}

catch (IOException ioe) {

throw new IllegalStateException("Can not read file", ioe);

}

Ha olvasás közben bármilyen hiba történik, a metódus IOException kivételt dob. Az olvasott adatok karakterként értelmezése alapértelmezetten UTF-8 kódolás szerint történik. Amennyiben a fájl más kódolással készült, akkor a readString() metódus második paramétereként megadhatunk egy Charset objektumot is. Ezt kétféleképpen is létrehozhatunk: \* a Charset.forName() statikus metódussal, mely paraméterként a karakterkészlet nevét várja, vagy \* a StandardCharsets osztályban található konstansokkal.

Charset latin2 = Charset.forName("ISO-8859-2");

Charset utf8 = StandardCharsets.UTF\_8;

String employees = Files.readString(file, latin2);

Amennyiben soronként külön-külön szeretnénk látni a fájl tartalmát, akkor használjuk a Files.readAllLines() metódust! Ez List<String> objektummal tér vissza. Hasonlóan az előzőhöz ez is Path objektumot vár paraméterként, hiba esetén pedig IOException kivételt dob.

try {

List<String> employees = Files.readAllLines(file); for (String employee: employees) {

System.out.println(employee);

}

}

catch (IOException ioe) {

throw new IllegalStateException("Can not read file", ioe);

}

Ellenőrző kérdések

* Hogyan lehet rendszerfüggetlenül létrehozni egy Path objektumot?
* Hogyan tudod egy szöveges fájl tartalmát egyszerre beolvasni egy változóba?
* Mi az alapértelmezett karakterkódolás, és hogyan lehet más kódolású fájlokat is beolvasni?

Feladat

Emberek

A feladat egy szöveges állományból nevek beolvasása és eltárolása egy listába. A megoldáshoz két osztály kell megvalósítanod. A Human osztály reprezentál egy embert. Két adattagja vezeték- illetve keresztnév. A FileManager osztály felelős a fájl feldolgozásért. Egy Path típusú attribútumon keresztül érjük el a fájlt, míg a readFromFile() metódus felelős a beolvasásért és a Human objektumok létrehozásáért.

Banki tranzakciók

Ehhez a feladathoz két fájl tartalmát is fel kell dolgoznod. Az accouts.txt tartalmazza a bankszámla(BankAccount) adatokat. A transactions.txt állomány tartalmazza az utalásokat azaz, hogy melyik számlára mennyit utaltunk. A feladat, hogy olvasd be a számlaadatokat egy listába, majd egy másik metódusban hajtsd végre a tranzakciókat!

String kiírása szöveges állományba (iowritestring)

Szöveges fájl írása szintén a Files osztály segítségével történik. A writeString() metódusának átadva a fájl elérhetőségét és a bele írandó szöveget az egy lépésben kiírja a teljes szöveget. Bármilyen hiba esetén a metódus IOException kivételt dob.

Path file = Path.of("employees.txt"); try {

Files.writeString(file, "John Doe\nJane Doe\n");

}

catch (IOException ioe) {

throw new IllegalStateException("Can not write file", ioe);

}

A fájl alapértelmezett kódulása UTF-8 lesz, de a metódus harmadik paramétereként egy Charset példányt megadva ezen módosíthatunk.

Files.writeString(file, "John Doe\nJane Doe\n", Charset.forName("ISO-88592"));

Amennyiben az írandó fájl még nem létezik, akkor a metódus hívásának hatására létrejön, ha viszont létezik, akkor az előző tartalma törlődik, és csak az újonnan beleírt szöveg lesz benne. Amennyiben az új szöveget a már meglévő tartalom végéhez szeretnénk hozzáfőzni, akkor paraméterként a StandardOpenOption.APPEND enum értéket is át kell adnunk. A StandardOpenOption enum az OpenOption interfészt implementálja, és sok más beállítás is átadható vele a writeString() metódusnak.

Files.writeString(file, "John Doe\nJane Doe", StandardOpenOption.APPEND);

Nem csak egyetlen szöveg, hanem szövegek listája is kiírható fájlba a Files.write() metódus segítségével. Ebben az esetben minden listaelem külön-külön sorba kerül.

List<String> employees = List.of("John Doe", "Jane Doe");

Files.write(file, employees);

A Files.write() metódusnak is lehet karakterkészletet, illetve OpenOption példányokat átadni paraméterként.

Ellenőrző kérdések

* Hogyan lehet szöveges fájlba tartalmat írni?
* Milyen karakterkészlet az alapértelmezett és hogyan lehet ezt módosítani?
* Mi történik, ha az írandó fájl már létezik?
* Hogyan lehet lézető fájl végére hozzáfűzni az új tartalmat?

Feladat

Napló

Ebben a feladatban egy iskolai naplózó rendszert kell megvalósítanod.

* Készíts a school csomagban egy Diary nevű osztályt benne egy newMark() metódussal, ami paraméterül várja a tanuló nevét és egy jegyet! Ha létezik a “tanulo\_neve.txt” akkor a jegyet hozzáfűzi az állomány végére. Ha nem, akkor egy új állományt hoz létre “tanulo\_neve.txt” formátumban, és beleírja a jegyet. Azt, hogy létezik-e egy fájl a Files.exists(path) metódussal tudod eldönteni. A fájlok a src/main/resources/ könyvtárban legyenek!
* Az év végén a tanár szeretné a tanuló fájl utolsó sorába az átlagot beírni. Írj egy metódust average() névvel, ami a fájl utolsó sorába a jegyek átlagát írja be! Például “average: 5”.

Bájtok beolvasása fájlból és kiírása fájlba (ioreadwritebytes)

Bináris állományok kezelésére is találhatunk metódusokat a Files osztályban. Ezek segítségével az állomány tartalma byte[]-be olvasható, illetve a fájlba byte[] közvetlenül írható.

A Files.readAllBytes() metódus csak a fájl elérhetőségét várja egy Path objektumban, és az abban található összes bájtot felolvassa és tömbként visszaadja. Hiba esetén IOException kivételt dob.

Path file = Path.of("data.dat");

try { byte[] bytes = Files.readAllBytes(file); System.out.println(Arrays.toString(bytes));

}

catch (IOException ioe) {

throw new IllegalStateException("Can not read file", ioe);

}

A Files.write() metódus alkalmas bináris állomány írására is, amennyiben második paraméterként byte[] típusú adatot adunk át.

Path file = Path.of("data.dat");

try {

Files.write(file, new byte[]{97, 98, 99, 100, 101});

}

catch (IOException ioe) {

throw new IllegalStateException("Can not write file", ioe);

}

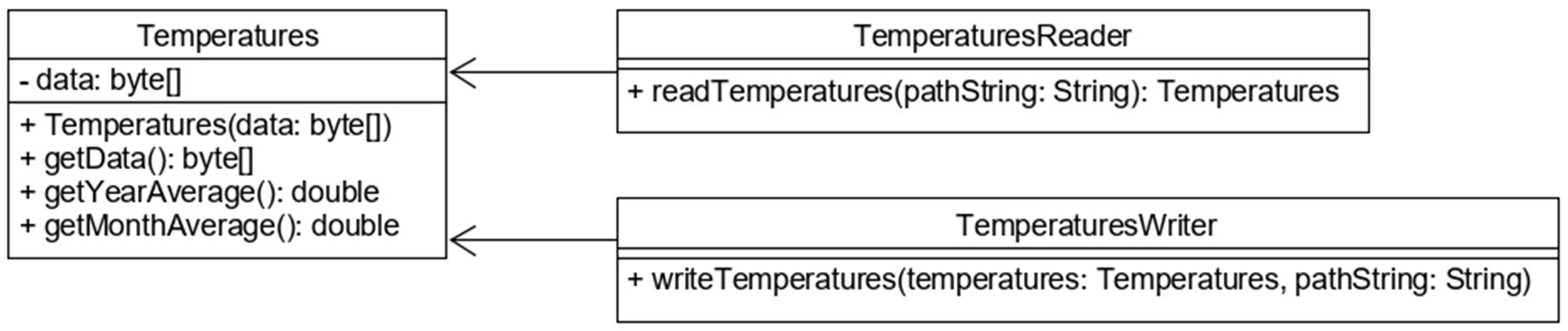
Ellenőrző kérdések

* Hogyan lehet bináris fájl tartalmát beolvasni a Files osztállyal? Milyen típusú értékkel tér vissza a metódus?
* Hogyan lehet bináris fájlba byte[] típusú adatot kiírni?

Feladat

Hőmérsékleti statisztika

Az előző 365 nap hőmérsékleti adatait fájlban tároljuk. A Temperatures osztály attribútumában ezek találhatók egy byte[]-ben. Az osztály konstruktorban kapja meg a hőmérsékleti adatokat. A getYearAverage() metódusa a teljes év átlaghőmérsékletét adja vissza, a getMonthAverage() az utolsó 30 napét. Készíts egy TemperatureReader osztályt, mely egyetlen readTemperatures() metódusa bináris fájlból beolvassa az eltárolt hőmérsékleteket, és egy Temperatures példánnyal tér vissza! Ugyanennek mintájára készíts egy TemperaturesWriter osztályt, amelynek a writeTemperatures() metódusa a paraméterként kapott Temperatures példányból bináris fájlba írja a hőmérsékleti adatokat!



UML osztály diagram

String olvasása Readerrel (ioreader)

Nagyméretű szöveges fájlok esetén nem célravezető, ha a teljes tartalmat egyszerre olvassuk be a memóriába. Ebben az esetben jobb, ha olvasás közben dolgozzuk fel az adatokat. Olvashatunk karakterenként, soronként vagy bármekkora egységenként, ehhez pusztán egy Reader példányra van szükségünk. A Reader egy absztrakt osztály, melynek több konkrét megvalósítása is van. Ezek közül az alapján választhatunk, hogy milyen módon szeretnénk a szöveges fájlt feldolgozni. Másik nagy előnye a darabokban olvasásnak, hogy bármikor megszakíthatjuk, nem kell feltétlenül a teljes fájlt felolvasni.



Szöveges állomány olvasása fájlrendszerből

Először mindenképpen egy Path objektumra van szükségünk, mely az olvasandó fájlt reprezentálja. Ha soronként szeretnénk feldolgozni a fájlt, akkor ezt a BufferedReader segítségével tehetjük meg. Ennek egy példányát a Files.newBufferedReader() metódussal készíthetjük el, amelynek átadjuk a Path objektumot. A BufferedReader readLine() metódusa a fájlnak egyetlen sorát adja vissza. Amikor vége a fájlnak, akkor a metódus visszatérési értéke null lesz, ezért a sorok olvasása történhet while ciklusban.

A BufferedReader objektumot le kell zárni, ezért try-with-resources szerkezetben hozzuk létre. Megnyitáskor és olvasás közben IOException kivétel keletkezhet, melyet megfelelően kezelnünk kell.

Path file = Path.of("employees.txt");

try (BufferedReader reader = Files.newBufferedReader(file)) { String line; while((line = reader.readLine()) != null) {

System.out.println(line);

}

}

catch (IOException ioe) {

throw new IllegalStateException("Can not read file", ioe);

}

Alapértelmezetten UTF-8 kódolású fájlt vár, de ez felülírható, ha a

Files.newBufferedReader() metódusnak paraméterként egy Charset példányt is átadunk.

BufferedReader reader = Files.newBufferedReader(file, Charset.forName("ISO8859-2"))

Java 8 előtt

Mint láttuk, a Files osztály itt is a segítségünkre volt, de a newBufferedReader() metódusa csak a Java 8-ban jelent meg, míg a Reader és a BufferedReader osztályok mindig is léteztek. A különböző Reader megvalósításokat két szintre sorolhatjuk: \* Alacsony szintű a FileReader, mely minimális műveletre képes, de közvetlen hozzáférést biztosít az erőforráshoz. Ezzel karakterenként olvashatunk. \* A magas szintű Reader-ek valamilyen szempontból hatékonyabb olvasást biztosítanak. Ilyen a pufferelt olvasást támogató BufferedReader.

Magas szintű Reader más, már létező Reader-t burkol be, ezért amikor mi magunk akarunk BufferedReader példányt készíteni, akkor először egy FileReader-t kell példányosítanunk.

BufferedReader reader = new BufferedReader(new FileReader("employees.txt"))

BufferedReader reader = new BufferedReader(new FileReader(new File("employees.txt")))



FileReader és BufferedReader kapcsolata

Elsőre ez bonyolultnak tűnhet, de nagy előnye ennek a felépítésnek, hogy egy magasabb szintű Reader nem csak a fájlrendszerből való olvasást tudja optimalizálni, hanem például a hálózatról érkező adatok olvasását is, azaz független az adat forrásától.

A FileReader példányosításakor az alapértelmezett karakterkódolás mindig a futtató platformon alapértelmezett lesz, de mi is megadhatjuk Charset példány átadásával.

BufferedReader reader = new BufferedReader(new FileReader("employees.txt",

StandardCharsets.UTF\_8))

Ellenőrző kérdések

* Hogyan lehet egy szöveges fájlt részletekben olvasni?
* Milyen karakterkódolással dolgozik a Files.newBufferedReader() metódussal létrehozott Reader? Hogyan lehet ezt megváltoztatni?

Feladat

Személyi igazolvány számok

A idread csomagban készítsd el az IdManager osztályt. Ez az osztály felelős az idnumbers.txt (kitalált) személyi igazolvány számokat tartalmazó szöveges állomány feldolgozásáért. A readIdsFromFile() metódus megkap egy fájlnevet, és annak sorait tárolja el az List<String> ids listában.

USA tagállamok

A következő feladatban a states csomagban kell dolgoznod! A stateregister.txt állomány tartalmazza az Amerikai Egyesült Államok államait és azok fővárosait kötőjellel elválasztva. Neked ezt a fájlt kell feldolgoznod és egy keresés algoritmust készítened!

* Hozzd létre a State java osztályt melynek két attribútuma az állam neve stateName és a főváros neve capital! (Konstruktor,getterek!)
* A StateRegister osztály felelős a file beolvasásáért és a State objektumok létrehozásáért, valamint a keresésért. Legyen benne egy lista states , State generikussal, amibe a readStatesfromFile(stringFileName) eltárolja a beolvasott adatokat. A findCapitalByStateName(String stateName) megkeresi a paraméterül kapott állam fővárosát. Ha nincs ilyen állam akkor IllegalArgument exception-t dob!

Osztálynapló

A követező feladat egy osztálynapló nyilvántartása. A grades.txt fájl minden sora tartalmaz egy nevet és utána a tanuló jegyeit. Készíts egy Student osztályt mely a tanuló nevét és jegyeinek listáját képes tárolni! Legyen benne egy átlagszámító metódus, valamint egy metódus, ami képes eldönteni, hogy a tanuló jegyei emelkednek-e.

Készíts egy SchoolRecordsManager osztályt, ami beolvassa fájlból az adatokat, eltárolja, és ezen felül még képes egy osztályátlag számítására is! ### String írása Writerrel (iowriter)

Fájlt nem csak olvasni, de írni is tudunk darabokban egyetlen megnyitás és lezárás között, nem kell a teljes tartalmát először összegyűjteni a memóriában.



Szöveges fájlok írása

Erre szolgálnak a Writer absztrakt osztály különböző megvalósításai. A pufferelt írást a

BufferedWriter osztály valósítja meg, melynek egy példányát a

Files.newBufferedWriter() metódussal tudunk előállítani. A metódus a fájlt reprezentáló Path objektumot várja paraméterként. A fájlba szöveget írni a write() metódussal lehet. Sorvége jelet külön karakterként nekünk kell kiírni. Mivel ez rendszerfüggő, a BufferedWriter külön newLine() metódust biztosít hozzá. Hiba esetén mind a BufferedReader létrehozása, mind a write() metódus IOException kivételt dob.

List<String> employees = List.of("John Doe", "Jane Doe", "Jack Doe"); Path file = Path.of("employees.txt");

try (BufferedWriter writer = Files.newBufferedWriter(file)) { for (String employee: employees) { writer.write(employee + "\n");

}

}

catch (IOException ioe) {

throw new IllegalStateException("Can not write file", ioe);

}

A BufferedWriter példányt le kell zárni, ezért try-with-resources szerkezetben hozzuk létre. Alapértelmezett karakterkódolása UTF-8, de a Files.newBufferedWriter() metódusnak paraméterként egy Charset is átadható.

Java 8 előtti írás

A Java 8 előtt még nem lehetett a Files osztállyal BufferedWriter példányt gyártatni, a Reader-ekhez hasonlóan itt is egy alacsony szintű Writer-re, a FileWriter-re építettük az olyan magas szintű Writer megvalósításokat, mint a BufferedWriter.



FileWriter és BufferedWriter

BufferedWriter writer = new BufferedWriter(new

FileWriter("employees.txt"));

A pufferelt írás annyit jelent, hogy a write() metódus hívása nem jelent azonnali háttértárra írást, a BufferedWriter először egy bizonyos mennyiséget összegyűjt, és csak utána adja át az egészet a FileWriter objektumnak írásra.

Ellenőrző kérdések

* Hogyan lehet Writer segítségével szöveges fájlt írni?
* Hogyan lehet a tartalmat sorokra tördelni?
* Milyen az alapértelmezett karakterkódolás? Hogyan lehet az írott fájl kódolását megadni?
* Mit jelent a pufferelt írást?
* Java 8 előtti verziójával hogyan hozunk létre olyan Writer-t, mely fájlba ír?

Feladat

Nevek

Az első feladatban egyszerűen neveket fogunk eltárolni fájlban és listában egyaránt. A NameWriter osztály konstruktorban megkapja az írni kívánt fájlt. Az addAndWrite() metódus egy nevet vár paraméterül, amelyet hozzáfűzi a fájlhoz. A hozzáfűzéshez használd a korábban megismert StandardOpenOption.Append paramétert a newBufferedWriter() metódusban.

Zenekarok

Ebben a fájl olvasását és írását is gyakorolhatod. Adott a bands\_and\_years.txt állomány, melyben zenekarok nevét és alapítási évét találod. Készíts egy metódust melynek a paramétere egy fájl és egy évszám! Ez a metódus ki fogja írni a fájlba az évszámnál régebben alakult zenekarokat. Megoldási javaslat, hogy készíts egy privát metódust, ami kigyűjti ezeket a zenekarokat egy listába. Minden zenekart egy Band objektum reprezentál, melynek attribútumai a név és az évszám.

Különböző típusok írása PrintWriterrel (ioprintwriter)

A különböző típusú adatok kiírása szöveggé konvertálás nélkül is lehetséges a

PrintWriter osztály segítségével. A print(), println() és a printf() metódusa overloadolt, minden primitív típusú, valamint String és általános Object típusú adattal is paraméterezhető.

PrintWriter objektumot BufferedWriter objektum becsomagolásával készíthetünk.



FileWriter, BufferedWriter és PrintWriter

A fájl megnyitása IOException típusú kivételt dobhat, melyet kezelnünk kell, az írást megvalósító metódusok azonban sosem dobnak kivételt.

List<String> employees = List.of("John Doe", "Jane Doe", "Jack Doe"); Path file = Path.of("employees.txt");

try (PrintWriter writer = new PrintWriter(Files.newBufferedWriter(file))) { for (String employee: employees) { writer.print(employee); writer.print(","); writer.println(200\_000);

}

}

catch (IOException ioe) {

throw new IllegalStateException("Can not read file", ioe);

}

Ellenőrző kérdések

* Milyen előnyei vannak a PrintWriter-nek a BufferedWriter-hez képest?
* Mi okozhat kivételt PrintWriter használata során?

Feladatok

Fizetések

Ebben a feladatban emberek fizetését kell meghatároznod titulus alapján. A SalaryWriter osztály konstruktorban kap egy név listát. A writeNamesAndSalaries(Path file) metódus kiírja a fájlba név: összeg formátumban. A fizetések a következő képpen alakulnak:

* Ha tartalmazza a név a “Dr” előtagot, akkor 500000
* Ha a “Mr” vagy “Mrs” előtagot akkor 200000
* Különben 100000

Szavazatszámlálás

Ebben a feladatban egy tehetségkutató showt fogunk szimulálni. A feladat kicsit összetettebb. A talents.txt tartalmazza az indulók listáját, míg a votes.txt a leadott szavazatokat, azaz az előadó kódját.

A te feladatod, hogy készíts egy kimutatást egy fájlba. A fájlnak tartalmaznia a kódot az előadás nevét illetve, hogy az adott előadás hány szavazatot kapott! Ezen felül az utolsó sornak tartalmaznia kell a győztes nevét a következő formátumban: Winner: győztes neve.

A megoldáshoz használj nyugodtan private metódusokat.

Kiírás Stringbe StringWriterrel (iostringwriter)

A StringWriter osztály az írást nem fájlba, hanem a memóriába egy String-be végzi. Az elkészült szöveget metódussal le tudjuk kérdezni. Tulajdonképpen ennél többre is képes a StringBuilder, de mivel a StringWriter Writer leszármazott, mindenhol használható, ahol egy létező metódus Writer-t vár. Főként tesztelési célból használjuk.

Paraméter átadása nélkül példányosítható, és írni a write() metódussal lehet. IOException kivételt egyedül a close() metódusa dobhat, ezért nem szükséges a trywith-resources szerkezetben létrehozni, de át kell adni neki a fejrészben.

A tartalmát a toString() metódussal kérdezhetjük le.

StringWriter sw = new StringWriter(); try (sw) {

for (String employee: employees) { sw.write(employee); sw.write(", ");

}

}

catch (IOException ioe) {

throw new IllegalStateException("Can not write", ioe);

}

System.out.println(sw.toString());

Amennyiben egy létező metódus vár Writer-t, az nem tudja, milyet is fog kapni.

public void writeTo(List<String> employees, Writer writer) { try {

for (String employee : employees) { writer.write(employee); writer.write(", ");

}

} catch (IOException ioe) {

throw new IllegalStateException("Can not read file", ioe); }

}

A fenti metódus híváskor kaphat például BufferedWriter példányt

<String> employees = List.of("John Doe", "Jane Doe", "Jack Doe"); try (BufferedWriter writer =

Files.newBufferedWriter(Path.of("employees.txt"))) { writeTo(employees, writer); } catch (IOException ioe) {

throw new IllegalStateException("Can not write file", ioe);

}

de kaphat StringWriter-t is.

StringWriter stringWriter = new StringWriter(); try (stringWriter) {

writeTo(employees, stringWriter);

} catch (IOException ioe) {

throw new IllegalStateException("Can not write", ioe);

}

System.out.println(stringWriter.toString());

Az első esetben a writeTo() metódus az employees.txt fájlba, a második esetben a memóriába írt.

Feladat

• Mire való a StringWriter osztály? Hasonlítsd össze a BufferedWriter-rel és a StringBuilder-rel!

Hosszú szavak

Az első feladatban hosszú szavakat kell kezelned StringWriter segítségével. Írj egy metódust, ami egy Writer-t és egy listát kap paraméterül, és minden lista beli elem után írja a szó hosszát! Majd írj egy metódust, ami csak egy listát vár, és az előzőleg megírt metódust használja a paraéterül kapott listával és egy példányosított StringWriter-rel! ### Bájtok olvasása InputStreammel (ioreadbytes)

Bináris állomány részletekben történő olvasásához InputStream-et használunk. Ekkor a közlekedő adatfolyam bájtok sorozata, nem pedig karaktereké.



InputStream és a fájlrendszer kapcsolata

Az InputStream absztrakt osztály melynek egy konkrét megvalósítása kérhető a

Files.newInputStream() metódussal, amely egy Path onjektumot vár paraméterként. A kapott példányon át elérhetjük és olvashatjuk a fájlt. Amennyiben egyszerre szeretnénk a teljes tartalmat beolvasni a memóriába, használjuk a readAllBytes() metódust.

Path file = Path.of("data.dat");

try (InputStream inputStream = Files.newInputStream(file)) {

byte[] bytes = inputStream.readAllBytes();

System.out.println(bytes.length);

}

catch (IOException ioe) {

throw new IllegalStateException("Can not read file", ioe);

}

Amennyiben részletekben szeretnénk a fájlt olvasni, akkor erre a legalkalmasabb a BufferedInputStream használata, amely egyszerre több bájtot is képes beolvasni pufferelve. Ez egy magas szintű stream, mely az alacsonyabb szintű FileInputStream osztályra épül.



FileInputStream és BufferedInputStream

A BufferedInputStream használatához mindössze egy előre elkészített byte[] tömböt kell átadni a BufferedInputStream read() metódusának. A metódus feltölti a tömböt és visszaadja a beolvasott bájtok számát. A fájl végéhez érve ez eltérhet a tömb méretétől, ezért a visszaadott érték vizsgálata alkalmas annak ellenőrzésére, hogy van-e még adat a fájlban.

Path file = Path.of("data.dat"); try (InputStream inputStream = new

BufferedInputStream(Files.newInputStream(file))) { byte[] bytes = new byte[1000]; int size;

while ((size = inputStream.read(bytes)) > 0) {

System.out.println(size);

}

}

catch (IOException ioe) {

throw new IllegalStateException("Can not read file", ioe);

}

Java 8 előtti olvasás



BufferedInputStream felépítése

A Java 8 verzió előtt a bináris fájlok olvasása nem sokban különbözött. Először egy

FileInputStream-et kellett példányosítani a fájl elérhetőségével, amit a BufferedInputStream konstruktorának adtunk tovább. Az elkészült objektum használata már egyezik a fent megismerttel.

InputStream inputStream = new BufferedInputStream(new

FileInputStream(file))

InputStream inputStream = new BufferedInputStream(new FileInputStream(new File(file)))

Ellenőrző kérdések

* Hogyan lehet bináris fájlokat darabokban beolvasni a memóriába?
* Milyen magas szintű streammel metódussal lehet adott méretű darabot beolvasni? Hogyan működik?

Feladat

“A” betűk

Adott a data.dat állomány melyben rengeteg karakter található. A feladatod, hogy számold meg benne az “a” betűket, a videóban látott eszközök segítségével.

Mátrix

Ebben a feladatban egy mátrix adatszerkezettel kell dolgoznod. Hozz létre egy listát, ami byte tömbök tárolására alkalmas! Tárold el ebben a listában a mátrix.dat állományból beolvasott, minden 1000 bájtot tartalmazó byte tömböt!

Az állományban csak egyesek és nullák vannak. Írj egy metódust, ami visszaadja azon oszlopok számát, ahol több nulla, mint egyes van a mátrixban!

