Tartalomjegyzék

Bevezető

Többször hallottam azt a számítástechnikai kiadványokkal kapcsolatos kijelentést, hogy az informatika esetében, ha valami már nyomtatásban is megjelent, akkor az már biztosan elavult, és ez halmozottan jellemző a magyar nyelvű könyvekre, kiadványokra. Természetesen a szakmabeliek tudják, hogy ez nem teljesen igaz. Bár valóban korunk legdinamikusabban fejlődő tudományterületéről van szó, ahol egymást követik a változások, újdonságok, nagyon sok dolog nem változik. A régi jó algoritmusokra vonatkozó könyvekben leírtak például ma is ugyanúgy érvényesek, mint megjelenésük pillanatában, és a tanulási folyamatban is következeteseknek kell lennünk: ahhoz, hogy alkalmazni tudjuk az új eredményeket, először meg kell értenünk az alapokat.

Mégis igaz lehet, hogy nehezebb helyzetben van az a szerző, aki olyan számítástechnikai témát akar tárgyalni, amelyre inkább jellemző a változás. Ebbe a kategóriába tartozhat egy programozási nyelv, vagy platform bemutatása is, nem beszélve a különböző keretrendszerekről, környezetekről, technológiákról. Azt hiszem, hogy joggal teheti fel az olvasó a kérdést, hogy miért kellene csaknem húsz évvel a Java programozási nyelv megszületése után a nyelvvel kapcsolatos alapfogalmakat tárgyaló könyvet írni. Miért lehet erre szükség, mi a célja?

A válasz a következő: a célom egy olyan jegyzet megírása volt, amely tanulmányi segédanyagként szolgálhat azok részére, akik most szeretnének megismerkedni a Java programozási nyelvvel. Elsősorban egyetemi hallgatókra gondolok, de bárki felhasználhatja, aki hasonló alapismeretekkel rendelkezik.

És miért éreztem úgy, hogy hivatott lennék egy ilyen jegyzet összeállítására? Egyetemi oktatóként az elmúlt néhány évben több Java programozási nyelvvel kapcsolatos tárgyat tanítottam a Babeș-Bolyai Tudományegyetem Matematika és Informatika Karán tanuló hallgatóknak. A jegyzet ezeken a tapasztalatokon, előadásokon, laboratóriumi és szemináriumi feljegyzéseken alapszik. Ez a tény és az előbbiekben meghatározott cél együttesen befolyásolja a jegyzet tartalmát és szerkezetét.

Elsőéves egyetemi hallgatóink az objektumorientált programozás tárgy keretein belül ismerkednek meg az objektumorientált programozási paradigmával és a C++ programozási nyelvvel. Másodévesen találkoznak a Java programozási nyelvvel (ennek a tárgynak az elsajátításához nyújthat segítséget a jelen jegyzet), majd ezután sajátítanak el (a másod- és harmadév folyamán) különböző, a Java platformhoz kapcsolódó ismereteket (ezek talán egy következő jegyzetnek lehetnek részei).

A jegyzetet úgy állítottam össze, hogy az első lépések megtételéhez nyújtson segítséget, de számításba vettem azt a tényt is, hogy az olvasók már ismerik az objektumorientált programozás alapfogalmait, illetve bizonyos mértékben járatosak a C és C++ programozási nyelvekben. Nem fogom részletesen tárgyalni az idekapcsolódó alapfogalmakat.

Nem fogom például részletezni a Java nyelv alapvető szintaxisát, vagy a nagyon egyszerű nyelvi elemeket, egyrészt mert ebben a tekintetben nagyon sok a hasonlóság a C++ nyelvvel, másrészt a különbségeket nagyon sok tanulmány részletezi. A jelen jegyzetet úgy próbáltam felépíteni, hogy inkább egy, a Java nyelv specifikumaira koncentráló gyakorlati megközelítést nyújtson.

Ennek a gondolatnak a szellemében alakítottam ki a könyv struktúráját. Minden fejezet tulajdonképpen egy-egy előadás kivonatának tekinthető, célja megalapozni az adott témával kapcsolatos gyakorlati tevékenységet (a szemináriumi és laborfeladatok megoldását). A szaktárgy egyetemi programjának megfelelően egy-egy fejezet egy-két heti tevékenységnek az alapja, de természetesen más „időzítés" is alkalmazható. A kialakított struktúra valószínűleg erőteljesen tükrözi az általunk alkalmazott oktatási programot, stratégiát.

Az első fejezetben a Java nyelv alapvető tulajdonságaival és rövid történetével ismerkedhetünk meg. Szó lesz a Java programok szerkezetéről és a futtatásukhoz szükséges környezetről, valamint a fejlesztéshez használható eszközökről. Ezután áttekintjük azokat az alapelemeket, amelyek feltétlenül szükségesek egy egyszerű Java program megírásához. Magyarázatokon és példaprogramokon keresztül megismerkedünk a primitív adattípusokkal, burkoló osztályokkal, a referenciák alkalmazásával, a tömbök kezelésével és a kivételkezelés alapjaival.

A második fejezet az objektumorientált programozási paradigma alapjait ismétli át, és tárgyalja a Java-val kapcsolatos jellemzőket. Melyek az objektumorientált fejlesztés jellemzői? Hogyan viszonyul a Java az olyan kérdésekhez, mint például a többszörös öröklődés? Hogyan valósítható meg a polimorfizmus? Miért kell arra törekednünk, hogy „interfészekben gondolkodjunk"? Ezekre és ezekhez hasonló kérdésekre próbál meg választ nyújtani a jegyzet második része.

A harmadik, negyedik és ötödik fejezetben grafikus felhasználói felületekkel kapcsolatos kérdések kerülnek tárgyalásra. Miért ez a „hirtelen ugrás"? Szintén a feladatmegoldás motiválását szolgálja: a tapasztalataink azt mutatják, hogy szívesebben programozunk, ha az eredményhez hozzátartozik egy barátságos és vizuális élményt is nyújtó felhasználói felület.

A harmadik fejezetben az AWT grafikus felhasználói felületek készítésére alkalmazható csomagról, illetve az eseménykezelés alapjairól lesz szó. A negyedik fejezet ugyanezt a témát folytatja: megtanulunk rajzolni, egyszerű kis Appletteket létrehozni, és megismerkedünk a Java biztonsági mechanizmusának alapjaival. Azt is meglátjuk, hogy milyen lehetőségeink vannak arra, hogy olyan felületeket készítsünk, amelyeket különböző földrajzi régiók különböző anyanyelvű felhasználói egyaránt kényelmesen tudnak használni. Ezután az ötödik fejezetben a SWING grafikus eszköztárat tárgyaljuk. Kiemeljük az AWT és SWING csomagok közötti alapvető különbségeket, és megpróbálunk választ adni arra a kérdésre, hogy mi lehetett a magyarázata az AWT alternatíva igényének, a SWING gyors megjelenésének és térhódításának.

Hogyan tudunk egy alkalmazáson belül több végrehajtási szállal dolgozni, több művelet párhuzamos végrehajtását lehetővé tenni? Hogyan tudnak ezek a szálak egymás között kommunikálni, és hogyan valósítható meg a műveletek szinkronizálása? Ezekre a kérdésekre próbál választ adni a hatodik fejezet.

A hetedik fejezet az adatfolyamokkal kapcsolatos alapfogalmakat, a bemeneti és kimeneti műveletek megvalósítására alkalmazható osztályokat, valamint az állománykezeléssel kapcsolatos kérdéseket tárgyalja, és röviden bemutatja a Java szerializációs mechanizmusát.

Alkalmazásainkban nagyon gyakran kell gyűjteményekkel, listákkal dolgoznunk. Milyen lehetőségeket, eszközöket biztosít erre a célra a Java? A gyűjtemény keretrendszerről (JCF – *Java Collections Framework*) a nyolcadik fejezetben lesz szó. Ugyanitt tárgyaljuk a témához szervesen kapcsolódó generikus típus fogalmát.

És ami kimaradt: a könyv első részében csak nagyon alapszinten tárgyalt témákkal kapcsolatos kiegészítéseket, részleteket mutat be példákon keresztül a kilencedik fejezet. Visszatérünk a kivétel- és eseménykezelésre, részletesebben bemutatva ezeket, és röviden áttekintjük a Java alapvető csomagjait.

A bevezetőben említett szaktárgy keretein belül a félév végén a hallgatóknak egy kis projekttel kell bizonyítaniuk, hogy elsajátították a fenti ismereteket. Az utolsó fejezet egy kivonatolt esettanulmány, célja segítséget nyújtani egy ilyen projekt megtervezéséhez.

Megjegyzendő, hogy a jegyzet egy többrészes anyag első kötetének volt tervezve. Remélhetőleg, hamarosan elkészülhet a folytatás is, amelyben „haladóbb" témák kaphatnának helyet: hálózati alkalmazások és osztott rendszerek, perzisztenciával kapcsolatos API-k és technológiák, dinamikus, komponens alapú és szolgáltatásorientált rendszerek fejlesztését célzó keretrendszerek, stb. De addig is remélem, hogy ez az első rész megfelelő segédeszközként fog szolgálni az alapok elsajátításához.

Jó olvasást, kellemes időtöltést, eredményes munkát, az élmények „Javát" kívánja

A szerző

1

A Java világa

Körülbelül két évtized telt el azóta, hogy egy kis mérnökcsoport megálmodta a „számítástechnika egy új hullámát", és nekilátott a megvalósításnak. Mára azt mondhatjuk, hogy álmuk több mint valóra vált. Megszületett a Java [1]. Előbb a programozási nyelv, majd a különböző platformok, keretrendszerek, technológiák. Valóban forradalmasította a számítástechnikát. Majdnem mindenütt jelen van körülöttünk, az egyszerű mobiltelefonoktól kezdve a bonyolult vállalati rendszerekig. Weboldalán azt írják, hogy megváltoztatta a világunkat és elvárásainkat. Igazuk van. De vajon minek köszönhető ez a sikertörténet? Tekintsünk be a Java világába…

1.1 Bevezetés a Java világába

Milyen programozási nyelv is a Java? Hol van a helye a programozási nyelvek világában? Melyek voltak történetének fontosabb állomásai, és mi vezetett ahhoz, hogy napjaink egyik vezető programozási nyelvévé váljon? Mi van a színfalak mögött? Ezekre a kérdésekre keressük a választ ebben az alfejezetben.

1.1.1 A Java helye a programozási nyelvek világában

A programozási nyelveket többféleképpen is kategorizálhatjuk. Az alábbiakban megpróbáljuk felsorolni a fontosabb osztályozási lehetőségeket, ezeken belül a teljesség igénye nélkül a fontosabb kategóriákat, és megpróbáljuk elhelyezni a Java programozási nyelvet ebben a rendszerben.

Az absztrakciós szint szempontjából beszélhetünk alacsonyszintű, középszintű és magas szintű programozási nyelvekről. A jelzők nem a nyelv hatékonyságára utalnak, hanem a gépi kódtól való elvonatkoztatás mértékére, így az alacsonyszintű programozási nyelveket hardware-közelinek is nevezik. Az alacsonyszintű kategóriába sorolhatnánk a gépi kódon kívül az assembly nyelveket, a közép szintre helyezhetnénk például a C-t, míg **a Java** a **magas szintű** kategóriába esne. A határok itt nem annyira élesek, ezt a fajta kategorizálást inkább viszonyítási alapként szokták használni (például a C alacsonyabb szintű, mint a Java). Az alacsony szintű nyelvek direkt hardware hozzáférési lehetőségeket biztosítanak, így bizonyos hardware-közeli problémák megoldásában (automatizálási feladatok, biztonsági megoldások stb.) hatékonyabbak. Ez nem jelenti azt, hogy az ilyen jellegű feladatok esetében teljesen el kell felejtenünk a Javat. Bizonyos mechanizmusokon, interfészeken keresztül lehetőségünk van más programozási nyelvekben megírt kódrészleteket integrálni Java programjainkba, és ez egy megoldás lehet a fentebb említett hátrány kompenzálására.

A végrehajtás szempontjából beszélhetünk értelmezett (interpretált), fordított és bájtkód (*bytecode*) alapú nyelvekről. Az első esetben a kód futtatásához egy futtató környezet (értelmező) szükséges, amely utasításonként értelmezi a kódot (pl. a régi BASIC nyelvek esetében). A fordított nyelvek esetében a forráskódból egy fordító (*compiler*) gépi kódot generál, amely már közvetlenül végrehajtható az operációs rendszer által (pl. Pascal vagy C/C++ esetében). A gépi kód mindig egy adott operációs rendszer által futtatható, platformfüggő, így natív kódnak (*native code*) is nevezzük. A sebesség szempontjából előnyt jelent a fordítás, mivel az interpretált nyelvek esetében a többszöri végrehajtás mindig a kód újraértelmezésével jár, míg az egyszer lefordított kódot bármikor újrafuttathatjuk.

A **Java** a **bájtkód alapú** nyelvek kategóriájába tartozik. Ennek a kategóriának az esetében először egy fordítási fázis során egy fordító a forráskódot egy átmeneti bájtkóddá alakítja. A bájtkód alapú nyelvek legnagyobb előnye, hogy, ellentétben a gépi kóddal, a bájtkód platformfüggetlen, így a programok könnyen hordozhatóak különböző architektúrák és operációs rendszerek között. A bájtkódba történő fordítás kisebb kódméretet, hatékonyabb szintaktikai ellenőrzést, és az értelmezett nyelvekhez viszonyítva jelentős sebességnövekedést eredményez. A fordítási fázist követheti egy értelmezési/végrehajtási fázis. Általában a bájtkód egy virtuális géphez (VM – *virtual machine*) továbbítódik, amely egy számítógép szoftver-implementációjának tekinthető, és, mint ilyen, képes végrehajtani a bájtkód utasításait. Lehetőség van továbbá dinamikus fordításra, amikor egy JIT fordító (*just in time compiler*) a végrehajtás előtt gépi kódra fordítja a bájtkódot, vagy annak bizonyos részeit. Ez a megoldás többszöri futtatás esetén jelentős sebességnövekedést eredményez, és ezen kívül kódoptimalizálásra is lehetőséget ad: a fordításnál figyelembe veheti az adott hardware tulajdonságait, valamint futtatási statisztikák alapján újrafordíthat bizonyos kódrészeket. Érdemes megemlíteni továbbá az AOT fordítók (*ahead of time compiler*) alkalmazásának lehetőségét is. Ezek a fordítók a natív fordítókhoz hasonlóan az átmeneti bájtkódot gépi kódba fordítják. Természetesen ez a platformfüggetlenség kárára mehet, de néhány speciális esetben haszonnal is járhat. Fontos kiemelni azt is, hogy a JIT vagy AOT fordítók alkalmazásától függetlenül maga a bájtkód platformfüggetlen és hordozható marad (pl. egy másik rendszeren a megfelelő AOT fordítóval egyszerűen újrafordíthatjuk a kódunkat).

A harmadik nagyon fontos kategorizálási lehetőség az alkalmazott programozási paradigma szerinti besorolás. Itt beszélhetünk szekvenciális programozási nyelvekről (ahol az utasítások sorról sorra, egymást követve kerülnek végrehajtásra, például a régi BASIC nyelvek esetében), procedurális programozási nyelvekről (ahol az utasításokat eljárásokba és függvényekbe csoportosíthatjuk, mint például a PASCAL vagy C esetében), funkcionális programozási nyelvekről (amelyek az alprogramokat mint matematikai függvényeket definiálják, mint például a Lisp), logikai programozási nyelvekről (logikai műveleteken és döntéseken alapulnak, mint például a Prolog), adatorientált programozási nyelvek (amelyek relációs adatokkal kapcsolatos műveletek hatékony végrehajtását teszik lehetővé, mint például az SQL), objektumorientált programozásról (mikor az adatok végrehajtható kódot is tartalmazó objektumokban vannak definiálva, mint például a Smalltalk esetében), prototípus alapú programozásról, aspektusorientált programozásról és így tovább.

A **Java objektumorientált** programozási nyelv. Ellentétben a C++-al, amely ebből a szempontból hibrid jelleget mutat (dolgozhatunk osztályokkal, de lehetőségünk van a hagyományos eljárásokon és függvényeken alapuló procedurális programozás alkalmazására is, amely tulajdonság főként a C-vel való kompatibilitás megőrzésére való törekvésnek köszönhető),

a Java egy „tisztán" objektumorientált nyelv, ahol „nincsen élet" az objektumokon kívül. Az objektumorientált programozással kapcsolatos alapfogalmakra, illetve a Java objektumorientáltságához kapcsolódó tulajdonságokra a későbbiekben részletesen kitérünk.
Összegzésként elmondhatjuk tehát, hogy **a Java** egy **magas szintű, bájtkód alapú, objektumorientált** programozási nyelv. A továbbiakban egy kicsit járjuk körül születésének körülményeit és felnőtté válásának fontosabb állomásait.

1.1.2 Java életút, röviden

Miután az 1960-as években a Simula 67 programozási nyelv bevezette az objektumok fogalmát, az 1970-es években a XEROX PARC Smalltalk programozási nyelvével együtt megszületett az objektumorientált programozás fogalma, és 1983-ban napvilágot látott a C++, az objektumorientált programozási paradigma egyre szélesebb teret hódított, és nagyon gyorsan a vezető programozási modellé nőtte ki magát.

Az 1980-as években egyre erőteljesebben jelentkezett az igény arra, hogy a számítógép-hálózatok által biztosított előnyöket a mindennapi életben is kamatoztatni lehessen. Ennek a törekvésnek a hozadékaként indult el 1991-ben a Sun Microsystem Stealth Project nevű vállalkozása, amely később Green Project néven vált ismertté. A projekt a számítógépek alkalmazásainak lehetőségeit vizsgálta a mindennapokban használt elektronikai készülékek piacán. Gyakorlatilag olyan „okos" elektronikai készülékek megtervezését célozták, amelyek „hálózatba" köthetőek, így egy nagy kiterjedésű, heterogén, osztott rendszer részeiként kommunikálni tudnak egymással, és ez a rendszer központilag vezérelhető. A projektben résztvevők közül megemlíthetjük Bill Joy, James Gosling, Mike Sheridan és Patrick Naughton neveit (a Green Team tagjai). Közülük James Goslingnak [26] volt az a feladata, hogy egy megfelelő programozási nyelvet találjon a projekt kivitelezéséhez.

Gosling először a C++ nyelv kiterjesztésével próbálkozott, de nagyon hamar arra a következtetésre jutott, hogy nem ez a megfelelő irány, és elkezdte egy új, független programozási nyelv megalkotását, amelyet Oak-nak nevezett el az irodája előtti tölgyfáról. A későbbiekben kiderült, hogy már létezik egy ilyen nevű programozási nyelv, így a nyelvet átkeresztelték Java-ra. Az új név pontos eredetét egy kis homály fedi, lehetséges, hogy a helyi kávézó felé vezető úton ugrott be a fejlesztőcsapat valamelyik tagjának, de van olyan elképzelés is, miszerint csapattagok neveinek kezdőbetűiből állt össze a Java név.

A Green Project céljainak következményeként az új programozási nyelv több kritériumnak kellett megfeleljen. Mivel a projekt az elektronikai cikkek piacát célozta, különböző gyártókkal és készüléktípusokkal kellett számolni, a platformfüggetlenség alapkövetelménynek számított, ezért döntöttek egy bájtkód alapú nyelv mellett.