String olvasása classpath-ról (ioreaderclasspath)

A felhasználóknak általában jar állományban adjuk át az alkalmazást. Ebben nem csak osztályok, hanem más állományok is találhatóak. Hogyan tudjuk ezeket a fájlokat beolvasni, hiszen ezek nem képezik a fájlrendszer részét?

Ezek a fájlok a classpath-on találhatók, a JVM itt keresi őket. Maven használata esetén ez a src/main/resources könyvtárat jelenti, ezek az állományok a jar állomány gyökerébe kerülnek. A könyvtárba alkönyvtárak hozhatók létre. A fájlok útvonala ekkor egy harmadik módon is megadható: a gyökér útvonal / jellel kezdődik, és mindig a classpath gyökér könyvtárához viszonyított útvonalat tartalmaz.

A classpath-on elhelyezett fájlokat a Class osztály getResourceAsStream() metódusával lehet megnyitni. A metódust az aktuális osztályon kell meghívni, szövegként várja az útvonalat, és egy InputStream-mel tér vissza. Ha ezt szöveges fájlként szeretnénk olvasni, akkor Reader-ré kell alakítani. Erre alkalmas az InputStreamReader osztály. Létrehozásakor egy InputStream-et vár, és egy Reader-rel tér vissza. Ezt BufferedReader-be csomagolva már akár soronként is olvashatjuk.



InputStream átcsövezése BufferedReaderbe try (BufferedReader reader = new BufferedReader(new InputStreamReader( EmployeeService.class.getResourceAsStream("/employees.txt")))) { String line; while((line = reader.readLine()) != null) {

System.out.println(line);

}

}

catch (IOException ioe) {

throw new IllegalStateException("Can not read file", ioe);

}

Mindig figyeljünk az útvonal helyes megadására! Ha elhagyjuk a kezdő / jelet, akkor a relatív útvonal az osztály csomagjának megfelelő classpath-on lévő mappához viszonyított útvonalat jelent!

Tehát ha van egy ioreaderclasspath.EmployeeReader osztályon meghívott getResourcesAsStream("employees.txt"), akkor az employee.txt állományt az src/main/resources/ioreaderclasspath/ könyvtárban kell elhelyezni.

Ellenőrző kérdések

* Hova kell tenni az alkalmazás részét képező erőforrás állományokat Maven használata esetén? Jar készítésekor hova kerülnek ezek az állományok a jar-on belül?
* Hogyan lehet hivatkozni ezekre a fájlokra? Mi az a gyökér útvonal?
* Hogyan lehet a classpath-on lévő fájlokat megnyitni?
* Hogyan lehet a classpath-on lévő szöveges állományokat soronként beolvasni?

Feladat

Országok

A src/main/resources/country.txt állományban országnevek és a szomszédos országok száma található. Hozz létre egy az ország tárolására alkalmas osztályt Country néven!

Hozz létre egy CountryStatistics osztályt, ahol beolvasod file tartalmát egy listába, amjd írj metódusokat amelyek visszatérési értéke választ ad a következő kérdésekre: \* Hány országot olvastál be? \* Melyik országnak van a legtöbb szomszédja? ### Bájtok írása OutputStreammel (iowritebytes)

Bináris fájlok írásához az absztrakt OutputStream osztály egy megvalósítását kell használnunk. Alacsony szintű stream a FileOutputStream, melyet pufferelt íráshoz egy BufferedOutputStream példányba csomagolhatunk.



FileOutputStream és BufferedOutputStream

Megnyitás után a write() metódussal írhatjuk ki a paraméterként átadott byte[] típusú adatot. Az OutputStream objektumot le kell zárni, ezért try-with-resources szerkezetben nyitjuk meg. A megnyitás és az írás közben IOException kivétel keletkezhet, melyet megfelelően kezelnünk kell.

Path file = Path.of("data.dat"); try (OutputStream outputStream = new

BufferedOutputStream(Files.newOutputStream(file))) { for (int i = 0; i < 1100; i++) {

outputStream.write("abcde".getBytes());

}

}

catch (IOException ioe) {

throw new IllegalStateException("Cannot write file", ioe);

}

Java 8 előtti írás (iowritebytes)

A Files.newOutputStream() metódussal csak Java 8 óta lehet OutputStream objektumot előállítani, előtte a FileOutputStream konstruktorát nekünk kellett hívni egy File típusú objektumként vagy String-ként átadva a fájl elérhetőségét.

OutputStream outputStream = new BufferedOutputStream(new FileOutputStream("data.dat"))

OutputStream outputStream = new BufferedOutputStream(new

FileOutputStream(new File("data.dat")))

Ellenőrző kérdések

• Hogyan lehet bináris fájlt részletekben írni? Milyen osztályokat használunk ehhez?

Feladat

Szövegek írása bájt tömbként

Hozz létre egy StringToBytes osztályt és készíts benne egy writeAsBytes() metódust! A metódus a paraméterként kapott szöveg listából az "\_" jellel kezdődő szövegeken kívül minden mást kiír a szintén paraméterül kapott path-ra bináris fájlba.

Darabokban érkező kép

Hozz létre egy ImageAssembler osztályt, melyben a makeImageFile() metódus a paraméterben a kép részleteit byte[][]-ként kapja meg, majd a részeket kiírja egyetlen image.png nevű fájlba a Path-ként kapott mappába!

Adatok írása OutputStreamre, és olvasás (iodatastream)

DataOutputStream

Bináris állományba nem csak bájtokat, hanem egyéb primitív típusú adatokat és szöveget is írhatunk DataOutputStream segítségével. A metódusai az adatokat bájtsorozattá alakítják, majd átadják az alatta lévő OutputStream-nek. Minden adattípusnak saját metódusa van, például writeInt(), writeDouble, szöveghez a writeUTF().

A DataOutputStream magas szintű OutputStream, ezért más OutputStream megvalósításokon át dolgozik. Tipikusan már létező BufferedOutputStream objektummal hozzuk létre.



FileOutputStream, BufferedOutputStream és DataOutputStream együttműködése

A DataOutputStream-et a többihez hasonlóan try-with-resources szerkezetben hozzuk létre, így az automatikusan lezárja. Az íráshoz használt változót DataOutputStream-ként kell deklarálni, hogy a speciális metódusait elérhessük. Megnyitás és írás közben IOException kivétel keletkezhet, melyet kezelnünk kell.

Path file = Path.of("data.dat"); try (DataOutputStream outputStream = new DataOutputStream( new BufferedOutputStream(Files.newOutputStream(file)))) { outputStream.writeUTF("John Doe");

outputStream.writeInt(200\_000);

}

catch (IOException ioe) {

throw new IllegalStateException("Can not write file", ioe);

}

DataInputStream

A bináris fájlok tartalmát nem csak byte[]-be, de közvetlen primitív típusú és szöveges típusú változókba is beolvashatjuk. Az átalakítást a DataInputStream teszi lehetővé, mely ugyanúgy magas szintű stream, mint a DataOutputStream.



FileInputStream, BufferedInputStream és DataInputStream együttműködése

Adattípustól függően olvassa be a bájtokat, azaz a metódusai különböző méretű byte[]öt alakítanak át a megfelelő típusú adatra. Például int típusú adat olvasásakor 4 bájt kerül beolvasásra és átalakításra. A primitív adatok mind előre meghatározott méretűek, azonban a szöveg esetén ez nem működik. A DataOutputStream writeUTF() metódusa először két bájton kiírja a szöveg bájtban vett hosszát, csak ezután következik a String értéke. Beolvasáskor ezért az első két bájt értéke határozza meg, hogy mekkora mennyiségű adatot kell szövegként értelmeznie a readUTF() metódusnak.

Path file = Path.of("data.dat");

try (DataInputStream inputStream = new DataInputStream( new BufferedInputStream(Files.newInputStream(file)))) {

String name = inputStream.readUTF(); System.out.println(name);

int salary = inputStream.readInt();

System.out.println(salary);

}

catch (IOException ioe) {

throw new IllegalStateException("Can not read file", ioe);

}

Ellenőrző kérdések

* Milyen osztály segítségével lehet bináris fájlba primitív típusú adatokat és szöveget írni?
* Hogyan történik ezek írása?
* Milyen osztállyal lehet bináris fájlból bármilyen primitív típusú adatot és szöveget olvasni?
* Hogyan történik az olvasás?
* Hogyan alakítja a szövegeket a DataOutputStream byte[] típusú adattá?

Feladat

Számok

Az iodatastream.statistics csomagba készíts egy HeightStatistics osztályt, bele pedig egy saveHeights() metódust. A metódus egy kosárcsapat tagjainak testmagasságát kapja meg List<Integer> típusú paraméterben. A paraméterben kapott Path objektumként reprezentált fájlba menti először a lista méretét, majd egyenként a lista elemeit.

BankAccount

Az iodatastream.bank csomagba készítsd el a BankAccount osztályt, melyben attribútumként a számlaszám (String), a tulajdonos neve (String) és az egyenleg (double) szerepel! A konstruktor mindhárom attribútumot várja, valamint mindegyikhez van getter.

Készíts egy BankAccountManager osztályt, melyben csak 2 metódus van. A saveAccount() metódus egy BankAccount példány állapotát menti a bankszámla számával megegyező nevű .dat kiterjesztésű fájlba DataOutputStream segítségével. Paraméterként megkapja a mentési mappát is Path objektumknt. A loadAccount() ugyanilyen szerkezetű fájlból betölti egy BankAccount adatait, és paraméterként csak egy InputStream-et vár.

A fájl minden adatot tartalmazzon az alábbi sorrendben: számlaszám, tulajdonos neve, egyenleg!

Bájtok írása tömörítéssel (iozip)

A Java lehetőséget biztosít arra, hogy a fájlokat tömörített állományba írjunk. A tömörített zip állomány önmagában is fájlokat és könyvtárakat tartalmaz, melyek létrehozása a ZipEntry osztály használatával lehetséges.

A ZipOutputStream teszi lehetővé, hogy a tömörített állományba új ZipEntry kerüljön elhelyezésre, illetve ebbe byte[] típusú adat kerüljön. A ZipOuputStream magas szintű stream, tipikusan BufferefOutputStream-re épülve használjuk.



FileOutputStream, BufferedOutputStream és ZipOutputStream együttműködése

Új ZipEntry-t a fájl vagy mappa nevének a megadásával hozhatunk létre. A mappa neve utáni / jel jelöli, hogy a létrehozandó oobjetum mappa lesz. Ha a fájlt valamelyik tömörített mappába akarjuk elhelyezni, akkor a nevének megadásakor a zip fájl gyökeréhez képesti relatív útvonalat kell megadnunk. Például a new

ZipEntry("folder/") egy folder nevű mappát hoz létre, a new

ZipEntry("folder/data.dat") egy data.dat nevű fájlt hoz létre a folder mappában.

ZipEntry-t a tömörített állományba a ZipOutputStream putNextEntry() metódusával lehet. Ezután az írás mindaddig ebbe az állományba történik, amíg új ZipEntry nem kerül elhelyezésre vagy a ZipOutputStream lezárásra nem kerül.

Path file = Path.of("data.zip");

try (ZipOutputStream outputStream = new ZipOutputStream( new BufferedOutputStream(Files.newOutputStream(file)))) { outputStream.putNextEntry(new ZipEntry("data.dat")); for (int i = 0; i < 1100; i++) {

outputStream.write("abcde".getBytes());

}

}

catch (IOException ioe) {

throw new IllegalStateException("Can not write file", ioe);

}

Ellenőrző kérdések

* Milyen osztályok segítik zip tömörített állományok írását?
* Hogyan hozhatunk létre új fájlt egy tömörített állományba?
* Hogyan írhatunk egy fájlt a tömörített állományban?
* Hogyan hozhatunk létre egy új mappát egy tömörített állományba?

Feladat

Alkalmazottak

Az iozip.names csomagba készíts egy EmployeeFileManager osztályt! Egyetlen metódusa van, a saveEmployees(), mely a paraméterként kapott Path által reprezentált zip fájlba létrehoz egy names.dat nevű bináris fájlt, és a szintén paraméterként kapott névlista tartalmát beleírja.

Napi tranzakciók

Az iozip.transactions csomagba készíts egy Transaction immutable osztályt, melyben a tranzakció azonosítója (long id), a tranzakció pontos időpontja

(LocalDateTime time), az érintett bankszámla száma (String account) és a tranzakció összege (double amount) található.

A TransactionFileManager osztály saveTransactions() metódusa a paraméterként kapott Path-ra elmenti a Transaction listában kapott tranzakciókat tömörítve. Minden tranzakció külön szöveges fájlba kerüljön! A fájl neve a tranzakció azonosítója legyen!

Tartalma sortöréssel (\n) elválasztva az időpont, a bankszámlaszám és az összeg.

Például:

2018-02-13T15:08:43

10092365-37255375-33310000

23000.0

Konvertálás a típusok között (ioconvert)

Fájlokat kezelhetünk streamként, azaz bájtfolyamként és karakterek sorozataként is. Ha például már rendelkezünk egy streammel, és azt szöveges adatként akarjuk kezelni, szükségünk van egy olyan eszközre, amely a kettő közötti konverziót végrehajtja.

Minden fájlkezelő osztály az alábbi négy absztrakt osztály valamelyikének a leszármazottja, és ez meghatározza az alapvető funkcióját:

* InputStream: bájtok olvasása
* OutputStream: bájtok írása
* Reader: karakterek olvasása
* Writer: karakterek írása

InputStream-ből Reader konverzióhoz az InputStream-re egy InputStreamReader példányt kell csatolni, ez végzi el a bájt sorozat karakteres adattá történő konvertálását.



InputStream Reader konverzió

OutputStream-ből Writer konverzióhoz az OutputStreamWriter csatolása szükséges a létező OutputStream-re. Ez végzi el a szöveges adatok bájt sorozattá történő konvertálását.



OutputStream Writer konverzió

Speciális osztály a PrintStream, mely tulajdonképpen egy stream, de metódusai az adatok szöveges reprezentációjának, azaz karakterek sorozatának az írását teszik lehetővé. Metódusai hiba esetén nem dobnak IOException kivételt, és képes az automatikus flush hívásra, azaz a puffer tartalmát automatikusan kiírja és utána üríti. Ilyen típusú a System.out is.

A PrintStream magas szintű stream, tipikusan BufferedOutputStream-hez csatolva használjuk.



FileOutputStream, BufferedOutputStream és PrintStream együttműködése

A PrintStream metódusai bármilyen típusú adat szöveges reprezentációját képes kiírni a print(), println() és printf() overloadolt metódusaival. Habár ezek a metódusok nem dobnak kivételt, a fájl megnyitásakor keletkező esetleges hibát kezelnünk kell. A PrintStream automatikus lezárásához try-with-resources szerkezetben nyitjuk meg.

Path file = Path.of("employees.txt");

List<String> employees = List.of("John Doe", "Jane Doe", "Jack Doe"); try (PrintStream outputStream = new PrintStream(

new BufferedOutputStream(Files.newOutputStream(file)))) { for (String employee: employees) { outputStream.print(employee); outputStream.print(","); outputStream.println(200\_000);

}

}

catch (IOException ioe) {

throw new IllegalStateException("Can not write file", ioe);

}

Ellenőrző kérdések

* Hogyan lehet InputStream-et Reader-ré konvertálni?
* Hogyan lehet OutputStream-et Writer-ré konvertálni?
* Mire való a PrintStream osztály és hogyan használjuk?

Feladat

Bevásárlólista

Az ioconvert.shopping csomagba készíts egy ShoppingListManager osztályt! A saveShoppingList() metódusa a paraméterül kapott OutputStream-re kiírja a szintén paraméterül kapott List<String> tartalmát szövegként, minden elemet külön sorba. A loadShoppingList() metódusa a paraméterül kapott Inputstream-ből beolvassa a bevásárlólista tartalmát, amit szöveglistaként ad vissza.

Termékek

Az ioconvert.products csomagba hozd létre a ProductWriter osztályt, és benne a saveProduct() metódust, mely két paramétert kap: egy OutputStream-et és egy List<Product>-ot. A feladata a lista adatait kiírni csv formátumban az OutputStream-re, azaz pontosvesszővel elválasztva a Product adatait. Minden termék külön sorba kerüljön, a sor végére nem kell ;. Kiíráshoz használd a PrintStream osztályt!

A Product tartalmazza a termék nevét (String) és az árát (int).

Files osztály használata (iofiles)

A Java 7 verzióban megjelenő Path és Files osztályt a korábbi File osztály kiváltására hozták létre. Nevét meghazudtoló módon a File osztály nem csak fájlokat reprezentál, hanem mappákat is, illetve a fájlrendszer kezeléséhez szükséges metódusokat is tartalmazza. A két funkció az új API-ban kettévált: a Path osztály kizárólag egy útvonalat reprezentál, és ezek kezeléséhez (összevonás, abszolút útvonallá alakítás) szükséges metódusokat tartalmaz, míg a fájlok és mappák műveletei a Files osztályba kerültek. Ezek a műveletek kiegészültek újabbakkal, mint például mappák bejárása, de még az olyan alapvető művelet, mint a fájl másolása, is újonnan került bele.

A File és a Path közötti konverzióra új metódus jelent meg a File osztályban, illetve a régebbi rendszerekkel való kompatibilitás miatt egy Path objektum File típusúvá alakítható.

File file = new File("docs/foo.txt"); Path path = file.toPath();

Path path = Paths.get("docs/foo.txt");

File file = path.toFile();

A Files osztály gyakori műveletei

A Files osztály fájltartalmat kezelő (pl. readAllLines()) és fájlkezelő objektumokat gyártó (pl. newBufferedReader()) metódusain kívül még nagyon sok más, fájlrendszert kezelő metódussal is rendelkezik. Ezek közül a leggyakrabban használtak:

Lekérdező műveletek:

* exists(Path): létezik-e a paraméterként átadott Path által reprezentált fájl vagy könyvtár
* isDirectory(Path): az átadott Path könyvtár-e
* isRegularFile(Path): az átadott Path normál fájl-e • size(Path): a fájl méretét adja vissza bájtban mérve Módosító műveletek:
* createDirectory(Path): létrehozza a Path által reprezentált könyvtárat feltéve, hogy annak minden szölőkönyvtára már létezik
* createDirectories(Path): létrehozza a Path által reprezentált könyvtárat, és a hiányzó szülőkönyvtárakat is
* copy(Path source, Path target, CopyOption... options): fájl másolása
* copy(InputStream in, Path target, CopyOption... options): InputStream tartalmának fájlba írása
* copy(Path source, OutputStream out): fájl tartalmának OutputStream-re helyezése
* move(Path source, Path target, CopyOption... options): fájl vagy könyvtár mozgatása
* delete(Path): könyvtár vagy fájl törlése, ha nem létezik kivételt dob
* deleteIfExists(Path): könyvtár vagy fájl törlése, ha létezik

CopyOption interfész

A másolás, mozgatás műveletek működése finomhangolható. A különböző beállításokat CopyOption implementációkkal adhatjuk át. Például a fájl másolásakor mi történjen, ha a cél helyen már létezik az adott fájl. A CopyOption-t váró metódusoknak például a StandardCopyOption enum különböző értékei adhatók át.

* StandardCopyOption.REPLACE\_EXISTING: a célfájl létezése esetén azt felül kell írni
* StandardCopyOption.COPY\_ATTRIBUTES: a fájlattribútumokat is át kell másolni
* StandardCopyOption.ATOMIC\_MOVE: a művelet atomi műveletként kezelendő Ellenőrző kérdések
* Sorold fel a Files osztály legalább két lekérdező műveletét! Mire valók?
* Sorold fel a Files osztály legalább öt módosító műveletét! Mire valók?
* Mire való a CopyOption interfész? Milyen implementációját ismered?

Feladat

Telepítés

Készíts egy Installer osztályt, egyetlen install() metódussal, mely paraméterként a telepítési mappát kapja meg String-ként. A megadott mappán belülre másold át a classpathról az install mappában lévő fájlokat és mappákat az install mappából! Az install.txt fájlban a / jelre végződő sorok mappákat jelölnek, ezeket létre kell hoznod, mielőtt fájlokat másolnál bele. Az esetleges sikertelen telepítésből visszamaradt fájlokat mindig írd felül! Ha a paraméterül kapott mappa nem létezik vagy nem is mappa, dobj IllegalArgumentException kivételt!

Fájlkezelés tesztelése (iofilestest)

A JUnit keretrendszer beépített támogatást nyújt a fájlkezelések tesztelésére. A TemporaryFolder objektum gondoskodik a mappák és fájlok ideiglenes létrehozásáról, majd a teszt lefutása után ezeket automatikusan törli.

Amennyiben a tesztelendő metódus a Path objektumot paraméterként kapja, akkor a tesztelés során a TemporaryFolder-rel létrehozott ideiglenes fájl vagy mappa könnyen átadható neki.

Tegyük fel, hogy az alábbi metódust kell tesztelni:

public class EmployeeService {

public void writeEmployeesToFile(List<String> employees, Path file) { try (BufferedWriter writer = Files.newBufferedWriter(file)) { for (String employee: employees) { writer.write(employee + "\n");

} }

catch (IOException ioe) {

throw new IllegalStateException("Can not write", ioe);

}

}

}

Az elvárt működés az, hogy az átadott szöveglistából az alkalmazottak neveit a szintén paraméterben átadott fájlba írja, minden nevet külön sorba.

A tesztosztályban létre kell hozni egy TemporaryFolder típusú publikus attribútumot, melyet a @Rule annotációval kell ellátni. A teszt metódusban ennek segítségével létrehozunk egy fájlt reprezentáló Path objektumot a newFile() metódus által létrejött File átkonvertálásával, és azt adjuk át a tesztelendő metódusnak. A metódus lefutása után az elkészült fájlt például a Files.readAllLines() metódussal felolvassuk és összehasonlítjuk az elvárt tartalommal.

public class EmployeeServiceTest {

@Rule

public TemporaryFolder temporaryFolder = new TemporaryFolder();

@Test

public void testWrite() throws IOException {

Path file = temporaryFolder.newFile("employees.txt").toPath();

System.out.println(file);

List<String> employees = List.of("John Doe", "Jane Doe", "Jack Doe");

new EmployeeService().writeEmployeesToFile(employees, file);

List<String> content = Files.readAllLines(file); assertEquals(3, content.size()); assertEquals("Jane Doe", content.get(1));

}

}

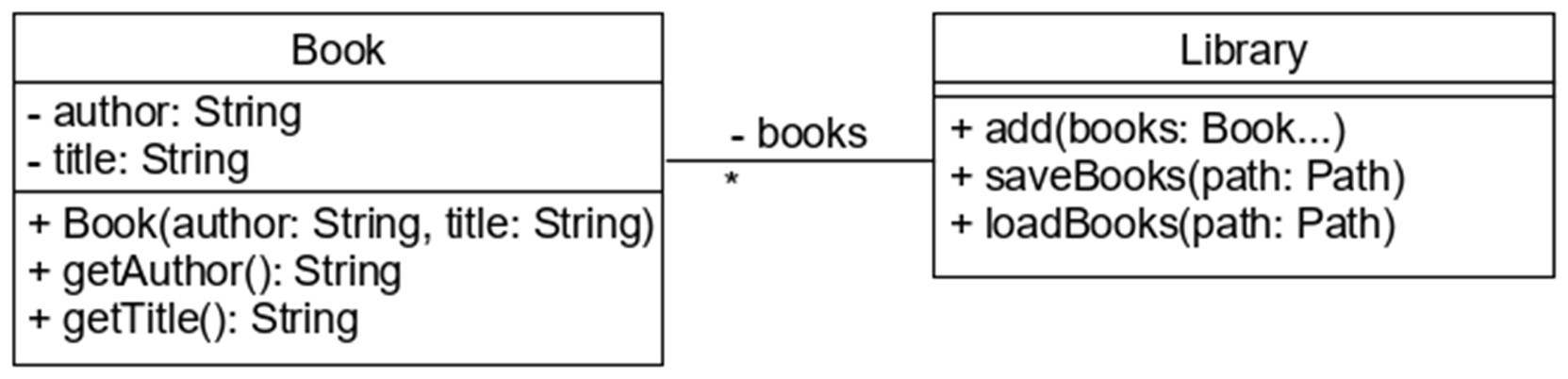
Ellenőrző kérdések

* Milyen JUnit osztály segíti a fájlkezelés tesztelését?
* Hogyan működik ez az osztály?
* Hogyan kell az osztályt használni a teszt esetben?

Feladat

Könyvtár

Készíts egy Book osztályt, amely a könyv címét és szerzőjét tartalmazza. A Library osztály a könyvek listáját attribútumként tárolja. Könyveket lehet hozzáadni az add(Book...) metódusával, mely csak azokat a könyveket adja a listához, amelyek még nem voltak benne. Az osztály aktuális állapotát szöveges fájlba lehet menteni, illetve fájlból be lehet tölteni az ott lévő könyveket.



Library osztály diagram

Készítsd el a teszteket is a Library osztályhoz.

Sajtok

Készíts egy immutable Cheese osztályt, mely a sajt nevét (String) és laktóztartalmát

(double) tartalmazza. A CheeseManager osztályba készíts egy olyan saveToFile() metódust, amely a paraméterül kapott Path által reprezentált bináris fájlba menti a szintén paraméterül kapott Cheese lista tartalmát. A findCheese() metódusa a Pathként kapott fájlban megkeresi név alapján a sajtot, és találat esetén egy Cheese objektumként adja vissza.

Készíts az elkészült CheeseManager osztály metódusaihoz teszteket!

# Kollekciók és osztálykönyvtárak

## Kollekció típusok

Generikusok használata, diamond operator (genericsusage)

Amikor kollekcióban szeretnénk sok elemet tárolni, relációs jelek között kell megadnunk az elemek típusát. Mi van, ha nem adjuk meg? Mivel a Java 5 előtt nem léteztek generikusok, ezért a kollekciók alapértelmezetten is működnek, Object típussal (raw type). Ebben az esetben bármilyen objektum beletehető a kollekcióba, hiszen minden osztály Object leszármazott. A probléma ott kezdődik, amikor valamilyen műveletet szeretnénk az elemen végezni, de nem tudjuk, hogy az pontosan micsoda. Mivel a deklarált típusa Object, ezért csak az Object metódusai érhetőek el. Ha mást szeretnénk, akkor az elemet típuskényszeríteni kell, hogy meghívhassuk a megfelelő metódust. De mi történik, ha az adott elem nem az általunk feltételezett típusú?

List list = new ArrayList(); list.add(Integer.valueOf(5));

System.out.println((String) list.get(0)).toUpperCase());

A programunk elvileg hibátlan, fordításkor nem történik semmi, de futás közben ClassCastException kivételt dob. Ez elkerülhető, ha az elemen először típusellenőrzést végzünk az instanceof operátorral.

A Java 5 bevezette a generikusok használatát, amivel már korlátozni lehet, hogy milyen típusú elem kerülhet a kollekcióba. Ez az ellenőrzés már fordításkor végbemegy. A kollekció deklarációjakor megadhatjuk, hogy milyen típusú elemeket fogad be, és csak ezek, illetve ezek leszármazottai, interfész esetén pedig csak az adott interfészt implementáló objektumok kerülhetnek bele. Java 7 óta a kollekció példányosításakor nem kell a befogadható típust megismételnünk, elég, ha diamond operátort (<>) használunk.

List<String> list = new ArrayList<>();

Még mindig használhatjuk a kollekciókat generikus típus nélkül, de a fordító figyelmeztet ennek veszélyeire.

ArrayList is a raw type. References to generic type ArrayList<E> should be parameterized

Sajnos, abban nem segít, hogy hol van a hiba. -Xlint:unchecked paraméterrel hívva a fordítót már pontosan beazonosítja a helyet is.

Main.java uses unchecked or unsafe operations.

A kollekció generikus típusa nem kovariáns, azaz kizárólag olyan kollekció tehető bele, ahol az elemek deklarált típusa pontosan megegyezik.

List<Number> list1 = new ArrayList<Integer>(); // Nem fordul le! List<Number> list2 = new ArrayList<Number>(); // Lefordul

list2.add(Integer.valueOf(2)); // Lefordul, az elem lehet bármi, ami Number

Ellenőrző kérdések

* Mi a generics jelentősége?
* Mi van a régebbi Java programokkal, ahol a generics még ismeretlen volt?
* Mi az a diamond operátor?
* Mi történik, ha generikussal ellátott osztályon próbálsz úgy műveletet végezni, hogy nem definiáltad a generikus típust?

Gyakorlati feladat - Library

Implementálj egy könytárat, ahol könyveket tárolhatunk. Valósítsuk meg a getFirstBook() metódust generics használata nélkül, és generics használatával is.

Hibakezelés

* Ha a könyvtárat reprezentáló kollekció null, dobjon NullPointerException

kivételt

* Ha a könyvtárat reprezentáló kollekció üres, dobjon IllegalArgumentException

kivételt equals, hashCode (collectionsequalshash)

Az Object osztálytól két nagyon fontos és gyakran használt metódust örököl minden osztály, az equals és a hashCode metódust.

Az equals() metódus

Az equals metódus két objektum egyezőségét vizsgálja. Mivel az == operátor a változók tartalmát, ami objektum esetén a referencia, hasonlítja össze, ezért csak azt tudjuk megnézni, hogy két referencia ugyanarra az objektumra mutat-e. Ha az objektumok állapotát szeretnénk összehasonlítani, akkor az equals metódust kell használnunk. Az Object osztályban lévő equals még az == operátorral egyezően viselkedik, de az olyan osztályokban, mint a String, ArrayList, primitív burkolók már felülírták. Amikor saját osztályt készítünk, amit össze akarunk állapot szerint hasonlítani, akkor nekünk is felül kell definiálnunk az alapértelmezett működést, hogy az attribútumokat vegye figyelembe egyezőség vizsgálatakor, még ha nem is az összeset. Például két személyt tekinthetünk azonosnak, ha megegyezik a személyi igazolvány számuk.

Mivel az equals objektumokon dolgozik, ezért példányból hívható. De vajon mi történik az alábbi esetben?

public static void main(String[] args){ String str = null;

str.equals("Text");

}

A program NullPointerException kivételt dob.

Amikor egy String típusú változót String literállal hasonlítunk össze, akkor jobb megoldás a literálon meghívni az equals-t, de ez nem megoldás, ha két változót akarunk összehasonlítani. Ilyenkor mindig vizsgáljuk meg előbb, hogy nem null értéken akarjuk hívni az equalst, illetve használhatjuk az Objects osztály statikus equals metódusát, mely a két összehasonlítandó objektumot várja paraméterként. Ha valamelyik null, akkor az Objects.equals false értékkel tér vissza, ha mindkettő null, akkor true-val.

Minden már esetben az első paraméter equals metódusát használja.

Szabályok az equals() írásakor

1. Az equals reflexív, azaz ha x nem null, akkor x.equals(x) true értéket kell visszaadjon.
2. Szimmetrikus, azaz ha x és y nem null, akkor x.equals(y) pontosan akkor true, ha az y.equals(x) true.
3. Tranzitív, azaz bármely x, y és z nem null változóra ha x.equals(y) és

y.equals(z) is true értéket ad, akkor az x.equals(z) is true értéket kell adjon.

1. Konzisztens, azaz minden x és y nem null értékű változóra, ha a két objektum állapota nem változik, akkor két független x.equals(y) hívás ugyanazt az eredményt kell adja.
2. Minden x nem null változóra az x.equals(null) false értékkel tér vissza.

public class Card { private String suit; private String rank;

@Override

public boolean equals(Object o) { if (this == o) return true; //1. szabály if (o == null || getClass() != o.getClass()) return false; //5.

szabály és 2. szabály alapfeltétele

Card card = (Card) o;

if (!suit.equals(card.suit)) return false; //Most minden attribútumot figyelembe vesz

return rank.equals(card.rank);

}

}

A 3. és a 4. szabály magától értetődik, ha a fenti mintára készítjük el az equals metódust.

A hashCode() metódus

A hash() függvényt tipikusan kriptográfiai algoritmusok használják. Tulajdonképpen bármilyen hosszú adatot egy adott hosszú adatra (pl. 0 és 65535 közötti egész számra) leképező függvény, amelyből maga az objektum nem állítható vissza. Elvárás, hogy ugyanarra az adatra ugyanazt az értéket adja, de két különböző adatra is kaphatjuk ugyanazt, még ha kis valószínűséggel is.

Javaban a hashCode metódus int értékkel tér vissza. Az Object toString metódusa ezt írja ki a @ jel után. Kollekciók esetén azért jó, mert az objektumok a hashCode-juk alapján osztályozhatóak, így sok adat esetén egyetlen visszakeresése sokkal gyorsabb lehet.

A fenti kártya példában ha konkrét kártyát keresek egy összekevert pakliban, akkor végig kellene nézni egyenként a lapokat, míg a keresettet meg nem találjuk. Ha valamilyen szempont szerint rendezni lehetne a lapokat, akkor nagyon gyorsan megtalálnánk a keresettet, mintha egy bejegyzést keresnénk a szótárban. Sajnos a rendezettség nem mindig megoldható, de csoportokat képezni valamilyen tulajdonság alapján könnyebb. Például ha a lapok értéke szerint csoportosítunk, akkor a pikk 7-est nagyon gyorsan megtaláljuk, mert először csak az érték szerinti 13 csoport között kell megtalálnunk a 7-est, majd csoporton belül a pikket. A hashCode tulajdonképpen a csoportosítást végzi.

Szabályok a hashCode() implementálásakor

Ha nem változik az objektum állapota, akkor a hashCode ugyanazt az értéket kell adja.

Ha két nem null változóra az equals metódus true értéket ad, akkor a hashCode metódus is ugyanazt kell adja.

Ha két nem null változóra az equals metódus false értéket ad, attól még a hashCode nem feltétlenül ad más értéket, de nagy valószínűséggel eltér.

Érdemes ugyanazon attribútumokra támaszkodni, mint az equals írásakor, de mindenképp csak az ott figyelembe vett attribútumok használhatóak hashCode generálásakor. Lehetőleg olyan attribútumokat vegyünk figyelembe, amelyek nem változnak a program futása során.

Az IDEA képes legenerálni az equals és a hashCode metódust is. Ehhez nyomjuk le az

Alt + Insert billentyűkombinációt és válasszuk az equals and hashCode pontot. Az IntelliJ Default (vagy akár egy külső library) template-et választva megadhatjuk, hogy mely attribútumok legyenek figyelembe véve a metódusok generálása során.

Segítségünkre lehet az Objects.hash(Object... values) metódus is.

Ellenőrző kérdések

* Mi az equals() és hashCode() metódusok szerepe? Hogyan kapcsolódik az állapot fogalmához?
* Milyen szabályokat kell betartani az equals implementálásakor?
* Ki/mi határozza meg az objektum azonosságát?
* Mit ad az öröklődés az objektumoknak ezek esetében?
* A felülírás elmulasztása mit eredményez?
* Mire való a hash függvény?
* Milyen tulajdonságokkal rendelkezik egy hash függvény?
* Milyen szabályoknak kell megfelelni a hashCode() metódusok implementálásakor?
* Mi van, ha eltérnek az equals és a hashCode által felhasznált mezők?
* Miért nem dob kivételt a fenti metódusok hiánya illetve a felülírás elmulasztása?

Gyakorlati feladat - equals és hashCode metódusok kipróbálása

A Book osztály objektumait rakjuk bele List és Set kollekció típusokba. Vizsgáljuk meg, hogy mely tesztesetek futnak le sikeresen, és melyek nem a Book osztályban felülírt, vagy éppen kihagyott equals és hashCode metódusok esetén. Értelmezzük a tapasztaltakat!

A BookCatalog osztályban a következő publikus metódusok találhatók (ezek persze a tesztesetekből is következnek):

public Book findBookByTitleAuthor(List<Book> books, String title, String author)

public Book findBook(List<Book> books, Book book) public Set<Book> addBooksToSet(Book[] books)

Hibakezelés

Használjuk ki a kollekciók boolean contains(Object o) metódusát! Ha a metódus nem találja a keresett objektumot, a visszatérési érték legyen null.

Megvalósítás

A Book osztályban az equals(Object o) és hashCode() metódusok megírásával és kikommentelésével (összes kombináció!) próbáljuk ki az egyes tesztesetek sikeres vagy sikertelen lefutását. Értelmezzük a tapasztaltakat!

Tippek

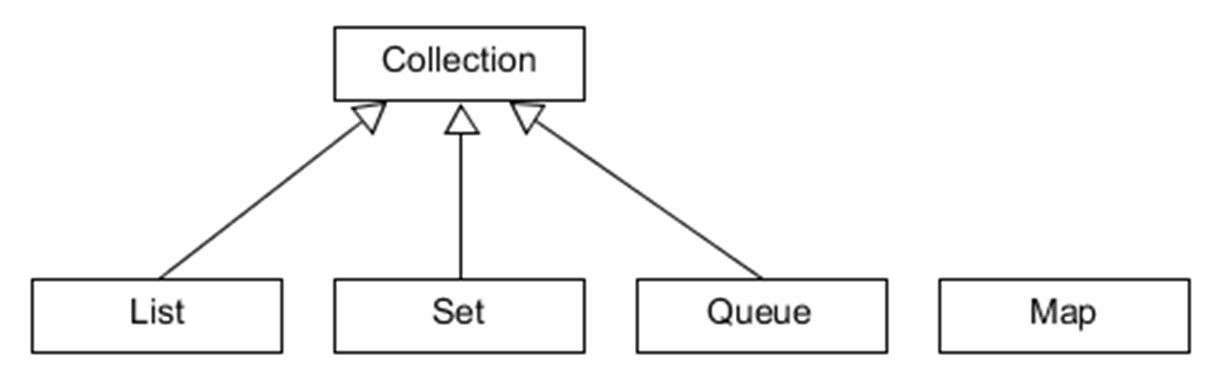
Értelmezzük, hogy mi történik akkor, ha a List kollekcióba azonos állapotú objektumokat töltünk. Mit kapunk vissza keresés esetén?

List, ArrayList kontra LinkedList (collectionslist)

A Java Collections Framework

Amikor több objektum együttes kezeléséről beszélünk, valamilyen kollekcióra gondolunk. Láttuk, hogy a tömb nem a leghatékonyabb megoldás elemek tárolására. A Collections Framework olyan interfészeket és osztályokat tartalmaz, amelyek különböző módon segítik az elemek rendszerezését, kezelését, miközben mind biztosítja a kollekció méretének dinamikusságát.

Öt fő interfészt definiál:



collections

A List jellemzője, hogy az elemeknek adott sorrendje van, index alapján elérhetőek és duplikált elemeket is tartalmazhat.

Ezzel szemben egy Set-ben az elemeknek nincs sorrendje, és duplikációt sem tartalmazhat.

A Queue-t akkor használjuk, ha az elemeket egy megadott sorrendben szeretnénk feldolgozni, mely lehet a beszúrás sorrendje, vagy valamilyen prioritás szerint meghatározott.

A Map kicsit különbözik a többitől, mert az elemei kulcs-érték párok, ahol a kulcsok mindig egyediek.

Minden interfészt több osztály is implementál, de közös jellemzőjük, hogy az Object-ben definiált toString metódust mind felülírja, hogy az a tárolt elemeket szöveges reprezentációjának sorát adja vissza. A Collection interfészben deklarált metódusok boolean add(E element): új elem hozzáadása.

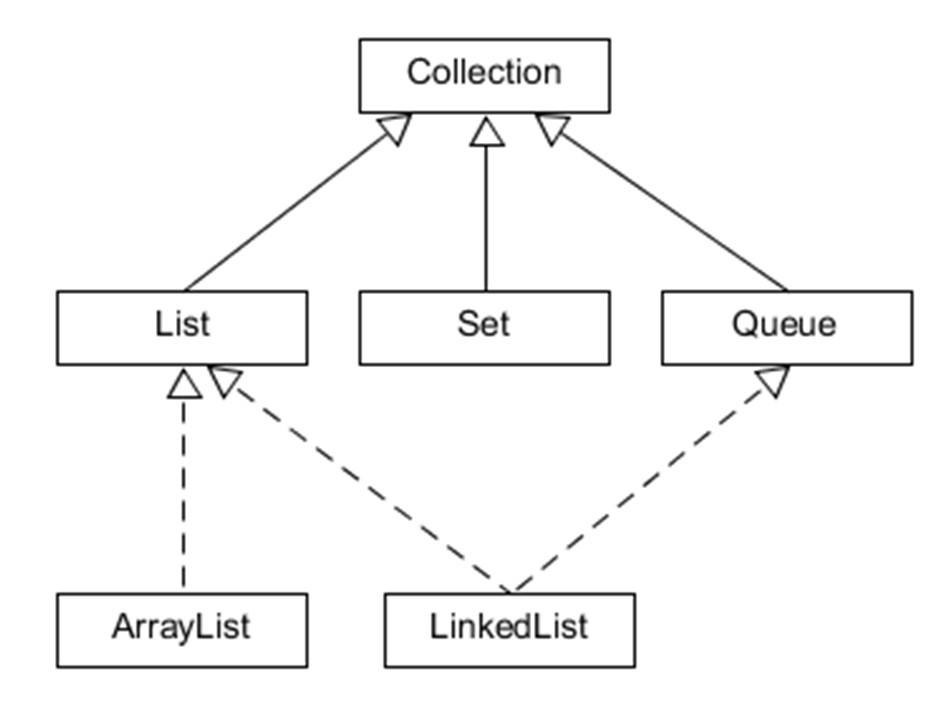
boolean remove(Object o): a megadott elem eltávolítása; igazzal tér vissza, ha volt olyan elem, amit el akartunk távolítani. int size(): az elemek számát adja meg.

boolean isEmpty(): igazzal tér vissza, ha a kollekció elemszáma nulla. void clear(): eltávolítja az összes elemet.

boolean contains(Object o): megadja, hogy a paraméterként átadott elem megtalálható-e a kollekcióban.

A List interfész és implementációi

A List interfészt implementáló osztályok közös jellemzője, hogy az elemek sorrendje adott, és mind befogad duplikációt. Hozzáférni bármelyik elemhez közvetlenül lehet az indexe segítségével, de hogy mely műveletek végezhetők el hatékonyabban, az az implementációtól függ.



lists

ArrayList

Az ArrayList osztály a háttérben tömbben tárolja az elemeket, ezért könnyen és gyorsan lehet az elemeket lekérdezni. Az elemek törlése és beszúrása a többi elem eltolásával jár, vagy akár az egész tömb nagyobb tömbbe átmásolásával, ezért ezek erőforrás igényesebb műveletek. A lista kapacitása tulajdonképpen a háttérben lévő tömb mérete, ezért ha előre tudjuk, hogy sok eleme lesz, érdemes eleve nagy kapacitással létrehozni. A lista telítettsége az elemek számának és a tömb méretének az aránya. Amikor a lista telítettsége elér egy bizonyos mértéket, akkor a kapacitás automatikusan megnő.

LinkedList

A háttérben kétirányú láncolt lista áll, amit úgy kell elképzelni, mintha minden elem csak azt tudná, hogy hol van a következő és az előző, de az elemek fizikailag nem egybefüggő területen vannak a memóriában. Bármely elem lekérdezéséhez be kell járni a lista első vagy utolsó elemétől kezdve a keresett elemig az összeset, ezért ez erőforrásigényes művelet, viszont elem beszúrása vagy törlése nem jár elemmozgatással, ezért ez gyorsabb, mint ArrayList estében. A Queue interfészt is implementálja. List metódusok void add(int index, E element): új elemet szúr be a meghatározott pozícióra.

E element(int index): elem lekérdezése az indexe alapján.

int indexOf(Object o): megkeresi a paraméterben átadott elem első előfordulását, és az indexével tér vissza. Ha az elem nincs a listában, a visszaadott index -1.

int lastIndexOf(Object o): megkeresi a paraméterben átadott elem utolsó előfordulását, és az indexével tér vissza. Ha az elem nincs a listában, a visszaadott érték 1.

void remove(int index): törli a listából a megadott indexű elemet.

E set(int index, E element): lecseréli a megadott indexű elemet a paraméterként átadottra, és a régi elemmel tér vissza.

Ellenőrző kérdések

* Mi a Java Collections Framework? Milyen interfészeket tartalmaz? Milyen viszonyban állnak ezek egymással?
* Milyen közös metódusokat ismersz?
* Milyen tulajdonságai vannak a List interfésznek?
* Mi a különbség az ArrayList és a LinkedList között?
* Milyen további metódusai vannak a List-nek a Collection interfészhez képest?

Gyakorlati feladat 1 - Lottó

Implementálj egy lottó gépet, ahol megadható a lottó típusa (ötös, hatos, stb.) és az is, hogy hány számból válasszon (golyók száma). Írj egy osztályt, amely létrehozza a golyókat (List<Integer> értékek), összekeveri, majd megcsinálja a “húzást”, azaz kiválaszt a lottó típusának megfelelő számú golyót, majd a kihúzott számokat növekvő sorba rendezi és úgy adja vissza.

A húzást a selectWinningNumbers(int lotteryType, int ballCount) metódus végzi, amely egy (List<Integer>) értéket ad vissza.

Hibakezelés

Ha a selectWinningNumbers(int lotteryType, int ballCount) metódust rossz paraméterekkel hívták meg (több vagy ugyanannyi számot kellene kihúzni, mint amennyit generál), a metódus dobjon egy IllegalArgumentException kivételt.

Megvalósítás

Maximálisan használjuk ki a megfelelő kollekció nyújtotta lehetőségeket! A visszaadott számok pozitív egész számok, és nem lehetnek nagyobbak, mint amennyi golyó van. A golyók számozása egytől kezdődik. Ugyanaz a szám nem szerepelhet kétszer.

publikus metódusok:

public List<Integer> selectWinningNumbers(int lotteryType, int ballCount) Tippek

Nézz utána, hogyan lehet List típusok esetén az elemeket rendezni (mi ennek a feltétele?), a rendezettséget megszüntetni.

Gyakorlati feladat 2 - Számsorsolás

Hasonlít a korábbi feladatra, itt is véletlen számok kihúzásáról van szó, ám más az implementáció. A véletlen számok létrehozása azonos is lehet, de ArrayList helyett itt LinkedList a konkrét implementáció. A húzásnál használjuk ki azt, hogy a kapott LinkedList mint Queue is kezelhető, és a Queue poll() metódusával szedhetők ki a nyerő számok.

Hibakezelés

Ha a public Set<Integer> drawNumbers(int drawCount, int maxNumber) metódust rossz paraméterekkel hívták meg (több vagy ugyanannyi számot kellene kihúzni, mint amennyit generál), a metódus dobjon egy IllegalArgumentException kivételt.

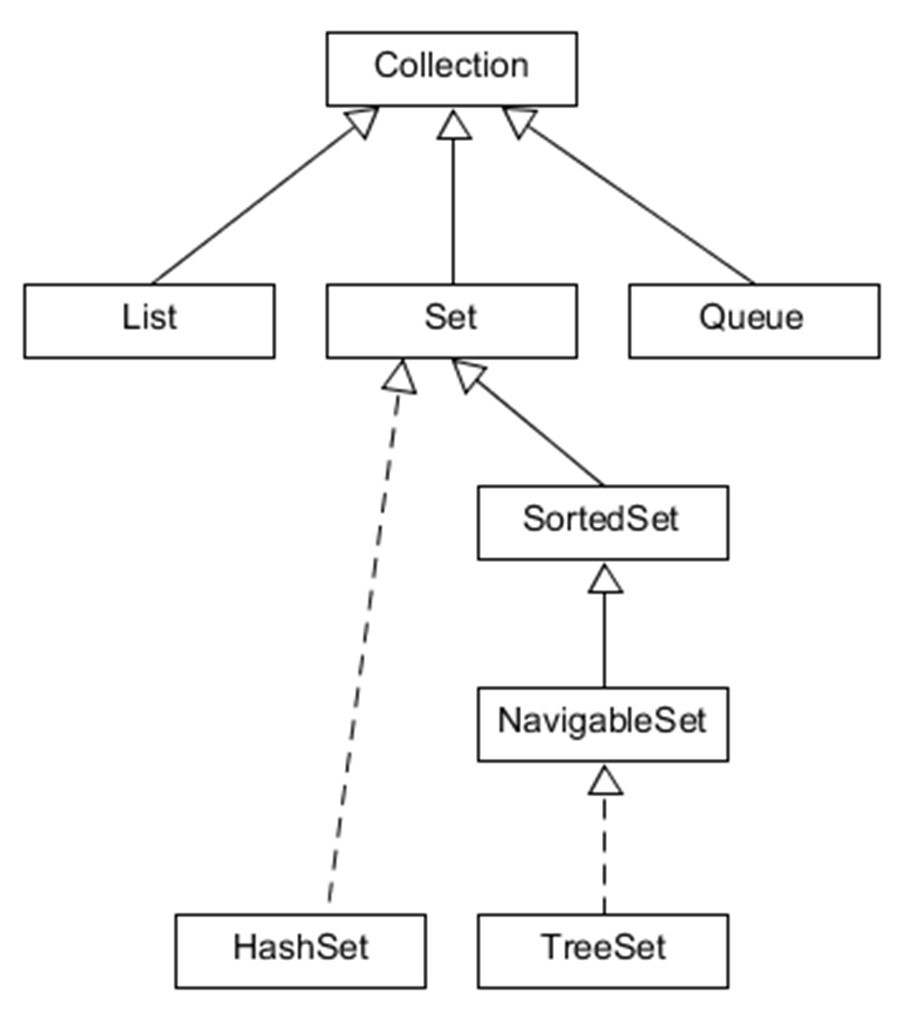
Megvalósítás

A kapott List típust át kell alakítani Queue típussá (közvetlen cast nem lehetséges!) és a kihúzott számok sorba rendezése TreeSet alkalmazásával történjen.

publikus metódusok: public Set<Integer> drawNumbers(int drawCount, int maxNumber)

Set (collectionsset)

A Set megfelel a matematikai halmaz fogalmának, ezért nem enged meg duplikációt. Duplikációnak számít, ha a két elem esetén az e1.equals(e2) igazzal tér vissza. Két gyakran használt implementációja a HashSet és a TreeSet. Ez utóbbi a NavigableSet-et is implementálja. Set-et főként akkor használunk, ha a hangsúly pusztán a tartalmazáson van, az nem számít, hogy a keresett elem hányadik, vagy milyen “szomszédjai” vannak.



sets

HashSet

A HashSet az elemeket a hashCode-juk alapján úgynevezett bucket-ekbe, kupacokba sorolja, ezért a visszakeresés gyorsabb, mint lista esetén, hiszen ha megtaláltuk a megfelelő kupacot, akkor már csak az abban lévő elemeket kell átnézni. Nagyon fontos, hogy egy osztályban deklarált equals() és hashCode() metódusok betartsák a szabályokat, hiszen ha két egyenlőnek tekintett objektum nem ugyanazt a hashCode-ot adja vissza, akkor egy HashSet-ben sosem fogjuk megtalálni. Lehet benne null elem, de csak egyszer.

TreeSet

A TreeSet az elemeket keresőfában tárolja, ezért képes azokat nagyság szerint rendezetten visszaadni. Ehhez persze az kell, hogy az elemek összehasonlíthatók legyenek egymással. Természetes rendezettsége van a számoknak és a String típusnak is, de saját osztályban is definiálhatunk rendezettséget. A null elem nem megengedett, hiszen ez nem hasonlítható össze a többivel.

Ellenőrző kérdések

* Milyen tulajdonságokkal rendelkezik a Set interfész?
* Hogyan vizsgálja a duplikációt?
* Milyen implementációkat ismersz, miben térnek el ezek egymástól?

Gyakorlati feladat 1 - HashSet alkalmazása

Szolgáltatásokat készítünk, egyrészt egy nagy elemszámú, random módon generált String kollekcióból akarjuk kiszűrni az egyedi elemeket, másrész két különböző String halmazból akarjuk kinyerni a közös elemeket.

A StringsHandler osztályban a következő publikus metódusok találhatók (ezek persze a tesztesetekből is következnek):

public Set<String> filterUniqueStrings(Collection<String> stringCollection) public Set<String> selectIdenticalStrings(Set<String> setA, Set<String> setB)

* Feltételezzük, hogy a véletlenszerűen generált String objektumok között valamennyi azonos is található.
* Feltételezzük, hogy két (vagy több) egyénileg, véletlenszerűen generált String kollekcióban számos azonos String található.

Gyakorlati feladat 2 - TreeSet alkalmazása

Egy String tömbből akarjuk kiszűrni az egyedi elemeket, és ezeket sorba is akarjuk rendezni, natív order, azaz itt abc szerint.

A WordFilter osztályban a következő publikus metódus található (ez persze a tesztesetekből is következik):

public Set<String> filterWords(String[] randomStrings)

Megvalósítás

Használjuk ki a TreeSet rendezettségét. A tesztelés során megvizsgáljuk a Set várható méretét, valamint a kollekció első és utolsó elemét!