Fontos kritérium volt a megbízhatóság is. Ha a számítógépünkön „lefagy" egy program újraindítjuk azt, vagy legrosszabb esetben a számítógépet, „hozzá vagyunk szokva", amennyiben nem egy kritikus rendszerről van szó, nem fogjuk tragédiaként felfogni. Egy mindennapokban használt elektronikai készülék esetében, amikor a készülék tulajdonosa nem is feltétlenül gyakorlott felhasználó, már más a helyzet. Elfogadhatatlan lenne egy olyan készülék, amelyet minduntalan újra kell indítani (gondolhatunk itt egy egyszerű mobiltelefonra, vagy esetleg egy bankautomatára). A Java fejlesztői ezért megpróbálták kiszűrni a nyelvből a potenciális hibaforrásokat, beleértve ebbe a célba a programozók „tévedési" lehetőségeinek lecsökkentését is,

és a hibák kezelése fontos szempont volt. Ennek a törekvésnek köszönhető számos eltérés a C++-hoz képest: csak objektumokat használhatunk adatstruktúraként, a kivételkezelés nagyobb szerepet kap, a többszörös öröklés nem engedélyezett, a memóriakezelést az automatikus szemétgyűjtés (*garbage collection*) teszi hatékonyabbá, stb. Ezekről a későbbiekben részletesebben is szó esik.

A Green Project hálózati alkalmazások kifejlesztését célozta, ezért a biztonság is az alapvető kritériumok közé tartozott. Nagyrészt ennek köszönhető például, hogy a Java-ban nem használunk pointereket, így kizárt a lehetősége annak, hogy „kártékony" programok tiltott memóriarészekhez férjenek hozza. A Java biztonsági mechanizmusának az alapja a „homokverem" (*sandbox*) modell. Ennek megfelelően az alkalmazások a JVM által létrehozott elszigetelt környezetben futnak, így nincs lehetőség arra, hogy külső (hálózaton keresztül elérhető), nem megbízható programok közvetlenül, ellenőrzés nélkül férjenek hozza lokális erőforrásokhoz. Az ilyen programok, csak a *sandbox*-on belül biztosított korlátozott erőforrásokat használhatják. A mechanizmusnak köszönhetően különböző programokhoz különböző jogosultságok rendelhetőek.

Láthatjuk, hogy a Green Project alapcéljai nagyban befolyásolták a Java nyelv tulajdonságait, és ezek a tulajdonságok vezettek később a nyelv elsöprő sikeréhez. Ezen kívül, óriási a hozzájárulása a sikerhez a Világhálónak. Nagyon könnyen belátható, hogy a World Wide Web természetéből adódóan ugyanazokat a tulajdonságokat várja egy programozási nyelvtől, amelyeket a Java fejlesztői tűztek ki célul: platformfüggetlenség, megbízhatóság, biztonság. Rövid idővel azután, hogy 1994-ben Patrick Naughton közreműködésével megszületett a HotJava Browser (eredeti nevén WebRunner), a Netscape jelezte, hogy Java támogatást épít be saját böngészőjébe, és ezt később a legtöbb internetes böngésző fejlesztői is megtették.

1995-ben a Sun Microsystem megalapította a Sun JavaSoft részlegét, amely a Java-hoz kapcsolódó fejlesztésekért felelős, és 1996-ban kiadták a JDK (Java Development Kit) 1.0 verzióját. Ettől kezdve a Java egymás után aratta a sikereket. Nagyon sokan hamar belátták, hogy ez a projekt „sikerre van ítélve", és a Java egyre több támogatóra talált, egyre több fejlesztő és vállalat csatlakozott a Java alapú fejlesztésekhez.

Gyakorlatilag már a kezdetektől fogva többről volt szó, mint egy egyszerű programozási nyelvről, architektúrájának köszönhetően a Java tulajdonképpen egy teljes software fejlesztési platform, amelyre az elmúlt években nagyon sok technológia és rendszer épült.

1998-ban jelent meg a J2SE (Java 2 Platform, Standard Edition) 1.2, amely a JDK 1.2-es verziójának felel meg. Az új név bevezetésére azért volt szükség, hogy az alap platformot megkülönböztessék az időközben megjelent többi platformtól, a J2ME-től (Java 2 Platform, Micro Edition) és a J2EE-től (Java 2 Platform, Enterprise Edition).A J2ME a J2SE egyszerűsített változata, amely elsősorban korlátozott erőforrásokkal rendelkező eszközök, beágyazott rendszerek (például mobiltelefonok, PDA-k) programozására alkalmas. A J2EE a J2SE kiterjesztése, amely a bonyolultabb vállalati rendszerekkel, alkalmazás szerverekkel (*application server*) kapcsolatos fejlesztéseket, és a komponens alapú programozást célozza. Többek között olyan plusz csomagokat, API-kat tartalmaz, mint az osztott rendszerek fejlesztését támogató EJB (Enterprise Java Beans), a tranzakciók menedzsmentjét elősegítő JTA (Java Transaction API), a programok közötti üzenetküldést elősegítő JMS (Java Message Service), vagy a webes fejlesztéseknél alkalmazható JSF (Java Server Faces).

Több új verzió után 2004-ben a Tiger kódnevű projekt eredményeként adták ki a J2SE 5.0-t (az eredeti számozásban ez 1.5-nek felelne meg), amely több jelentős változtatást, újítást hozott magával, melyek közül néhányra a későbbiekben részletesebben is kitérünk. Ekkora (a hivatalos Java weboldal szerint) már több mint 500 millió személyi számítógépen jelen volt a Java technológia, a J2EE SDK már rég túl volt a kétmillió letöltésen, és egyes statisztikák szerint a professzionális fejlesztők 75%-a a Java-t tekintette elsődleges programozási nyelvének. 2005-re már több mint 2.5 milliárd elektronikus eszközön volt jelen a Java, és körülbelül 4.5 millió fejlesztő használt Java technológiát.

A Java platform fejlesztésével párhuzamosan óriási iramban követték egymást a Java alapú technológiákkal, keretrendszerekkel (*framework*) és integrált fejlesztési környezetekkel (IDE – Integrated Development Environment) kapcsolatos fejlesztések. Érdekességként megjegyezhetjük itt, hogy a Microsoft is nagyon gyorsan részt szeretett volna kapni a Java piacából, és kiadta a saját Java verzióját, a J++ -t, amely Microsoft specifikus volt. A programok a saját fejlesztésű MSJVM virtuális gépen futottak, és a Microsoft saját környezetet is kifejlesztett a nyelvhez, a Visual J++ -t, amely részévé vált a Visual Studionak. A J++ nem felelt meg teljes mértékben a Sun standard specifikációinak, és olyan Windows specifikus kiterjesztéseket tartalmazott, amelyek sértették a Java platformfüggetlenségét. Ebből viták és perek származtak, amelyek végül egy kiegyezéshez vezettek, amelynek értelmében a Microsoft leállította a J++ fejlesztését. Bár a Visual J++ utoljára a Visual Studio 6.0 –nak volt a része, a J++ némiképp tovább él a Microsoft későbbiekben kiadott J# nyelvében, amely szintaxisában nagyon hasonló a Java-hoz és a .NET keretrendszer részét képezi.

Nem a Microsoft volt az egyedüli cég, amely korán felismerte a Java lehetőségeit. Mint ahogyan azt már említettük, szinte mindegyik nagyvállalat nagyon hamar megtette a szükséges lépéseket azért, hogy hozzájárulhasson a Java folyamathoz. Egymást követték például a különböző fejlesztői környezetek: Symantec Visual Cafe, Borland JBuilder, Oracle JDeveloper, IntelliJ IDEA, Xinox JCreator, stb. Jelen pillanatban talán a két legelterjedtebb környezet a NetBeans [20] és az Eclipse IDE [19], mindkettő nyílt forráskódú (*open source*) fejlesztés.

A NetBeans egy teljes egészében Java-ban megírt IDE, fejlesztése 1997-ben indult egy egyetemi projektként, majd 1999-ben a Sun megvásárolta, és a következő évben nyílt forráskódúvá tette. Ettől kezdve a NetBeans körül kialakult egy fejlesztői közösség, és a környezet folyamatos fejlődésnek indult. A 6.0-ás verziójától kezdve már több mint IDE, az Eclipse-hez hasonlóan *plug-in* (beépülő modul) szerkezetének köszönhetően egy teljes keretrendszernek tekinthető, amely könnyebbé teszi komplex, összetett felhasználói felülettel rendelkező applikációk készítését.

Az Eclipse szintén Java alapú, az IBM fejlesztéseként indult 1998-ban. 2001-ben alakult egy több nagyvállalatból álló Eclipse fejlesztéseket támogató konzorcium, és az Eclipse ettől kezdve nyílt forráskódú projekt lett. 2004-ben megalakult egy Eclipse fejlesztésekért felelős non-profit szervezet, az Eclipse Foundation, amelynek mára több száz tagja van, köztük több nagyvállalat. A projekt körül egy óriási és nagyon aktív fejlesztői közösség alakult ki, amely máig biztosítja a gyors fejlesztést, a minőséget, a könnyű információszerzést és a fejlesztők támogatását. Mára már egyre nehezebb választ adni arra a kérdésre, hogy „mi az Eclipse?". Egyben több óriásprojekt gyűjtőneve, egy közösség, egy szervezet, egy keretrendszer. Az Eclipse lelke az Equinox, amely az OSGi (Open Service Gateway Initiative) [21] standard *framework* specifikációjának egy implementációja.

Plug-in szerkezetének köszönhetően az Eclipse lehetőséget ad komplex projektek, általános célú RCP (*Rich Client Platform*) alkalmazások könnyű és hatékony fejlesztésére. Itt még talán érdemes megjegyezni, hogy az Eclipse egy saját eszköztárat is biztosít GUI fejlesztésekhez, az SWT-t, amely hasonlóan a közismert SWING-hez általánosan alkalmazható. Az SWT-re épülő magasabb szintű GUI komponenseket tartalmazó JFace eszköztár elősegíti komplex grafikus felszínnel rendelkező alkalmazások készítését. Az Eclipse JDT (Java Development Tools) és a Java IDE csak egy kicsiny része az Eclipse projektek teljes halmazának, ugyanakkor egy nagyon fontos része is. Az összes funkcionalitást biztosítja, amelyet egy modern fejlesztői környezettől elvárhatunk (kódkiegészítés, *refactoring*, *debugging*, stb.), és a beépülő moduloknak köszönhetően könnyen kiterjeszthető, így a fejlesztéseinkhez használt eszközöket egy rendszeren belül alkalmazhatjuk.

A Java technológiák, keretrendszerek és környezetek mellett érdemes talán megemlíteni a „rokonokat". A Java robbanásszerű fejlődésével párhuzamosan megjelentek, és fejlődésnek indultak olyan programozási nyelvek, amelyek azonos filozófián alapulnak, a Java alapokra épülnek, kiegészítve a lehetőségek tárházát.Ezeken kívül megjelentek ismert programozási nyelvek „Java változatai". Közös jellemző, hogy ezeknek a nyelveknek az esetében a forráskód JVM kompatibilis Java bájtkódba fordítható, a Java virtuális gépen futtatható. Az ilyen nyelvekben megírt programok kiválóan együttműködnek a Java programokkal, így talán inkább tekinthetőek kiegészítőnek, mint alternatívának. Kiemelhető a Groovy nyelv [27], amely néhány hasznos kiegészítést vezet be, és használata néhány esetben kompaktabb, elegánsabb kódot eredményezhet. Megemlíthető a JRuby [28], amely egy Java-ban megírt Ruby értelmező. A JRuby programok teljes mértékben együttműködnek a Java programokkal: Java osztályokat használhatunk fel a JRuby kódon belül, és fordítva, gyakorlatilag az értelmező beépíthető Java alkalmazásokba. A JRuby esetében is lehetőségünk van a kódot JVM kompatibilis bájtkódra fordítani. Hasonló Java megvalósítás létezik a Python programozási nyelvhez is. A Jython [29] nyelvben megírt kód szintén Java bájtkódra fordítható.

Az eddig kidolgozott programozási nyelvek száma meglepően nagy (a [22] weboldal az egyes nyelvek különböző változatait is beleszámítva jelenleg több mint 1300 nyelvet tart nyilván). Ezek közül csak nagyon kevesen futottak be a Java-hoz hasonló sikertörténetet. Az, hogy hogyan lesz egy nyelv sikeres, bonyolult folyamat, amelyben a nyelv tulajdonságain kívül még nagyon sok más tényező is szerepet játszik, mint például az aktuális piac, vagy a támogató közösség, és így tovább. A Java teljesítette a hozzáfűzött reményeket (mára a beágyazott rendszerekkel kapcsolatos fejlesztésekben vezető helyen szerepel), és jócskán fölül is múlta azokat, hiszen új korszakot indított el a webes fejlesztések területén, és mára a legtöbb statisztika az első helyre sorolja a programozási nyelvek piacán. Nehéz egy teljes választ adni a Java sikerének miértjére, de természetesen, ahogyan azt már említettük, az alapelveknek köszönhető tulajdonságoknak és főként a platformfüggetlenségnek óriási a szerepe. A továbbiakban megpróbáljuk megvizsgálni azt, hogy hogyan is valósulhat meg ez a platformfüggetlenség.

1.1.3 A színfalak mögött

Ahogyan azt már említettük, a Java a bájtkód alapú nyelvek kategóriájába tartozik. Ez azt jelenti, hogy az átmeneti kódba lefordított forráskód egy virtuális géphez továbbítódik, amely képes azt értelmezni, és végrehajtani az utasításait.

Ennek megfelelően, ahhoz, hogy egy felhasználó Java programokat futtathasson, szüksége van egy futási környezetre. Ez a környezet a JRE (Java Runtime Environment), a Java Platform nagyon fontos része. A felhasználók ezt ingyenesen letölthetik a Sun weboldaláról [1], és a fejlesztő cégeknek lehetőségük van csatolni a programjaikhoz, telepítő csomagjaikhoz.

A JRE tartalmazza a Java virtuális gépet (JVM – Java Virtual Machine), a Java programok futtatásához szükséges indítót (*launcher*), és a futási időben dinamikusan betölthető osztályokat tartalmazó osztálykönyvtárakat.

A JVM egy nyílt specifikációjú, jól dokumentált rendszer, így több változata létezik, több cég implementált saját Java virtuális gépet a Sun specifikációi alapján. A Sun saját JVM implementációja jelenleg a HotSpot, amely a J2SE fontos része (léteznek más speciális Sun JVM implementációk is), és az előző verzióknál alkalmazott Classic VM –et váltotta fel.

Természetesen a modern JVM-ek esetében, már nem egyszerűen a bájtkód soronkénti értelmezéséről és végrehajtásáról van szó. Már a Classic VM is lehetőséget adott a JIT fordítók előnyeinek kihasználására. Nagyon leegyszerűsítve mondhatnánk, hogy a JIT fordítók hasonlóak a megszokott natív fordítókhoz, csak futási időben fordítanak és optimalizálnak. A gyakorlatban a Java esetében nem ilyen egyszerű a dolog, a Java nyelv tulajdonságai miatt a kód optimalizálása és gépi kódra fordítása számos problémát vet fel. Ezekre itt nem térünk ki részletesen, annyiban maradunk, hogy minden JIT fordító legjobb tudása szerint megpróbálja megoldani őket.

A HotSpot a fordítás szempontjából két lehetőséget is kínál számunkra: egy kliens és egy szerver típusú fordítót. A kliens fordító az általánosan használt kliens programokra, asztali *(desktop)* applikációkra koncentrál, így kevesebb optimalizálási lehetőséget biztosít, viszont a fordítási időt (újra megjegyzendő, hogy a JIT fordítók esetében a fordítás futási időben történik) lecsökkenti. Gyakorlatilag a bájtkód azonnali végrehajtásra kerül a VM értelmezője által (így a kliensnek a program indításakor nem kell várakoznia), és a futtatás során születik döntés arról, hogy melyik gyakran használt kódrészletek kerülnek lefordításra. Ennek köszönhető a Java programok futtatásakor megfigyelhető „bemelegedési" fázis. Ezzel szemben a szerver fordító szerver alkalmazásokra koncentrál, és több optimalizálási mechanizmust biztosít. A szerver programokra általában jellemző, hogy az egyszeri indítás után hosszú ideig futnak, így az elindulást megelőző fordítási és optimalizálási fázis miatti időveszteség elhanyagolható, a későbbiekben bőven „visszahozza az árát".

Alternatív lehetőségként újra megemlíthetjük, az AOT fordítókat, amelyeknek használata bizonyos speciális esetekben előnyökhöz vezethet. Egy ilyen eset lehet például, ha fontos a szerzői jogok védelme. A bájtkód sokkal könnyebben visszafejthető, mint a natív kód, az Interneten nagyon könnyű találni olyan programokat, amelyek képesek a Java bájtkódból visszanyerni a forráskódot. Elképzelhető olyan eset, amikor nem szeretnénk, ha a kliens megkapná a bájtkódunkat, és egy AOT fordító ilyenkor megoldást jelenthet a problémánkra. Az AOT fordítók alkalmazása ezen kívül megkönnyítheti felhasználóbarát telepítési csomagok készítését, és néhány esetben a futási sebesség növekedését is eredményezheti. A Sun nem adott ki saját AOT fordítót, de léteznek külső cégek által fejlesztett fordítók, például az Excelsior JET [23] vagy a GNU Compiler for Java. Az AOT fordítók alkalmazása némileg sérti a platformfüggetlenség alapelvét, és ezen kívül a JIT fordítókkal ellentétben kevés lehetőséget adnak az architektúrát is figyelembe vevő optimalizálásra, így használatuk csak indokolt esetben javasolt.

A klasszikus esetben tehát a Java programunk bájtkód formájában jut el a felhasználóhoz, *.class* kiterjesztésű állományokban, vagy csomagokban, amelyek lehetnek egyszerűen az osztályokat tartalmazó könyvtárak, vagy *.zip*, illetve *.jar* csomagolt állományok (a jar a Java saját csomagolója, amelyről a későbbiekben még szó esik).

Indításkor a JRE indító programja megkapja egy ilyen *.class* állomány nevét, a JVM elvégez bizonyos ellenőrzéseket a bájtkódon, és ha mindent rendben talál, amennyiben az illető osztály tartalmaz egy *main* metódust, végrehajtja azt. A *main* metódus lesz az elsőként végrehajtott metódus, hasonlóan a C *main* függvényéhez (fontos megjegyezni, hogy itt a *main* mindig egy osztályon belül van). Ahhoz, hogy egy egyszerű Java program futtatható legyen, kell tartalmaznia minimum egy olyan osztályt, amely rendelkezik egy ilyen *main* metódussal.

A legtöbb program esetében a program futtatásához szükség van külső osztályokra. Ezek lehetnek a Java Platform által biztosított standard, úgynevezett *bootstrap* osztályok (például a *java.lang*, vagy a *java.util* csomagok osztályai), kiterjesztések (opcionális csomagok), vagy a felhasználó saját osztályai, csomagjai. Ahhoz, hogy a JVM futtatni tudja programunkat, tudnia kell, hogy hol találja meg a szükséges osztályokat.

A *bootstrap* osztályok és kiterjesztések keresése automatikus (a JRE aktuális verziójának megfelelően), ezek az osztályok a JRE főkönyvtárának *lib* és *lib\ext* alkönyvtáraiban találhatóak (a legtöbb fontos, központi osztály a *lib\rt.jar* csomagban található). A saját osztályok és csomagok esetében meg kell határozni az elérési útvonalakat. Ezt az operációs rendszer CLASSPATH környezeti változójának beállításával tehetjük meg (a különböző operációs rendszereknek megfelelő beállítási módszerekkel), vagy a *launcher* megfelelő opciójának segítségével.