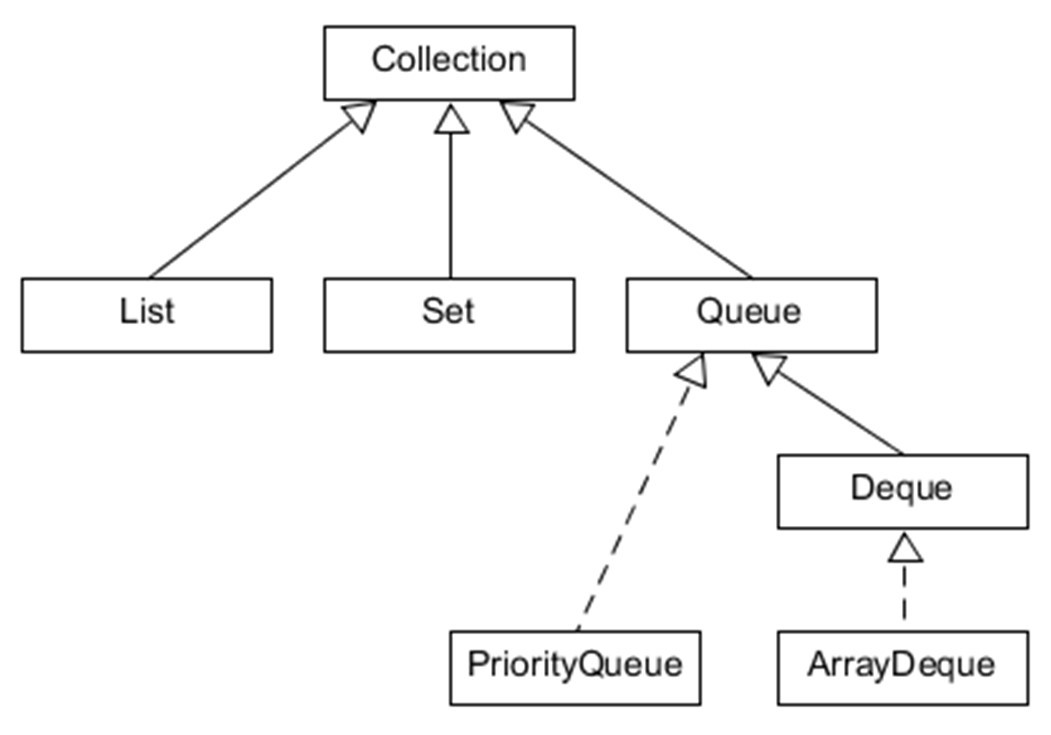
Queue (collectionsqueue)

A Queue olyan adatszerkezet, amely esetén az elemek hozzáadása és eltávolítása meghatározott sorrendben történik. Az eltávolítás sorrendje lehet

* a beszúrás sorrendje (FIFO - First In First Out) (sor),
* a beszúrás sorrendjével fordítottan (LIFO - Last In First Out) (verem), • rendezettség esetén az elsőt (azaz “legkisebbet”) először (prioritási sor).

A Deque interfész kiterjeszti a Queue-t, és elemek hozzáadását és kivételét a sor mindkét végénél lehetővé teszi (kétvégű sor). A sor elejét fejnek, a végét faroknak is hívják.

Queue implementációja a már megismert LinkedList, valamint a PriorityQueue, a Deque implementációja az ArrayDeque.



queues

Queue metódusok

A műveleteket két nagy csoportba sorolhatjuk. Az első csoportba azok tartoznak, ahol sikertelen művelet esetén kivételt kapunk, a másodikba azok, amelyek sikertelenség esetén valamilyen speciális értékkel (pl. null) térnek vissza.

Kivételt dob Speciális érték

Elem hozzáadása boolean add(E e) boolean offer(E e)

Elem kivétele E remove() E poll()

Fej lekérdezése E element() E peek()

A beszúrás a végére, vagy prioritási sor esetén a sorrend által meghatározott helyére történik, a kivétel azonban mindig a sor elejéről.

Deque metódusok

Amennyiben veremként (LIFO) kezeljük a kétvégű sort, akkor az alábbi műveleteket használhatjuk:

void push(E e): elem hozzáadás a verem elejéhez

E pop(): elem kivétele a verem elejéről

A kétvégű sor mindkét végéhez adhatunk hozzá, és vehetünk ki onnan elemeket. Ehhez a Queue metódusainak First és Last végződésű változatait használhatjuk. Pl. addFirst(E e), peekLast() metódusok.

Ellenőrző kérdések

* Milyen jellemzői vannak a Queue interfésznek?
* Milyen implementációit ismered?
* Mit jelentenek a FIFO és LIFO fogalmak?
* Milyen metódusokat ismersz?

Gyakorlati feladat 1 - JobDispatcher

A feladatokat (job) tetszés szerinti sorrendben felvisszük egy PriorityQueue típusú kollekcióba, és azt várjuk, hogy mindig a soron következő legfontosabb feladatot adja ki.

Írj egy JobDispatcher osztályt, amely tárolja és rendezi a Job típusú objektumokat.

A kollekció feltöltését a (Queue<Job> addJob(Job... jobs)) metódus végzi, amely egy

(PriorityQueue<Job>) értéket ad vissza. A soron következő feladatot a (Job dispatchNextJob(Queue<Job> jobs) throws NoJobException) metódus adja ki a queue-ból.

Hibakezelés

Üres queue dobjon NoJobException kivételt.

Megvalósítás

Hozz létre egy saját kivétel osztályt NoJobException néven, ezt akkor dobja a megfelelő metódus, ha a queue üres.

Hozzunk létre egy Job osztályt, a következők szerint: Legyen 3 final attribútuma:

int priority // 1 - 10 skálán osztályozzuk a prioritást String jobDescription boolean urgent // minden olyan job esetén true, ahol a prioritás < 5

Legyen egy public Job(int priority, String jobDescription) konstruktora. Implementálja a Comparable interfészt, a kisebb számértékű prioritás van előbb a sorban.

Tippek

A queue feltöltésekor használj varargs szerkezetet. Így tetszés szerinti számú objektum átadható paraméterként.

Gyakorlati feladat 2 - Deque implementáció

A Deque interfész egy kettős végű sor szerkezetet ír le, ennek az egyik implementációja az ArrayDeque kollekció. A feladatokat most kétféle minősítéssel látjuk el, fontos az, amelynek a prioritása kisebb, mint 5 a többi nem fontos. Fontos feladatok a sor elejére

(head) addFirst() a nem fontosak a végére kerülnek addLast(). A feltöltés a fentivel azonos módon működhet.

Hibakezelés

Üres queue dobjon NoJobException kivételt. Megvalósítás publikus metódusok:

public Deque<Job> addJobByUrgency(Job... jobs) public Job dispatchUrgentJob(Deque<Job> jobs) throws NoJobException public Job dispatchNotUrgentJob(Deque<Job> jobs) throws NoJobException

Az ArrayDeque implementáció rendelkezik getFirst() és getLast() metódusokkal Tippek

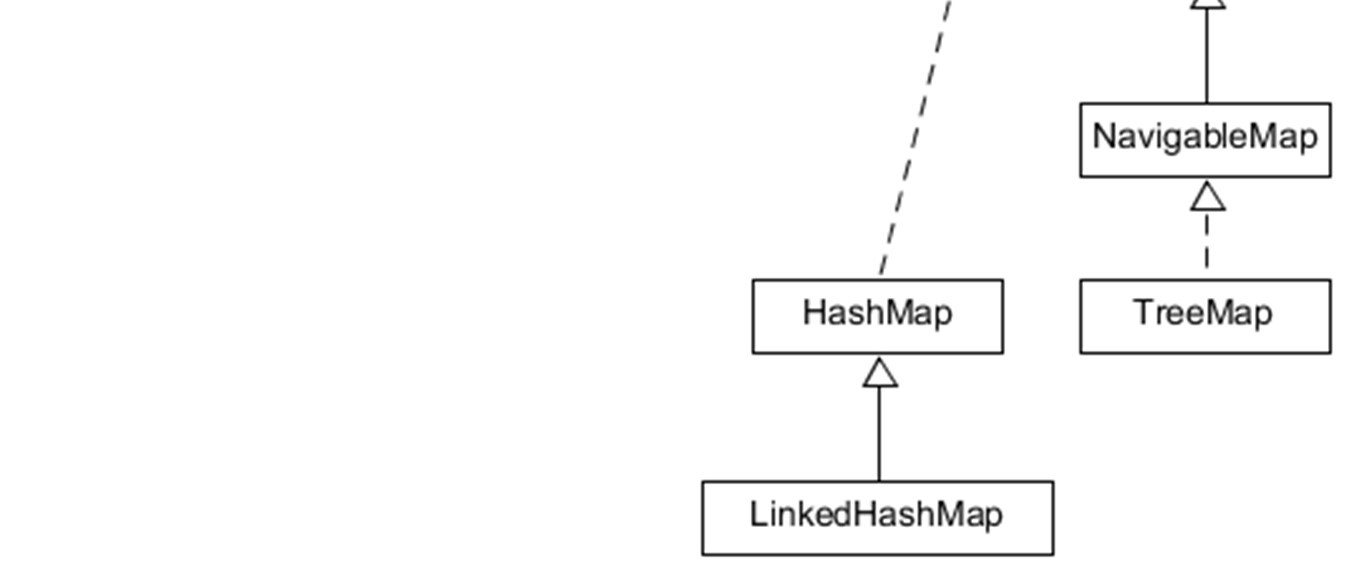
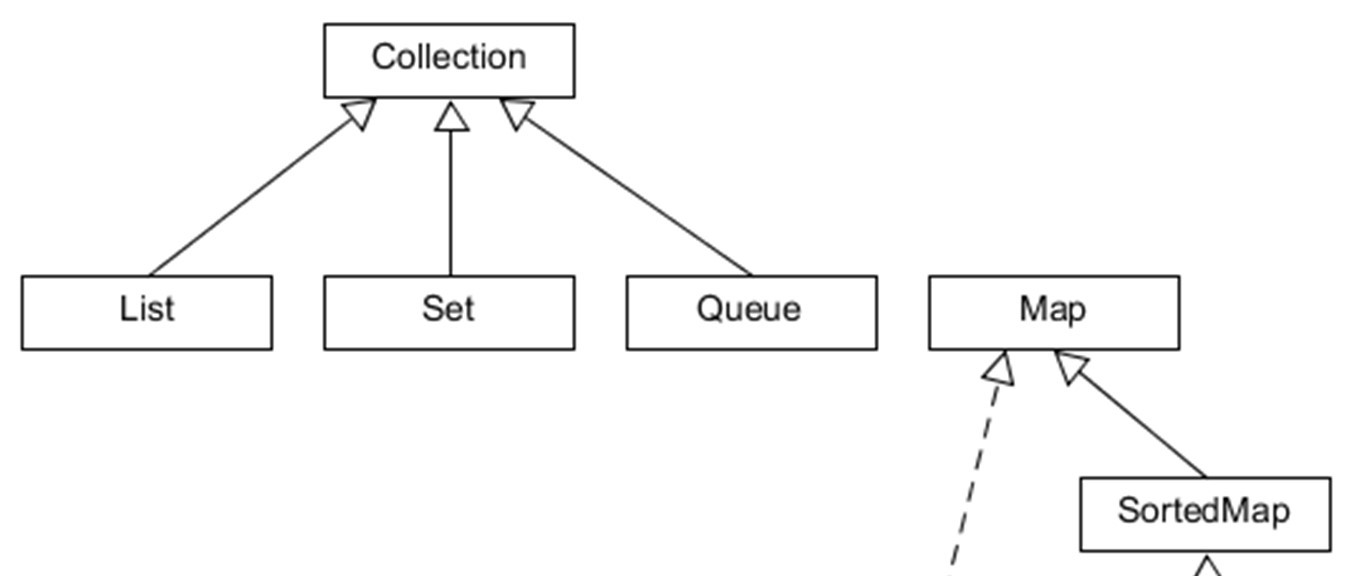
Hozzunk létre egy Job osztályt, a következők szerint: Legyen 3 final attribútuma:

int priority // 1 - 10 skálán osztályozzuk a prioritást String jobDescription boolean urgent // minden olyan job esetén true, ahol a prioritás < 5

Legyen egy public Job(int priority, String jobDescription) konstruktora. Implementálja a Comparable interfészt, a kisebb számértékű prioritás van előbb a sorban.

Map (collectionsmap)

A Map kulcs-érték párok tárolását valósítja meg. Előnye, hogy az eltárolt érték a rá jellemző kulcs alapján nagyon gyorsan visszakereshető. A kulcs mindig egyedi kell legyen, és csak egyetlen érték társítható hozzá. A kulcs és az érték adattípusa a Map deklarációjakor generikusként adható meg. Mivel a két típus eltérhet egymástól, mindkettőt külön meg kell adnunk. Például a Map<Integer, String> deklaráció szerint a kulcs Integer, míg az érték String típusú. Ugyanazon kulcshoz csak egy érték tartozhat, de ez lehet akár kollekció is. A párok a Map.Entry interfészt implementálják, azaz mondhatjuk, hogy egy Map<Integer, String> kollekcióban Map.Entry<Integer, String> elemek vannak. Ebben az interfészben két metódus segíti külön a kulcs és külön az érték lekérdezését. A kulcsok halmaza és az értékek kollekciója külön-külön is kinyerhető a Map-ből. Három gyakran használt implementációja a HashMap, a LinkedHashMap és a TreeMap.



maps

HashMap

A kulcsokat hashCode-juk alapján kupacokba (bucket) szervezi, így az értékek megtalálása kulcs alapján ugyanolyan gyors, mint HashSet esetén. Az elemek sorrendje nem meghatározott. Elfogad null értéket kulcsként és értékként is.

LinkedHashMap

A HashMap olyan változata, ahol a bejárás sorrendje állandó, és a beszúrás sorrendjét tükrözi. Megjegyzendő, hogy ha létező kulccsal szúrunk be új értéket, akkor az nem a map végére, hanem az eredeti bejegyzés helyére kerül.

TreeMap

A map elemei keresőfába kerülnek, azért kulcs alapján rendezetten kapjuk vissza a bejegyzéseket. A rendezéshez a kulcsoknak összehasonlíthatónak kell lenniük, ezért a kulcs értéke sosem lehet null. Map<K, V> interfész metódusai int size(): a Map bejegyzéseinek száma.

boolean isEmpty(): igaz, ha a Map mérete nulla. void clear(): törli az összes bejegyzést.

V get(Object key): a paraméterként átadott kulcshoz hozzárendelt értéket adja vissza, vagy null-t, ha a kulcs nem található a Map-ben.

V put(K key, V value): új kulcs-érték pár beszúrása. Ha a Map-ben már létezett a kulcs, akkor a hozzárendelt értéket lecseréli az újra, és a régi értékkel tér vissza. Ha még nem létezett a kulcs, akkor új bejegyzést készít, és null-lal tér vissza.

V remove(Object key): törli az adott kulcshoz tartozó bejegyzést. Ha a kulcshoz tartozott érték, akkor azzal tér vissza, ha nem tartozott, akkor null-lal.

boolean containsKey(Object key): igazzal tér vissza, ha a megadott kulcs létezik a Map-ben.

boolean containsValue(Object value): igazzal tér vissza, ha a megadott érték legalább egy kulcshoz hozzá van rendelve.

Set<Map.Entry<K, V>> entrySet(): a Map bejegyzéseinek halmazával tér vissza.

Set<K> keySet(): a kulcsok halmazát adja vissza.

Collection<V> values(): az értékek sokaságát adja vissza.

Az entrySet(), keySet() és values() műveletek által visszaadott kollekciók mögött ott van az eredeti Map, így ha bármelyik változik, az hatással van a másikra is. A visszaadott kollekciókon nem hívható meg az add és addAll művelet.

Ellenőrző kérdések

* Miket tárolnak a Map interfész implementációi?
* Hogyan lehet végigiterálni az elemeken?
* Milyen implementációkat ismersz?
* Hogyan működik a HashMap?
* Milyen metódusokat ismersz?

Gyakorlati feladat - Napló állomány elemzése

Gyűjtsd ki az ugyanahhoz az IP-címhez tartozó napló bejegyzéseket egy szöveges naplóból. A napló sorokban tárolja az adatokat, egy sor egy Entry objektumnak felel meg, IP cím, dátum és login mezőkből áll. Írj egy log feldolgozást implementáló osztályt, LogParser néven, amely elvégzi a dátum String konvertálását, a sor feldolgozását, és Map-be szortírozását. A fő tevékenységet a (parseLog(String log)) metódus végzi, amely egy (Map<String, List<Entry>>) értéket ad vissza.

Hibakezelés

Ha a dátum String nem dolgozható fel, a metódus dobjon egy IllegalArgumentException kivételt.

Ha a parseLog(String log) metódus bemenete olyan String, amely nem dolgozható fel, mint Entry objektum, szintén IllegalArgumentException kivételt dobjon.

Megvalósítás

A sorfeldolgozáshoz szükséges adatok - elemek száma, pozíciója, mező szeparátor, dátum String formátuma - statikus final változóként kerüljenek be a LogParser osztályba.

Tippek

Ha a sor nem dolgozható fel, vagy a dátum nem konvertálható, az egész feldolgozást töröljük és kivételt dobunk a probléma megjelölésével.

Ellenőrző kérdés

Mi várható, ha nem HashMap, hanem TreeMap az implementáció? Mikor előnyös egyik, vagy másik?

Autoboxing használata kollekcióknál (collectionsautoboxing)

Kollekciók csak objektumokat tudnak tárolni, primitív típusokat nem. Minden primitív típusnak megvan a csomagoló osztálya, amely már megadható a kollekció generikus típusaként. Amikor primitív típusú elemet adnánk hozzá, az automatikusan a neki megfelelő osztályba “csomagolódik” be (autoboxing), és amikor a kivett elemet primitív típusú változóba tennénk, akkor a Java futtatókörnyezet “kicsomagolja” a kivett objektumot (autounboxing).

List<Integer> numbers = new ArrayList<>(); int a = 1; numbers.add(a); int b = numbers.get(0);

Primitív típusok és csomagoló osztályaik

Primitív típus Csomagoló osztálya

|  |  |
| --- | --- |
| boolean | Boolean |
| byte | Byte |
| short | Short |
| int | Integer |
| long | Long |
| float | Float |
| double | Double |
| char | Character |

A null érték használata

A kollekciók többsége megengedi a null érték tárolását. Ez alól csak a TreeSet és a TreeMap a kivétel, hiszen ezek az elemeket rendezetten tárolják, a null értékről pedig nem dönthető el, hogy egy elemnél kisebb vagy nagyobb. (Egyedi Comparator készítésével ez kiküszöbölhető, de erről később lesz szó.) Problémába ütközik a futtatókörnyezet, amikor a kollekcióban lévő null elemet kellene kicsomagolnia primitív típussá, mert ennek megfelelő érték nem létezik. Ilyenkor futási időben NullPointerException-t kapunk.

List<Integer> numbers = new ArrayList<>(); numbers.add(null); //ez megengedett művelet

int a = numbers.get(0); //ez is megengedett művelet, de futás közben NullPointerException-t dob

Ellenőrző kérdések

* Mi az autoboxing szerepe a kollekciók esetén?
* “Feltalálása” előtt hogyan lehetett egy kollekcióba primitív értékeket rakni és kivenni?
* Hogyan kezelik a null értéket a kollekciók?
* Mikor kaphatunk autoboxing esetén NullPointerException kivételt?

Gyakorlati feladat - Az autoboxing működése Integer objektumok összeadása és kollekció esetében.

Implementáld az IntegerOperations osztályban a következő metódusokat!

public List<Integer> createList(int... numbers) public int sumIntegerList(List<Integer> integerList) public int sumIntegerObjects(Integer... integers)

Az egyes kollekciók esetében vizsgáljuk meg és értelmezzük a toString() metódus működését, a kiírás sorrendjének elemzésével.

Megvalósítás

Használjunk varargs típusokat a paraméterek megadásához.

## Alapvető algoritmusok

Bejárás (collectionsiterator)

A kollekciók bejárása iterátor alkalmazásával is lehetséges. Az Iterator interfészt implementáló osztály képes a kollekció elemeit bejárni. Az iterátort a kollekció iterator() metódusával lehet lekérni.

Műveletei:

boolean hasNext(): igazzal tér vissza, ha van még elem a kollekcióban

E next(): a következő elem referenciájával tér vissza. Ha nincs több elem, NoSuchElementException-t dob.

void remove(): törli az utoljára visszaadott elemet. Nem minden kollekció engedi, ezért dobhat UnsupportedOperationExceprion-t.

A kollekció bejárása közbeni módosítás (elem hozzáadása, törlése) az iterátor inkonzisztens állapotához vezet, és a program ConcurrentModificationException-t dob. A következő kód törli a páros számokat a listából.

List<Integer> values = new ArrayList<>(Arrays.asList(1, 2, 3, 4, 5)); for (Iterator<Integer> i = values.iterator(); i.hasNext()) { int value = i.next(); if (value % 2 == 0) { i.remove();

}

}

Ellenőrző kérdések

* Mire való az Iterator interfész?
* Hogyan lehet hozzájutni egy Iterator példányhoz?
* Milyen metódusait ismered?
* Mi történik, ha iterátoros bejárás közben akarsz módosítani egy Collection példányt?

Gyakorlati feladat - LibraryManager

Készítsünk olyan osztályt, ami egy könyvtárban tárolt könyvek szoftveres menedzselését valósítja meg. A könyvtárat feltöltjük egyedi könyvekkel, és egyedi regisztrációs szám alapján kikereshetjük, eltávolíthatjuk a könyvtár állományából, illetve szerző szerint kigyűjthetünk könyveket.

Megvalósítás

Book osztály int regNumber String title és String author attribútumokkal.

Az equals() és hashCode() metódusokat az igényeknek megfelelően készítsük el, egyedi a regNumber attribútum.

LibraryManager osztály Set<Book> libraryBooks attribútummal. Ezt konstruktorból tudjuk feltölteni.

publikus metódusok:

public Book findBookByRegNumber(int regNumber) public int removeBookByRegNumber(int regNumber) public Set<Book> selectBooksByAuthor(String author)

Mindhárom metódus saját MissingBookException-t (RuntimeException) dob, ha a regisztrációs számnak megfelelő könyv nem található, vagy adott szerzővel nem található könyv.

Tipp

Használjunk iterátoros bejárást a könyvek megtalálására és kigyűjtésére. A tesztelés segítésére készítsünk egy public int libraryBooksCount() metódust is, ami visszaadja a Book kollekció méretét.

[rating feedback=java-collectionsiterator-librarymanager]

Comparator és Comparable (collectionscomp)

Ahhoz, hogy egy osztály példányait rendezni tudjuk, tudnunk kell, melyik számít kisebbnek. Ezt számok esetén már természetesnek vesszük, a String-ek esetén viszont már nem működik mindig jól az alfabetikus rendezés, hiszen a karakterkódok szerint az a betű nagyobb, mint a Z, az ékezetesekkel meg még több probléma van. Két Person objektum közül meg nem is tudjuk, melyik számít nagyobbnak. Aki idősebb, vagy aki magasabb? Vagy esetleg a névsorrend számít?

Bárhogy is döntünk, ezt tudatnunk kell a futtatókörnyezettel, hogy el tudja végezni a rendezést. TreeSet és TreeMap megköveteli tőlünk, hogy az elemek, illetve a kulcsok rendezhetőek legyenek. Döntésünket kétféle módon közölhetjük. Az osztályunk implementálhatja a Comparable interfészt vagy átadhatunk a rendező metódusnak vagy a kollekció konstruktorának egy Comparator interfészt implementáló osztályt.

Comparable interfész

public interface Comparable<T> {

public int compareTo(T o);

}

Amikor az osztályunk implementálja ezt az interfészt, akkor az általa megvalósított rendezettséget hívjuk természetes sorrendnek (natural order). Ilyen módon kizárólag egyféle rendezettséget valósíthatunk meg, de az akár több attribútum értékét is figyelembe veheti. A compareTo() metódus az osztály egy példányát kapja paraméterként és egy egész számot kell visszaadjon, mely negatív, ha a hívó példány kisebb, mint a paraméterül kapott, 0, ha egyenlők és pozitív, ha a paraméter számít nagyobbnak. Jó gyakorlat úgy megalkotni ezt a metódust, hogy a megfelelő attribútumra delegáljuk a hívást.

Figyeljünk arra, hogy a compareTo() és az equals() metódus összhangban legyen! Ha az equals() metódus igazzal tér vissza, akkor a compareTo() metódus értéke 0 legyen és viszont.

A következő példa a Trainer objektumokra az id alapján definiál növekvő rendezettséget.

class Trainer implements Comparable<Trainer> {

private int id;

private String name;

@Override public int compareTo(Trainer other) { return id.compareTo(other.id);

}

}

Comparator interfész

A Comparator előnye, hogy egyféle osztályhoz többféle rendezettséget is definiálhatunk, illetve olyan osztályok rendezettségét is definiálhatjuk, amelyek nem implementálják a Comperable interfészt. A Comparator interfész egyetlen metódust tartalmaz, a compare()-t, mely a két összehasonlítandó objektumot kapja meg paraméterként, és egy egész számmal tér vissza.

public interface Comparator<T>{

int compare(T o1, T o2);

}

Ezt az osztályt felhasználhatjuk a kollekciók rendezésekor, illetve TreeSet és TreeMap esetén a konstruktornak átadva ez adja meg a rendezési szempontot.

A Trainer osztályhoz definiálhatunk egy név szerinti rendezettséget is.

public class NameComparator implements Comparator<Trainer> {

public int compare(Trainer o1, Trainer o2) { return o1.getName().compareTo(o2.getName());

}

}

Collator absztrakt osztály

Említettük, hogy a String-ek rendezése a különböző ékezetes és speciális karakterek miatt nehézkes. Erre nyújt megoldást a Collator absztrakt osztály, mely implementálja a Comparator interfészt, és figyelembe veszi a helyi adottságokat, vagy az átadott Locale objektumot.

Ha a fenti Trainer osztályban a nevek a magyar ékezeteseket is tartalmazhatnak, akkor a NameComparator már nem fog helyesen működni, módosítanunk kell a kódon. A megfelelő Collator-t a Collator.getInstance() metódussal kérhetjük le.

Ellenőrző kérdések

* Mit takar a “natural ordering”?
* Hogyan kell használni a Comparable interfészt?
* A compareTo() metódus visszatérési értékei mit jelentenek?
* A compareTo() metódusnak milyen más metódussal kell összhangban lennie?
* Milyen osztályokat ismersz, amelyek implementálják a Comparable interfészt?
* Mire való a Collator?
* Hogyan lehet hozzájutni egy Collator példányhoz?