Például Windows esetén:

```
set classpath = utvonal1; utvonal2
```

vagy

```
java –classpath utvonal1; utvonal2 osztalynev
```

ahol *java* az indítóprogram (a JRE *bin* alkönyvtárában található *java.exe*), az *utvonal1* és *utvonal2* a saját osztályok elérési útvonala, az *osztalynev* a *main* metódust tartalmazó osztály neve. Amennyiben a második konstrukciót alkalmazzuk a CLASSPATH környezeti változó nem lesz figyelembe véve. Név nélküli csomagok esetében az útvonal utolsó eleme az osztályokat tartalmazó könyvtár neve, névvel rendelkező csomagok esetében a csomagnak megfelelő gyökérkönyvtárat tartalmazó könyvtár neve, csomagolt állományok esetében az állomány neve. Az osztályok keresése a következő sorrendben történik: a rendszer először a *bootstrap* osztályok között, majd a kiterjesztések között keres, és utoljára a saját osztályok kerülnek sorra.

Láthatjuk, hogy egy felhasználó számára a JRE biztosítja a Java programok futtatásának lehetőségét. Egy fejlesztőnek ennyi azonban nem elégséges. Ahhoz, hogy Java programokat írjunk, szükségünk van még egy fejlesztői csomagra, amely esetünkben a JDK (JavaDevelopment Kit.)

A JDK tartalmazza a JRE-et is, de ezen kívül még számos, a fejlesztők számára hasznos eszközt és csomagot biztosít: fejlesztői eszközöket (fordító, *debugger*, dokumentáció generátor, stb.), *demo* applikációkat és példakódokat, a Java Platform fontosabb osztályainak forráskódját, stb.

Néhány fontosabb program a JDK főkönyvtárának fejlesztési eszközöket tartalmazó *bin* alkönyvtárából:

- *javac*: a fordító, amely forráskódunkat bájtkódba fordítja;
- *jdb*: a debugger;
- *javadoc*: a dokumentáció generátor, amely a forráskód megfelelő formátumú kommentárjai alapján képes a standard formátumú, *html* alapú dokumentáció generálására;
- *appletviewer*: segítségével webes böngésző nélkül futtathatunk appletteket;
- *jar*: a Java saját csomagoló programja;
- más eszközök, például a JVM statisztikáinak monitorizálásához, a memória monitorizálásához, natív kódrészletek integrálásához, osztott rendszerek fejlesztésének támogatásához, stb.

A JDK mellé szükségünk lehet még egy dokumentációra [2], a Java API specifikációjára, amely elérhető az Interneten, vagy ingyenesen letölthető a Sun oldaláról. Az API specifikációi is az előbb már említett standard *html* alapú formátum szerint vannak elkészítve: a kezdőoldalon láthatjuk a különböző csomagokat, azokon belül megtalálhatjuk az alcsomagokat, osztályokat és interfészeket. Amennyiben megtaláltuk az általunk keresett osztályt a specifikáció első részében megtalálhatjuk az osztály helyét a Java osztályok hierarchiájában, ezután találunk egy rövid általános leírást (és esetleg még kapcsolódó linkeket, az illető osztály használatára vonatkozó cikkek, példák felé), majd következnek az osztály attribútumai, konstruktorai és metódusai, amelyeket követi az örökölt attribútumok és metódusok listája, majd végül megtalálhatjuk az attribútumok, konstruktorok és metódusok részletesebb leírását.

A tanulás és dokumentálódás szempontjából hatalmas előnyt jelent az a tény is, hogy a legtöbb alapvető Java osztály forráskódja szabadon elérhető. A JDK forráskódja letölthető a Java hivatalos oldaláról, és nagyon hasznos kiegészítőként szolgálhat a dokumentáció, nyelvi specifikáció [3], és a Sun által kiadott [4], vagy más oktatási anyagok mellett.

Természetesen eszközeinket a megoldandó feladat függvényében még kiegészíthetjük további fejlesztési eszközökkel, csomagokkal. Miután rendelkezésünkre áll a teljes Java SDK (Software Development Kit), szükségünk lesz még egy fejlesztői környezetre. Miután ez is megvan, nem marad más hátra, mint az, hogy nekilássunk megírni első Java applikációnkat.

1.2 Az alapok alapjai

Ebben a részben a Java programozási nyelv néhány alapelemét tárgyaljuk [5]. Amint arról már a bevezetőben szó volt, ez a jegyzet feltételez az olvasó részéről némi jártasságot a C/C++ programozási nyelvek, illetve az objektumorientált programozás területén. A Java szintaxisa C alapú, az alapvető nyelvi elemek sok pontban megegyeznek, vagy nagyon hasonlóak, így ebben a részben nem térünk ki az olyan részletekre, mint például a feltételes, vagy ciklusvezérlő utasítások szintaxisa, az alapvető aritmetikai és logikai operátorok, vagy hasonló alapfogalmak. Természetesen a nyelv alaptulajdonságainak megfelelően van néhány eltérés a C és a Java között (például nincsenek pointer operátorok, stb.), ezek közül a fontosabbakat a megfelelő helyeken megemlítjük. Most viszont egy egyszerű kis programmal indítunk.

1.2.1 *Hello World* helyett

Tekintsük az alábbi forráskódot:

```
public class Example {
      public static void main(String[] args) {
            for (int i = 0; i < args.length; i++)
                  System.out.println(args[i]);
      }
}
```

A fenti kis program a parancssor argumentumait írja ki a konzolra.

Láthatjuk, hogy az *Example* nevű osztályunkon belül (mint azt már említettük Java-ban mindig egy-egy osztályon belül vagyunk) egyetlen metódusunk van, a *main*, amely a JVM által végrehajtott első (és esetünkben egyetlen) metódus.

Megfigyelhetjük, hogy ez a *main* metódus publikus, azaz bárhonnan hozzáférhetünk, statikus (osztályszintű), mivel végrehajtásához nincs szükség az osztály példányosítására, és nem térít vissza értéket (a hozzáférés- és típusmódosítókat a következő fejezetben részletesebben is tárgyaljuk). A paraméterlistában látható *String* tömb fogja tartalmazni a parancssor argumentumait.

A Java nyelvben minden tömbnek van egy *length* tulajdonsága, amely megadja az illető tömb hosszát, így egy egyszerű *for* ciklussal bejárhatjuk a parancssor argumentumait tartalmazó tömböt, és egyenként, egymás alá kiírjuk az argumentumokat a konzolra. A kiírást elvégző kódsor pontos értelme így első ránézésre homályos lehet. Egyelőre maradjunk annyiban, hogy „ezt így kell írni", és majd később az adatfolyamokkal (*streamek*) kapcsolatos fejezet áttanulmányozása után, ígérjük, kikristályosodik a kép.

Amennyiben ezt a kis példaprogramot elmentjük egy *Example.java* állományba, bájtkódba fordíthatjuk, például a JDK *javac* fordítójával:

```
javac Example.java
```

A fordítás után az *Example.java* forráskód állományunk mellett megjelenik egy *Example.class* állomány, amely a bájtkódot tartalmazza.

Amennyiben a "Hello" és "Java" paraméterekkel futtatjuk a programot, a

```
java Example Hello Java
```

parancs beírása után (a *java* a JRE indítóprogramja, az *Example* a *class* állomány neve, a "Hello" és a "Java" a parancssor argumentumai) a konzolon a

```
Hello
Java
```

kimenet jelenik meg, és elégedettek lehetünk, ugyanis megírtuk az első Java programunkat.

Megjegyzések:

- megfigyelhettük, hogy a forráskódot tartalmazó állomány neve megegyezett az osztály nevével. Egy forráskód állományon belül több osztályunk is lehet, de ezek közül csak egy lehet publikus (a belső osztályokat leszámítva, amelyekről a későbbiekben még szó esik), és az állomány nevének kötelező módon meg kell egyeznie a publikus osztály nevével.
- a Java forráskódok Unicode karakterekből állnak, tetszőleges karakter előfordulhat bennük (beleértve az ékezetes betűket és az Unicode *escape* szekvenciákat is), és a Java különbséget tesz a kis- és nagybetűk között (*case sensitive*). Az ezzel kapcsolatos elnevezési konvenciókat a későbbiekben tárgyaljuk.

1.2.2 Primitív adattípusok és burkoló osztályok

A Java esetében az adattípusokat két csoportba oszthatjuk: primitív típusok és referencia típusok. A primitív típusok a következők:

Név	Típus	Méret	Megjegyzések
byte	előjeles egész	8 bit	
short	előjeles egész	16 bit	
int	előjeles egész	32 bit	
long	előjeles egész	64 bit	
char	egész	16 bit	UTF 16 kódolás
float	lebegőpontos	32 bit	
double	lebegőpontos	64 bit	
boolean	logikai	8 bit	Értékkészlet: *true* és *false*

1.1 táblázat: primitív adattípusok Java-ban

Mivel a primitív adattípusok nem objektumok, ezért mindegyiknek megfelel egy úgynevezett burkoló osztály (*wrapper class*). Ezek a következők: *Byte, Short, Integer, Long, Character, Float, Double, Boolean*, mindegyik az azonos, vagy hasonló nevű, kis kezdőbetűvel írt primitív adattípusnak felel meg.

Számos olyan esettel találkozunk, amikor objektumokat kell használnunk primitív adattípusok helyett. A legkézenfekvőbb példa, a Java gyűjtemény keretrendszerének alkalmazása (erről a későbbiekben részletesebben is szólunk), de számos más esetben is találkozunk ezzel a helyzettel. Tételezzük fel például, hogy egész értékeket szeretnénk tárolni egy listában, például *Vector* –ban, de a *Vector* elemei csak objektumok lehetnek. Az ilyen esetekben lehetnek segítségünkre a burkoló osztályok.

A burkoló osztály egy példánya egy objektum, amely egy, a megfelelő primitív típussal rendelkező adatot „burkol”.

Példa:

```
int i1 = 5;
Integer i2 = new Integer(5);
Integer i3 = new Integer(i1);
```

Míg az *i1* változó típusa az *int* primitív adattípus, értéke 5, addig az *i2* objektum az *Integer* burkoló osztály egy példánya, amely egy 5 értékű *int*-et burkol. Így az előbbi példához visszatérve az *i1*-el ellentétben az *i2* már lehet például egy *Vector* eleme. Hasonlóan az *i3* objektum szintén az *Integer* burkoló osztály egy példánya, amely az *i1* változó értékét burkolja.

A burkoló osztályok számos hasznos metódust is biztosítanak számunkra, amelyek segítségével primitív típusokon végezhetünk el különböző (például átalakító vagy ellenőrző) műveleteket.

Példa:

```
String s = "10";
int i1 = Integer.parseInt(s);
Integer i2 = new Integer(i1);
float f = i2.floatValue();
char c = 'a';
if (Character.isLowerCase(c)) c = Character.toUpperCase(c);
```

A második sorban az *Integer* burkoló osztály *parseInt* osztályszintű metódusának segítségével egész értékbe konvertáltuk a paraméterként kapott *s* karakterláncot. Esetünkben ennek nem volt semmi akadálya, mivel a "10" karaktersor probléma nélkül átkonvertálható a 10 egész számba. Természetesen előfordulhatott volna, hogy az *s* tartalma nem numerikus (például "10" helyett "tíz"), ebben az esetben az átalakítás kivételt eredményezett volna (lásd későbbi példánkat).

A harmadik sorban létrehoztuk az *Integer* burkoló osztály egy példányát, az *i2*-t, amely az *i1* értékét (azaz a 10 –et) „burkolja". A negyedik sorban az *Integer* osztály által biztosított *floatValue()* metódussal *float* típusba konvertáltuk az *i2* tartalmát, és ettől kezdve az *f* változónk a 10.0 értéket fogja tárolni.

A hatodik sorban a *Character* burkoló osztály statikus metódusait használtuk arra, hogy a *c* karakterről eldöntsük, hogy kisbetű-e, és amennyiben kisbetű, nagybetűvé alakítsuk.

A J2SE 5.0 (Tiger) és a későbbi Java verziók biztosítják számunkra az automatikus be- és kidobozolási (*autoboxing* és *auto-unboxing*) mechanizmusokat, amelyek megkímélnek bennünket a primitív adattípusok és burkoló osztályok példányai közötti manuális átalakításokkal járó kellemetlenségektől.

Például a

```
int i;
Integer j;
i = 1;
j = new Integer(2);
i = j.intValue();
j = new Integer(i);
```

kódsort az *autoboxing* mechanizmus segítségével egyszerűbb formában írhatjuk:

```
int i;
Integer j;
i = 1;
j = 2;
i = j;
j = i;
```

Ugyanígy az *autoboxing* és auto-*unboxing* segítségével most már a következő kód is helyes:

```
int i1 = 10;
Vector v = new Vector();
v.add(i1);
```

Figyelem: bár az *autoboxing* és *auto-unboxing* mechanizmus kényelmesebb írásmódot biztosít, a megfelelő óvatossággal kezeljük. Például a burkoló osztályok példányainak esetében a „==" logikai operátor továbbra is a referenciák azonosságát fogja vizsgálni. Amennyiben érték szerinti összehasonlítást szeretnénk végezni az *equals()* metódus szolgáltathat megoldást. Ezt a megjegyzést a következő fejezetben részletesebben is kifejtjük. Azért említettük meg már itt is, mert amennyiben hibamentes programot szeretnénk, erre nagyon oda kell figyelnünk. Főként, hogy a kérdésben további félreértések forrása lehet például a J2SE 5.0 „*wrapper class caching*" mechanizmusa, amelyről a későbbiekben szintén szó lesz.

1.2.3 Referencia típusok

Míg a primitív adattípusok kezelése Java-ban érték szerint, direkt módon történik, addig az objektumok és tömbök (amelyek a Java nyelvben szintén objektumok) esetében ez nem lehetséges. Az objektumoknak nincs jól meghatározott, fix méretük, és általában több helyet foglalnak néhány bájtnál. Ezért azonosításuk és kezelésük minden esetben referenciák segítségével történik.

Általában véve a referencia egy fix méretű változó, amely egy másik változóról tartalmaz információt. A legalapvetőbb példa a referenciára a mutató *(pointer)*, amely egy olyan változó, amely egy másik változó memóriacímét tartalmazza.

A referencia fogalma már a C++-ból is ismerős lehet, de a Java esetében a referencia fogalma közelebb áll a C++ *pointer* fogalmához. A C++ referenciák tulajdonképpen egy alternatív névnek, „*alias*"-nak tekinthetőek. Deklarálásukkor meg kell határoznunk a referált objektumot is (kötelező módon inicializálnunk kell őket), és a referencia a pointerrel ellentétben ettől kezdve „nem átirányítható", és *null* sem lehet. Használatuk a függvények paraméterlistáiban jelenthet előnyt, amikor a referencia szerinti átadás lehetővé teszi, hogy a referált objektumot módosíthassuk a függvényen belül.

Mint mondtuk, ezzel szemben a Java referencia fogalma inkább a *pointer* fogalmához áll közelebb, azzal az alapvető különbséggel, hogy a Java referenciái nem adják meg számunkra a pointerek esetében rendelkezésünkre álló szabadságot, semmilyen módon nem manipulálhatjuk a referenciákat, nem alkalmazhatjuk például a mutatók esetében alkalmazható aritmetikai operátorokat.

A primitív adattípusok és referenciák kezelése közötti alapvető különbségek az értékek másolásában és összehasonlításában keresendőek. Erről a későbbiekben részletesen is szólunk (lásd *clone()* és *equals()* metódusok leírása a második fejezetben). Egyelőre nézzünk meg néhány példát a referenciák alkalmazására:

```
String s1 = "Hello";
String s2 = s1;
String s3;
s3 = new String("Hello");
```

Az első sorban az *s1* referenciának és a "Hello" karakterlánc-literálnak foglalunk le helyet. A második sorban az *s2* referenciának foglalunk le helyet, és inicializáljuk az *s1* referenciával. Ez azt jelenti, hogy ettől kezdve az *s1* és *s2* referenciák ugyanarra az objektumra (esetünkben a "Hello" karakterlánc-literálra) fognak „mutatni". A harmadik sorban egyszerűen az *s3* referenciának foglalunk le helyet, de még nem inicializáljuk, az általa mutatott objektumnak csak a negyedik sorban foglaljuk le a helyet. A helyfoglalás a *new* kulcsszó segítségével történik, és ezelőtt a referencia implicit értéke *null*.

A memória felszabadítása Java-ban automatikus, egy háttérben futó *daemon* szál, a *garbage collector* valósítja meg, amelynek feladata a nem használt (rájuk mutató referenciával nem rendelkező) objektumok által foglalt memóriaterületek felszabadítása.

Példa:

```
String s1;
s1 = new String("Hello");
s1 = new String("Java");
s1 = null;
```

Az első sorban lefoglaljuk a helyet az *s1* referencia számára. A második sorban lefoglaljuk a helyet a "Hello" karakterláncot tartalmazó *String* típusú objektum számára. Az új objektumra az *s1* referencián keresztül tudunk hivatkozni. A harmadik sorban azonban egy újabb helyfoglalás történik a "Java" karakterláncot tartalmazó *String* objektum számára, és ettől kezdve az *s1* referencia erre az objektumra fog mutatni. Ebben a pillanatban már nincs a "Hello" karakterláncot tartalmazó objektumra mutató referenciánk, így az objektum által foglalt helyet a *garbage collector* felszabadíthatja. Hasonlóan a negyedik sorban az *s1* referenciát *null* értékre állítjuk (amely egyébként az implicit értéke is volt), így a "Java" karakterláncot tartalmazó objektum által foglalt memória is felszabadíthatóvá válik.

Megjegyzendő, hogy az „objektumok azonosítása Java-ban minden esetben referenciák által történik" állításnak nincs köze a változók referenciák által történő átadásához (*pass by reference*). Java-ban a paraméterátadás mindig érték általi (*pass by value*), ami azt jelenti, hogy a változó függvénynek történő átadásakor az illető változóból másolat készül. Természetesen, amennyiben egy objektumot adunk át, akkor tulajdonképpen a referencia érték szerinti átadásáról beszélünk, így egy metóduson belül a referencián keresztül történő változtatások a metódusból történő kilépés után is érvényben maradnak. De ebben az esetben is a referencia átadásakor az illető referenciáról egy másolat készül, és nem egy a „referenciára mutató referencia" kerül átadásra, így továbbra is érték szerinti paraméterátadásról beszélünk.

1.2.4 Tömbök

Bár első látásra a Java tömbök hasonlóaknak tűnhetnek a C++ tömbjeihez, tekintve a deklarációt, vagy az elemekre történő hivatkozást, ha közelebbről megvizsgáljuk a kérdést, láthatjuk, hogy ebből a szempontból jelentős különbségek vannak a két nyelv között. Míg a C++ esetében a tömb azonos típusú, szomszédos memóriaterületeken tárolt elemek szekvenciája, és a tömb neve az első elemre mutató *pointer*, addig a Java esetében a tömbök objektumok, az *Object* osztály példányai.

Java-ban a tömbök elemei azonos típusúak, a tömbök méretei mindig ismertek, a tömb-határok mindig ellenőrzésre kerülnek, és a határok túllépése kivételt eredményez.

A Java tömbök elemei lehetnek tömbök, így a többdimenziós tömbök tulajdonképpen tömbökből álló tömbök:

```
int t[][] = new int[2];
t[0] = new int[3];
t[1] = new int[5];
```

Az első sorban egy két referenciából álló tömbnek foglaltunk helyet, a második, illetve a harmadik sorban egy három, illetve egy négy egész elemet tartalmazó tömbnek. Így a *t* tömbünk tulajdonképpen két egydimenziós tömböt fog tartalmazni.

1.2.5 Kivételkezelés dióhéjban

Ahogyan azt már említettük a Java esetében a biztonság az alapkritériumok között szerepelt, és a hibák kiszűrése, a hibakezelés nagyon fontos szerepet kapott. Java-ban a hibák kezelése teljes egészében kivételkezelésen alapszik, ezért szükséges, hogy röviden már most beszéljünk erről a témáról. Egyelőre az érthetőség kedvéért nem megyünk bele részletekbe, a későbbiekben viszont a jobb megértéshez szükséges alapfogalmak tárgyalása után még visszatérünk a témához.

A kivételkezelés Java-ban *try-catch* konstrukciók segítségével valósítható meg:

```
try {
      // Kódrészlet, amely kivételt eredményezhet
} catch (Exception1 object1) {
      // az Exception1 kivétel kezelésének megfelelő kód
} catch (Exception2 object2) {
      // az Exception2 kivétel kezelésének megfelelő kód
} finally {
      // Ez a kódrészlet minden esetben végre lesz hajtva
}
```

A *try* után következő programrész tartalmazza azt a „problémás” kódot, amely valamilyen kivételeket eredményezhet. Fogalmazhatnánk úgy is, hogy általában egy adott „viselkedést” várunk el ettől a programrésztől, de tudjuk azt, hogy bizonyos speciális, kivételes esetekben ez a viselkedés megváltozhat.

A *try*-részt követően mindegyik *catch* ág egy-egy konkrét kivételtípusnak felel meg. Amennyiben nem lép fel kivétel, a *try*-on belüli kódrész „probléma nélkül" lefut. Amennyiben valamilyen kivételt kapunk, akkor a kód helyett a kivételnek megfelelő *catch* részen belüli kód kerül lefuttatásra. Végül a *finally* részen belüli kód mindenképpen lefut (függetlenül attól, hogy a „problémás" kódrészlet dobott-e kivételt vagy nem).

A kivételek objektumok, egy-egy kivételosztály példányai. Fontos megjegyezni, hogy egy kivétel felléptekor a kivétel típusának megfelelő első *catch* ág kerül csak lefuttatásra. Erre oda kell figyelnünk akkor, ha a várt kivételek egymás leszármazottjai. Tételezzük fel például, hogy a példánk esetében az *Exception1* az *Exception2* leszármazottja. Amennyiben külön szeretnénk őket lekezelni, fontos a *catch* ágak sorrendje: ha fordítva írtuk volna őket, akkor minden esetben csak az első (az *Exception2* –nek megfelelő) kódot lehetett volna végrehajtani. A második *catch* ág így elérhetetlen kódnak minősülne, és a fordító hibát jelezne. Természetesen, tudva azt, hogy minden kivételtípus valamilyen ágon a *Throwable* ősosztály, és a legtöbb az ebből származó *Exception* ősosztály leszármazottja, megtehetnénk azt is, hogy csak egy *catch* ágat használunk, amely bármilyen kivételnek megfelel. Az általános javaslat mégis az, hogy amennyiben pontosan tudjuk, hogy milyen kivételekkel lehet dolgunk, akkor külön kezeljük a különböző kivételtípusokat.

A fenti megjegyzések érthetőbbé válnak már a következő fejezet osztályokkal és öröklődéssel kapcsolatos részénél. A kivételkezeléssel kapcsolatos „haladóbb" kérdésekre a későbbiekben még visszatérünk, de most inkább az alapok jobb megértésének érdekében vegyünk egy konkrét példát.

Nézzük a következő feladatot: írjunk egy programot, amely a parancssor első argumentumát átalakítja egy *i* egész számmá. Amennyiben a program indításakor egyáltalán nem kap bemeneti paramétert, akkor a program az *i* értékeként adjon meg 10-et. Amennyiben az első bemeneti paraméter nem numerikus, így nem történhet meg az átalakítás, akkor az *i* értéke legyen 20. Végül növeljük eggyel az *i* értékét és írjuk ki a konzolra.

```
public class ExceptionExample {
      public static void main(String[] args) {
            int i;
            try {
                  i = Integer.parseInt(args[0]);
            } catch (ArrayIndexOutOfBoundsException e1) {
                  i = 10;
            } catch (NumberFormatException e2) {
                  i = 20;
            } finally {
                  i++;
            }
            System.out.println(i);
      }
}
```

A *try* programblokkon belül az *Integer* osztály statikus metódusának alkalmazásával a parancssor argumentumait tartalmazó tömb első elemét egész számmá konvertáljuk. Megtörténhet, hogy a program nem kap bemeneti paramétert, ebben az esetben a tömb első

elemére, az *args[0]*-ra, történő hivatkozás által átlépjük a tömbindexek határait, így egy *ArrayIndexOutOfBoundsException* típusú kivételt kapunk, amelyet az első *catch* ágon belül kezelünk le, az *i* értékét ebben az esetben 10-re állítva. Kivételt kapunk akkor is, ha a tömb első eleme, bár létezik, nem numerikus, így az átalakítás nem működhet, és *NumberFormatException* típusú kivételhez vezet. Ezt a lehetőséget a második *catch* ágon belül kezeljük, az *i* értékét 20-ra állítva ebben az esetben. Végül a *finally* ágon belül mindenképpen növeljük eggyel az *i* értékét és a *try-catch* után kiírjuk a kapott értéket a konzolra. Megjegyzendő, hogy az első példánk, a „Hello Java" esetében, ahol szintén a parancssor argumentumait tartalmazó tömböt jártuk be, azért nem volt szükség a tömbindex határaira vonatkozó kivétel kezelésére, mert a *for* ciklus feltételében már megadtuk, hogy a bejárás biztosan csak a tömb méretéig történjen. Amennyiben nincs bemeneti paraméter, az *args.length* 0, így nem lépünk be a ciklusba, nincsen közvetlen hivatkozás a 0 méretű tömb valamelyik elemére.

A *try-catch* konstrukció *finally* részével kapcsolatban felmerülhet valakiben a kérdés, hogy miért van rá szükség, miért nem írhatjuk a kódot egyszerűen a *try-catch* után. A válasz: nem érnénk el ugyanazt az eredményt. Hangsúlyozzuk ki újra, hogy a *finally* kódrész *mindenképpen* végrehajtódik. Például akkor is, ha a *try* blokkon belül olyan kivételt kapunk, amelyet nem kezelünk a *catch* ágakon belül. Vagy elképzelhető olyan eset, amikor a *try-catch* konstrukción belül kilépünk a metódusból (*return*), és a *try-catch* utáni, metóduson belüli kód már nem futhat le. A *finally* kódrész ezekben az esetekben is végre lesz hajtva.

1.3 Javasolt gyakorlatok

1. A Java fejlesztői csomag (JDK) letöltése és telepítése után töltsük le a megfelelő dokumentációt, és első feladataink megoldásával párhuzamosan tanulmányozzuk át annak szerkezetét, hogy a későbbiekben könnyen használhassuk. Töltsünk le, és (ha szükséges) telepítsünk két-három fejlesztői környezetet (pl. Eclipse, NetBeans). Az első feladataink megoldása során próbáljuk ki, és próbáljuk meg összehasonlítani ezeket, kiválasztva a számunkra legmegfelelőbbet a munkánk hatékonyabbá tételének szempontjából (még ezelőtt, egy-két feladat esetében próbáljuk ki a konzolból történő fordítás és futtatás lehetőségét is, elvégezve a megfelelő konfigurációs beállításokat).
2. Amennyiben szükséges, próbáljuk meg felfrissíteni az objektumorientált programozással és az általunk ismert objektumorientált programozási nyelvekkel (pl. C++) kapcsolatos ismereteinket. További javaslat, amely a jegyzet következő részeire is érvényes, hogy próbáljuk ki (futassuk le) a fejezeteken belül megadott példaprogramokat.
3. Írjunk programot, amely kiszámolja a parancssor argumentumainak összegét, csak az egész számokat véve figyelembe (kivételkezelést alkalmazzunk). Írjuk meg a programnak egy másik változatát is, amelyik minden numerikus argumentumot figyelembe vesz. Ezt az utóbbit egészítsük ki olyan módon, hogy a program külön számolja ki a páratlan, illetve páros argumentumok összegeit.
4. Írjunk programot, amely kiírja a konzolra a parancssor argumentumait, a kisbetűket nagybetűkbe, a nagybetűket kisbetűkbe alakítva. Útmutatás: egy *String* objektum esetében egy adott karaktert a *charAt(index)* metódus segítségével kérdezhetünk le, az ellenőrzés és átalakítás a *Character* osztály statikus metódusainak segítségével történhet.

5. Hozzunk létre egy tömbökből álló tömböt, amelynek első sora 1, második sora 2, *n*-edik sora *n* elemet tartalmaz. Az elemek egész számok 1-től *n*(n+1)/2*-ig, ahol *n*, a sorok száma, a parancssor argumentuma. Amennyiben nem adunk meg argumentumot (vagy az nem egy numerikus érték), a sorok alapértelmezett száma legyen 10. Figyeljünk arra, hogy minden tömb esetében csak annyi elemnek foglaljunk helyet, amennyire valóban szükség van. A tömb elemeit írassuk ki a konzolra az alábbi példához hasonlóan:

```
1
2 3
4 5 6...
```

2

Az objektumorientált programozás „Java”

Leegyszerűsítve azt mondhatnánk, hogy az objektumorientált szemlélet lényege, hogy a világot egymással kapcsolatban álló, egymással kommunikáló objektumok halmazaként látja, ahol minden objektum valamilyen osztály példánya. Ebben a részben túllépünk egy kicsit ezen az egyszerű meghatározáson, röviden ismertetjük az objektumorientált tervezés és modellezés alapelveit, majd megmagyarázunk néhány objektumorientált programozással kapcsolatos alapfogalmat, és tárgyaljuk a Java nyelv jellegzetességeit.

2.1 Objektumorientált modellezés, tervezés, fejlesztés

A klasszikus meghatározás szerint a szoftverfejlesztés folyamatát öt különböző fázisra bonthatjuk: elemzés (analízis), tervezés (*design*), megvalósítás (implementálás), tesztelés és karbantartás. Természetesen a modern fejlesztési stratégiák ennél sokkal bonyolultabbak, de ha jobban megnézzük, általában az alapelvük a felosztás, valamint a különböző fejlesztési fázisok ciklikus ismétlése, és az egyes részeken belül valószínűleg ugyanezzel az öt alapfázissal (vagy azoknak egy részével) találkozunk. Ezt figyelembe véve, esetünkben egyelőre elégséges, ha megmaradunk az öt alapszakasznál.

Minden alapszakasz esetében rendelkezésünkre állnak speciális modellek és eszközök, amelyek a munkánkat segítik. Megjegyzendő, hogy nagyobb szoftverek esetében egy projekten több ember dolgozik, a különböző fejlesztési szakaszokon belül különböző csoportok tevékenykednek, egy projektvezető irányításával. Természetesen a szakaszok között vannak átfedések, a csoportok között elengedhetetlen a kommunikáció, és ezért is fontos egységes modellek, egy egységes „nyelv” alkalmazása, amely lehetővé teszi, hogy a különböző csoportok különböző felkészültségű tagjai megértsék egymást.

A szoftverek általában megrendelésre készülnek. Az elemzés során történnek meg az első egyeztetések a klienssel (a majdani felhasználóval), és ezeknek alapján az elemzők elkészítik a projekt specifikációját, meghatározzák a használati eseteket és elkészítik ezek leírásait, illetve a megfelelő *use-case* diagramokat, azonosítják a projekten belüli központi entitásokat, alaposztályokat, elkészítik az ezeknek megfelelő osztálydiagramokat (*domain analysis* és *domain class diagrams*), és azonosítják a rendszer főbb részeit.

A második szakaszban a tervezők az elemzés során elkészült dokumentumokból kiindulva elkészítik a projekt részletes tervét. Ekkor készülnek el az osztálydiagramok, a különböző használati eseteknek megfelelő szekvencia diagramok, az állapotátmenet diagramok, illetve a kollaborációs diagramok, amelyek alapján később megvalósulhat az implementáció.

Az első két szakaszban a CASE (Computer Aided System Engineering) modellek és eszközök lehetnek segítségünkre. A modellek szempontjából érdemes megemlítenünk az 1991-ben James Rumbaugh és társai által javasolt OMT-t (Object Modelling Technique), a szintén 1991-ben Grady Booch által javasolt módszert, illetve az 1992-ben Ivar Jacobson által bevezetett OOSE-t (Object Oriented System Engineering). A három szakember ezután a Rational Software Corporation céghez került, és egy csapatban (Three Amigos) dolgoztak tovább, hogy kifejlesszenek egy egységes modellező nyelvet, mely mindhárom modell előnyeit egybefogja. 1997-ben munkájuk eredményeként megszületett a Unified Modeling Language (UML), amely jelenleg a legelterjedtebb objektumorientált modellezési nyelv. Ezen, illetve a cég szoftverfejlesztési tapasztalatain alapszik a közismert Rational Unified Process (RUP) [35] nevet viselő iteratív szoftverfejlesztési stratégia. A modellező eszközök közül talán a legismertebb a Rational vállalat Rational Rose nevű szoftvere, de számos más modellezési környezetet fejlesztettek ki az elmúlt időszakban, közülük több nyílt forráskódú rendszer, mint például a StarUML [25].

A harmadik fejlesztési szakaszban, a tervezési fázisban elkészített dokumentumok és diagramok alapján a programozók implementálják a rendszer komponenseit, majd integrálják azokat a rendszerbe. Ebben a fázisban kapnak szerepet a programozási nyelvek, illetve a megfelelő integrált fejlesztési környezetek, és más, a fejlesztési folyamatot elősegítő eszközök.

A negyedik szakasz, a tesztelés, a verifikációs és validációs folyamat része, amely gyakorlatilag a szoftverfejlesztés minden szakaszához kapcsolódik. A tesztelés esetében beszélhetünk a komponensek teszteléséről (a legelterjedtebb módszer a *unit testing*), vagy integrációs tesztelésről (*integration testing*), fekete doboz és üvegdoboz tesztelésről, stb. A modellek és stratégiák mellett itt is megtalálhatjuk a megfelelő eszközöket: hibajelentési (*bug reporting*) rendszerek, *debugger*-ek, *unit testing* keretrendszerek (a Java esetében az egyik legközismertebb a JUnit [30]), automatikus ellenőrzéseket végrehajtó szoftverek, stb.

Miután a rendszert a fejlesztők stabilnak tekintik, következik annak telepítése, valós körülmények közötti tesztelése és az esetleges hibák kijavítása, módosítások beiktatása, tehát a folyamatos karbantartás.

Újra érdemes kihangsúlyozni, hogy a valóságban ezek a szakaszok általában nem egyszerűen, szekvenciálisan követik egymást, hanem a fejlesztő csapat által alkalmazott fejlesztési stratégia alapján ismétlődnek, és elkerülhetetlenek az átfedések. Azt már említettük, hogy a verifikációs és validációs tevékenység minden szakaszban jelen van, például már a *design* fázisban elkészített terveket is ellenőrzik. Ugyanígy, amennyiben egy komponens tesztelése során hibát találnak, a hibajavítás visszamutat az implementálási fázisra, a javítások más komponensekre is hatással lehetnek, és az is megtörténhet, hogy a terven is változtatásokat kell eszközölni. Nem beszélve arról, hogy szinte elképzelhetetlen, hogy az elemzési fázisban az elemzők mindent figyelembe vegyenek, ez minimum azt feltételezné, hogy a kliens is informatikus, és tökéletesen tisztában van a saját elvárásaival, illetve azok technológiai kivitelezhetőségével. A valóságban a fejlesztés során folyamatosan megjelennek új elvárások, vagy módosítási javaslatok. És persze még ott vannak a több kiadást megélő szoftverek, ahol a különböző verziók több fejlesztési ágra bontják a fejlesztési folyamatot.

Az egyszerű, klasszikus, szekvenciális fejlesztési stratégiát vízesés modellnek (*waterfall model*) is szokták nevezni [31]. A modell nagyon sok más módszernek az alapja, ezek közül megemlíthető például a nagyobb rendszerek esetében gyakran alkalmazott spirális fejlesztési modell (*spiral model*) [34]. Napjainkban nagyon elterjedtek az agilis szoftverfejlesztési

módszerek (*agile software development*) [33].Ezek az iteratív módszerek az inkrementális fejlesztést támogatják, a projektet rövid idő alatt kifejleszthető kis részekre bontják, és ez egy rugalmasabb fejlesztési folyamatot tesz lehetővé, a fejlesztők könnyebben alkalmazkodhatnak a változó követelményekhez. A közismertebb agilis módszerek közül megemlíthetőek például az *Extreme Programming* és a *Scrum* módszerek, a fejlesztési folyamat fontosabb jellemzői közül kiemelhető a teszt alapú megközelítés (*Test Driven Development*).

Nagyon sok fejlesztési stratégia létezik [32], és ennek a jegyzetnek nem célja részletesebben tárgyalni ezeket. A mi célunk a Java programozási nyelv alapjainak elsajátítása, tehát a könyvben tárgyalt témák többnyire csak a szoftverfejlesztés implementációs fázisához kötődnek. Ennek ellenére szükségesnek tartottam ezt a kis kitérőt. Miért? Mert az objektumorientált programozás esetében a programozónak mindig szem előtt kell tartania a tényt, hogy a fejlesztés több mint kódolás. Mindig gondoljuk át a feladatot, és készítsünk egy kis tervet, legalább egy osztálydiagramot, mielőtt „nekiugranánk” a forráskódnak. Pontosan ezért, a következő részben a modellezést tárgyaljuk, és a továbbiakban több példa és feladat osztálydiagramok segítségével lesz leírva.

2.2 Osztályok és objektumok

Mielőtt az osztályokról szólnánk ismételjük át az absztrakt adattípusok (*Abstract Data Types* – ADT) definícióját: egy absztrakt adattípus adatoknak és az ezekkel az adatokkal végrehajtható műveletek halmazának implementációtól független (erre vonatkozik az absztrakt jelző) specifikációja. Az objektumorientált programozás eszköztárát tekintve az absztrakt adattípusokat egy-egy interfész segítségével tudnánk definiálni (az interfészekről a későbbiekben bővebben is szó esik). A megvalósításra a legtöbb programozási nyelv biztosít számunkra lehetőségeket, így például a C++-ban struktúrák (*struct*), vagy osztályok (*class*) alkalmazásával tudtunk absztrakt adattípusokat implementálni. Az osztályok hasonlóak a struktúrákhoz. Az alapvető különbség, hogy az osztályok megfelelnek az objektumorientált programozási paradigma alapelveinek, segítségükkel megvalósítható az adatrejtés, az öröklődés, a polimorfizmus.

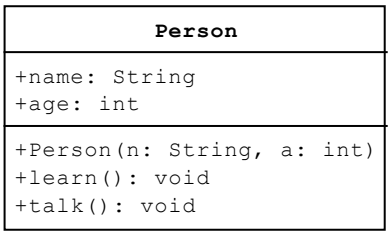
Egy osztály objektumok absztrakt tulajdonságait és viselkedési módját definiálja, a tulajdonságokat attribútumok (adattagok) írják le, a viselkedést metódusok (tagfüggvények).

Az adatrejtés megvalósításának érdekében az attribútumokat és metódusokat hozzáférés-módosítókkal láthatjuk el, amelyek a láthatóságot határozzák meg. Ezek Java-ban a következők:

- *public*: az illető attribútum vagy metódus mindenki számára látható;
- *protected*: az illető attribútum vagy metódus, a csomagon belül és az osztályból származtatott (csomagon kívüli) osztályokon belül látható;
- *private*: az illető attribútum vagy metódus csak az osztályon belül látható;
- csomag szintű (*package private*), vagy barát (*friend*): amennyiben nem használunk hozzáférés módosítót, Java-ban az illető attribútum vagy metódus a csomag szintjén lesz látható (és nem látható a csomagon kívüli származtatott osztályokból sem).