Gyakorlati feladat - OrderedLibrary

Készítsünk olyan osztályt, ami egy könyvtárban tárolt könyvek szoftveres menedzselését valósítja meg. A könyvtárat feltöltjük könyvekkel, és kereséshez cím vagy szerő szerint rendezhetjük a könyveket. Alapértelmezett a cím szerinti rendezés. Egy külön metódusban valósítsuk meg a könyvcímek magyar ékezetes karaktereket is figyelembe vevő rendezését. Csak a könyvcímek kellenek, nem a könyvek!

Megvalósítás

Book osztály int regNumber String title és String author attribútumokkal.

A public int compareTo(Book o) metódust az igényeknek megfelelően készítsük el.

OrderedLibrary osztály List<Book> libraryBooks attribútummal. Ezt konstruktorból tudjuk feltölteni.

publikus metódusok:

public OrderedLibrary(List<Book> libraryBooks) public List<Book> orderedByTitle() public List<Book> orderedByAuthor()

public List<String> orderedByTitleLocale(Locale locale)

Egy külön osztályban (AuthorComparator) valósítsuk meg a szerző szerint történő rendezéshez szükséges Comparator osztályt.

Bónusz feladat - Collator

Az ékezetes karaktereket is figyelembe vevő String rendezéshez a Collator osztály egy példányára lesz szükségünk. Nézz utána, ez hogyan állítható elő és hogyan paraméterezhető!

Bónusz feladat - Comparator

A Comparator objektumot többnyire névtelen osztály példányaként állítjuk elő. Nézz utána, ez hogyan valósítható meg!

Keresés (searching)

Sok-sok objektum között nem könnyű keresni. Laikusként szépen egyesével megvizsgálnánk mind, míg meg nem találjuk a számunkra megfelelőt. Ez bizony hosszadalmas lehet, ha nincs szerencsék, csak az utolsó megvizsgált elem lesz jó. N elem esetén legrosszabb esetben n, de átlagosan is n/2 elemet meg kell vizsgálnunk. Sokkal könnyebb a keresés, ha az elemek a keresési szempont szerint rendezettek. Ebben az esetben ha megvizsgáljuk a középső elemet, és az nem a keresett, máris tudjuk, hogy csak az előtte vagy csak az utána lévőket között lehet, amit keresünk. Máris megfeleztük az átvizsgálandó elemek számát. Ha ezt a stratégiát folytatjuk, akkor legrosszabb esetben is log(n) lépés alatt megtaláljuk a keresettet, vagy éppen biztosan állíthatjuk, hogy nincs a sokaságban. Az első módszert lineáris, a másodikat bináris keresésnek nevezzük. Bármilyen hatékonynak is tűnik a második módszer, az sajnos csak rendezett sorozatra használható, különben hibás eredményt ad.

Keresés tömbben

Rendezett tömbben az Arrays.binarySearch() statikus metódussal tudunk egy adott elemet binárisan megkeresni. Első paraméterként a tömböt, másodikként a keresett elemet kell átadni. A kereséshez kell, hogy az elemeknek legyen természetes rendezettsége, illetve ha nincs, akkor harmadik paraméterként egy Comparator-t is át kell adni.

Amennyiben nem találja meg az elemet, akkor negatív számmal tér vissza. A szám azt is mutatja, hogy hol kellene lennie a keresett elemnek. Ha például a kapott index -6, akkor az azt jelenti, hogy a keresett elemnek a 6.-nak, azaz az 5-ös indexű elemnek kellene lennie.

Keresés kollekciókban

Arra a kérdésre, hogy egy adott elem létezik-e a kollekcióban, a kollekció contains() metódusa ad választ. Ha egyszerre több elem létezését szeretnénk megvizsgálni, akkor a containsAll() metódust kell használnunk, mely a keresett elemek kollekcióját várja paraméterként, és csak akkor ad vissza igazat, ha mind megtalálta. Lineáris kereséssel a keresett elem helyét az indexOf() metódussal kaphatjuk meg. Ez csak listákra értelmezett, mert csak itt létezik az index fogalma. Ha a keresett elem nem létezik, -1-et ad. Ezek a keresések kizárólag az elemeken értelmezett equals() metódussal működnek.

Bináris keresésre a Collections.binarySearch() statikus metódust használhatjuk, amely listákban képes egy elemet megkeresni és a találat indexével tér vissza. A metódus a rendezettséget definiáló Comparator-t is megkaphatja, ha az elemeknek nincs természetes rendezettsége vagy az eltérő, mint ami szerint keresünk. Természetesen most is a rendezettnek kell lennie a listának, különben nem kapunk helyes eredményt.

Kollekciókban megkereshetjük a legnagyobb és legkisebb elemet is a

Collections.min() és a Collections.max() statikus metódusokkal. Paraméterként a kollekciót kell átadnunk. Az elemeknek természetes rendezettsége alapján fogja visszaadni a kollekció legnagyobb, illetve legkisebb elemét, de magunk is adhatunk meg Comparator-t.

Ellenőrző kérdések

* Milyen keresési algoritmusokat ismersz?
* Hogyan keresel egy rendezett tömbben?
* Hogyan nézed meg egy kollekcióban, hogy egy elem benne van-e?
* Hogyan vizsgálod meg, hogy egy elem egy listában hányadik indexen szerepel?
* Hogyan keresel egy rendezett kollekcióban?
* Mely metódus működik natural ordering, és mely metódus Comparator alapján?

Gyakorlati feladat - BookArraySearch

Készítsünk olyan osztályt, ami egy könyvtárban tárolt könyvek között tud keresni adott szerzőre és címre. Az osztályban a könyveket objektum tömb formájában tároljuk, a keresés legyen bináris keresés.

Megvalósítás

Book osztály int id String title és String author attribútumokkal.

A public int compareTo(Book o) metódust az igényeknek megfelelően készítsük el. Alapértelmezetten szerző és ezen belül cím alapján rendezi a Book objektumokat.

BookArraySearch osztály private Book[] bookArray attribútummal. Ezt konstruktorból tudjuk feltölteni.

publikus metódusok:

public BookArraySearch(Book[] bookArray) public Book findBookByAuthorTitle(String author, String title)

Kivételkezelés

A két String paramétert le kell ellenőrizni, üres String esetén dobjon IllegalArgumentException-t. Ha a bináris keresés nem talál könyvet, dobjon IllegalArgumentException-t a megfelelő tájékoztató szöveggel.

Rendezés (sorting)

Rendezési algoritmusok

Nagyon sokféle rendezési algoritmus létezik. Az, hogy melyik a leghatékonyabb, nagyban függ attól, hogy processzor- vagy memóriaerős gépünk van, illetve milyen a rendezendő tömb (kollekció) felépítése.

Rendezhetünk új kollekcióba másolással és ugyanazon sorozaton helyben is. Míg az első sokkal természetesebb és könnyebb, addig ez utóbbi helykímélőbb módja a rendezésnek.

Beszúrásos rendezés (Insertion sort)

A módszer lényege, hogy vesszük az elemeket egyenként, és a már rendezett sorozatban a helyére tesszük. Egy elem mindig rendezett, tulajdonképpen csak a második elem beszúrásától kezdve kell a helyét megkeresni. Helyben rendezésnél a sorozat eleje rendezett, és a még nem rendezett részből választott elemet szúrjuk be a helyére.

https://www.youtube.com/watch?v=kU9M51eKSX8

Buborékos rendezés (Bubble sort)

A módszer lényege, hogy csak az egymás melletti elemeket hasonlítjuk, és ha szükséges, megcseréljük őket. Így a legnagyobb elem a sorozat legvégére, a helyére kerül. Most újra elkezdjük a szomszédosok cseréjét az utolsó előtti elemig. A sorozat vége így mindig rendezett, csak az elején kell az elemeket újra cserélgetni. A nagy elemek így a sorozat vége felé “szállnak”, mint egy buborék.

https://www.youtube.com/watch?v=RT-hUXUWQ2I

Gyorsrendezés (Quicksort)

A sorozatot egy kiemelt eleménél (általában az utolsó) kisebb és nagyobb két részsorozatra bontjuk, a kisebbeket elé, a nagyobbakat mögé mozgatjuk, így a kiválatszott elem a helyére kerül. Az előtte és utána lévő még rendezetlen részsorozattal ugyanígy járunk el, míg a teljes sorozat rendezett nem lesz.

https://www.youtube.com/watch?v=aQiWF4E8flQ

Rendezés Javaban

Tömbök rendezésére az Arrays.sort() statikus metódust használhatjuk. Az elemeket természetes rendezettségük szerint rendezi, de Comparator-t is figyelembe tud venni.

A TreeSet és TreeMap eleve rendezetten adja vissza az elemeket, mert új elem hozzáadásakor már a helyére szúrja be.

Listák rendezésére a List sort() metódusát vagy bármilyen kollekcióhoz a Collections.sort() statikus metódust használhatjuk. Mindkettő az elemek természetes rendezettségét használja, illetve képes átadott Comparator-t is használni.

Ezek a fentieknél hatékonyabb Timsort algoritmust használják. Ha részletesebben is érdekel, nézd meg az alábbi videót. https://www.youtube.com/watch?v=jVXsjswWo44

Ellenőrző kérdések

* Milyen rendezési algoritmusokat ismersz?
* Hogyan rendezed le a tömbök elemeit?
* Milyen eleve rendezett kollekciókat ismersz?
* Hogyan rendezed le egy lista elemeit?

Gyakorlati feladat 1 - OrderedArrayLibrary

Készítsünk olyan osztályt, ami egy könyvtárban tárolt könyvek szoftveres rendezéseit valósítja meg. A könyvtárban a könyveket tömb segítségével tároljuk, ezt kell rendezni igény esetén különböző szempontok alapján.

Megvalósítás

Book osztály int id String title és String author attribútumokkal.

A public int compareTo(Book o) metódust az igényeknek megfelelően készítsük el. Alapértelmezett az id szerinti rendezés.

OrderedArrayLibrary osztály private Book[] bookArray attribútummal. Ezt konstruktorból tudjuk feltölteni.

publikus metódusok:

public OrderedArrayLibrary(Book[] bookArray) public Book[] sortingById() public Book[] sortingByTitle()

A rendezés során az eredeti tömb egy másolatát adjuk vissza, a megfelelő szempont szerint rendezve!

Tipp

A rendezéshez szükséges Comparator objektumot előállíthatjuk külön osztályból, vagy névtelen osztályból is.

[rating feedback=java-sorting-orderedarraylibrary]

Gyakorlati feladat 2 - OrderedLibrary

Készítsünk olyan osztályt, ami a könyvtárban tárolt könyveket alapvetően rendezett formában tárolja.

Megvalósítás

Book osztály int id String title és String author attribútumokkal.

A public int compareTo(Book o) metódust az igényeknek megfelelően készítsük el. Alapértelmezett az id szerinti rendezés.

OrderedLibrary osztály Set<Book> library attribútummal. Ezt konstruktorból tudjuk feltölteni.

publikus metódusok:

public OrderedLibrary(Set<Book> library) public Book lendFirstBook()

A rendezett kollekcióból adjuk ki az első könyv referenciáját “kölcsönzéshez”!

Tipp

Amennyiben a kollekció üres, a lendFirstBook() metódus dobjon egy NullPointerException-t a megfelelő tájékoztató szöveggel.

Collections osztály (keresésen, rendezésen felüli metódusok) (collectionsclass)

A Collections osztály ún. utility osztály, azaz kollekciókhoz tartalmaz metódusokat.

Már megismerkedtünk a kereső és rendező metódusaival, most ismerjük meg a többit is.

Az emptyXxx() metódusok üres kollekciót adnak vissza. Lehet emptyList(), emptySet() vagy emptyMap(), sőt néhány másik is. Ezek immutable kollekciókat adnak.

A singletonXxx(T o) metódusok egyelemű immutable kollekciót adnak vissza. Lehet singleton(T o), singletonList(T o) és singletonMap(K key, V value). Az első neve nem tartalmazza a kollekció típusát: Set-tel tér vissza.

A fenti két metóduscsoportnál figyeljünk arra, hogy elemeket sem hozzáadni, sem elvenni, sem módosítani nem lehet.

A synchronizedXxx() metódusok a paraméterül kapott kollekciók szálbiztos burkoló osztályával térnek vissza. Ezek mind nézetek, ami azt jelenti, hogy kapcsolatban vannak az eredeti kollekcióval, azaz az ezen történő változások megjelennek abban is és viszont. A szálbiztos kollekciók akkor is jól működnek, ha egyszerre több szál is módosítani próbálja a tartalmát.

Az unmodifiableXxx() metódusok a paraméterként átadott kollekció csak olvasható nézetét adják vissza, azaz ezek tartalma nem módosítható. Az eredeti lista módosítható marad, ha abban változtatunk, akkor az látszani fog az itt visszakapott listán is.

A listákon még további két gyakran használt metódust tartalmaz. A reverse(List<?> list) a paraméterül kapott listában megfordítja az elemek sorrendjét, a shuffle(List<?> list) pedig véletlenszerűen összekeveri az elemeket. Ez utóbbihoz második paraméterként átadhatjuk a keveréshez használt Random példányt is, ami mint már láttuk, seeddel irányítható véletlenszám generátor. Az így kapott keverés már kiszámítható és jól tesztelhető.

Ellenőrző kérdések

* Mire való a Collections osztály?
* Milyen metódusokat ismersz? Keresésre, rendezésre?
* Hogyan hozol létre üres kollekciókat?
* Hogyan hozol létre egy elemből álló kollekciókat?
* Hogyan hozol létre szinkronizált burkoló példányokat?
* Hogyan hozol létre módosíthatatlan burkoló példányokat?
* Hogyan fordítod meg egy lista elemeinek sorrendjét?
* Hogyan kevered meg egy lista elemeit? Hogy lehet ez pszeudorandom?
* Az elemek sorrendjének megfordítása, vagy a keverés miért csak listákon működik?
* Mit jelent az, hogy a burkoló példányok nézetek? Milyen viselkedéssel rendelkeznek?

Gyakorlati feladat - CollectionManager

Készítsünk olyan osztályt, ami a Collections osztály kiválasztott metódusai segítségével “könyvtári szolgáltatásokat” nyújt.

Megvalósítás

Book osztály int id String title és String author attribútumokkal.

A public int compareTo(Book o) metódust az igényeknek megfelelően készítsük el. Alapértelmezetten id alapján rendezi a Book objektumokat és az equals() metódus is ezen az attribútumon alapul.

CollectionManager osztály private List<Book> library attribútummal. Ezt konstruktorból tudjuk feltölteni.

publikus metódusok:

public CollectionManager(List<Book> library)

public List<Book> createUnmodifiableLibrary() //módosíthatatlan listát eredményez public List<Book> reverseLibrary() //az eredeti lista másolatán dolgozik! public Book getFirstBook() //a legrégebbi (legkisebb id) könyvet adja vissza public Book getLastBook() // a legújabb (legnagyobb id) könyvet adja vissza

A clone() metódus, deep clone (clone)

Az objektum klónozásának célja, hogy olyan új objektumot hozzunk létre, amely állapota megegyezik az eredeti objektum állapotával. A Cloneable marker interfésszel jelezhetjük, hogy az osztályunk példányai klónozhatók. A klónozást a clone() metódus implementálásával érhetjük el, mely nem a Cloneable interfészben, hanem az Object osztályban van deklarálva protected metódusként. Ennél jobb gyakorlat, ha copy konstruktort készítünk.

public class Auction {

private Product product;

private User user;

private LocalDateTime start;

private double price;

// Konstruktorok, getter, setter metódusok

public Auction(Auction auction) { product = auction.product; user = auction.user;

start = auction.start; price = auction.price;

}

}

Láthatjuk, hogy a copy konstruktor az Auction egy példányából úgy készít újat, hogy a referencia változói ugyanarra az objektumra mutatnak. Ezt a másolási módot shallow copy-nak nevezzük. A primitív típusú változókból másolat készül, a referencia típusú attribútumokon viszont osztozkodik a klónozott objektummal. Ha ez immutable, akkor nem jelent gondot, de ha nem az, akkor bármelyiken történő változtatás kihat mind a klónozott, mind a klón objektumra.

Másik megvalósítási mód a deep copy, amikor a referencia típusú attribútumokat is klónozzuk.

public Auction(Auction auction) { product = new Product(auction.product); user = auction.user; start = auction.start; price = auction.price;

}

Ellenőrző kérdések

* Mit értünk objektum klónozás alatt?
* Mi a standard megoldás?
* Mit érdemes inkább alkalmaznunk?
* Mit kell eldöntenünk a klónozás implementálásakor?

TimeSheetItem klónozása (clone)

A clone.timesheet.TimeSheetItem osztály tartalmazza, hogy egy alkalmazott mikor, min dolgozott. Van egy employee, project, from és egy to attribútuma.

Legyen egy konstruktora, mely ezekkel az adatokkal inicializálja. Azonban legyen egy copy konstruktora is, mely paraméterül kap egy TimeSheetItem példányt, és annak adatait átmásolja az új példányba.

Legyen egy statikus withDifferentDay(TimeSheetItem, LocalDate) metódusa is, mely lemásolja a paraméterként átadott bejegyzést, azonban a from és to attribútumokban szereplő dátumokat kicseréli a másodikként megadott napra, de az időket érintetlenül hagyja. Ez a statikus metódus hívja a copy konstruktort.

[rating feedback=java-clone-timesheetitem]

Gyakorlat - Issue klónozása (clone)

Egy hibabejelentő rendszer egy alkalmazással kapcsolatosan bejelentett hibákat tartalmazza.

Legyen egy clone.issuetracker.Issue osztály, mely a rendszerben lévő hibákat reprezentálja, egy name attribútummal, LocalDateTime time és egy Status status attribútummal. A Status egy enum NEW, IN\_PROGRESS és CLOSED értékekkel.

Az Issue tartalmazzon egy clone.issuetracker.Comment listát. A Comment tartalmazzon egy String text és egy LocalDateTime time attribútumot.

A Issue osztálynak legyen egy copy konstruktora, mely kap egy másik Issue példányt, valamint egy CopyMode enum értéket. Ez vagy CopyMode.WITH\_COMMENTS vagy CopyMode.WITHOUT\_COMMENTS. Előbbi esetben a megjegyzéseket is másolja, utóbbi esetben nem.

A megjegyzések is copy konstruktorral legyenek másolhatóak, és ez kerüljön meghívásra (deep copy).

[rating feedback=java-clone-issue]

Properties állományok (properties)

A Properties osztály olyan Map-hez hasonló adatszerkezet, mely kizárólag szöveges kulcs-érték párokat tartalmaz. Mivel ezek általában konfigurációs bejegyzéseket tartalmaznak, a bejegyzések streambe kiírhatók, illetve streamből betölthetők, ami alkalmassá teszi fájlba mentésre és betöltésre. Két támogatott szöveges fájlformátuma a properties és az XML.

A Properties osztály metódusai

String getProperty(String key) : az adott kulcshoz tartozó értéket kérdezi le. Ha nem létezik a kulcs, akkor null-al tér vissza.

String getProperty(String key, String defaultValue): a megadott kulcshoz tartozó értéket kérdezi le. Ha nem létezik a kulcs, akkor a defaultValue értékét adja vissza.

Object setProperty(String key, String value): új kulcs-érték pár beszúrása. Ha a kulcs már létezett, akkor a hozzá tartozó értéket lecseréli az itt megadottra, és a régi értéket adja vissza.

Set<String> stringPropertyNames(): az összes létező kulcs halmazát adja vissza. void load(InputStream is): kulcs-érték párok betöltése byte streamből. void load(Reader reader): kulcs-érték párok betöltése karakter streamből. void loadFromXML(InputStream is): kulcs-érték párok betöltése XML fájlból.

void store(Outputstream os, String comment): a kulcs-érték párok mentése OutputStream-be.

void store(Writer writer, String comments): a kulcs-érték párok mentése karakteres streambe.

void storeToXML(OutputStream os, String comment): a kulcs-érték párokkal XML dokumentumot készít.

void storeToXML(OutputStream os, String comment, String encoding): a kulcsérték párokkal XML dokumentumot készít a megadott karakterkódolással.

Properties állomány formátuma

A kulcs-érték párokat egyszerű .properties kiterjesztésű szöveges fájlba menthetjük. Az összetartozó kulcs és érték egy sorba kerül, közte = vagy : karakterrel. A különböző párokat külön-külön sorba írjuk. A sor eleji és az elválasztó karakter körüli szóközöket a load() metódus nem veszi figyelembe, de az érték után lévőket igen.

host = 192.168.0.1

port =80 // Ez ugyanaz, mint a port=80 protocol = http

Egy properties fájlon belül csak egyféle elválasztó karaktert használj!

XML állomány formátuma

Ahhoz, hogy az XML dokumentumból Propeties-be töltsünk adatokat, a fájlnak speciális szerkezettel kell rendelkeznie. A gyökérelem neve mindig properties, ebben helyezkednek el az entry tagek, amelyeknek a key attribútuma tartalmazza a kulcsot, és a hozzá tartozó értéket írjuk a nyitó és záró tag közé. Ha tartozik megjegyzés hozzá, akkor az az első entry tag előtt lévő comment tagbe kerül.

<?xml version="1.0" encoding="UTF-8" standalone="no"?>

<!DOCTYPE properties SYSTEM "http://java.sun.com/dtd/properties.dtd">

<properties>

<comment>XML properties file</comment>

<entry key="host">192.168.0.1</entry>

<entry key="port">80</entry>

<entry key="protocol">http</entry>

</properties>

Ellenőrző kérdések

* A Properties osztály milyen értékek tárolására való?
* Mi a specialitása más kollekciókhoz képest?
* Milyen állományformátumokat ismersz? Hogyan épülnek fel?
* Hogyan lehet állományból beolvasni az értékeket?
* Hogyan lehet állományba kiírni az értékeket?

Adatbázis konfiguráció beolvasása

Írj egy properties.DatabaseConfiguration osztályt, mely properties állományból betölti az adatbázis beállításokat, majd le lehet ezeket kérdezni. Létre lehet hozni paraméter nélküli konstruktorral, ekkor a classpath-ról tölti be a properties állományt. Van egy File paramétert váró konstruktora is, melyet megadva a beállításokat a paraméterként megadott fájlból tölti be.

Megvalósítási javaslatok

Figyelj a karakterkódolásra. IDEA-ban, hogy a properties fájlt UTF-8 kódolással hozza létre, át kell állítani, File / Settings / Editor / File Encodings ablakon a “Default encoding for properties files” értékét kell UTF-8-ra állítani. A karakterkódolás konstans értékként legyen megadva.

A betöltés a konstruktorban történjen. Érdemes felvenni egy Properties típusú attribútumot. A getHost(), getPort() és getSchema() metódusok ezt hívják.

[rating feedback=java-properties-adatbazisconfig]

Java eszközök nyilvántartása

Egy properties állományban tárold el a különböző Java eszközök leírásait a következő formátumban:

jdk.name = Java Development Kit

jdk.url = http://www.oracle.com/technetwork/java/javase/downloads/indexjsp-138363.html maven.name = Apache Maven maven.url = https://maven.apache.org/ junit.name = JUnit

junit.url = http://junit.org/junit4/

Írj egy properties.JavaTools osztályt, mely betölti ezen eszközöket a properties állományból. Legyen egy Set<String> getToolKeys() metódusa, mely visszaadja az eszközök kulcsait (pl. jdk, maven). Legyen egy Set<String> getTools() metódusa, mely visszaadja az eszközök neveit. Legyen egy String getName(String) metódusa, mely kulcs alapján visszaadja annak nevét, és egy String getUrl(String) metódusa, mely kulcs alapján visszaadja annak címét.

## Lambda kifejezések

Bevezetés a lambda kifejezések használatába (lambdaintro)

Funkcionális nyelvek

A Java nyelv tisztán objektumorientált, azaz a program építőelemei az objektumok. A funkcionális nyelvek alap építő kövei a függvények. A deklaratív nyelvek csoportjába tartoznak, azaz a programozónak nem azt kell megmondani, hogy hogyan szeretné elérni a célját, hanem csak azt, hogy mit szeretne elérni. Legjobban a matematikai függvényekre hasonlít, mivel a bemenethez egyértelmű kimenetet rendel. Mivel állapotot nem tárol, nincs értékadás. Ebből következően nincs semmilyen mellékhatása, a függvényeket a program bármely pontján hívva ugyanazt az eredményt kapjuk.

Funkcionális programozás Javaban

A Java 8-ban jelentek meg először funkcionális programozási elemek: lambda kifejezések és funkcionális interfészek.

A lambda kifejezés olyan kódblokk, amelyet paraméterként adhatunk át egy metódusnak. Tulajdonképpen egy névtelen metódus. Kollekciók esetén nagyon hasznos, például sorba rendezéskor egy egész Comparator objektum lecserélhető egyetlen lambda kifejezésre.