Tekintsük a következő példát: definiálunk egy személy osztályt a név és életkor attribútumokkal, egy konstruktorral, amely paraméterként kap egy karakterláncot és egy egész értéket, amelyek segítségével inicializálni tudja az előbb említett attribútumokat, valamint a tanul

és beszél metódusokkal (az egyszerűség kedvéért a példa esetében most csak publikus adattagokkal és tagfüggvényekkel dolgozunk, a későbbiekben kitérünk a láthatóság meghatározásával kapcsolatos szabályokra is):



2.1. ábra: a Person osztály attribútumai és metódusai

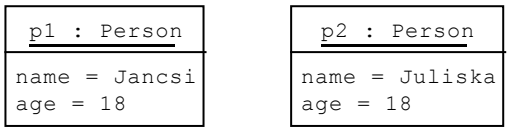
Megjegyzések:

- a UML osztálydiagramokon belül az osztályokat a fenti módon ábrázoljuk. A felső részbe írjuk az osztály nevét, a középső rész tartalmazza az attribútumokat, az alsó rész a metódusokat. Az attribútumok és metódusok neve előtt megjelenő jel az illető adattag vagy tagfüggvény láthatóságát mutatja (+ - *public*, - - *private*, # - *protected*, ~ - *package private/friend*), a kettőspont után következik az attribútum típusa, vagy a metódus által visszatérített érték típusa.
- a példa esetében az osztály, illetve az attribútumok és metódusok nevét angolul írtuk. Napjaink szoftver projektjei általában programozó csapatok együttműködésén alapulnak, és ez az együttműködés többször országhatárokon keresztül mutat. Törekednünk kell arra, hogy terveink és programjaink mindenki számára érthetőek legyenek, anyanyelvtől függetlenül. Jelenleg az angol nyelvet tekintik a számítástechnika nyelvének, javasolt tehát, hogy terveinkben és forráskódjainkban angol megnevezéseket, angol nyelvű kommentárokat használjunk. Informatikusokként azért is meg kell barátkoznunk az angol nyelv használatának gondolatával, mivel a legtöbb dokumentáció először ezen a nyelven válik hozzáférhetővé. Ezeknek a megállapításoknak a szellemében a további példák (tervek és forráskódok) esetében is az angol nyelvű elnevezések használatára fogunk törekedni.

Nézzük, hogyan történne ennek a példának az implementálása Java-ban:

```
public class Person {
      //az attributumok:
      public String name;
      public int age;
      //a konstruktor:
      public Person(String n, int a) {
            name = n;
            age = a;
      }
      //a metodusok (nem adunk meg konkret implementaciot):
      public void talk() {…}
      public void learn() {…}
}
```

Az objektumok az osztályok példányai, azonosításuk referenciák segítségével történik. A személy osztályból példányosítunk két objektumot:



2.2 ábra: a *Person* osztályból példányosított *p1* és *p2* objektumok

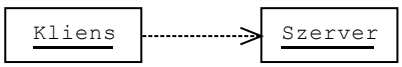
A létrehozott objektumok a *Person* osztály példányai, azonosításuk a *p1*, illetve a *p2* referenciák segítségével történik. Az attribútumok értékeket kapnak: a *p1* objektum esetében a név "Jancsi", az életkor 18, a *p2* objektum esetében a név "Juliska", az életkor szintén 18.

Az attribútumok aktuális értékei határozzák meg az objektumok állapotát. Két objektum lehet azonos állapotban, de ettől még különböző objektumokról beszélünk: különböző memóriatartományokat foglalnak, és különböző referenciák segítségével hivatkozunk rájuk (például tételezzük fel, hogy a *p2* objektum esetében szintén "Jancsi" értéket adtunk volna a *name* attribútumnak). Megjegyzendő viszont, hogy azonos állapotú objektumok azonos kérésekre azonos módon válaszolnak.

A példányosítás Java-ban:

```
Person p1 = new Person("Jancsi", 18);
Person p2 = new Person("Juliska", 18);
```

Az objektumok közötti kommunikáció kliens-szerver modell alapján valósul meg, az egyik objektum (a kliens) bizonyos műveletek végrehajtását kérheti egy másik objektumtól (a szervertől):



2.3 ábra: az objektumok kommunikációja kliens-szerver modell alapján történik

A kliens kéréseket intéz a szerverhez, aki válaszol azokra. A kérések tulajdonképpen üzenetek, a szerver publikus metódusainak meghívásai. Létezhetnek tipikusan kliens, vagy szerver objektumok, de ez általában nem jellemző: a program futásának bizonyos pillanataiban egy objektum viselkedhet kliensként, egy másik időpillanatban betölthet szerver szerepet.

A megvalósítás szempontjából fontos, hogy a kliensnek mindig ismernie kell a szervert, ami tulajdonképpen azt jelenti, hogy a kliens osztály tartalmaz egy szerver típusú attribútumot:

```
class Server {
…
}
class Client {
      Server s;
      …
}
```

Hangsúlyoztuk, hogy az objektumok azonosítása minden esetben referenciák segítségével történik. Az aktuális példány azonosítására Java-ban a ***this*** predefiniált referencia alkalmazható. Ezt nagyon gyakran használjuk olyan metódusokon vagy konstruktorokon belül, ahol a paraméterek nevei megegyeznek az attribútumok neveivel. Alkalmazása más helyzetekben is szükséges lehet, például egy adott konstruktoron belül a *this* kulcsszó segítségével meghívhatunk egy másik konstruktort.

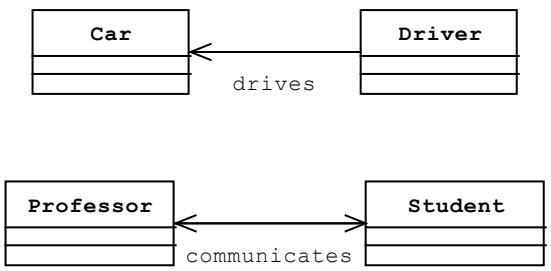
2.3 Osztályok közötti kapcsolatok

Az osztályok közötti kapcsolat lehet egyszerű ismeretségi viszony, vagy tartalmazási viszony.

Az ismeretségi viszonyt az irányítottsága szerint jellemezhetjük. Például egy sofőrnek „ismeretségi" viszonyban kell lennie az autójával, ahhoz, hogy vezethesse azt. Ebben az esetben a vezetés tulajdonképpen bizonyos metódusok meghívását jelentené, és ezekhez a metódushívásokhoz szükségünk van egy autó objektumra mutató referenciára. Ezzel szemben az autónak nem kell meghívnia a sofőr objektum metódusait, az autónak nem kell ismernie vezetőjét (nincsen szüksége személy objektumra mutató referenciára), így itt egy egyirányú ismeretségi viszonyról beszélhetünk.

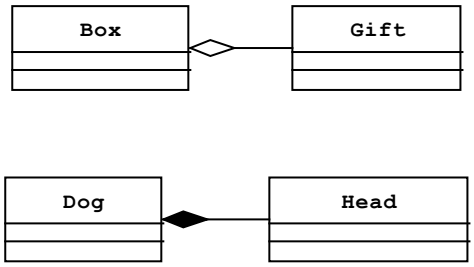
Amennyiben egy diák beszélget a tanárral, a kommunikáció kölcsönös üzenetküldéseken (metódushívásokon) keresztül valósulhat meg. Itt már egy kétirányú ismeretségi kapcsolatra van szükségünk: mindkét osztályon belül szükségünk van egy, a másik osztály egy példányára mutató referenciára, ahhoz, hogy megvalósítható legyen a kommunikáció.

A fentebb leírt ismeretségi viszonyra vonatkozó példákat az alábbi diagramokkal szemléltethetnénk:



2.4 ábra: osztályok közötti ismeretségi viszony szemléltetése

A tartalmazási viszony jellege lehet erős (*composition*) vagy gyenge (*aggregation*). Gyenge tartalmazási viszonyról beszélhetünk például a doboz és az ajándék között: a doboz létezik ajándék nélkül is. Erős tartalmazási viszonyról beszélhetünk például a kutya és valamilyen testrésze, például a feje közötti viszony esetében: a kutya nem létezhet a feje nélkül. Ezeket a példákat az alábbi diagramok segítségével szemléltethetnénk:

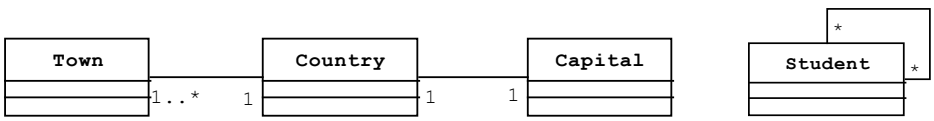


2.5 ábra: osztályok közötti tartalmazási viszony szemléltetése

Az osztályok közötti kapcsolatokat jellemezhetjük számosságuk szerint is, így beszélhetünk:

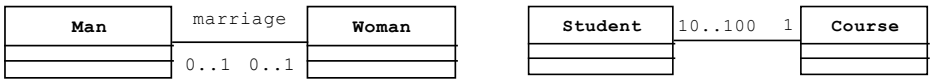
- egy az egyhez kapcsolatokról (*one to one association*), például egy ország és fővárosa között;
- egy a többhöz (*one to many*) kapcsolatokról, például egy ország és városai között;
- több a többhöz (*many to many*) kapcsolatokról, például diákok közötti kapcsolat.

Néhány példa:



2.6 ábra: az osztályok közötti kapcsolatok számosság szerinti jellemzése

Láthatjuk, hogy az osztálydiagramokon a számosságot a kapcsolatokat jelölő vonalakra (vagy nyilakra) írt számokkal jelöljük, a „több” jelölésének a * felel meg (amennyiben nincs felső határ). Amennyiben a kapcsolatokat jelölő szakaszokon nincsenek az irányítottságot mutató nyilak, kétirányú kapcsolatoknak tekintjük őket. Azt is megfigyelhetjük, hogy a diákok esetében reflexív kapcsolatról beszélünk. Ugyanakkor az első példák esetében mindig megszabtunk egy alsó határt, az 1-et, de ez nem feltétlenül kell így legyen. Ezek szerint beszélhetünk kötelező és opcionális kapcsolatokról. Kötelező kapcsolat például az ország és fővárosa közötti kapcsolat: minden országnak van egy fővárosa, és minden fővároshoz tartozik egy ország. Opcionális kapcsolatról beszélhetnénk például a férfi és nő között, amennyiben a kapcsolatot a házasság jelentené: a kapcsolat számosságának alsó határa 0, felső határa 1 lenne (legalábbis a keresztény kultúrkörben). Ugyanakkor a kapcsolatok számának az alsó határon kívül felső határt is megszabhatunk. Például a diákok és a kurzusok közötti kapcsolat esetében feltételezzük, hogy egy kurzust nem indítanak el 10 hallgatónál kevesebb érdeklődővel, de nem is vehet részt egy előadáson 100-nál több hallgató. Az említett példákat az alábbi diagramokkal szemléltethetjük:



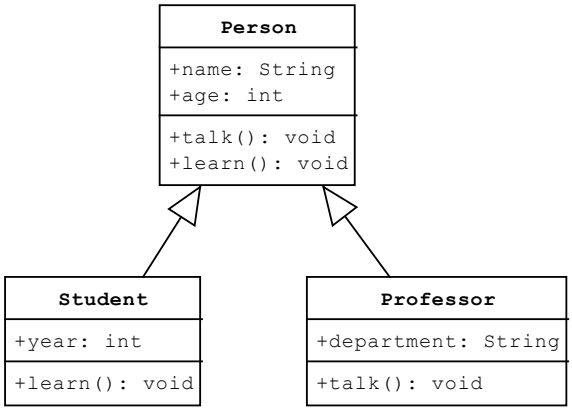
2.7 ábra: opcionális és kötelező jellegű kapcsolatok szemléltetése

2.4 Öröklődés

Az öröklődés (*inheritance*) az a folyamat, amelynek során egy osztályból további osztályokat származtatunk. A származtatott osztályok öröklik az alaposztály tulajdonságait (attribútumait és metódusait), és ezeken kívül további specifikus attribútumokat és metódusokat deklarálhatnak, valamint újradefiniálhatják az alaposztály bizonyos metódusait.

Ismételjük át röviden a következő fogalmak jelentését: metódusok túlterhelése (*method overloading*) és metódusok újradefiniálása (felülírása) (*method overriding*). Az első fogalomról akkor beszélünk, amikor egy osztályon belül több metódus jelenik meg azonos névvel, csak a paraméterek listája, és esetenkét a visszatérített érték típusa különbözik. A második fogalomról beszélünk akkor, ha a származtatott osztályok újradefiniálják az alaposztály metódusait: egy adott metódusnak megváltozik az implementációja a származtatott osztályon belül.

Egy egyszerű származtatási viszony bemutatására tekintsük az alábbi példát:



2.8 ábra: származtatási viszony szemléltetése

A személy (*Person*) osztályból származtattunk egy hallgató (*Student*) és egy tanár (*Professor*) osztályt. A hallgatók és tanárok öröklik a *Person* osztály tulajdonságait: ők is rendelkeznek névvel és életkorral, de ezen kívül a hallgatók esetében még rögzíteni szeretnénk az évfolyamot is, és a tanárok esetében a részleget, ahol tanítanak. Ezeknek az adatoknak a tárolására a származtatott osztályok új attribútumokat vezetnek be. Ugyanakkor egy diák esetében a tanulás új értelmet kap: egy diák esetében mást értünk tanulás alatt, mint általánosan egy személy esetében. Ugyanúgy, ha a tanár előadást tart, a beszéd fogalma is átminősül, ezért a *talk* metódust újradefiniáljuk a származtatott osztályban.

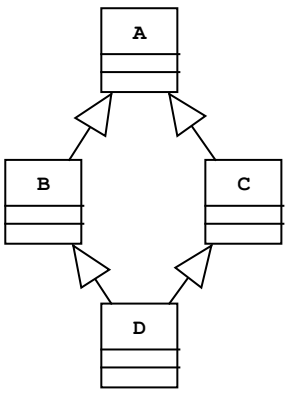
Amennyiben egy származtatott osztályból példányosítunk az alapértelmezett (paraméterrel nem rendelkező) konstruktor segítségével, először az alaposztály alapértelmezett konstruktora kerül meghívásra, és ezután következik a származtatott osztály konstruktora. A következő példánk azt szemlélteti, hogy hogyan valósítható meg a származtatás Java-ban, és hogyan alkalmazhatunk paraméterrel rendelkező konstruktorokat:

```
class A {
      A(String a) {
            System.out.println(
                  "Az A osztály konstruktora " + a);
      }
}
class B extends A {
      B(String b) {
            super(b);
            System.out.println(
                  "A B osztály konstruktora " + b);
      }
}
public class Example {
      public static void main(String[] args) {
            B b = new B("Hi");
      }
}
```

Létrehoztunk tehát egy *A* osztályt, amelynek a konstruktora egy *String*-et kap paraméterként, és kiírja azt a konzolra. A *B* osztályt az *A* osztályból származtatjuk (az *extends* kulcsszó segítségével), és konstruktorának első sorában a *super(b)* konstrukció segítségével hívjuk meg az alaposztály paraméteres konstruktorát, úgy, hogy paraméterként átadjuk neki a *b* karakterláncot. A *main* metóduson belül létrehozzuk a *B* osztálynak egy példányát a "Hi" karakterláncot adva meg a konstruktor paramétereként. Az eredmény:

```
Az A osztály konstruktora Hi
A B osztály konstruktora Hi
```

A C++ nyelvvel ellentétben a Java nem támogatja a többszörös öröklést. Ennek az oka a többszörös öröklődés esetében felmerülő gyémánt öröklődés által felvetett problémáknak a kiküszöbölése. Ismétlésként tekintsük az alábbi szerkezetet:



2.9 ábra: a gyémánt öröklődés problémájának szemléltetése

Az alapkérdés: ha az *A* osztály valamelyik metódusát mind a *B*, mind a *C* osztályokban újradefiniáljuk, a *D* osztály melyik verziót örökli?

A probléma megoldására a különböző programozási nyelvek különböző megközelítéseket javasolnak, de minden esetben a programozóra hárul a feladat, hogy odafigyeljen, és gondoskodjon a helyes megoldásról. Ezért, mint az ilyen esetek általában, a gyémántöröklés megjelenése potenciális hibaforrás lehet. A Java programozási nyelvnek az egyik alapelve pontosan a programozói hibalehetőségek minimalizálása volt, és ez a magyarázat arra, hogy a nyelv nem engedélyezi a többszörös öröklés alkalmazását. A későbbiekben látni fogjuk, hogy ez nem jelent számunkra lényeges megkötéseket, mivel azokat a feladatokat, amelyek esetében indokoltnak tűnhet a többszörös öröklés alkalmazása, megoldhatjuk interfészek segítségével.

2.5 Polimorfizmus

Az objektumorientált programozási terminológiában a polimorfizmus (többalakúság) kifejezést annak a jelenségnek a leírására használjuk, amikor egy *B* típusú objektum egy adott helyzetben *A* típusúként jelenik meg, és *A* típusú objektumként használjuk. Természetesen ez csak akkor lehetséges, ha a két osztály (*A* és *B*) között származtatási viszony áll fent.

Mivel a származtatott osztály örökli az alaposztály tulajdonságait, használható minden olyan helyzetben, ahol az ős használható. Az angol terminológiában ezt a helyzetet a „*B is A*" kifejezés érzékelteti (amennyiben *B* az *A* leszármazottja). Vigyázat: a kijelentés fordítottja már nem igaz. Egy egyszerű példa: a négyszög (*A*) osztályból származtatjuk a négyzet (*B*) osztályt. A négyzetről elmondható, hogy négyszög, de természetesen nem minden négyszög négyzet. A származtatott osztály az őshöz képest új tulajdonságokkal is rendelkezhet, így előállhatnak olyan helyzetek, amikor az ős nem helyettesítheti utódját. Egy négyzet (*B*) objektumot, viszont bármilyen helyzetben „kezelhetünk" négyszögként, így semmi akadálya, hogy egy négyszög típusú referencia egy négyzet objektumra mutasson.

A fentieknek megfelelően egy referencia esetében beszélhetünk statikus és dinamikus típusról, vagy kötésről. A statikus típus az, amely a deklarációban szerepel, a dinamikus típus pedig a referencia által aktuálisan beazonosított objektum típusa.

A kérdés: mi történik akkor, ha egy *A* típusúként deklarált referencia adott pillanatban egy *B* típusú objektumra mutat, és segítségével meghívunk egy metódust, melyet a *B* osztály újradefiniált, a metódusnak „melyik változata" fog érvényesülni? Természetesen, ha újradefiniáltuk a metódust, valószínűleg azt szeretnénk, hogy az új, a konkrét típusnak megfelelő implementáció érvényesüljön. De ez nem minden nyelvben történik automatikusan így. Gondoljunk például a C++ esetére, ahol hasonló esetben (*A* típusúként deklarált pointer *B* típusú objektumra mutat) az alaposztály metódusa érvényesül. A problémát itt a virtuális tagfüggvények segítségével oldhatjuk meg (ha az alaposztályban az érintett metódusokat virtuálisként deklaráljuk). A Java nyelvben erre nincs szükség.

A Java nyelvben a metódusok újradefiniálásának esetében mindig az objektum konkrét típusának megfelelő implementáció érvényesül (dinamikus kötés). Azt is mondhatnánk, hogy a Java-ban minden tagfüggvény virtuális.

A 2.2 alfejezetben említettük az aktuális példány azonosítására alkalmazható *this* referenciát. Hasonlóan az öröklődéssel és polimorfizmussal kapcsolatban meg kell említenünk a ***super*** referenciát, amely az alaposztály példányára mutat. Ha például egy metódust a származtatott osztály újradefiniál, de adott helyzetben mégis szükséges az alaposztály metódusának a meghívása, a *super* referencia lehet segítségünkre. Hasonlóan a *super* kulcsszó segítségével hívhatjuk meg az alaposztály konstruktorát a származtatott osztály konstruktorán belül.

2.6 Absztrakt osztályok

A származtatás mechanizmusának és a polimorfizmusnak egyik nagyon kézenfekvő felhasználási lehetősége egy rendszeren belül a közös alap létrehozása különböző, de azonos alaptulajdonságokkal is rendelkező objektumok részére. Ez lehetővé teszi, hogy azokban az esetekben, amikor csak a közös tulajdonságok relevánsak, azonos módon hivatkozzunk ezekre az objektumokra. Ezt egyszerűen megtehetjük akkor, ha az osztályok rendelkeznek egy közös ősel.

Előállhatnak olyan esetek is, amikor ennek az ősosztálynak nem lehetnek példányai, vagy a rendszer szempontjából értelmetlen lenne a példányosítás. Gondolhatunk példaként egy mértani alakzatokat megjelenítő felületre. Az alakzatok rendelkeznek közös tulajdonságokkal (mindegyiknek ismerhetjük például a pozícióját, színét, stb.), de mégsem lenne értelme annak, hogy egy általános alakzat objektumot hozzunk létre. A közös tulajdonságok nem lennének elegendőek ahhoz, hogy megjelenítsünk valamit a felületen. Mégis hasznos lenne, ha a különböző alakzatoknak megfelelő osztályaink rendelkeznének egy közös ősel, hogy bizonyos esetekben egységesen hivatkozhassunk rájuk.

Az ilyen és ehhez hasonló esetekben lehetnek segítségünkre az absztrakt metódusok és absztrakt osztályok. Java-ban az absztrakt metódusokat az *abstract* kulcsszóval jelöljük, és esetükben nem adunk meg konkrét implementációt. Ha egy osztályon belül van legalább egy ilyen absztrakt metódus, akkor absztrakt osztályról van szó, és ezt a tényt szintén az *abstract* kulcsszóval jelezzük (kötelező módon). A C++ programozók itt hasonlóságot fedezhetnek fel a tisztán virtuális tagfüggvények alkalmazhatóságával.

Az előbbiekben említett mértani alakzatokkal kapcsolatos példánk esetében gondolhatunk arra, hogy szeretnénk, ha minden konkrét alakzattípus rendelkezne bizonyos metódusokkal (megjelenítés, forgatás, újraméretezés stb.), de nem tudnánk egy-egy egységes implementációt adni ezekre a műveletekre. A megoldás, hogy az alaposztályban absztrakt metódusokként deklaráljuk ezeket.

Megjegyezhetjük, hogy az absztrakt osztályok lehetnek más absztrakt osztályok leszármazottjai. Az absztrakt metódusokat általában a származtatott osztályok implementálják. Ha ezt mégsem teszik, akkor az illető származtatott osztályt is absztraktnak kell deklarálni.

Újra hangsúlyozzuk ki azt, hogy az absztrakt osztályokból nem példányosíthatunk. Tulajdonképpen csak egy közös felületet, egy adott közös viselkedési módot határoznak meg a származtatott osztályok részére. Megjegyezhetjük, hogy ilyen szempontból hasonlóságot mutatnak az interfészekkel. Az alapvető különbséget az interfészeket tárgyaló következő részben írjuk le.

2.7 Interfészek

A számítástechnikában (és máshol is) általánosan akkor beszélünk interfészekről, amikor különböző rendszerek (hardware vagy software) közötti kommunikációról van szó. Egy adott rendszer interfésze írja le azt, hogy kívülről hogyan lehet hozzáférni az illető rendszerhez. Például, egy tévénéző és a készülék közötti „kommunikációban" a távirányítót tekinthetjük egy interfésznek. Az objektumorientált paradigmán belül egy osztály interfésze azt határozza meg, hogy más osztályok hogyan férhetnek hozzá az illető osztályhoz, vagy annak példányaihoz, milyen üzenetküldések jöhetnek szóba. Tehát, tulajdonképpen, az osztály interfészét a publikus adattagok és metódusok alkotják.

A Java programozási nyelv az interfész fogalmát egy további jelentéssel ruházza fel. A Java esetében az interfész egy típus deklaráció, amely egy bizonyos viselkedési módot határoz meg. Konstansokból és nem implementált metódusokból áll. A metódusokat az interfészt megvalósító („implementáló") osztályok fogják kötelező módon implementálni. Úgy is tekinthetünk az interfészekre, mint „szerződésekre", amelyeket az implementáló osztályoknak kötelező módon be kell tartaniuk. Ha egy osztály megvalósít egy adott interfészt, „vállalja azt", hogy az interfésznek megfelelően fog „viselkedni".

Az interfészek hasznosak lehetnek abban az esetben, ha hasonlóságokkal bíró osztályok részére egy közös alapot szeretnénk létrehozni, az absztrakt osztályoknál leírtakhoz hasonlóan. Az alapvető különbség, hogy az absztrakt osztályok esetében öröklődésről, az interfészek esetében megvalósításról beszélünk. Az interfészek alkalmazása nem vezet egy kötött osztályhierarchiához. Java-ban különösen fontos szerep jut az interfészeknek, mivel a nyelv nem támogatja a többszörös öröklést. Így, egy adott osztályt nem származtathatunk több különböző absztrakt alaposztályból, viszont az osztály több különböző interfészt is megvalósíthat (természetesen implementálva azok minden metódusát).

Az interfészek használatával kapcsolatosan tekinthetjük a következő egyszerű példát: az előbbiekben már említett mértani alakzatokkal kapcsolatos alkalmazásunk esetében biztosak akarunk lenni abban, hogy minden alakzat átméretezhető, azaz mindegyik biztosít számunkra egy átméretezést megvalósító metódust, amely paraméterként egy *Dimension* típusú objektumot kap, az alakzatot tartalmazó téglalap alakú felület méreteinek meghatározására. Létrehozunk egy *Resizable* interfészt, melyet majd a konkrét alakzatoknak megfelelő osztályok valósítanak meg.

```
public interface Resizable {
      public void resize(Dimension d);
}
```

Az interfészt megvalósító *Circle* osztály:

```
public class Circle implements Resizable {
      public void resize(Dimension d) {
            //a kör újraméretezését megvalósító kód
            ...
      }
}
```

Alkalmazás:

```
public static void main(String[] args) {
      Circle c = new Circle();
      c.resize(new Dimension(100,100));
      Resizable s = new Circle();
      s.resize(new Dimension(100,100));
}
```

Láthatjuk, hogy a *main* metódus első két sorában egyszerűen létrehoztunk egy kör objektumot, majd meghívtuk az újraméretező metódust. A *main* metódus második része érdekesebb, ahol az objektum típusának deklarációját az interfész segítségével adtuk meg. Amennyiben csak újraméretezési műveleteket akarunk elvégezni az alakzatokon, a deklaráció pillanatában még nem feltétlenül szükséges tudnunk az objektumok konkrét típusát, elegendő tudnunk, hogy implementálják a megfelelő interfészt, és ezáltal elvárhatunk tőlük egy adott viselkedési módot, esetünkben nevezetesen azt, hogy újraméretezhetőek (megvalósítják a megfelelő metódust). Az implementációval kapcsolatos részletek nem relevánsak. Természetesen interfészből nem példányosíthatunk, így a példányosítás pillanatában már egy konkrét, az interfészt megvalósító osztály példányát hozzuk létre.

Megjegyzendő, hogy az interfészek metódusai alapértelmezetten nyilvánosak (a *public* kulcsszó használata opcionális). Az interfészek esetében is beszélhetünk öröklődésről, egyik interfész lehet egy másik interfész leszármazottja. Ezt sok esetben érdemes is kihasználni. Tekintsük például azt a helyzetet, amikor egy létező interfészt, amelynek már léteznek megvalósításai (osztályok, amelyek implementálják az interfész metódusait) ki szeretnénk egészíteni további metódusokkal. Ha megváltoztatjuk az interfészt, akkor az összes érintett osztályon belül változtatnunk kell: minden osztálynak kötelező módon implementálnia kell az újonnan bevezetett metódusokat. Sok esetben szerencsésebb megoldást szolgáltathat a származtatás: az új metódus egy származtatott interfészben kaphat helyet.

Egy további hasznos észrevétel lehet az is, hogy az interfészek megvalósításánál a „szerződés” betartása, csak azt jelenti, hogy az interfész metódusait a megvalósító osztály implementálja, tehát egy, az osztály példányára mutató referencia segítségével ezek a metódusok meghívhatóak. Ez a „vállalás” az implementációs részletekkel, a megvalósítás hogyanjával kapcsolatban semmiféle garanciát nem jelent. Például, az is lehetséges, hogy egy adott interfészt megvalósító osztály valamelyik metódust olyan módon implementálja, hogy kivételt dob a metóduson belül, vagy egyszerűen üresen hagyja a metódus törzsét.

Az interfészek kiválóan alkalmasak arra, hogy feloldjuk a különböző programrészek közötti fölösleges függőségeket. Gondoljunk például egy olyan esetre, amikor egy metódus egy karakterláncokból álló listát vár, hogy a lista elemeit kiírja a konzolra. Maga a lista többféleképpen megvalósítható. Lehet, például, hogy az elemeket tömbben tároljuk, de az is lehet, hogy láncolt listát alkalmazunk. A feladat szempontjából ezek az implementációs részletek nem relevánsak. Csak az fontos, hogy a paraméterként kapott objektum rendelkezzen a listák alapvető tulajdonságaival, például lehessen egy iterátor segítségével bejárni, hogy kiírhassuk az elemeit. Fölösleges lenne megkötnünk a metódusunkat használó programozó kezét azzal, hogy rákötelezzük egy adott implementáció alkalmazására (ami esetleg szükségtelen átalakítások kényszerével is terhelné). A megoldás: a paraméter típusát interfész segítségével határozzuk meg.

Az említett példához hasonló megoldások gyakran előfordulnak majd a jegyzet további részeiben. Gondoljunk erre például a gyűjtemény keretrendszerrel foglalkozó résznél, és remélhetőleg világossá válik majd, hogy miért is játszik nagyon fontos szerepet az interfészek alkalmazása az objektumorientált tervezésben.

A komponens alapú programozás térhódításával az interfészek még fontosabb szerephez jutottak, segítségükkel valósítható meg a különböző komponensek közötti kommunikáció. Vagy ‚gondolhatunk a különböző alkalmazás-programozási felületekre (API – *Application Programming Interface*), arra a tipikus példára, amikor egy cég komplex műveleteket megvalósító osztályokat tartalmazó szoftvercsomagot készít. A csomagot egy másik cég fogja felhasználni saját alkalmazásának fejlesztésekor. A tipikus eljárás, hogy az osztályok publikus interfészeknek lesznek a megvalósításai. A publikus interfészek segítségével lehet majd meghívni az osztályokon belül implementált metódusokat az implementációs részletek ismerete nélkül.

Összefoglalóként kijelenthetjük tehát, hogy érdemes „interfészekben gondolkodni". Persze, mint olyan sok esetben, itt is fontos a „mértéktartás": mindig elemezzük a problémát, és olyan eszközöket alkalmazzunk, amelyek használata az adott helyzetben legindokoltabb. Lehet, hogy interfészek bevezetésére van szükségünk, lehet, hogy absztrakt osztályok alkalmazása szolgáltat az adott helyzetben optimálisabb megoldást, de az is lehet, hogy egyáltalán nem jár előnyökkel az adott rendszert „bonyolultabbá tenni".

2.8 Beágyazott osztályok

A Java programozási nyelv megengedi számunkra, hogy osztályokat hozzunk létre más osztályokon belül. Ezek az osztályok lehetnek statikusak, vagy nem statikusak. A szakterminológia elnevezés szintjén is különbséget tesz ezek között: a statikusokat egyszerűen statikus beágyazott osztályoknak (*static nested classes*), a nem statikusokat belső osztálynak nevezzük (*inner classes*).

Az egyik alapvető különbség, hogy a belső osztályok hozzáférhetnek a külső osztály minden adattagjához és metódusához, beleértve a privát adattagokat és metódusokat is. Figyelem: ez a statikus beágyazott osztályokra nem igaz. A statikus beágyazott osztályok ebből a szempontból ugyanúgy viselkednek, mint az egyszerű osztályok, a beágyazás ebben az esetben tulajdonképpen csak a kód struktúrájának szempontjából fontos. A belső osztályok nem tartalmazhatnak statikus metódusokat, és példányosításukat mindig a külső osztály példányosítása előzi meg.

Példa belső osztály létrehozására és használatára Java-ban:

```
class A {
      A() {}
      class B {
            B() {}
            public void doIt() {
                  System.out.println("Hello");
            }
      }
}
```

```
public class Example {
      public static void main(String[] args) {
            A a = new A();
            A.B b = a.new B();
            b.doIt();
      }
}
```

Példa statikus beágyazott osztály létrehozására és használatára:

```
class A {
      A() {}
      static class B {
            B() {}
            public static void doIt() {
                  System.out.println("Hello");
            }
      }
}
public class Example {
      public static void main(String[] args) {
            A.B.doIt();
      }
}
```

Tudjuk azt, hogy egy forráskód állományon belül csak egyetlen publikus osztály kaphat helyet, de ez a kijelentés a beágyazott osztályokra nem terjed ki. Míg az osztályok csak nyilvánosak, vagy csomag szinten nyilvánosak lehetnek, a beágyazott osztályok esetében minden hozzáférés módosító alkalmazható.

A belső osztályok hasznosak lehetnek arra, hogy logikailag csoportosítsuk az osztályainkat. Például, ha egy osztályt csak egyetlen másik osztályon belül használunk, kézenfekvő, hogy belső osztályként hozzuk létre. Ha a kis terjedelmű, és csak egyetlen másik osztály számára „hasznos” osztályainkat belső osztályként hozzuk létre, a kódunk is áttekinthetőbbé válik. Ezen kívül a beágyazott osztályok az adatrejtés elvének szempontjából is hasznosak lehetnek. Például, ha egy osztály bizonyos attribútumaihoz és metódusaihoz csak egyetlen másik osztálynak kellene hozzáférnie, fölösleges lenne ezeket a tagokat teljesen nyilvánossá tenni. Ha az őket használó osztályt belső osztályként hozzuk létre, megoldjuk a problémát: az adattagok és metódusok maradhatnak rejtve, a belső osztály így is hozzájuk férhet.

A belső osztályok külön kategóriáját képezik a név nélküli belső osztályok. A következő példát még nem szükséges részleteiben megérteni (a későbbiekben még visszatérünk rá, és több hasonlóval is találkozunk), de egy „első találkozásként” azért vethetünk rá egy pillantást:

```
this.addWindowListener(new WindowAdapter() {
      public void windowClosing(WindowEvent e) {
            System.exit(0);
      }
});
```

Mi is történt itt? Tulajdonképpen egy egyszerű metódushívás, az *addWindowListener* nevű metódus meghívása az aktuális objektumra mutató *this* referencia segítségével. Ami érdekes, hogy a metódus paraméterlistáján belül tulajdonképpen létrehoztunk egy új osztályt, amely a *WindowAdapter* osztály leszármazottja, és „helyben" példányosítottunk is belőle. A származtatott osztályon belül újradefiniáltuk a *windowClosing* nevű metódust, és mivel ráadásul a *WindowAdapter* alaposztályunk a *WindowListener* interfész megvalósítása, és az *addWindowListener* metódus pontosan ilyen típusú paramétert vár, minden talál.

A példa teljes megértéséhez szükségesek a grafikus felhasználói felületekkel és eseménykezeléssel kapcsolatos alapismeretek. Ezeket a következő fejezetben tárgyaljuk. Ami számunkra most fontos, hogy az új származtatott osztályt névnélküli (*anonymus*) belső osztályként hoztuk létre. Ez akkor lehet hasznos, amikor nincsen szükségünk egy névvel rendelkező referenciára, amely az illető osztály példányára mutat. Csak „helyben" van szükségünk az illető példányra, a későbbiekben nem akarunk közvetlenül hivatkozni rá. Példánk esetében az *addWindowListener* metódushívás azt eredményezi, hogy a paraméterként kapott objektum bekerül egy listába, és így majd más, „ebben érdekelt" programrészek hivatkozhatnak rá. Számunkra nem szükséges, hogy a későbbiekben direkt módon hozzáférjünk ehhez az objektumhoz, így nincsen szükségünk rá mutató referenciára, megfelelő megoldás, ha név nélküli belső osztályként hozzuk létre, és „helyben" példányosítunk belőle.

A konstrukció első ránézésre bonyolultnak tűnhet, de valójában elegáns kódhoz, kompakt írásmódhoz vezet, így nagyon hasznos lehet. Remélhetőleg, ezt majd a további példákon keresztül sikerül megfelelő módon magyarázni és alátámasztani.

Végezetül még egy észrevétel: a belső osztályok használatának esetében is fontos a „mértéktartás". Akkor használjunk belső osztályokat, amikor az előbb felsorolt szempontokat figyelembe véve a rendszer tervének szempontjából az valóban indokolt. Túlzott használatuk ne vezessen nagyon hosszú, nehezen áttekinthető forráskód-állományokhoz, és ne sértse a kód újrahasznosíthatóságának elvét. Az objektumorientált tervezés szempontjából ezek az elvek, valamint a rendszer megfelelő struktúrája nagyon fontos: ne törekedjünk a feltétlen „sűrítésre", ne féljünk attól, hogy túl sok csomagunk, osztályunk, interfészünk lesz, ez az elvek szempontjából előnyökkel járhat. Minél modulárisabban tervezzünk, mindig figyelembe véve az általánosság, újrafelhasználhatóság és átláthatóság szempontjait.

2.9 Típusmódosítók

A 2.2 alfejezetben leírt hozzáférés-módosítók mellett fontos megemlítenünk a különböző típusmódosítókat is. Most, hogy megismerkedtünk a jelentésük megértéséhez szükséges alapfogalmak többségével, soroljuk fel a Java programozási nyelvben alkalmazható típusmódosítókat.

Ezek az osztályok esetében a következők:

- *final*: az illető osztályból nem lehet származtatni;
- *abstract*: absztrakt osztályok esetében alkalmazzuk.

A metódusok esetében alkalmazható típusmódosítók a következők:

- *static*: osztályszintű metódusok, meghívásukhoz nincs szükség példányosításra. Csak statikus attribútumokkal végezhetnek műveleteket, és nem hívhatnak meg nem statikus metódusokat. Meghívásuk az osztály nevének segítségével történik (*Osztalynev.metodus()*);
- *abstract:* az absztrakt osztályokban deklarált, implementációval nem rendelkező metódusok. Ezeket a származtatott osztályok fogják megvalósítani;
- *final*: nem újradefiniálható metódusok;
- *native*: valamilyen platformfüggő (pl. C++) nyelvben implementált metódusok (mint említettük léteznek különböző eljárások, amelyek segítségével Java kódunkba natív kódrészleteket integrálhatunk);
- *synchronized*: különböző erőforrásokhoz történő hozzáférés szinkronizálásakor alkalmazzuk. Részletesebben tárgyaljuk a jegyzet végrehajtási szálakkal kapcsolatos részénél.