Rendezés anonymous inner class Comparatorral trainers.sort(new Comparator<Trainer> {

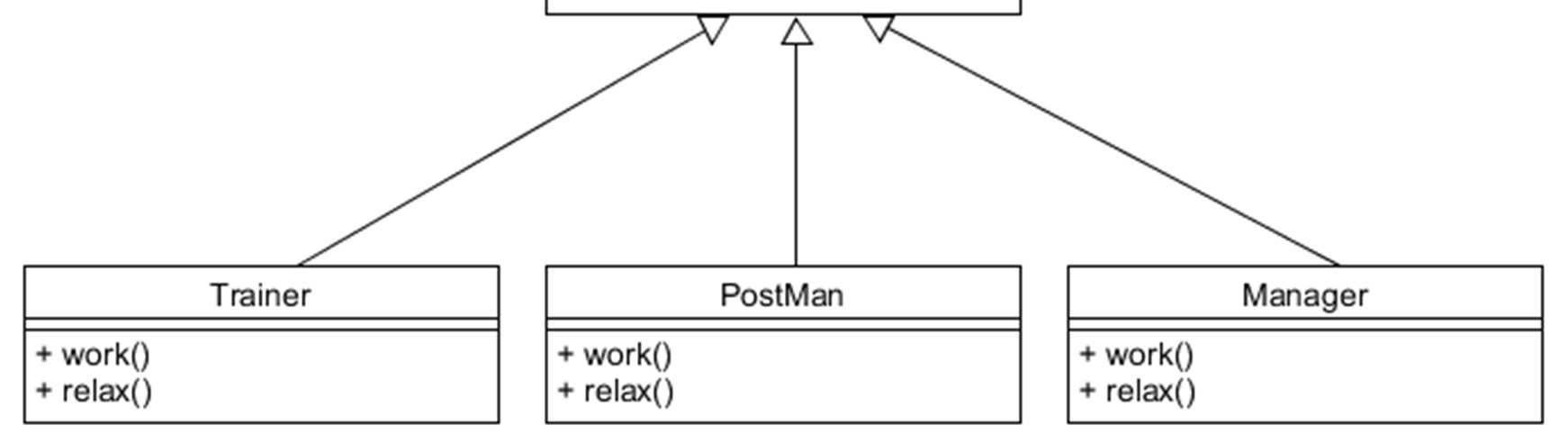
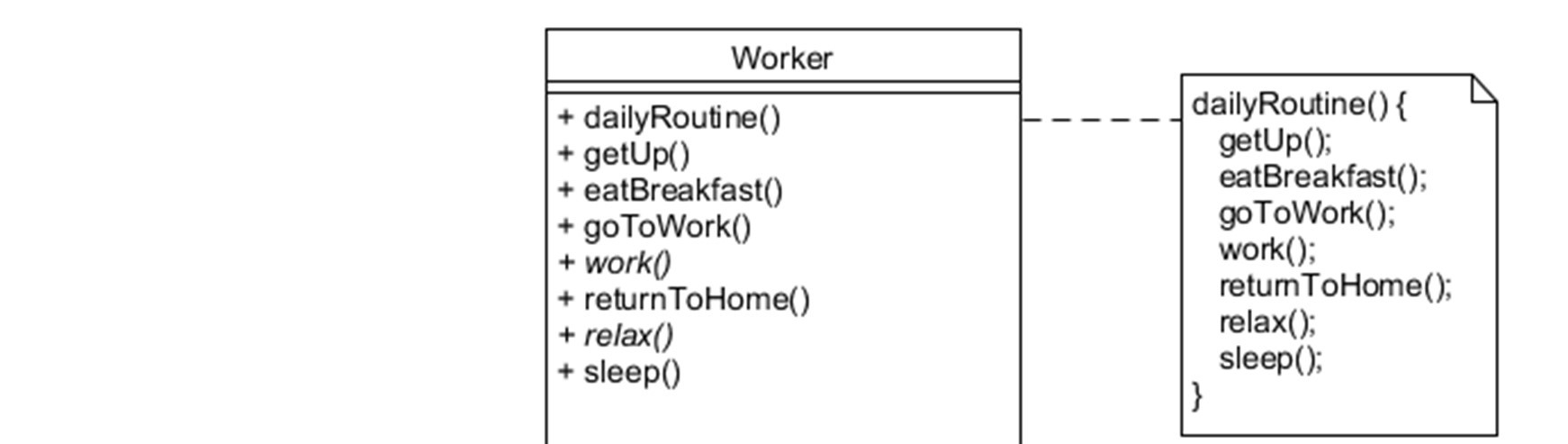
@Override public int compare(Trainer trainer1, Trainer trainer2) { return trainer1.getName().compareTo(trainer2.getName()); }

});

Rendezés lambda kifejezéssel trainers.sort((trainer1, trainer2) -> trainer1.getName().compareTo(trainer2.getName()));

Template method tervezési minta

Amikor egy algoritmus sokféle objektumra nagyon hasonló, akkor elég lenne az algoritmus nagy részét megírni, a különbözőségeket kívülről kellene beadni. Ez objektumorientált nyelvek esetében azt jelenti, hogy egy ősosztály deklarálja az absztrakt metódust és az azt használó közös algoritmust, míg a leszármazottak mind implementálják az absztrakt metódust. A leszármazott lehet egy konkrét osztály vagy egy anonymous belső osztály.



template\_method

A hiányzó rész tulajdonképpen csak egy metódus. Nem kellene egy egész osztályt létrehozni, elég lenne ezt megadni valahogy kívülről, például paraméterként. Ezt két módon tehetjük meg: vagy egy létező implementált metódust adunk át (metódus referencia), vagy egy lambda kifejezést, mely a hiányzó működést írja le.

Lambda kifejezés szintaktikája

A lambda kifejezés két fő részből áll, melyeket -> (nyíl operátor) választ el egymástól. Az első rész a bemenő paraméterek listája, a második rész a törzs, amely az utasításokat tartalmazza.

(trainer1, trainer2) -> { return trainer1.getName().compareTo(trainer2.getName());}

A paraméterek típusa megadható, de nem kötelező. Pontosan egy paraméter esetén, ha a típust sem adjuk meg, a zárójel elhagyható.

(Trainer trainer) -> {return trainer.getName();} trainer -> {return trainer.getName();}

A törzs {} közötti utasítások sorozata. Ha pontosan egy utasítást tartalmaz (amely legtöbbször egy return), a kapcsos zárójel, a return kulcsszó és a pontosvessző is elhagyható.

trainer -> trainer.getName()

Funkcionális interfész

Láthattuk az első példában, hogy mennyivel rövidebb és áttekinthetőbb lesz a kód, ha lambda kifejezést használunk. De mikor használhatunk lambda kifejezést paraméterként?

A funkcionális interfész olyan interfész, amely pontosan egy absztrakt metódust tartalmaz. Az interfész megjelölhető a @FunctionalInterface annotációval. Az így megjelölt interfész nem fordul le, ha nem egy absztrakt metódust tartalmaz.

A Comparator funkcionális interfész, mivel csupán a compare absztrakt metódust tartalmazza. A metódus két objektumot vár és int típussal tér vissza, ezért bármi olyan lambda kifejezés átadható Comparator helyett, amely két bemenő paramétert vár és int típussal tér vissza. Minden lambda kifejezés mögött egy funkcionális interfész áll, melynek absztrakt metódusát implementálja, ezért funkcionális interfész mindig kaphat lambda kifejezést értékül.

Comparator<Trainer> comp = (trainer1, trainer2) -> trainer1.getName().compareTo(trainer2.getName());

Metódus referencia

Előfordul, hogy a lambda kifejezés nem új műveletet ír le, arra már létezik metódus. Ilyenkor átadhatjuk a metódust is lambda kifejezés helyett.

Szövegösszefűzés lambdával: (str1, str2) -> str1.concat(str2)

Ugyanez metódusreferenciával: String::concat

Az átadott metódus lehet statikus, konkrét objektumé vagy paraméterként átadott objektumé, illetve lehet konstruktor referencia is.

Az előző példában a paraméterként átadott str1-en hívja meg a concat függvényt str2 paraméterrel. A metódusreferencia nem statikus, de azt sem mondtuk meg, konkrétan melyik String-et kell használnia, azért az a paraméterként átadott objektumon fog meghívódni.

Ha mindig egy konkrét objektumot szeretnénk használni, akkor a metódus neve előtt nem az osztályt, hanem az objektumot kell feltüntetni. Például ha létezik egy String str = "alma" szövegünk, akkor a str::concat metódusreferenciánk mindig ehhez fűzi hozzá a paraméterként átadottat.

Konstruktor referenciát akkor használunk, ha új objektumot szeretnénk létrehozni a metódusból. Formátuma: Osztály::new

Saját és beépített funkcionális interfészek

A Java nyelv számos funkcionális interfészt tartalmaz, de sajátot is készíthetünk. Amennyiben megfelel annak a szabálynak, hogy pontosan egy absztrakt metódust tartalmaz, akkor funkcionális interfész.

public interface Condition<T> {

boolean apply(T t); }

Az interfészt használhatjuk például arra, hogy egy lista elemei közül megtaláljuk az első feltételnek megfelelőt. Figyeld meg a template method tervezési mintát! A kereső algoritmus állandó, de mögötte a feltételt vizsgáló metódus implementációja cserélhető. Amiben biztosak lehetünk a keresés írásakor, hogy a condition paraméternek van apply nevű metódusa, ami most éppen egy Trainer objektumot kap és boolean-t ad vissza.

public Trainer findFirst(List<Trainer> trainers, Condition<Trainer> condition) {

for (Trainer trainer: trainers) { if (condition.apply(trainer)) { return trainer;

} }

throw new IllegalArgumentException("Cannot find trainer applied to the condition: " + condition); }

A keresési feltételt elég a findFirst metódus hívásakor definiálnunk. Például, ha a “John Doe” nevű Trainer-t keressük: findFirst(trainers, trainer -> trainer.getName().equals("John Doe"));

Beépített funkcionális interfészek

Attól függően, hogy hány és milyen típusú paraméter megy be, és milyen típussal tér vissza a metódus, többféle funkcionális interfészt készítettek el a Java fejlesztői, melyeket a java.util.function csomagban találunk meg. Ahhoz, hogy ezeket használni tudjuk, ismernünk kell a bennük található metódus nevét is.

Paraméterek száma és Visszatérési Metódus

Interfész típusa típus neve

|  |  |  |  |
| --- | --- | --- | --- |
| Supplier<T> | 0 | T | get |
| Consumer<T> | 1 (T) | void | accept |
| BiConsumer<T, U> | 2 (T, U) | void | accept |
| Predicate<T> | 1 (T) | boolean | test |
| BiPredicate<T, U> | 2 (T, U) | boolean | test |
| Function<T, R> | 1 (T) | R | apply |
| BiFunction<T, U,  R> | 2 (T, U) | R | apply |
| UnaryOperator<T> | 1 (T) | T | apply |

BinaryOperator<T> 2 (T, T) T apply

Láthatjuk, hogy létezik a fenti funkciónak megfelelő beépített interfész a nyelvben, és ez a Predicate. A generikus típusnak megfelelő paramétert kap, és logikait ad vissza. Ezt használva a fenti keresés:

public Trainer findFirst(List<Trainer> trainers, Predicate<Trainer> condition) {

for (Trainer trainer: trainers) { if (condition.test(trainer)) { return trainer;

} }

throw new IllegalArgumentException("Cannot find trainer applied to the condition: " + condition);

}

Csupán az interfész és a benne található metódus nevét kellett módosítanunk, a findFirst metódus hívása ugyanaz marad.

Ezek az interfészek csak objektumokkal tudnak dolgozni. Ha az a feladat, hogy egy listában található egész számokat összegezzünk, akkor erre a Function<List<Integer>, Integer> funkcionális interfészt használhatjuk. Primitív típusú adatokat sem befogadni, sem visszaadni nem tudnak, arra külön interfészeket hoztak létre. Nézz utána ezeknek!

Ellenőrző kérdések

* Mik a jellemzői a funkcionális programozásnak? Mi az alapegysége?
* Melyik Java verzióban jelentek meg funkcionális elemek?
* Java nyelvben milyen eszköz van a funkcionális programozásra?
* Hogyan épül fel egy lambda kifejezés?
* Mi áll minden lambda kifejezés mögött?
* Hogyan tudsz saját funkcionális interfészt készíteni? Milyen szabályoknak kell megfelelnie?
* Milyen már beépített interfészeket ismersz?

Bankszámlák lekérdezése

Készíts egy BankAccount osztályt accountNumber, nameOfOwner és balance attribútumokkal. Készíts egy BankAccounts osztályt, mely konstruktorban BankAccount listát vár.

A metódusok rendre új listát készítenek, rendezik a következő szabályok alapján, és adják vissza.

* Rendezés bankszámlaszám alapján
* Rendezés rendelkezésre álló összeg alapján, de előjeltől függetlenül
* Rendezés rendelkezésre álló összeg alapján, az előjel számítson, de csökkenő sorrendben
* Név alapján, de ha a név megegyezik, akkor bankszámlaszám alapján. Amennyiben nincs kitöltve a név (értéke null, elől szerepeljen)

Implementációs javaslat

Kizárólag lambda kifejezéseket vagy method reference-eket használj.

Használd a következő metódusokat (persze a megfelelő paraméterezéssel): Comparator.naturalOrder(), Comparator.comparing(), Comparator.reversed(), Comparator.nullsFirst(), Comparator.thenComparing().

Fájlnevek

Írj egy OfficeDocumentReader osztályt, abba egy List<File>

listOfficeDocuments(File path) metódust, mely visszaadja a paraméterként átadott könyvtárban található összes docx, pptx és xlsx kiterjesztésű fájlt, név szerint sorbarendezve.

Implementációs javaslat

Kizárólag lambda kifejezéseket vagy method reference-eket használj.

[rating feedback=java-lambdaintro-fajlnevek]

Szülinapok

Írj egy FamilyBirthdays osztályt, mely konstruktor paraméterül kap születésnapokat. Implementáld benne az isFamilyBirthday és nextFamilyBirthDay metódusokat, a tesztben szereplő method reference-ek alapján.

A isFamilyBirthday visszaadja, hogy a paraméterként átadott dátum születésnap-e. A nextFamilyBirthDay metódus visszaadja, hány nap van a legközelebbi születésnapig.

Implementációs javaslat

Nézd meg LocalDate query() metódusát, hogy mit kap paraméterül. Használd a ChronoUnit osztályt annak meghatározására, hogy két dátum között hány nap telt el.

Közösségi háló

Hozz létre egy közösségi hálózatot, melyben a fejvadászok mindenféle műveletet tudnak végezni a tagokkal.

Hozz létre egy Member osztályt, name, skills (mely egy List<String>), gender és messages (List<String>) attribútumokkal. Hozz létre benne egy sendMessage(String) metódust, mely a paraméterként kapott üzenetet beteszi a messages listába.

Hozz létre egy SocialNetwork osztályt, mely Member objektumokat képes tárolni. A findMembersBy() metódusa a paraméterként átadott feltétel alapján kigyűjti a tagokat. A applyToSelectedMembers() a paraméterként átadott feltételnek megfelelő tagokon végez valamilyen műveletet (2. paraméterként átadva). A transformMembers() metódus minden tagon valamilyen transzformációt végez.

Implementációs javaslat

Egy feltétel átadásához használj Predicate interfészt. Egy művelet átadásához használj Consumer interfészt. Konvertáláshoz használj Function interfészt.

A transformMembers() metódus elég erősen használ generikust, nézd meg, hogy kell a metódusban definiálni, és használni is generikust. ### Optional osztály (lambdaoptional)

Megismerkedtünk pár olyan algoritmussal, ahol az, hogy el tudjuk-e végezni a számítást, erősen függött a bemenettől. Például a minimum- és maximumkiválasztás előfeltétele, hogy a bemenő listának vagy tömbnek legyen legalább egy eleme, és az elemek összehasonlíthatók legyenek. Mit tegyünk, ha nincs egy elem sem vagy nem összehasonlíthatók? Mit adjon vissza a keresés, ha nincs meg a keresett elem?

Amikor valami kivételes történik, dobhatunk kivételt. Ilyen az, ha úgy akarok szélsőértéket keresni, hogy az elemek nem összehasonlíthatók. Az azonban nem kivételes eset, ha a listánk üres, vagy nincs meg a keresett elem. Ekkor jelezhetjük valamilyen speciális visszatérési értékkel ezt. Például a hiányzó elem esetén null-t adunk vissza, vagy index esetén -1-t. A probléma csak az, hogy az első

NullPointerException-höz, a második IndexOutOfBoundsException-höz vezethet, ha nem kezeljük a speciális visszatérési értéket a hívó metódusban.

Van, amikor nem is tudjuk meghatározni, mi legyen az a speciális érték. Például nulla darab szám összege 0, de mennyi az átlaguk?

public static double average(int... scores) { if (scores.length == 0) { //??? } int sum = 0; for (int score: scores) { sum += score;

} return (double) sum / scores.length; }

Amikor lehetséges, hogy valamilyen speciális bemenő adatra nem tudunk értelmes eredményt adni, akkor használjuk az Optional<T> generikus burkoló osztályt. Az osztály definiálja a “nincs eredmény” állapotot, így az vizsgálható a hívó kódban. Az értéket az Optional osztály factory metódusaival tudjuk előállítani:

Optional.empty(): üres Optional-t ad vissza, amely a “nincs eredmény” állapotnak felel meg.

Optional.of(T t): az adott objektumot Optional osztályba csomagolja. A paraméter nem lehet null.

Optional.ofNullable(T t): az adott objektumot Optional osztályba csomagolja. A paraméter lehet null, ekkor az üres Optional-t állítja elő.

public static Optional<Double> average(int... scores) { if (scores.length == 0) { return Optional.empty();

} int sum = 0; for (int score: scores) { sum += score;

}

return Optional.of((double) sum / scores.length);

}

Azt, hogy tartalmaz-e értéket az isPresent() metódussal tudjuk megvizsgálni. Ha ez hamis, akkor nincs értelmes eredmény, ha igaz, akkor az eredményt a get() metódussal kapjuk meg. Ellenőrzés nélküli get() hívás NoSuchElementException-höz vezet, ha az Optional üres.

Optional<Double> opt = average(90, 100);

if (opt.isPresent()) { Double d = opt.get(); }

További metódusok

void ifPresent(Consumer<? super T> consumer): ha az Optional tartalmaz értéket, meghívja rajta a paraméterként átadott metódust.

T orElse(T other): ha az Optional tartalmaz értéket, akkor azt adja vissza, ha nem, akkor a paraméterül kapottat.

T orElseGet(Supplier<? extends T> other): ha tartalmaz értéket, akkor azt adja vissza, különben meghívja a paraméterül kapott other-t és az onnan kapott értéket adja vissza.

<X extends Throwable> T orElseThrow(Supplier<? extends X> exceptionSupplier): ha tartalmaz értéket, akkor azt adja vissza, ha nem, akkor a paraméterül kapott exceptionSupplier által előállított kivételt dobja.

Ellenőrző kérdések

* Mire való az Optional osztály?
* Milyen metódusait ismered?

Gyakorlat - Közösségi háló

Hozz létre egy közösségi hálózatot, melyen kereséseket lehet végezni.

Hozz létre egy Member osztályt, name, skills (mely egy List<String>), gender attribútumokkal.

Hozz létre egy SocialNetwork osztályt, mely Member objektumokat képes tárolni.

A findFirst metódusa paraméterként egy keresési feltételt kap. Visszatérési típusa Optional. Ha talál a keresési feltételnek megfelelő tagot, akkor az elsőt adja vissza, ha nem talál, üres értékkel tér vissza.

A averageNumberOfSkills üres értékkel tér vissza, ha a közösségi háló nem tartalmaz tagot. Ellenkező esetben átlagolja a tagok szakértelmének számát, és azzal tér vissza.

Comparator módosítások (lambdacomparator)

A Java 8 verzióval bejött statikus interfész metódusok lehetővé tették, hogy a

Comparator interfészbe olyan metódusok kerüljenek be, melyek elkészítik a Comparator implementációt bármilyen osztályra, amennyiben az összehasonlítás logikáját megadjuk neki.

Comparator.comparing()

A comparing() metódus többféle bemenetből el tudja készíteni a szükséges Comparatort. Elég egy olyan Function-t adnunk neki, amely megmondja, hogy az osztály két példányát milyen kulcs alapján kell összehasonlítani, amennyiben a kulcs maga már implementálja a Comparable interfészt. Például ha egy List<Person> people listát az emberek neve szerint szeretnénk rendezni, akkor az ehhez szükséges Comparator-t az alábbi hívás állítja elő:

Comparator.comparing(Person::getName)

Ennek persze feltétele, hogy a Person getName() metódusa által visszadott String a Javában összehasonlítható. Amennyiben az adott kulcs nem implementálja a Comparable interfészt, vagy nem az implementált logika alapján szeretnénk összehasonlítani, szükséges az ezeket összehasonlító logika, azaz egy újabb Comparator átadása második paraméterként.

Comparator.comparing(Person::getName, (s, t) ->

s.trim().toLowerCase().compareTo(t.trim().toLowerCase()));

Amennyiben nem egy szempont alapján akarunk rendezni, akkor eddig egy elég bonyolult Comparator implementációt kellett megfogalmazni, amely az elsődleges szempont egyezősége esetén külön vizsgálta a másodlagos szempontot, annak egyezősége esetén a harmadlagosat stb. Most elég az elsődleges szempont megadása a comparing() metódusban, majd láncoltan hívható sorban a thenComparing() metódus a többi szemponttal.

Például ha a Person objektumokat elsődlegesen vezetéknév szerint, azok egyezősége esetén pedig a keresztnév szerint szeretnénk rendezni:

Comparator.comparing(Person::getLastName).thenComparing(Person::getFirstNam e); int, long és double típusú kulcs

A comparing() metódus objektumokat összehasonlító Comparator-t ad, nem használható primitív értékek összehasonlítására. Amennyiben az embereket valamilyen primitív kulcs szerint szeretnénk rendezni, akkor más metódusokat kell használnunk. Ha az összehasonlítás int típusú kulccsal történik, akkor a comparingInt() metódust használjuk, de létezik még comparingLong() és comparingDouble() is.

Az embereket a nevü hossza szerint rendező Comparator előállítása:

Comparator.comparingInt(p -> p.getName().length());

Itt a lambda kifejezéssel a Person objektumból a nevük hosszát vontuk ki kulcsnak. null, natural order és fordított rendezés

Az eddigiekben nem foglalkoztunk azzal az esettel, ha a kulcsot kivonó függvény null értéket ad vissza. Ebben az esetben a rendezés csődöt mondana, mert nem tudja a null-t összehasonlítani a nem null értékellel. Ebben az esetben megadhatjuk, hogy a null mindennél kisebbnek vagy nagyobbnak számít-e a Comparator.nullsFirst(), illetve a Comparator.nullsLast() metódusokkal.

Amennyiben a nem kötelező középső név szerint szeretnénk rendezni, és a null értéket minden elé szeretnénk tenni, az alábbi Comparator-t kell elkészítenünk:

Comparator.comparing(Person::getMiddleName,

Comparator.nullsFirst(Comparator.naturalOrder()));

Láthatjuk, hogy a Comparator.nullFirst() egy újabb Comparator-t vár. Amennyiben a természetes rendezettséget szeretnénk megtartani, akkor ehhez használhatjuk a Comparator.naturalOrder() metódust. Ezt mindenhol használhatjuk, ahol egy metódus Comparator-t vár, de mi nem akarunk változtatni a típusban definiált rendezettségen.

Ha a természetes rendezettséghez képest pont fordítottan szeretnénk az elemek sorrendjét, akkor a Comparator.reverseOrder() metódusa által legyártott Comparator-t kell használnunk, míg ha egy bármilyen más, Comparator-ral definiált sorrendet szeretnénk megfordítani, akkor a reversed() metódusra van szükségünk.

people.sort(Comparator.comparing(Person::getMiddleName,

Comparator.reverseOrder()));

people.sort(Comparator.comparing(Person::getName, Comparator.comparingInt(String::length)).reversed());

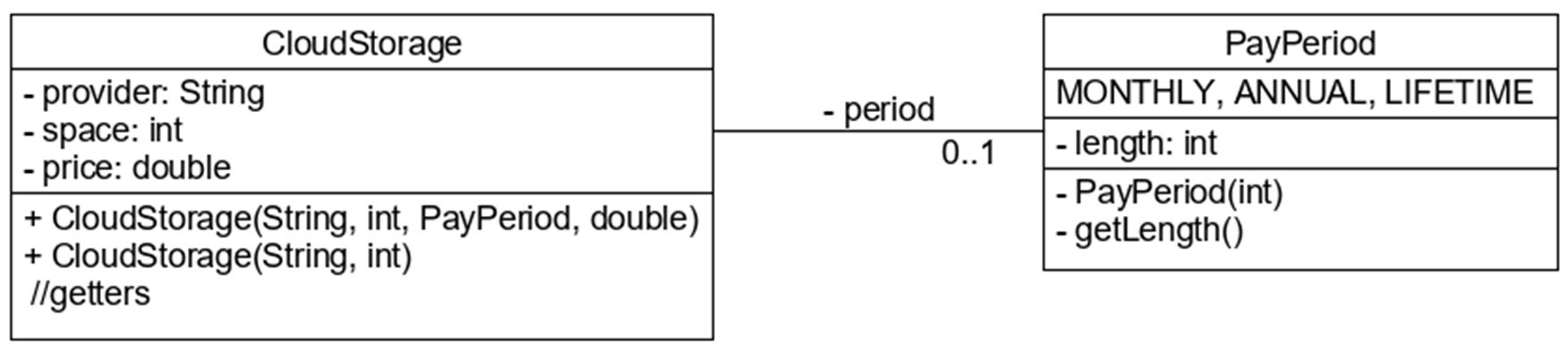
Ellenőrző kérdések

* Milyen új statikus metódusokkal bővült a Comparator interfész? Mikor hasznosak?
* Hogyan lehet egy listát rendezni, ha az elemek nem implementálják a Comparable interfészt, és több szempont szerint is rendezni akarunk?
* Hogyan lehet egy Comparator objektumban definiált rendezettséget megfordítani?

Feladat

Felhő tárhelyek

Különböző felhő tárhely szolgáltatókat szeretnénk összehasonlítani, ezért a CloudStorage osztályban eltároljuk a különböző adataikat. A tárhely mérete GB-ban adott, az árak pedig mindenhol ugyanabban a pénznemben. A PayPeriod enum a fizetési gyakoriság, ahol a length értéke a hossz hónapokban megadva (lifetime esetén 60 hónap). Az ingyenes csomagok esetén a fizetési gyakortiság nincs megadva. A CloudStorage implementálja a Comparable interfészt, a természetes rendezettségét az 1000 GB-ra eső éves díj nagysága adja.



CloudStorage UML

A Clouds osztály metódusai a paraméterként kapott listából bizonyos szempont szerint a legjobba(ka)t adják vissza. Amennyiben több ugyanolyan van, akkor közülük bármelyik visszaadható.

* alphabeticallyFirst(): a szolgáltató neve alapján betűrendben a legelső CloudStorage. Kis-nagybetű nem számít.
* bestPriceForShortestPeriod(): a legrövidebb időszakra vonatkozó legolcsóbb CloudStorage. Ha van ingyenes, akkor azok közül bármelyik megadható.
* worstOffers(): a természetes rendezettség szerinti 3 legrosszabb ajánlat.

## Streamek

Streamek (lambdastreams)

A streamek a Java nyelvben adatfolyamok. Úgy kell elképzelni, mint egy futószalagot, ahol sorban jönnek az adatok, és minden művelet csinál velük valamit. A közbenső műveletek (intermediate operations) eredménye szintén stream, így folytathatjuk a feldolgozást, míg a záró művelet (terminal operation) kimenete valamilyen kollekció vagy más objektum. A feldolgozás ún. lusta kiértékelésű (lazy evaluation), azaz a műveletek csak akkor hajtódnak végre, amikor szükség van rájuk, és csak azokon az adatokon, amelyeken feltétlenül szükséges. Az egész feldolgozást a záró művelet indítja el. E nélkül nem történik semmi, az adatok a futószalag előtt várakoznak feldolgozásra.



stream\_pipeline

A stream lehet véges vagy végtelen, attól függően, hogy hány adat van a forrásban. Ez persze nem jelenti azt, hogy végtelen adatot kell feldolgozni. Előfordulhat, hogy a kíván eredmény eléréséhez csak az első öt kerül a futószalagra.