Az attribútumokra alkalmazható típusmódosítók a következők:

- *static*: osztályszintű attribútum, minden példány ugyanazt az értéket használja;
- *final*: csak egyszer történhet értékadás, konstansok deklarációjánál alkalmazhatjuk. Figyelem: egy referencia esetében a *final* használata nem jelenti azt, hogy a referencia által azonosított objektum állapota (attribútumainak értéke) nem változhat, csak azt, hogy a referencia nem „átirányítható”;
- *transient*: olyan attribútum, amely nem képezi részét az objektum perzisztens állapotának. A nem szerializálható adattagok esetében alkalmazzuk (a szerializálásról az adatfolyamokkal kapcsolatos fejezetben lesz szó);
- *volatile*: olyan adattagok, amelyek értékét több végrehajtási szál is változtathatja (a végrehajtási szálakról egy későbbi fejezetben esik szó).

2.10 Az *Object* ősosztály

Ahogyan azt már elmondtuk, a Java egy tiszta objektumorientált nyelv, „nincs élet” az osztályokon kívül. Minden osztály egy jól meghatározott osztályhierarchiának a része, és ennek a hierarchiának a csúcsán, a legfelső szinten, a „Nagy Közös Előd”, az *Object* osztály található. Ez az osztály definiálja a legalapvetőbb metódusokat, és minden osztály direkt vagy közvetett módon ennek az osztálynak leszármazottja.

Az *Object* ősosztály által definiált metódusok nevei: *clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString* és *wait*. Ezek közül néhányra már most kitérünk, a többiekkel a későbbiekben fogunk megismerkedni.

A **getClass() metódus** az objektum típusának futási időben történő meghatározására szolgál. Egy *Class* típusú objektumot térít vissza, amelynek segítségével az osztályról kérhetünk különböző információkat.

Az **equals() metódus** az objektumok egyenlőségét vizsgálja. Általában a metódust a tartalom, az állapot összehasonlítására szokás használni. Bár az *Object* osztály alapértelmezett implementációja csak a referenciák azonosságát vizsgálja, a származtatott osztályokban ez a metódus általában olyan módon van újradefiniálva, hogy mélyebb, tartalmi összehasonlításra adjon lehetőséget. Például *String* objektumok esetében akkor fog igaz eredményt (*true* értéket) adni, ha a karakterláncok azonos karakterekből állnak.

A **hashCode() metódus** egy egész értéket, az objektum hash kódját téríti vissza, amely gyakran szükséges, amikor az objektumokat hasító táblákban tároljuk. A metódus a kódot a példány aktuális állapotának függvényében képezi. Ha két objektum állapota azonos (tartalma megegyezik), hash kódjuk is megegyezik. Következményként, ha újradefiniáljuk az *equals* metódust, akkor általában a *hashCode* metódust is újra kell definiálnunk. Az *Object* osztály alapértelmezett metódusa a memóriacím alapján képezi a kódot. A metódus újradefiniálásánál fontos, hogy minden olyan adattagot felhasználjunk a kód generálásához, amelyet az *equals* metóduson belül felhasználtunk az összehasonlításhoz.

A **toString() metódus** az osztály egy szöveges reprezentációját téríti vissza. Az *Object* osztály implementációja az osztály nevét és hash kódját fűzi egybe, de természetesen ez a metódus is újradefiniálható.

A **finalize() metódus**t az objektum által foglalt memóriaterület felszabadítása előtt hívja meg a szemétgyűjtő. Az *Object* osztály nem ad implementációt erre a metódusra. A származtatott osztályokban tipikusan az objektum által foglalt erőforrások felszabadítására alkalmazzák.

A **clone() metódus**t akkor használjuk, ha egy másolatot szeretnénk készíteni az osztály egy már létező példányáról. A metódus fejléce:

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

A másolat készítése csak akkor valósítható meg, ha az osztály implementálja a *Cloneable* interfészt. Maga az *Object* ősosztály ezt nem teszi meg, így ha egy, az interfészt nem implementáló osztály példányáról szeretnénk a *clone* metódushívás segítségével másolatot készíteni, a *CloneNotSupportedException* típusú kivételt kapjuk futási időben.

Ha a másolat elkészíthető (az osztály implementálja a *Cloneable* interfészt), akkor egy, az eredeti objektummal megegyező állapotú új objektumot kapunk eredményül. Megjegyzendő, hogy az adattagokról nem készül másolat (a maguk során ezek nem lesznek „klónozva"), így alapértelmezetten a metódushívás egy sekély másolást (*shallow copy*) eredményez, nem készül mély másolat (*deep copy*). Természetesen az osztályokban a *clone* metódus újradefiniálható, és így mély másolat is készíthető.

A másolatokkal kapcsolatosan van néhány elvárás, de (mivel a *clone* tetszőlegesen újradefiniálható) ezek nem kötelező érvényűek. Az általános elvárások a következők:

- a másolat egy teljesen új, független objektum legyen (`x.clone() != x`);
- a másolat az eredetivel azonos típusú legyen (ezért a konvenciónak megfelelően a metóduson belül az új példányt nem a *new* kulcsszóval hozzuk létre, hanem az alaposztály *clone* metódusának meghívásával)
- (`x.clone().getClass() == x.getClass()`);
- A másolat és az eredeti objektum legyen azonos állapotban (tartalmilag egyezzenek) (`x.clone().equals(x)`).

A *wait* és *notify* (valamint *notifyAll*) metódusokat a későbbiekben, a végrehajtási szálakkal foglalkozó fejezet szinkronizálással kapcsolatos részénél tárgyaljuk.

Már korábban is említettük, de érdemes újra kihangsúlyozni, hogy az objektumok másolásánál, összehasonlításánál nagyon figyeljünk a helyesen alkalmazott kódra. Ha két objektum tartalmát (állapotát) szeretnénk összehasonlítani, semmiképpen se az egyszerű logikai operátort (==) alkalmazzuk, mivel azzal csak a referenciák azonosságát vizsgáljuk (ugyanarra a példányra mutatnak-e). Használjuk az *equals* metódust. Mint már említettük, néhány a Java nyelv által bevezetett, és a munkánk segítését célzó mechanizmus további félreértések forrása lehet ebben a tekintetben, és az odafigyelés (vagy megértés) hiánya komoly hibákhoz vezethet. Röviden szemléltessük ezt a dolgot néhány példával.

Tekintsük az alábbi hibás kódot:

```
String s1 = "Hello";
String s2 = "Hello";
if (s1 == s2) System.out.println("azonos karaktersorok");
```

A fenti kód hibás, mivel mi az objektumok tartalmát, a karaktersorok egyezését szerettük volna vizsgálni, de ehelyett csak a referenciákat hasonlítottuk össze. Ha kipróbáljuk a fenti kódot, mégis „működik” a dolog. Vajon miért?

Most nézzük az alábbi változatot:

```
String s1 = new String("Hello");
String s2 = new String("Hello");
if (s1 == s2) System.out.println("azonos karaktersorok");
```

Első ránézésre nincsen nagy különbség, de ebben a második esetben már mégsem áll fent az egyenlőség. Hogyan lehetséges ez?

A megoldás a JVM által a memória hatékonyabb kihasználása céljából alkalmazott *String Pool* mechanizmus. Ennek a mechanizmusnak megfelelően, ha a referenciát egy karakterlánc-literál segítségével inicializáljuk, nem keletkezik feltétlenül egy új objektum. A rendszer egy gyűjteményt („*pool*”-t) használ az ilyen karakterláncok tárolására. Amikor egy karakterlánc-literál segítségével inicializáljuk a referenciát, először ellenőrzi, hogy az illető karakterlánc már megtalálható-e ebben a gyűjteményben, és ha igen, akkor egyszerűen visszatérít egy rámutató referenciát. Ellenkező esetben megtörténik a példányosítás, és az új példány bekerül a gyűjteménybe. Ez természetesen nem így történik akkor, ha a *new* kulcsszót használjuk. Ilyenkor mindig új objektum jön létre.

Visszatérve a példánkhoz, az első esetben a két referencia valóban ugyanarra az egy objektumra mutatott, a második esetben viszont, tartalmilag egyező, de különálló objektumok kerültek összehasonlításra. Természetesen mi hibáztunk akkor, amikor objektumok tartalma helyett a referenciák azonosságát vizsgáltuk, nem alkalmaztuk az *equals* metódust. A kódot helyesen a következő formában kellett volna írnunk:

```
String s1 = new String("Hello");
String s2 = new String("Hello");
if (s1.equals(s2)) System.out.println("azonos karaktersorok");
```

Hogy jobban érzékeltessük a hibánk „komolyságát” tekintsünk még egy példát:

```
Integer i1 = 127;
Integer i2 = 127;
if (i1 == i2) System.out.println("a szamok egyenloek");
```

Ismét hibáztunk. Bár a burkoló osztályok leírásánál már említett *auto-boxing* mechanizmus lehetővé teszi számunkra, hogy a fenti formában használjuk a burkoló osztályok és primitív adattípusok közötti megfeleltetést, ettől itt még objektumokról, a burkoló osztály példányairól van szó, tehát, ha a tartalmukat szeretnénk összehasonlítani, szintén az *equals* metódust kellene használnunk. És itt jön a meglepetés: a fenti kódrészlet szintén „működik”. De nézzük csak: ha a 127-et mindkét sorban mondjuk 128-ra cseréljük, akkor már nem áll fent az egyenlőség. Hogyan lehetséges ez? A 127 egyenlő 127-el, de a 128 már nem egyenlő 128-al? Itt valami gondnak kell lennie.

Valóban. De természetesen nem valami matematikai paradoxonról van szó, csak annyi a gond, hogy nem az *equals* metódust használtuk az összehasonlításra. Hogy az első esetben mégis működött a dolog, egy, az előbbiekben említett *String Pool* mechanizmushoz hasonló megoldásnak köszönhető, az 5-ös Java (Tiger) által bevezetett *wrapper class caching* mechanizmusnak. A mechanizmust szintén a teljesítmény növelésével kapcsolatos meggondolásból vezették be. A megoldásnak megfelelően a JVM használ egy *IntegerCache* osztályt, amelyen belül egy tömbben tárolja a -128 és 127 közötti egész számokat burkoló objektumokat. A *String Pool* mechanizmushoz hasonlóan az előbbi példa esetében is literálokkal inicializáltuk a referenciákat (nem történt a *new* kulcsszó segítségével példányosítás), és az *IntegerCache*-ben tárolt értékek esetében nem történt példányosítás, hanem mindkét referencia a már tárolt példányra mutatott.

A helyes megoldás természetesen újra csak az *equals* metódus alkalmazása:

```
Integer i1 = 128;
Integer i2 = 128;
if (i1.equals(i2)) System.out.println("a szamok egyenloek");
```

Összefoglaló tanulságként azt mondhatjuk tehát, hogy nagyon figyeljünk a kódunk minden részletére, ha meg akarjuk kímélni magunkat és társainkat, felhasználóinkat a csúnya, nehezen felfedezhető hibáktól. Az „odafigyelés” természetesen a megfelelő gyakorlattal automatizmussá válik.

2.11 Bean-ek, POJO-k, konvenciók

Az objektumorientált programozásban (és így a Java-ban is) nagyon fontos szerepet játszanak a különböző minták és konvenciók. Egy általánosan elfogadott elv például, hogy az osztályok attribútumait lehetőleg privátnak deklaráljuk (vagy ha a helyzet megkívánja, *protected*-nek). Ha nem feltétlenül szükséges, ne hagyjunk direkt hozzáférési lehetőséget a külső osztályoknak. Az osztályok „belső ügyei” maradjanak az „osztályon belül”. Ha mégis szükség van arra, hogy kívülről hozzáférjünk az attribútumokhoz, akkor ez a hozzáférés speciális metódusokon, úgynevezett *getter* és *setter* metódusokon keresztül történjék. A *getter* visszatéríti egy adott attribútum aktuális értékét, a *setter* beállítja azt egy paraméterként kapott értékre.

Egy kézenfekvő magyarázat az adatrejtés támogatása: lehetséges, hogy bizonyos attribútumok esetében csak olvasási, vagy csak írási joggal szeretnénk felruházni a külső osztályokat, és egy ilyenfajta különbségtétel nem lenne lehetséges, ha egyszerűen nyilvánossá tennénk az illető attribútumot. Az attribútumnak megfelelő *getter* és *setter* metódusok közül viszont bármelyik elhagyható.

Hogy még inkább meggyőződjünk a javaslat megalapozottságáról, tekintsük a következő példát. Egy személy osztályt hozunk létre a vezetéknév és keresztnév attribútumokkal, és ezeket egyszerűen nyilvánosnak deklaráljuk. A későbbiekben megjelenik az igény, hogy a nevek megadásakor ellenőrizzük, hogy nagy kezdőbetűvel kezdődnek-e a nevek, és ha nem, akkor automatikusan konvertáljuk a kezdőbetűket nagybetűkbe. Ha a neveket úgy adtuk meg, hogy egyszerűen hozzárendeltük őket a publikus attribútumokhoz, ennek az új funkcionalitásnak a bevezetésére nincsen lehetőségünk. Ha viszont *setter* metódusokat alkalmaztunk volna, ezek a metódusok egyszerűen kiegészíthetőek lennének. És hasonlóan, ha bármilyen számításokat, átalakításokat vagy ellenőrzéseket akarunk végrehajtani az érték-hozzárendelésekkel, vagy lekérdezésekkel egy időben, a *getterek* és *setterek* használata lehet a megoldás.

Fontos, hogy osztályainkon belül betartsuk a Java kódolási és elnevezési konvenciókat [6] is. Kódolási konvenció meglehetősen sok van, de ezek többnyire általánosak, betartásuk mindegyik programozási nyelv esetében fontos, és ezekre vonatkozó leírás, felsorolás nagyon sok létezik a szakirodalomban. Ide tartozik az, hogy megfelelően tördeljük (*indentation*) a kódunkat (pl. ne használjunk nagyon hosszú sorokat), használjunk kifejező változóneveket, lássuk el a kódot magyarázatokkal, kommentárokkal (a vonatkozó konvenciók figyelembevételével), de az olyan „apróságok” is idetartoznak, hogy például ne hagyjunk ki helyet a metódus neve és paraméterlistája között, stb., stb. A fontosabb Java-specifikus elnevezési konvenciók a következő szabályokban foglalhatóak össze:

- az osztályok és interfészek nevei mindig nagybetűvel kezdődnek. Ezen kívül, ha a név több szóból tevődik össze, minden belső szó kezdőbetűje szintén nagybetű. Például: *ArrayIndexOutOfBoundsException;*
- az attribútumok és metódusok nevei kis kezdőbetűvel kezdődnek, de, ha több szóból tevődnek össze, a belső szavakat szintén nagy kezdőbetűvel írjuk. Például: *addWindowListener*. A *getterek* és *setterek* esetében a metódusok neve tipikusan a „get” és „set” (vagy a *boolean* típusú visszatérített értékek esetében „is”) előtagból, és a megfelelő attribútum nevéből áll össze. Például *getName*, *setName*, *isMarried* stb.

A Java szakirodalomban gyakran fogunk találkozni a *bean* (bab), vagy *JavaBean* fogalommal. A *JavaBean* egy újrafelhasználható Java szoftverkomponens. Gyakorlatilag olyan osztályokról van szó, amelyek megfelelnek bizonyos konvencióknak: rendelkeznek alapértelmezett konstruktorral, privát adattagokkal, valamint ezeknek megfelelő publikus *getter* és *setter* metódusokkal (ezek elnevezései megfelelnek a Java elnevezési konvencióknak), és szerializálhatóak (a szerializálásról a jegyzet adatfolyamokkal kapcsolatos fejezetében esik majd szó).

A *JavaBean*-ek használatának előnye, hogy ezek a komponensek, mivel betartják a fenti szabályokat, egyszerűen manipulálhatóak különböző környezeteken, keretrendszereken belül, ezek komponensei által. Például, vizuális fejlesztői eszközök használatának esetében jelenthetnek előnyt, de a minta alkalmazása hasznos lehet tipikusan adattárolásra használt,

vagy *domain* objektumok (*business objects/domain objects* – lásd a jegyzetben későbbiekben leírt kivonatolt esettanulmányt) esetében is.

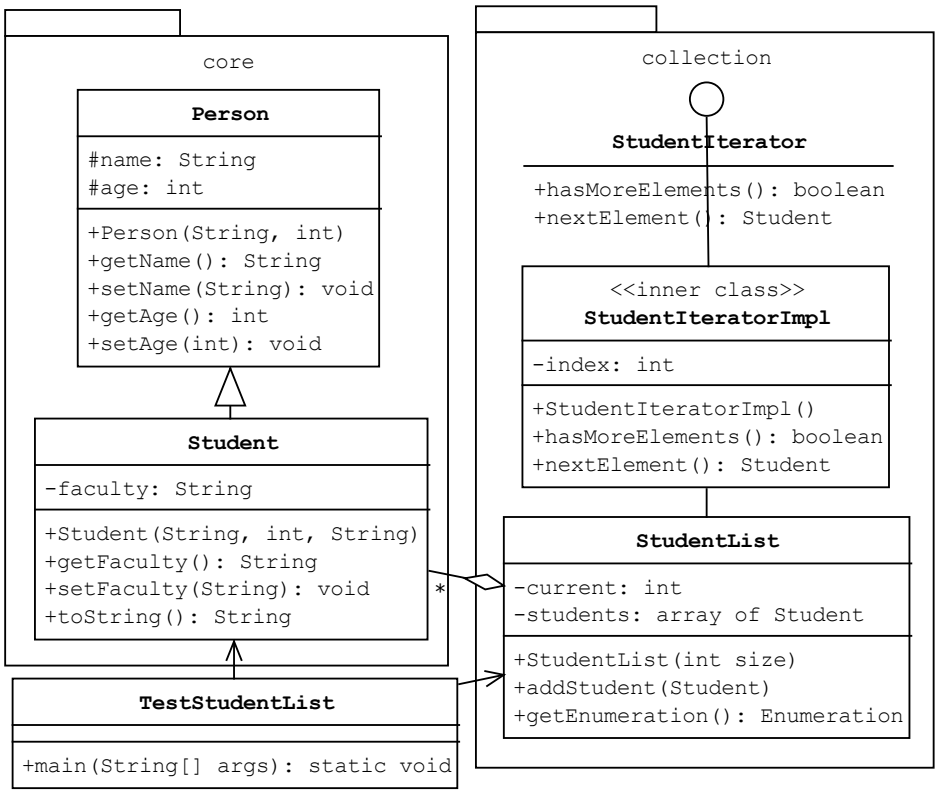
Figyelem: a fogalom nem összetévesztendő az *Enterprise JavaBean* (EJB) fogalommal. Az *Enterprise JavaBean*-ek szintén újrafelhasználható szoftverkomponensek, de esetükben főként osztott vállalati rendszereknél alkalmazott szerver oldali komponensekről van szó, amelyek egy komplexebb komponens-modellnek felelnek meg.

A POJO rövidítés a *Plain Old Java Object* kifejezésből származik. Azokra az egyszerű Java osztályokra szokták használni ezt a megnevezést, amelyek nem rendelkeznek „speciális tulajdonságokkal", nem akarnak megfelelni különböző bonyolult szabályoknak (és semmiképpen sem *Enterprise JavaBean*-ek). Természetesen az alapvető szabályoknak (például privát attribútumok, publikus *getterek* és *setterek*), valamint a kódolási és elnevezési konvencióknak ezeknek az osztályoknak is meg kell felelniük. Tulajdonképpen azt is mondhatjuk, hogy egy egyszerű *JavaBean* és egy POJO között csak annyi a különbség, hogy az utóbbinak nem feltétlenül kell betartania az alapértelmezett konstruktorral és szerializálhatósággal kapcsolatos szabályokat.