Stream létrehozása

A java.util.stream csomagban található Stream interfész metódusaival tudunk streameket létrehozni.

Stream.empty(): üres stream létrehozása.

Stream.of(T... values): a felsorolt elemekből készít streamet.

Stream<Integer> numbers = Stream.of(1, 2, 3);

Stream.generate(Supplier<T> s): végtelen hosszú streamet generál a megadott Supplier segítségével.

Stream<Double> randoms = Stream.generate(Math::random);

Stream.iterate(T seed, UnaryOperator<T> f ): végtelen streamet készít a megadott seed értéktől kezdve. A következő elemet mindig az előző elem és a paraméterként átadott függvény segítségével állítja elő.

Stream<Integer> oddNumbers = Stream.iterate(1, n -> n + 2);

Már létező kollekcióból is tudunk streamet gyártani a Collection.stream() illetve a Collection.paralellStream() metódusaival. A párhuzamos feldolgozású stream esetén a forrásból az elemek több ugyanolyan műveleteket tartalmazó futószalagra kerülnek, és csak bizonyos műveleteknél válnak ismét eggyé. Mivel a futószalagok sebessége eltérhet, nem tudhatjuk, hogy a záró művelethez milyen sorrendben érkeznek az elemek. Ez leszámítva a feldolgozás nagy mennyiségű adat esetén sokkal gyorsabb is lehet, mint soros stream esetén, ezért ha nem számít a sorrend, érdemes használni.

List<String> list = Arrays.asList("a", "b", "c");

Stream<String> fromList = list.stream();

Stream<String> fromListParallel = list.parallelStream();

Záró műveletek

A feldolgozást a záró művelet indítja, ezért záró műveletnek mindig lennie kell. A záró művelet után a stream megszűnik létezni, új művelet nem végezhető rajta. A záró műveletek között vannak redukciók, amelyek az egész streamből egyetlen objektumot, értéket gyártanak.

long count(): a stream elemszámát adja meg.

Stream<String> s = Stream.of("monkey", "gorilla", "bonobo"); System.out.println(s.count()); // 3

Optional<T> min(Comparator<? super T> comparator): az elemek közül a paraméterként átadott comparator szerinti legkisebbet adja vissza

Optional<T> max(Comparator<? super T> comparator): az elemek közül a paraméterként átadott comparator szerinti legnagyobbat adja vissza.

Stream<String> s = Stream.of("monkey", "ape", "bonobo");

Optional<String> min = s.min((s1, s2) -> s1.length() — s2.length()); min.ifPresent(System.out::println); // ape

Optional<T> findAny(): a stream egyik elemével tér vissza, ha van ilyen. Párhuzamos streamek esetén gyakran használjuk, ha mindegy, melyik elem kerül először feldolgozásra.

Optional<T> findFirst(): a stream első elemét adja vissza, ha van ilyen.

Stream<String> s = Stream.of("monkey", "gorilla", "bonobo"); Stream<String> infinite = Stream.generate(() -> "chimp"); s.findAny().ifPresent(System.out::println); // monkey infinite.findAny().ifPresent(System.out::println); // chimp

boolean allMatch(Predicate<? super T> predicate): igaz, ha a stream minden eleme megfelel a paraméterként átadott predicate-nek.

boolean anyMatch(Predicate<? super T> predicate): igaz, ha a stream valamelyik eleme megfelel a paraméterként átadott predicate-nek.

boolean noneMatch(Predicate<? super T> predicate): igaz, ha a stream egyik eleme sem felel meg a paraméterként átadott predicate-nek.

List<String> list = Arrays.asList("monkey", "2", "chimp");

Stream<String> infinite = Stream.generate(() -> "chimp");

Predicate<String> pred = x -> Character.isLetter(x.charAt(0));

System.out.println(list.stream().anyMatch(pred)); // true

System.out.println(list.stream().allMatch(pred)); // false

System.out.println(list.stream().noneMatch(pred)); // false System.out.println(infinite.anyMatch(pred)); // true

void forEach(Consumer<? super T> action): a stream minden elemén elvégzi a paraméterben megadott műveletet.

Stream<String> s = Stream.of("Monkey", "Gorilla", "Bonobo"); s.forEach(System.out::print); // MonkeyGorillaBonobo

A reduce() művelet a stream elemeit egyetlen értékké gyúrja össze a paraméterben megadott accumulator művelet segítségével. Mindig új objektumot gyárt az előző részeredmény és az accumulator művelet segítségével, míg el nem jut a végeredményig. Három variációja van:

Optional<T> reduce(BinaryOperator<T> accumulator): üres stream esetén Optional.empty-vel tér vissza.

T reduce(T identity, BinaryOperator<T> accumulator): az összesítést a identity értékkel kezdi, ezért üres stream esetén ezt adja vissza.

<U> U reduce(U identity, BiFunction<U,? super T,U> accumulator, BinaryOperator<U> combiner): főleg párhuzamos streameknél használjuk. A párhuzamosan feldolgozott és az accumulator művelettel egyesített elemeket végül a combiner művelettel egyetlen végeredménnyé gyúrja.

BinaryOperator<Integer> op = (a, b) -> a \* b;

Stream<Integer> stream = Stream.of(3, 5, 6);

System.out.println(stream.reduce(1, op, op)); // 90

A collect() a stream kimenetén megjelenő elemeket gyűjti össze és/vagy rendszerezi valamilyen kollekcióba.

<R,A> R collect(Collector<? super T, R, A> collector): az elemeket a megadott collector gyűjti össze.

<R> R collect(Supplier <R> supplier, BiConsumer<R, ? super T> accumulator, BiConsumer<R, R> combiner): a supplier által szolgáltatott kollekcióba gyűjti az elemeket az accumulator függvény segítségével. Párhuzamos streamek esetén az esetlegesen létrejövő több kollekció eggyé kovácsolásában játszik szerepet a combiner.

Például ha az eredményt listába szeretnénk összegyűjteni, akkor azt három féle képpen tehetjük meg:

stream.collect(ArrayList::new, List::add, List::addAll); stream.collect(Collectors.toCollection(ArrayList::new)); stream.collect(Collectors.toList());

A Collectors osztály többféle kollekciót is tud szolgáltatni, akár szortírozni is tudja az elemeket valamilyen szempont szerint.

Ellenőrző kérdések

* Mi a stream?
* Hogyan lehet streamet létrehozni?
* Hogyan dolgozzuk fel a streameket?
* Mit jelent az, hogy a stream kiértékelése lusta (lazy evaluation)?
* Sorolj fel öt záró műveletet! Mit csinálnak?

Gyakorlat - Alapműveletek

Készíts egy Numbers osztályt, amely egész számokból álló listát tárol. A listát a konstruktoron át kapja meg. Készíts metódusokat az alábbi számítások elvégzésére:

* min(): a legkisebb szám,
* sum(): az elemek összege,
* isAllPositive(): megvizsgálja, hogy minden elem pozitív-e,
* getDistinctElements(): az összes különböző elemet pontosan egyszer tartalmazó kollekció.

Az összes metódus kizárólag stream műveleteket használjon.

gyakorlat - Könyvesbolt

Készíts egy Book osztályt a könyv címével, kiadási évével, árával, darabszámával! A konstruktor is ebben a sorrendben kapja meg az adatokat.

A BookStore osztály tartalmazza a könyvek listáját. A listát a konstruktorban kapja meg. Készítsd el benne az alábbi metódusokat streamek segítségével:

getNumberOfBooks(): a könyvek számát adja meg findNewestBook(): a legújabb kiadású könyvet adja vissza getTotalValue(): a könyvek összértékét adja meg

Bónusz feladat

Nézz utána a Collectors osztály groupingBy() metódusának, és készítsd el az alábbi metódust is!

getByYearOfPublish(int year): adott évben kiadott könyvek listáját adja vissza

Részletek (lambdaintermediate)

A közbenső műveletek (intermediate operations) kimenete mindig stream, bár a benne lévő elemek típusa, száma, sorrendje változhat. Intermediate műveleteket láncolva hívhatunk, de vigyázzunk az olvashatóságra.

filter()

Stream<T> filter(Predicate<? super T> predicate)

Csak a predicate-nek megfelelő elemeket engedi át, a többit kiszűri. A bemenő és a kijövő adatok típusa megegyezik.

Stream<String> s = Stream.of("monkey", "gorilla", "bonobo");

s.filter(x -> x.startsWith("m")).forEach(System.out::print); //monkey distinct()

Stream<T> distinct()

Csak az equals metódus szerint különböző elemeket engedi át, azaz kiszűri a duplikációkat. A bemenő és a kijövő adatok típusa megegyezik.

Stream<String> s = Stream.of("duck", "duck", "duck", "goose"); s.distinct().forEach(System.out::print); // duckgoose limit() és skip()

Stream<T> limit(long maxSize)

Csak a paraméterként átadott darabszámú elemet engedi át.

Stream<T> skip(long n)

A megadott számú elemet kihagyja, és csak a többit engedi át.

Stream<Integer> s = Stream.iterate(1, n -> n + 1);

s.skip(5).limit(2).forEach(System.out::print); // 67 map()

<R> Stream<R> map(Function<? super T, ? super R> mapper)

Az elemeket a függvény segítségével egyenként átkonvertálja más típusú elemre. Primitív típusú streammé konvertáláshoz nem ezt a metódust használjuk. A stream feldolgozásának további részében az eredeti adat már nem elérhető.

Stream<String> s = Stream.of("monkey", "gorilla", "bonobo");

s.map(String::length).forEach(System.out::print); // 676

flatMap()

<R> Stream<R> flatMap(Function<? super T, ? extends Stream<? extends R>> mapper)

A bemeneti stream tartalmazhat kollekciókat és streameket is. Ezek elemeit egyetlen streammé tudjuk konvertálni. Primitív streammé konvertáláshoz nem ezt használjuk.

List<String> zero = Arrays.asList();

List<String> one = Arrays.asList("Bonobo");

List<String> two = Arrays.asList("Mama Gorilla", "Baby Gorilla"); Stream<List<String>> animals = Stream.of(zero, one, two); animals.flatMap(l -> l.stream()).forEach(System.out::println);

flatmap nélkül a kimeneten a listák jelennének meg, így minden sorba egy-egy String kerül. sorted()

Stream<T> sorted()

Stream<T> sorted(Comparator<? super T> comparator)

A bemenő elemeket természetes sorrendjükben vagy a comparator által definiált sorrendben engedi át. Be kell várnia az összes elemet, ezért végtelen streamekre sosem áll le.

Stream<String> s = Stream.of("brown-", "bear-");

s.sorted().forEach(System.out::print); // bear-brown

Stream<String> s = Stream.of("brown bear-", "grizzly-"); s.sorted(Comparator.reverseOrder())

.forEach(System.out::print); // grizzly-brown bear- peek()

Stream<T> peek(Consumer<? super T> action)

Menet közben belepillanthatunk a streambe. Nem csak az első elemén, hanem mindegyiken végrehajtódik az adott action. Fontos, hogy az action ne változtasson a stream állapotán. Főként debug célokra használható.

Stream<String> stream = Stream.of("black bear", "brown bear", "grizzly"); long count = stream.filter(s -> s.startsWith("g"))

.peek(System.out::println) // grizzly

.count();

System.out.println(count); // 1

Ellenőrző kérdések

* Mi jellemzi az intermediate műveleteket?
* Sorolj fel öt közbenső műveletet! Mit csinálnak?

Gyakorlat - Kávézó

Készíts egy Coffee osztályt! Attribútumai: type a kávé típusa, price a kávé ára. A konstruktor is ebben a sorrendben kapja meg az adatokat. Az ár lehet tört, 2 tizedesjegy pontossággal számolj!

A kávé típusához készíts egy CoffeeType enum-ot. Lehetséges értékei: ESPRESSO, MACHIATTO, RISTRETTO, MOCHA, LATTE, CAPPUCCINO, AMERICANO.

A CoffeeOrder osztály tárolja egy vásárló által megrendelt és leszámlázott kávékat.

Attibútumai: coffeeList a megrendelt kávék listája, dateTime a vásárlás időpontja.

A Cafe osztály tartalmazza a kávézó összes rendelését egy listában. A listát a konstruktorban kapja meg, de legyen lehetőség új rendelést hozzáadni. Készítsd el benne az alábbi metódusokat streamek segítségével:

getTotalIncome(): az eddigi összes bevétel getTotalIncome(LocalDate date): adott napi teljes bevétel

getNumberOfCoffee(CoffeeType type): az adott típusú kávéból eladott összmennyiség getOrdersAfter(LocalDateTime from): a megadott időpont utáni rendelések listája getFirstFiveOrder(LocalDate date): adott napon az első 5 vásárlásban lévő kávék listája

Primitívek használata streamekben (lambdaprimitives)

A Stream interfészt objektumok feldolgozására hozták létre. Nagyon gyakran találkozunk azzal, hogy primitív típusú adatokat szeretnénk streamben feldolgozni. Mivel minden primitívhez létezik csomagoló osztály, ez nem probléma. Ha az elemek int típusúak, akkor abból Stream<Integer> lesz. Vannak azonban olyan számítások, amiket nagyon gyakran végzünk primitív típusú adatokkal, és némelyiket nem könnyű objektum streamen végrehajtani. Számok összegzésére már láttunk példát:

Stream<Integer> numbers = Stream.of(1, 2, 3);

int sum = numbers.reduce(0, (s, n) -> s + n); //autounboxing a visszaadott Integerre

Ez primitív streammel:

IntStream numbers = IntStream.of(1, 2, 3); int sum = numbers.sum();

Háromféle primitív stream létezik:

* IntStream
* LongStream
* DoubleStream

Mindháromban megtalálhatók a Stream eddig megismert műveletei, bár kisebb-nagyobb eltérések előfordulnak. A műveletek paraméterei nagyon gyakran funcionális interfészek, amelyek szintén objektumokkal működnek. Éppen ezért a funkcionális interfészeknek is megvan a maga primitívekkel dolgozó megfelelője. Például egy objektumot szolgáltató Supplier<T> int, short, byte és char primitívvel dolgozó megfelelője az IntSupplier, long primitívre a LongSupplier, double-ra és float-ra a DoubleSupplier. Ahol a be- és kimenet más-más típusú, ott lehet az egyik primitív, a másik objektum, vagy mindkettő primitív, akár ugyanolyan, akár különböző típusú. Elsőre kicsit bonyolultnak hangzik, de lambda kifejezés használata esetén ez teljesen automatikus.

Primitív stream létrehozása ugyanúgy történik, mint az objektum streamek esetén, kivéve, hogy kollekcióból nem tudunk közvetlen primitív streamet csinálni, mivel az csak objektumokat képes tárolni.

DoubleStream empty = DoubleStream.empty();

DoubleStream varargs = DoubleStream.of(1.0, 1.1, 1.2);

DoubleStream random = DoubleStream.generate(Math::random);

DoubleStream fractions = DoubleStream.iterate(.5, d -> d / 2);

Nagyon gyakori, hogy egymás utáni egészek sorozatával kell dolgoznunk, ezért ezek előállítására külön is van lehetőség.

IntStream.range(int start, int end): start és end között egyesével növekvő számsorozat, start-ot beleértve, end-et nem.

IntStream.rangeClosed(int start, int end): start és end között egyesével növekvő számsorozat, start-ot és end-et is beleértve.

IntStream numbers = IntStream.range(2, 6); //2 3 4 5

IntStream numbersClosed = IntStream.rangeclosed(2, 6); //2 3 4 5 6

Váltás különböző típusú streamek között

A Stream map metódusához hasonló, de primitív be- és/vagy kimenettel működő metódusok is léteznek.

Míg a map ugyanolyan típusú be- és kimenetet feltételez, a mapToInt() kimenete

IntStream, a mapToLong() kimenete LongStream, a mapToDouble() kimenete DoubleStream, a mapToObj() kimenete Stream. Attól függően, hogy mi a bemenet és a kimenet, a kettő között konvertáló függvényt kell megadnunk. Az ennek megfelelő funkcionális interfészek összefoglalását az alábbi táblázatban találod.

metódus bemenet kimenet paraméter típusa

|  |  |
| --- | --- |
| map | Stream<T> Stream<R> Function<? super T, ? extends R> |
| map | XStream XStream XUnaryOperator |
| mapToInt | Stream<T> IntStream ToIntFunction<? super T> |
| mapToInt | XStream IntStream XToIntFunction |
| mapToLong | Stream<T> LongStream ToLongFunction<? super T> |
| mapToLong | XStream LongStream XToLongFunction |
| mapToDouble | Stream<T> DoubleStream ToDoubleFunction<? super T> |
| mapToDouble | XStream DoubleStream XToDoubleFunction |
| mapToObj | XStream Stream<R> XFunction<? extends R> |
| X lehet Int, Long vagy Double | |

Stream<String> objStream = Stream.of("penguin", "fish"); IntStream intStream = objStream.mapToInt(s -> s.length()); Nézz utána a többi primitívekkel dolgozó funkcionális interfésznek is!

Gyakori számítások

Primitív típusú streameken a min() és max() metódusokon kívül létezik még kettő gyakran használt záró művelet, a sum() és az average(). A sum() üres streamre 0-t ad, az average() viszont nem tud értéket visszaadni. Erre találták ki az Optional burkoló osztályt, azonban az csak objektumokat tud fogadni. Primitívekre elkészült az

OptionalInt, OptionalLong és OptionalDouble burkoló osztály. Ezekből az eredményt a típusától függőengetAsInt(), getAsLong() vagy getAsDouble() metódussal lehet kinyerni, de minden más Optional metódus létezik benne.

Tudjuk, hogy a stream a záró művelet után megszűnik létezni, ezért ha több számítást szeretnénk végezni rajta, akkor a summaryStatistics() metódussal mindenféle összesítést tartalmazó XXXSummaryStatistics objektumot kapunk, ahol XXX a stream típusának megfelelően Int, Long vagy Double.

Műveletei:

* getMin(), • getMax(),
* getSum(),
* getAverage(),
* getCount().

Mindegyik mindig szolgáltat adatot, még üres streamre is. Üres streamre a minimum a típus legnagyobb értéke, a maximum a legkisebb, az átlag és az összeg pedig 0.

IntStream integers = IntStream.range(1, 6);

IntSummaryStatistics stats = ints.summaryStatistics(); int max = stats.getMax(); int min = stats.getMin();

Ellenőrző kérdések

* Milyen primitívekkel dolgozó streamek vannak, és melyik melyik primitív(ek)nek felel meg?
* Milyen módon lehet primitív streamet létrehozni?
* Hogyan lehet Double elemeket tároló listából primitív streamet készíteni?
* Milyen számítások végezhetők primitív streamekkel?
* Milyen primitívekkel dolgozó funkcionális interfészeket ismersz?

Gyakorlat - Sportbolt

Hozz létre egy Product osztályt, amely a sportszer nevét, árát, darabszámát tárolja! A konstruktora is ezeket kapja meg ugyanebben a sorrendben. A SportGadgetStore osztály tárolja a termékek listáját, és különböző statisztikákat készít belőle. Az osztály kapja meg a listát kívülről.

Készítsd el a következő metódusokat streamek segítségével: getNumberOfProducts(): összesen hány termék van a boltban,

getAveragePrice(): átlagosan mennyibe kerül egy termék. Ha nincs termék, 0-t adjon vissza.

getExpensiveProductStatistics(double minPrice): adott árnál drágább termékek darabszámáról szolgáltat statisztikát. Az összesítést szövegként adja vissza az alábbi formában:

Összesen 3 féle termék, amelyekből minimum 1 db, maximum 52 db, összesen 74 db van.

Ha nincs ilyen, akkor a visszaadott szöveg a Nincs ilyen termék. legyen!

Collectors (lambdacollectors)

Az leggyakrabban használt záró művelet a collect(), mely legtöbbször a stream elemeivel egy kollekciót ad vissza. Collectors paraméterrel hívva akár összesítést is tud végezni az elemeken.

Összesítések

Az elemek összefűzése egyetlen szöveggé. Az elemeknek CharSequence típusúaknak kell lennie (a String is ez).

joining()

Collector<CharSequence,?,String> joining(): atz elemek nincsenek elválasztva egymástól

Stream<String> ohMy = Stream.of("lions", "tigers", "bears");

System.out.println(ohMy.collect(Collectors.joining())); // lionstigersbears

Collector<CharSequence,?,String> joining(CharSequence delimiter): az elemek közé a delimiter szöveg kerül

Stream<String> ohMy = Stream.of("lions", "tigers", "bears"); System.out.println(ohMy.collect(Collectors.joining(";"))); // lions;tigers;bears

Collector<CharSequence,?,String> joining(CharSequence delimiter, CharSequence prefix, CharSequence suffix): az elemek közé a delimiter kerül, az egész szöveg elé a prefix, utána a suffix

Stream<String> ohMy = Stream.of("lions", "tigers", "bears");

System.out.println(ohMy.collect(Collectors.joining(";", "Állatok: ", "")));

// Állatok: lions;tigers;bears counting()

<T> Collector<T,?,Long> counting(): az elemek számát adja vissza.

Stream<String> ohMy = Stream.of("lions", "tigers", "bears");

System.out.println(ohMy.collect(Collectors.counting())); // 3

Önmagában nem használjuk streamekre, mert ugyanazt adja vissza, mint a count(), de nagyon hasznos, amikor csoportosításkor minden csoportra meg akarjuk határozni az elemszámot.

minBy(), maxBy()

<T> Collector<T,?,Optional<T>> minBy(Comparator<? super T> comparator): a comparator szerinti legkisebb elem, ha van

<T> Collector<T,?,Optional<T>> maxBy(Comparator<? super T> comparator): a comparator szerinti legnagyobb elem, ha van

Ha nincs egyetlen elem sem, akkor mindkettő üres Optional-t ad vissza.

summingXXX(), averagingXXX()

Mindkét metódusnak három verziója létezik, attól függően, hogy milyen típusú adatokra alkalmazzuk. XXX lehet Int, Long és Double.

Collector<T,?,XXX> summingXXX(ToXXXFunction<? super T> mapper): az elemeket XXX primitív típusúra konvertálva összegzi azokat. A konvertáló függvényt paraméterben adjuk át. Üres streamre az eredmény 0.

Collector<T,?,Double> averagingXXX(ToXXXFunction<? super T> mapper): az elemeket XXX primitív típusúra konvertálva átlagolja azokat. A konvertáló függvényt paraméterben adjuk át. Üres streamre az eredmény itt is 0.

Stream<Integer> numberStream = Stream.of(2,7,3,5,9,10,-4); System.out.println(numberStream.collect(Collectors.summingInt(x -> x.intValue()))); // 32

Stream<Integer> numberStream = Stream.of(2,7,3,5,9,10,-4);

System.out.println(numberStream.collect(Collectors.averagingInt(x -> x.intValue()))); // 5.5714 summerizingXXX()

<T> Collector<T,?,XXXSummaryStatistics> summarizingXXX(ToXXXFunction<? super T> mapper): az elemeket XXX primitív típusúra konvertálva statisztikát készít róluk. XXX lehet Int, Long és Double

Stream<Integer> numberStream = Stream.of(2,7,3,5,9,10,-4);

System.out.println(numberStream.collect(Collectors.summarizingInt(x -> x.intValue())));

// IntSummaryStatistics{count=7, sum=32, min=-4, average=4,571429, max=10}

Kollekcióba gyűjtés

<T> Collector<T,?,List<T> > toList(): a stream elemeit listába gyűjti.

<T> Collector<T,?,Set<T> > toSet(): a stream elemeit halmazba gyűjti.

<T,C extends Collection<T>> Collector<T,?,C> toCollection(Supplier<C> collectionFactory): a stream elemeit a paraméterként átadott collectionFactory által előállított kollekcióba gyűjti.

Stream<String> ohMy = Stream.of("lions", "tigers", "bears");

TreeSet<String> result = ohMy.filter(s -> s.startsWith("t"))

.collect(Collectors.toCollection(TreeSet::new));

System.out.println(result); // [tigers]

<T,K,U> Collector<T,?,Map<K,U>> toMap(Function<? super T, ? extends K> keyMapper, Function<? super T,? extends U> valueMapper): a stream elemeit Mapbe gyűjti. Paraméterként át kell adnunk a kulcsot és az értéket előállító függvényt. Ha két elemnél ugyanazt a kulcsot állítjuk elő, akkor IllegalStateException-t dob.

<T, K, U> Collector<T,?,Map<K,U> toMap(Function<? super T,? extends

K> keyMapper, Function<? super T,? extends

U> valueMapper, BinaryOperator<U> mergeFunction) : a stream elemeit Map-be gyűjti. Paraméterként át kell adnunk a kulcsot és az értéket előállító függvényt, valamint azt a függvényt, amely kulcsütközés esetén a már a Map-ben lévő és az új elemet egyesíti.

<T,K,U,M extends Map<K,U>> Collector<T,?,M> toMap(Function<? super T, ?

extends K> keyMapper, Function<? super T, ? extends

U> valueMapper, BinaryOperator<U> mergeFunction, Supplier<M> mapSupplier) :

ez előbbieken túl azt is átadjuk, hogy milyen Map-et szeretnénk visszakapni.

Stream<String> ohMy = Stream.of("lions", "tigers", "bears"); TreeMap<Integer, String> map =

ohMy.collect(Collectors.toMap(String::length, //keyMapper k -> k, //valueMapper (s1, s2) -> s1 + "," + s2, //mergeFunction

TreeMap::new)); //mapSupplier

System.out.println(map); // {5=lions,bears, 6=tigers}

System.out.println(map.getClass()); // class. java.util.TreeMap Partícionálás feltétel alapján: partitioningBy

static <T> Collector<T,?,Map<Boolean,List<T> >> partitioningBy(Predicate<? super T> predicate): a stream elemeit kettévállogatja aszerint, hogy megfelelnek-e az átadott predicate-nek. A két elkészült listát Map-be szúrj, ahol a kulcs a true és a false .

Stream<String> ohMy = Stream.of("lions", "tigers", "bears"); Map<Boolean, List<String>> map = ohMy.collect(Collectors.partitioningBy(s -> s.length() <= 5)); System.out.println(map); // {false=[tigers], true=[lions, bears]}

public static <T,D,A> Collector<T,?,Map<Boolean,D>>

partitioningBy(Predicate<? super T> predicate, Collector<? super T,A,D> downstream): ha lista helyett mást szeretnénk használni, akkor második paraméterként megadhatunk egy downstream Collector-t, amely azt határozza meg, hogy az elemek milyen kolleckióba kerüljenek, illetve hogy egyáltalán mi kerüljön a Map-be értékként.

Például ha Set-be szeretnénk megkapni a partíciókat, akkor

Stream<String> ohMy = Stream.of("lions", "tigers", "bears"); Map<Boolean, Set<String>> map = ohMy.collect(Collectors.partitioningBy(s -> s.length() <= 7,

Collectors.toSet()));

System.out.println(map);// {false=[], true=[lions, tigers, bears]}

Ha csak az elemek számát szeretnénk megkapni partíciónként, akkor

Stream<String> ohMy = Stream.of("lions", "tigers", "bears"); Map<Boolean, Long> map = ohMy.collect(Collectors.partitioningBy(s -> s.length() <= 7, Collectors.counting()));

System.out.println(map); // {false=0, true=3}

Tulajdonság alapján csoportosítás: groupingBy

Működése és a paraméterei nagyon hasonlítanak a partitioningBy metóduséhoz, azzal az eltéréssel, hogy itt a csoportok kulcsa nem csak logikai, hanem bármi más lehet, így akárhány csoportot képezhetünk. Az első paramétere éppen ezért Function, nem Predicate. Második paraméterként megadhatunk egy Map-et szolgáltató Supplier-t, ha meg akarjuk határozni, hogy milyen Map-et kapjunk.

<T,K> Collector<T,?,Map<K,List<T>>> groupingBy(Function<? super T,? extends K> classifier)

Stream<String> ohMy = Stream.of("lions", "tigers", "bears"); Map<Integer, List<String>> map = ohMy.collect(Collectors.groupingBy(String::length)); System.out.println(map); // {5=[lions, bears], 6=[tigers]}

<T,K,A,D> Collector<T,?,Map<K,D>> groupingBy(Function<? super T,? extends K> classifier, Collector<? super T,A,D> downstream)

Ha nem listába, hanem halmazba szeretnénk gyűjteni a csoport elemeit:

Stream<String> ohMy = Stream.of("lions", "tigers", "bears"); Map<Integer, Set<String>> map = ohMy.collect(Collectors.groupingBy(String::length,

Collectors.toSet()));

System.out.println(map); // {5=[lions, bears], 6=[tigers]}

Ha nem az elemeket, hanem az elemek számát szeretnénk a Map-be értékként:

Stream<String> ohMy = Stream.of("lions", "tigers", "bears");

Map<Integer, Long> map = ohMy.collect(Collectors.groupingBy(String::length,

Collectors.counting()));

System.out.println(map); // {5=2, 6=1}

<T,K,D,A,M extends Map<K,D>> Collector<T,?,M> groupingBy(Function<? super T,? extends K> classifier, Supplier<M> mapFactory, Collector<? super T,A,D> downstream)

Ha a csopportokat tartalmazó Map TreeMap legyen, a csoportok elemei pedig Set-be kerüljenek:

Stream<String> ohMy = Stream.of("lions", "tigers", "bears"); TreeMap<Integer, Set<String>> map = ohMy.collect(Collectors.groupingBy(String::length,

TreeMap::new,

Collectors.toSet()));

System.out.println(map); // {5=[lions, bears], 6=[tigers]}

Elemek konvertálása összegyűjtés előtt mapping

Ha az összegyűjtött elemek nem egyeznek a stream elemeivel, de azokból valamilyen függvénnyel előállítható, akkor használjuk a mapping metódust.

<T,U,A,R> Collector<T,?,R> mapping(Function<? super T,? extends U> mapper, Collector<? super U,A,R> downstream)

Stream<String> ohMy = Stream.of("lions", "tigers", "bears");

Map<Integer, Optional<Character>> map = ohMy.collect(

Collectors.groupingBy(

String::length, //a csoportosítás alapja a szöveg hossza Collectors.mapping( s -> s.charAt(0),

Collectors.minBy(Comparator.naturalOrder()))));

/\* Nem a szövegeket, hanem csak az első karaktereket gyűjtjük össze, és azokból is csak a "legkisebbet"\*/

System.out.println(map); // {5=Optional[b], 6=Optional[t]}

Ellenőrző kérdések

* Milyen fő csoportjai vannak a Collectors metódusainak?
* Sorolj fel öt összesítő metódust! Mit csinálnak?
* Hogyan lehet meghatározni a kimeneti kollekció fajtáját?
* Hogyan tudsz az eredményből csoportokat képezni?
* Hogyan lehet összegyűjtés előtt az elemeket másféle objektummá konvertálni?

Gyakorlat - Kávézó v2

Készíts egy Coffee osztályt! Attribútumai: type a kávé típusa, price a kávé ára. A konstruktor is ebben a sorrendben kapja meg az adatokat. Az ár lehet tört, 2 tizedesjegy pontossággal számolj!

A kávé típusához készíts egy CoffeeType enum-ot. Lehetséges értékei: ESPRESSO, MACHIATTO, RISTRETTO, MOCHA, LATTE, CAPPUCCINO, AMERICANO.

A CoffeeOrder osztály tárolja egy vásárló által megrendelt és leszámlázott kávékat.

Attibútumai: coffeeList a megrendelt kávék listája, dateTime a vásárlás időpontja.

A Cafe osztály tartalmazza a kávézó összes rendelését egy listában. A listát a konstruktorban kapja meg, de legyen lehetőség új rendelést hozzáadni. Készítsd el benne az alábbi metódusokat streamek segítségével:

Map<CoffeeType, Long> getCountByCoffeeType(): az eladott kávék mennyiségét adja vissza kávétípusonként double getAverageOrder(): átlagosan hány kávét rendelnek egyszerre Dátum és időkezelés

Új típusok, LocalDate, LocalTime, parse (datenewtypes)

Ezeket a típusokat Java-8 ban vezették be és a java.time csomagban találhatóak. A

LocalDate csak dátum, a LocalTime idő, míg a LocalDateTime idő és dátum tárolására alkalmas. Egyik sem tartalmaz időzónát erre a ZonedDateTime használható. Fontos, hogy mindegyik immutable.

Használatuk

A now() metódussal lehet létrehozni, egy új dátumot. A toString() metódus is megfelelően implementálva van. Konkrét dátum of() metódussal adható meg.

Például:

LocalDate date = LocalDate.of(2015, Month.JANUARY, 20); LocalDate date = LocalDate.of(2015, 1, 20);

Ezek az osztályok nem engedik meg a túlcsordulást, ebben az esetben DateTimeException-t dob.

Műveletek

* plusXxx(),minusXxx()
* A műveletek láncolhatók
* DayOfWeek, Month enumok
* isAfter(), isBefore() összehasonlításra használhatóak

LocalDate date = LocalDate.of(2014, Month.JANUARY, 20);

System.out.println(date); // 2014-01-20 date = date.plusDays(2);

System.out.println(date); // 2014-01-22 date = date.plusWeeks(1);

System.out.println(date); // 2014-01-29

LocalDate.of(2014, Month.JANUARY, 20).plusDays(2).plusWeeks(1);

Átjárás a típusok között

LocalDateTime localDateTime = LocalDateTime.now();

LocalDate localDate = localDateTime.toLocalDate();

LocalTime localTime = localDateTime.toLocalTime();

LocalDateTime newLocalDateTime = LocalDateTime.of(localDate, localTime);

Formázás és parse-olás

Itt is lehetőség van Stringből beolvasni illetve Stringé alakítani a dátumokat. Itt a

DateTimeFormatter osztályt kell használnunk. Egy ilyet többféleképpen létrehozhatunk:

* Konstanssal: DateTimeFormatter.ISO\_LOCAL\_DATE
* Lokalizált stílussal: DateTimeFormatter.ofLocalizedDate(FormateStyle.SHORT)
* Formátum Stringgel: DateTimeFormatter.ofPattern("MM dd, yyy, hh:mm")

A formázáshoz vagy a formatter vagy LocalDateTime metódusait hívhatjuk. A DateTimeFormatter immutable és szálbiztos.

Nézzünk néhány példát a formázásra:

DateTimeFormatter formatter = DateTimeFormatter.ofPattern("yyyy.MM.dd.

HH:mm");

LocalDateTime now = LocalDateTime.of(2017, Month.JANUARY, 1, 12, 0);

System.out.println(formatter.format(now));

System.out.println(now.format(formatter));

LocalDateTime start = LocalDateTime.parse("2017.01.01. 12:00", formatter);

System.out.println(start);

Ellenőrző kérdések

* Mire valók a LocalDate, LocalTime és LocalDateTime osztályok?
* Mit értünk az alatt, hogy ezen osztályok immutable-ök?
* Mi történik, ha rossz a hónap, vagy a nap paraméter?
* Május hónapnak milyen integer érték felel meg?
* Mi a szerepe a DayOfWeek és Month enumoknak?
* Hogyan lehet módosítani az időt, dátumot tároló objektumokat?
* Hogyan lehet összehasonlítani őket?
* Hogyan hozhatunk létre egy LocalDateTime objektumot az aktuális idővel?
* Hogyan hozhatunk létre egy LocalDateTime objektumot egy előre megadott értékkel?
* Milyen módokon adhatunk meg formátum leírásokat?
* Hogyan formázhatjuk a LocalDate, LocalTime és LocalDateTime osztályok példányait?
* Hogyan alkalmazhatjuk a Locale-t a formázásoknál?

Gyakorlati feladat - Születésnap

Életünk nevezetes dátumairól (DateOfBirth osztály) szeretnénk speciális információkat kapni, mint például: a hét mely napján születtünk, eddig összesen hány napot éltünk, vagy a saját és barátunk/párunk születése között pontosan hány nap telt el. A dátumokat tetszőlegesen formázott String formájában is szeretnénk látni.

Hibakezelés

Üres pattern esetén dobjon IllegalArgumentException-t az adott metódus. A Locale nem lehet null. Ha rossz dátumot adtunk meg születési dátumként, a countDaysSinceBirth() metódus dobjon IllegalStateException-t

Tippek

Használd a ChronoUnit enumot arra, hogy kiszámold, hány nap van két dátum között! Szervezzük külön metódusba az azonos paraméterek ellenőrzését!

Megjegyzés

Érdemes a dokumentációban átnézni a LocalDate, LocalTime osztályok nyújtotta lehetőségeket!

Gyakorlati feladat - Találkozó

Fontos találkozó előtt állunk, és nem akarjuk lekésni, ezért tudnunk kell, hány perc van addig. Azzal is számolunk, hogy az időpont módosulhat. Egy Rendezvous osztályt készítünk a funkciókhoz.

Hibakezelés

Üres pattern esetén dobjon IllegalArgumentException-t. Hasonlóképpen a sikertelen parszolás is dobjon IllegalArgumentException-t. Ha elfeledkezünk az időpontról és már késő elmenni, a countMinutesLeft() metódus dobjon

MissedOpportunityException-t. Ezt nekünk kell megírni, ez is egy RuntimeException.

Tippek

Használd a ChronoUnit enumot arra, hogy kiszámold, hány perc van két idő között!

Szervezzük külön metódusba a paraméterek ellenőrzését!

Gyakorlat - Szülinapok

Írj egy FamilyBirthdays osztályt, mely konstruktor paraméterül kap születésnapokat. Implementáld benne az isFamilyBirthday és nextFamilyBirthDay metódusokat, a tesztben szereplő method reference-ek alapján.

A isFamilyBirthday visszaadja, hogy a paraméterként átadott dátum születésnap-e. A nextFamilyBirthDay metódus visszaadja, hány nap van a legközelebbi születésnapig.

Implementáció

Nézd meg LocalDate query() metódusát, hogy mit kap paraméterül. Használd a ChronoUnit osztályt annak meghatározására, hogy két dátum között hány nap telt el.

Gyakorlati feladat

Egy olyan osztályt - DateOfBirth - akarunk létrehozni, amely életünk nevezetes dátumaival kapcsolatos extra információkat szolgáltat. Például meg tudja mondani, a hét melyik napján születtünk, vagy akár csak azt, hogy a nevezetes dátum hétköznapra vagy hétvégére esett. :) Az osztály dateOfBirth attribútuma tárolja a kérdéses nevezetes dátumot, ebben az esetben a születésünk dátumát. Arra is kíváncsiak lehetünk, hogy a 40. születésnapunk milyen napra fog esni (nem lehet elég korán elkezdeni a készülést).

Megvalósítás

Az osztály objektumait többféle módon is létre lehessen hozni, számokkal vagy akár szöveges dátum formátumból.

publikus metódusok:

public DateOfBirth(int year, int month, int day) public DateOfBirth(String dateString)

public String findDayOfWeek() public boolean isWeekDay() public boolean wasItALeapYear() public String findBirthDayOfWeekLater(int year)

Tippek

Privát metódus segíthet abban, hogy a findDayOfWeek metódust eltérő paraméterezéssel újra tudjuk hasznosítani. Érdemes utánanézni a LocalDate osztály leírásának a Java dokumentációban, és megismerkedni a DayOfWeek enum használatával is.

Gyakorlati feladat 2

Phileas Fogg egy rendkívül precíz angol gentleman, aki a napjait percnyi pontossággal osztja be. Nyelvtanára sajnos nem ennyire pontos, minden délelőtt 9 óra körül kezdi az órákat, és három 45 perces nyelvórát tart egy 25 perces szünettel.

Fogg úr percre pontosan szeretné tudni, hogy mikor fogják befejezni, és azt is, hogy be fogják-e fejezni déli 12 előtt az órákat. Segítsünk neki ebben!

Egy olyan osztályt - DailyRoutine - hozzunk létre, amelynek objektuma tárolja a mindenkori órakezdetet percnyi pontossággal, és ki tudja számolni, mikor fejeződik be az oktatás, valamint azt is meg tudja mondani, hogy ez még déli 12 óra előtt megtörténik-e.

Az osztály startTime attribútuma tárolja a nyelvórák kezdetét, ez megadható óra és perc, de megadható standard szöveges formában is (hh:mm).

Megvalósítás

Az osztály objektumait többféle módon is létre lehessen hozni, számokkal vagy akár szöveges dátum formátumból. Publikus metódusai segítségével a tárolt időpont módosítható a befejezés időpontjára, és lekérdezhető, hogy ez az időpont hogyan viszonyul a déli 12 órához, azt megelőzi-e.

publikus metódusok:

public DailyRoutine(int hour, int minute) public DailyRoutine(String timeString)

public void setFutureTime(int minutes) public boolean isBeforeNoon()

Tippek

Érdemes utánanézni a LocalTime osztály leírásának a Java dokumentációban, és megismerkedni az osztály által biztosított LocalTime.NOON final static értékkel.

Régi típusok, Date, Calendar, parse (dateoldtypes)

Időzóna fogalma

Az időzóna a földfelszín azon területe, ahol az időmérő eszközök azonos időt mutatnak. Technikai okok miatt ezek általában ország-, államhatárokhoz vannak igazítva szélességi kör alapján. Az „origo” a korrigált világidő (UTC), régebbi nevén greenwichi középidő (GMT). Magyarország a téli-nyári időszámítás miatt télen a közép-európai időzónához (CET, UTC+1), nyáron a középeurópai nyári időzónához (CEST, UTC+2) tartozik.

Date osztály

Javaban ez az osztály felelős a dátum kezelésre és az 1970 00:00:00 (GMT) óta eltelt időt tárolja ezredmásodpercben. A Date osztály a java.util csomagban helyezkedik el. Az osztály pédányosítható paraméter nélküli konstruktorral, de a háttérben valójában egy System.currentTimeMillis() hívás van amely visszaadja az 1970 óta eltelt időt, de ez egy getTime() metódussal le is kérdezhető. A Date nem tárol időzónát, a rendszer időzónája alapján dolgozik. Az osztály immutable és nem használható adott idő létrehozására vagy műveletekre. Ezekre a Calendar interface-t használjuk.

Calendar interface

Különböző naptárakkal tudunk dolgozni. A naptár definiálja év, hónap illetve nap fogalmát. A Calendar.getInstance() metódus visszaadja a rendszer által használt naptárt (Magyarországon Gergely naptár - GregorianCalendar).

Calendar beállításai

A Calendar implementációi módosíthatók. java c.set(2015, Calendar.JANUARY,1) A fenti metódussal állíthatunk be dátumot, illetve akár időt is.

A Calendar tartalmaz konstansokat is Calendar.YEAR, Calendar.MONTH. Ezeket természetesen get() metódussal le is tudjuk kérdezni.

Calendar műveletei

* getTime()-Date objektummá lehet konvertálni
* setTime(Date)-Dátum alapján beállítani
* add(Calendar.YEAR,3)-Háromévet hozzáad az adott évhez (nappal, hónappal, negatív számmal is működik)
* Összehasonlító műveletek: after(), before()
* Lenient: túl nagy érték esetén engedi a túlcsordulást például 36. hónapot (kikapcsolható)
* A hónapok 0-tól indexeltek

Stringből olvasás Stringé alakítás

A Stringből olvasáshoz a DateFormat interfacet és annak a SimpleDateFormat implementációját kell használni. Fontos a MM jelöli a hónapot és a mm jelöli a percet.

DateFormat dateFormat = new SimpleDateFormat("yyyy-MM-dd hh:mm"); try {

Date date = dateFormat.parse("2017-04-04 12:00");

System.out.println(dateFormat.format(date));

}

catch (ParseException pe) {

pe.printStackTrace();

}

A parse() metódussal lehet Stringből dátumot, a format() metódussal pedig dátumból Stringet konvertálni.

Ellenőrző kérdések

* Mi az az időzóna?
* Mi a különbség a Date és Calendar objektumok között? Melyik mire való?
* Mit tárol a Date?
* Date példányosításával mi lesz annak az értéke?
* Hogyan hozhatunk létre kifejezetten GregorianCalendar objektumot? Mitől függ a Calendar objektum konkrét típusa?
* Miért ne használjuk a Date deprecated metódusait?
* Milyen Calendar konstansokat ismersz?
* Hogy lehet két dátumot összehasonlítani?
* Hogy lehet egy dátumot eltolni? (Pl. 5 napot hozzáadni?)
* Hogyan lehet dátumot formázni? Hogy lehet egy dátumot tartalmazó szöveget dátummá alakítani?
* Mi a lenient beállítás szerepe és mi a default értéke?

Gyakorlati feladat - Nevezetes dátumok

Egy olyan osztályt akarunk létrehozni, amely életünk nevezetes dátumaival kapcsolatos extra információkat szolgáltat. Például meg tudja mondani, a hét melyik napján születtünk, vagy akár csak azt, hogy a nevezetes dátum hétköznapra vagy hétvégére esett. :)

Hibakezelés

Biztosítsuk a teszteseteknek megfelelően, hogy illegális év, hónap és nap paraméter értékek esetén, valamint hiányos dátum és formázó string paraméterek esetén dobjon IllegalArgumentException kivételt a megfelelő tájékoztató szöveggel, illetve null paraméter esetén dobjon NullPointerException-t, szintén a megfelelő szöveggel.

Megvalósítási javaslatok

Az osztály objektumait többféle módon is létre lehessen hozni. Figyeljünk a lenient flag beállítására!

publikus metódusok:

public DateOfBirth(int year, int month, int day)

public DateOfBirth(String dateString, String pattern, Locale locale) public DateOfBirth(String dateString, String pattern) public String findDayOfWeekForBirthDate(Locale locale) public String toString(String pattern) public boolean isWeekDay()

Tippek

A paraméter string ellenőrzésére készüljön külön metódus, amit többször is fel tudunk használni. Hasonlóképpen a konstrukció során többször használandó közös utasításokat szervezzük ki külön metódusba.

boolean isEmpty(String str) void setDateOfBirth(String dateString, DateFormat dateFormat) Architektúrák

JVM (jvm)

Ellenőrző kérdések

* Hogyan valósítja meg a Java a hordozhatóságot, platform függetlenséget?
* Hogy hívják a Sun/Oracle féle megvalósítást?
* Mi felel az osztályok betöltéséért? Honnan képes osztályokat betölteni?
* Mi biztosítja a Java alkalmazások gyors futását annak ellenére, hogy egy virtuális gép futtat egy interpretált nyelvet?
* Mi alapján érdemes választani, hogy 32 vagy 64 bites JVM-et telepítsünk?
* Hogyan működik a garbage collector? Mi az az elv, ami alapján felépítették?

JVM működésének naplózása

Indítsd el az alkalmazást parancssorból.

Írasd ki a GC működését a -verbosegc kapcsolóval. Milyen értékeket ír ki, és ezek mit jelentenek?

Írasd ki a JIT működését a -XX:+PrintCompilation kapcsolóval. Milyen értékeket ír ki, és ezek mit jelentenek?

Bónusz feladat 1.

Elemezd a forráskódot és a tesztesetet! Mit csinál az alkalmazás? Történhet olyan, hogy futás közben előbb-utóbb elfogy a memória?

Bónusz feladat 2.

Amennyiben van három objektum, mely körkörösen egymásra hivatkoznak, de kívülről rájuk más objektum nem hivatkozik, képes a GC eldobni őket egyszerre?

Bónusz feladat 3.

Indítsd el a Java VisualVM alkalmazást, és csatlakozz a futó virtuális géphez. Mit mutat meg az alkalmazás a JVM belső működéséről?

Bónusz feladat 4.

A Tools/Plugins menüpontban telepítsd a Visual GC plugint, majd csatlakozz újra a JVMhez. Miket mutat meg a Visual GC plugin?

Forrás

OCA - Chapter 1/Destroying Objects, Benefits of Java

Third party library-k (thirdparty)

Ellenőrző kérdések

* Mi szükség van third party library-kre?
* Mi az a library, ami nem a Java SE része, és eddig is használtuk?
* Mi az a CLASSPATH?
* Hogyan lehet third party library-t használni Mavennel? • Melyek a leggyakrabban használt third party library-k?

Szavak számlálása

Írj egy WordCounter osztályt, mely public int numberOfWords(String s) metódusa a paraméterként átadott szöveget szavakra bontja, és visszaadja, hogy mennyi szóból áll. Tegyük fel, hogy space, vessző és pont karakterek az elhatároló karakterek.

Próbáld ki először a String split() metódusát.

A Guava library Splitter osztálya paraméterezhető, használd azt! Először a függőséget kell definiálni a pom.xml állományban.

<dependency>

<groupId>com.google.guava</groupId>

<artifactId>guava</artifactId>

<version>21.0</version>

</dependency>

Adjon vissza egy listát (splitToList), és annak a méretét (size() metódus).

Futtasd az alkalmazást parancssorból!

A java -classpath target\classes thirdparty.WordCounterMain "foo bar" parancssor futtatásakor a következő hibaüzenetet kapjuk:

D:\vicziani\Work\yellowroad-java\thirdparty>java -classpath target\classes thirdparty.WordCounterMain "foo bar"

Exception in thread "main" java.lang.NoClassDefFoundError:

com/google/common/base/CharMatcher at thirdparty.WordCounter.numberOfWords(WordCounter.java:9) at thirdparty.WordCounterMain.main(WordCounterMain.java:10)

A JAR-t futtatva ugyanezt a hibaüzenetet kapjuk: java -jar target\thirdparty-1.0SNAPSHOT.jar "foo bar".

Megoldásként a CLASSPATH-ra rá kell tenni a Guava third party library-t. Töltsd le a következő címről:

http://repo1.maven.org/maven2/com/google/guava/guava/21.0/guava-21.0.jar, és másold a target könyvtárba. (Vigyázz, az mvn clean parancs futtatásakor a target könyvtár törlésre kerül.)

Futtassuk a következő parancssorral: java -classpath target\classes;target\guava-21.0.jar thirdparty.WordCounterMain "foo bar".

A pom.xml-t egészítsük ki a következőképpen:

<plugin>

<artifactId>maven-jar-plugin</artifactId>

<configuration>

<archive>

<manifest>

<mainClass>thirdparty.WordCounterMain</mainClass>

<addClasspath>true</addClasspath>

</manifest>

</archive>

</configuration>

</plugin>

Futtassuk a következő paranccsal: java -jar target\thirdparty-1.0-SNAPSHOT.jar "foo bar".

Mi változott a JAR állományban a META-INF\MANIFEST.MF fájlban?

Egészítsd ki a pom.xml állományt a következőképpen:

<plugin>

<artifactId>maven-assembly-plugin</artifactId>

<configuration>

<archive>

<manifest>

<mainClass>thirdparty.WordCounterMain</mainClass>

</manifest>

</archive>

<descriptorRefs> <descriptorRef>jar-with-dependencies</descriptorRef>

</descriptorRefs>

</configuration>

</plugin>

A következő parancsot kell futtatni: mvn clean compile assembly:single, majd az

alkalmazást: java -jar target\thirdparty-1.0-SNAPSHOT-jar-withdependencies.jar "foo bar".

Mi a különbség a két JAR között?

Bónusz feladat 1.

A Maven hol keresi a guava-21.0.jar állományt a tesztesetek futtatásakor?

Naplózás (logging)

Ellenőrző kérdések

* Miért használunk naplózó keretrendszert?
* Milyen elvárásaink lehetnek egy naplózó keretrendszerrel kapcsolatban?
* Milyen Java megvalósításokat ismersz?
* Mi az a logger és naplózási szint?

Gyakorlati feladat - karakterek keresése

Készíts egy logging.CharacterCounter osztályt, benne egy int countCharacters(String source, String chars) metódust, mely megszámolja hogy az első paraméterként megadott szövegben hányszor szerepel a második paraméterként megadott karakterek egyike.

A metódus elején naplózd ki, hogy a metódus meghívásra került, és milyen paraméterekkel.

Pl. Finding 'ae' characters in 'abcdabcdabcdabce'.

Amennyiben találat van, naplózd ki, hogy melyik karaktert találta meg az algoritmus, és hanyadik karakteren.

Pl. 'a' character found at 12. index

Tipp

Függőségként fel kell venni a pom.xml állományban az SLF4J-t a következőképpen:

<dependency>

<groupId>org.slf4j</groupId>

<artifactId>slf4j-api</artifactId>

<version>${slf4j.version}</version>

</dependency>

<dependency>

<groupId>org.slf4j</groupId>

<artifactId>slf4j-simple</artifactId>

<version>${slf4j.version}</version>

</dependency>

A verziószámot a properties tagen belül kell definiálni:

<slf4j.version>1.7.22</slf4j.version>