A fenti fogalmak ismerete, de főként a minták és konvenciók figyelembe vétele nagyon fontos. Gondoljunk arra, hogy a szoftverfejlesztésben rengeteg időt vesz fel a rendszerek javítása, karbantartása. Képzeljünk el egy olyan helyzetet, amikor megírunk egy terjedelmesebb szoftvermodult, és később, amikor mi már nem dolgozunk az illető projekten, ezt az alrendszert javítani kell, vagy át kell alakítani. Nagyon fontos, hogy a kód átlátható, könnyen érthető legyen. Ha mi száz sorban írtuk meg azt, amit tízben is lehetett volna, nem alkalmaztuk az alapvető mintákat, és ráadásul még a konvenciókat sem tartottuk be, súlyos problémát idéztünk elő. A kód minősége, és a megoldások eleganciája jelentik a programozó névjegyét. Figyeljünk erre!

2.12. Javasolt gyakorlat

Az alábbi osztálydiagram (2.10 ábra) alapján hozzuk létre a diagramban feltüntetett csomagokat, interfészt és osztályokat.



2.10 ábra: útmutatás a program megírásához: osztálydiagram

Magyarázat: létrehozzuk a *core* és *collection* csomagokat. A *core* csomagon belül létrehozunk egy személy (*Person*) osztályt (POJO) a név (*name*) és kor (*age*) adattagokkal, az adattagok értékeit beállító paraméteres konstruktorral, valamint az adattagoknak megfelelő *getter* és *setter* metódusokkal. A *Person* osztályból származtatjuk a hallgató (*Student*) osztályt, kiegészítve a Kar (*faculty*) adattaggal, a megfelelő konstruktorral és *getter/setter* metódusokkal. A *Student* osztályon belül újradefiniáljuk az *Object* ősosztály *toString* metódusát, visszatérítve a hallgatókkal kapcsolatos információk szöveges reprezentációját (például a név, kor és Kar tulajdonságoknak megfelelő karakterláncok összefűzéséből előállított *String*-et).

A *collection* csomagon belül létrehozzuk a *StudentList* osztályt, amelyből hallgató objektumokat tartalmazó gyűjtemények példányosíthatóak. Az osztályon belül egy *Student* típusú elemeket tartalmazó tömbben rögzítjük a hallgatókkal kapcsolatos információkat. A *current* adattag mindig a tömb aktuális elemére fog mutatni, így az új hallgató beillesztését végző *addStudent*

metódus felhasználhatja ezt az értéket az új elem pozíciójának meghatározásához, és a beillesztés után elvégezheti az érték frissítését.

A tömb méretét (a lista maximális kapacitását) a konstruktor paramétere határozza meg, az elemeknek a konstruktoron belül foglalunk helyet.

A lista bejárása egy iterátor segítségével történhet. Az iterátornak megfelelő osztály a *StudentIterator* interfésznek lesz a megvalósítása. Az interfész két metódust deklarál: a *hasMoreElements* metódus segítségével lekérdezhető, hogy van-e még elem a listában (vagy annak végéhez értünk), a *nextElement* metódus segítségével lekérdezhető a lista következő eleme. Ezt az interfészt a *StudentIteratorImpl* osztály valósítja meg, egy egész típusú attribútum segítségével rögzítve az aktuális pozíciót. Az osztályt a *StudentList* osztály belső osztályaként hozzuk létre. A *StudentList* osztály *getIterator()* metódusa példányosít a *StudentIteratorImpl* osztályból, és visszatéríti a létrehozott objektumra mutató referenciát (megjegyzés: a visszatérített érték típusa az interfész segítségével van meghatározva, így a metódust meghívó, a bejárást elvégző osztályokat nem befolyásolja, ha a későbbiekben lecseréljük az implementációt).

A létrehozott osztályok kipróbálásához írjunk egy rövid programot. A *TestStudentList* osztály *main* metódusán belül hozzunk létre néhány *Student* objektumot. Hozzunk létre egy listát (*StudentList* példány), és adjuk hozzá ehhez a létrehozott hallgató objektumokat. Az iterátor segítségével egy cikluson belül járjuk be a listát, és írjuk ki a konzolra a hallgatókkal kapcsolatos információkat.

3

Grafikus felhasználói felületek és eseménykezelés

Ebben a fejezetben egyszerű grafikus felhasználói felületek (GUI – *Graphical User Interface*) elkészítésének és működésre bírásának módjáról lesz szó. A fejezet első részében a Java AWT eszközkészletet [7] mutatjuk be, majd a második részben az eseménykezelés alapjait tárgyaljuk. Az olvasó felteheti a kérdést, hogy hogyan, miért jutunk el ilyen gyorsan ehhez a témához, hiszen még számos alapvető kérdéskört nem tárgyaltunk. Az indok a gyakorló feladatok megoldásának motiválása: tapasztalataink azt mutatják, hogy szívesebben oldunk feladatokat, ha egy látványosabb eredmény szolgáltat sikerélményt. Ezért gondoljuk azt, hogy a grafikus felhasználói felületek létrehozásának korai elsajátítása hasznos lehet a nyelv elsajátításához elengedhetetlenül szükséges gyakorlás szempontjából. Az elmaradt alapvető elméleti részekre a későbbiekben természetesen visszatérünk.

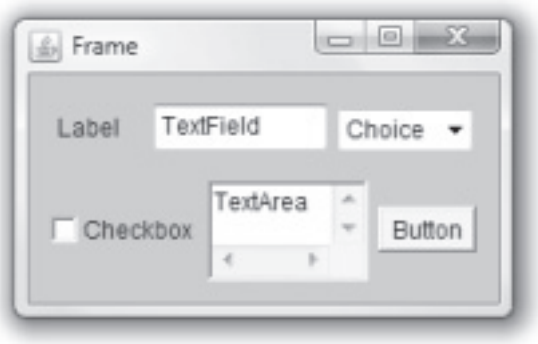
3.1 A *java.awt* csomag

Az AWT a Java első, 1995-ben kiadott grafikus felhasználói felületek készítésére alkalmas eszközkészletének a neve. A rövidítés az *Abstract Window Toolkit* elnevezésből származik, ami utal arra, hogy egy platformfüggetlen grafikus eszközkészletről van szó.

A grafikus felhasználói felületeket grafikus elemekből, komponensekből (*widgets*) rakjuk össze, és működésük tipikusan eseményalapú: a felhasználó kölcsönhatásba lép a felszínnel (pl. egy gombra kattint az egérrel) és ezt az eseményt a felszín a vezérlő modulhoz továbbítja, amely elvégzi a szükséges lépéseket (pl. mi történjen, ha a felhasználó kattintott a gombra). Az eseménykezelésről bővebben a fejezet második részében esik szó.

Az 3.1 ábrán néhány gyakori grafikus komponensre láthatunk példát: keret (*frame*), címke (*label*), jelölőnégyzet (*checkbox*), szövegmező (*textfield*), szöveges komponens (*text area*), választás komponens (*choice*), gomb (*button*). Természetesen ez csak néhány alapvető példa, az általunk megszokott szoftverek sokkal bonyolultabb felszínnel rendelkeznek, sokkal több komponenstípust ismerünk, és ennek megfelelően a GUI-ek felépítésére használható eszközkészletek (így az AWT is) jóval gazdagabbak. Ezt a későbbiekben látni is fogjuk, mikor megismerkedünk néhány komponenstípussal, illetve ezek használatával.

Megjegyzendő, hogy az AWT kevéssé népszerű, bizonyos értelemben túlhaladottnak számító eszközkészlet, és a biztosított komponenstípusok szempontjából nem túlságosan gazdag. Megjelenését nagyon hamar követte a SWING eszközkészlet kiadása (1996-ban), majd további eszközkészletek is megjelentek (például az SWT), és ezek jóval szélesebb körben alkalmazottak. Ennek oka az AWT működési háttere, amely számos hátránynak a forrása (ezt a későbbiekben látni fogjuk).



3.1 ábra: néhány gyakori grafikus komponens

A jegyzet további részeiben több szó fog esni a SWING eszközkészletről, de ezelőtt fontos az AWT bemutatása is. Ahhoz, hogy megértsük, hogy mi volt a SWING korai bevezetésének, az AWT alternatíva igényének a magyarázata szükséges látnunk a különbségeket. És, ami még fontosabb: nagyon sok dolog, amelyet ebben a fejezetben tárgyalni fogunk ugyanúgy érvényes más eszközkészletek (pl. SWING vagy SWT) esetében is.

3.1.1 AWT: a színfalak mögött

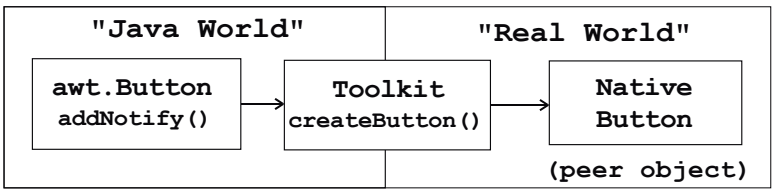
Az AWT csomagot használó fejlesztők absztrakt komponensekből rakják össze grafikus felhasználói felületeiket. A komponensek által biztosított funkcionalitások, és a létrehozásukhoz felhasznált osztályok azonosak, függetlenül attól, hogy milyen operációs rendszer alatt futtatjuk majd programunkat. A felszín megjelenítésében lesznek különbségek, a komponensek a platform ablakrendszerének megfelelően lesznek megjelenítve, de ez nem befolyásolja a programozó munkáját. Ez a függetlenség az AWT komponensek és a nekik megfelelő natív komponensek közötti megfeleltetést megvalósító köztes rétegnek köszönhető.

Minden absztrakt komponensnek van egy natív párja (*peer* objektum), amely az illető komponens adott natív környezetnek megfelelő implementációja. Egy felszín létrehozásánál számunkra csak a natív objektumnak megfelelő *peer* interfész fontos. Például egy AWT *Choice* komponens egy olyan natív komponensnek felel meg, amely lehetővé teszi a felhasználónak egy elem kiválasztását egy listából. Elegendő tudnunk, hogy a natív komponens megfelel ennek az elvárásnak, nem érdekelnek az implementációjával, megjelenítésével kapcsolatos részletek. Tehát elegendő, ha tudjuk, hogy a komponens implementálja a neki megfelelő *peer* interfészt. Minden AWT komponensnek megfelel egy ilyen interfész, és ezek a *java.awt.peer* csomagban találhatóak.

Az AWT komponensek és natív megfelelőik közötti kötést *Toolkit* objektumok teszik lehetővé. A *Toolkit* egy absztrakt alaposztály, amely a felszínek platform-specifikus részleteivel kapcsolatos tulajdonságok számára biztosít interfészt (például felbontás, rendelkezésre álló betűtípusok, stb.). Minden Java-t támogató platformnak biztosítania kell egy *Toolkit* implementációt.

Egy Java (AWT) komponens (pl. *Button*) létrehozása nem vonja maga után automatikusan a *peer* objektum létrehozását. Ezt csak később a *Toolkit* hozza létre, a Java komponens *addNotify()* metódusának hatására. Ez a metódus akkor kerül meghívásra, amikor az illető komponenst

meg kell jeleníteni, tipikusan akkor, amikor az őt tartalmazó tároló (*container*) komponens (ezekről a későbbiekben lesz szó) megjelenik a képernyőnkön. A metódus a maga során meghívja a *Toolkit createXXX()* metódusát (pl. *createButton()*), és ez a metódus hozza létre a komponens natív párját (a *peer* objektumot), amely megjelenik a képernyőnkön. Mondhatjuk tehát, hogy a *Toolkit* egy natív komponens gyárként (*Factory*) viselkedik.



3.2 ábra: az AWT működési alapelvének szemléltetése

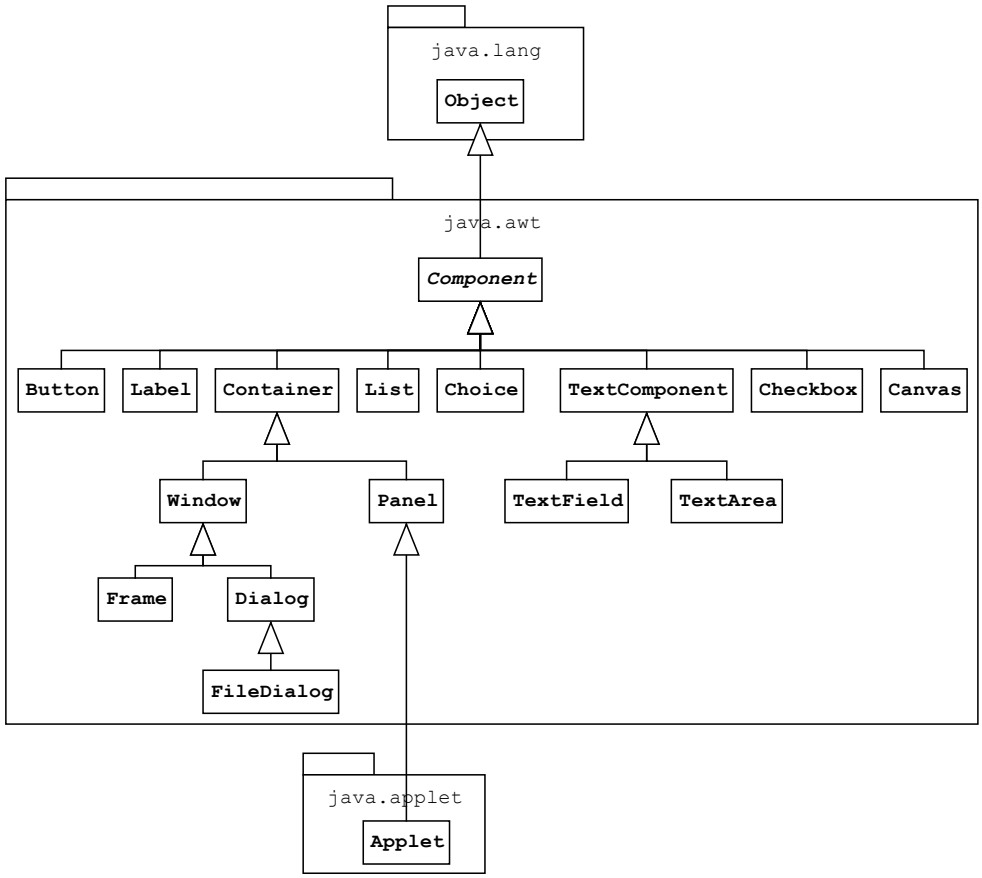
A különböző platformoknak megfelelő *Toolkit*-ek bonyolultak. Szerencsére Java programozóként, ha csak egyszerűen felhasználói felszíneket akarunk létrehozni AWT segítségével, és nem akarunk például egy új (a JDK által még nem támogatott) platformhoz írni *Toolkit*-et, vagy nem akarunk valami nagyon speciális megjelenítést, nem kell velük foglalkoznunk. Megjegyzendő viszont, hogy a bonyolultság hibalehetőségekhez vezetett és többek között ez volt az egyik oka az AWT „korai hanyatlásának".

3.1.2 AWT komponensek

Minden grafikus komponensnek megfelel egy az AWT csomagon belüli osztály. Az osztályhierarchia legfelső szintjén helyezkedik el a *Component* absztrakt alaposztály. Az osztály a komponensek közös alaptulajdonságainak (szín, méret, stb.) megfelelő attribútumokat, valamint a megjelenítéssel és viselkedéssel kapcsolatos metódusokat deklarálja. Az absztrakt metódusokat a különböző konkrét komponenseknek megfelelő osztályok fogják megvalósítani.

A *Component* alaposztályból származik, a *Container* osztály, amely a tároló (*container*) komponensek részére határoz meg egy közös alapot. Ezek olyan komponensek, amelyek több másik komponenst tartalmazhatnak (pl. panelek, ablakok). A *Container* alaposztályból származnak a különböző konkrét tároló típusoknak megfelelő osztályok, például a *Panel* és a *Window*. Természetesen a hierarchiának még vannak következő szintjei: a *Window* osztályból származik a *Frame* osztály a keretek (plusz tulajdonságokkal rendelkező, pl. lekicsinyíthető ablakok) részére, és a *Dialog* osztály a különböző párbeszédablakok létrehozására. Ha a *Dialog* osztálytól még tovább lépünk egy szinttel a hierarchiában, eljuthatunk a *FileDialog* osztályig, amely az állományok megnyitására szolgáló párbeszédablak megfelelője. A *Panel* osztályból származik az *Applet* osztály, amellyel a következő fejezetben részletesebben is foglalkozunk.

Az említett tároló osztályok közös tulajdonsága tehát, hogy más komponenseket tartalmazhatnak. Ezek a komponensek a maguk során lehetnek szintén tárolók (például egy ablakon belül elhelyezhetünk több panelt), vagy lehetnek egyszerű komponensek (címkék, gombok, stb.).



3.3 ábra: az AWT csomag részleges osztálydiagramja

Az egyszerű komponenseknek megfelelő osztályok közvetlenül a *Component* alaposztály leszármazottjai. Megemlíthetjük itt a *Label* (címke), *Button* (gomb), *Checkbox* (jelölőnégyzet), *Choice* (választás komponens), *List* (lista komponens), *Canvas* (rajzvászon) komponenseket. Ezen kívül a *Component* osztály leszármazottja a *TextComponent* osztály, amelyből származtatták a *TextField* (szövegmező) és *TextArea* (többsoros szöveg komponens) osztályokat.

A fontosabb AWT grafikus komponenseknek megfelelő osztályok hierarchiáját a 3.3 ábra szemlélteti.

A grafikus felhasználói felületek fontos részét képezik a menük. A menük létrehozásához használható osztályok a *MenuComponent* absztrakt alaposztály leszármazottjai. Ebből a *MenuComponent* osztályból származnak például a *MenuBar* és *MenuItem* osztályok.

Az AWT csomagban található osztályoknak a segítségével létrehozhatunk grafikus komponenseket, tárolókat és egyszerű komponenseket egyaránt (példányosítunk a megfelelő osztályokból). Az osztályok által biztosított metódusok segítségével beállíthatjuk a komponensek tulajdonságait: háttér színe, betűtípus, méret, pozíció, stb..

A tárolók által biztosított *add* metódus segítségével hozzáadhatunk komponenseket a tárolóinkhoz. A kérdés, hogy hogyan rendezzük a komponenseket a tárolón belül? Ebben a feladatban lehetnek segítségünkre a komponensek elrendezéséért felelős *layout manager* osztályok.

3.1.3 Komponensek elrendezése

A komponensek tárolókon belüli elrendezése speciális elrendezésmenedzser (*layout manager*) osztályok példányainak segítségével történik. Ezeknek az osztályoknak közös tulajdonsága, hogy implementálják a *LayoutManager* interfészt. Minden tárolóhoz alapértelmezetten hozzárendelődik egy *LayoutManager* objektum. A tárolók egy metódust is biztosítanak (*setLayout(LayoutManager manager)*) az elrendezésért felelős objektum beállítására.

Az AWT csomag biztosít számunkra elrendezésmenedzser osztályokat, de természetesen külső osztályok, speciális elrendezési stratégiák is alkalmazhatóak. Az AWT csomag *LayoutManager* implementációi a következők:

- *FlowLayout*: a komponenseket a tárolón belül balról jobbra, és fentről lefele haladva sorba rendezi. Ez a legegyszerűbb elrendezésmenedzser, a panelek esetében ez az alapértelmezett elrendezés.
- *BorderLayout*: a tárolót öt részre osztja, az északi (*north*), déli (*south*), keleti (*east*), nyugati (*west*) és középső (*center*) részekre. A komponensek tárolón belüli elrendezése a felosztásnak megfelelően történik. Az *add* metódus paramétereként meg kell határoznunk, hogy a tároló melyik részében szeretnénk megjeleníteni a komponensünket. A keret komponensek alapértelmezett elrendezése ilyen típusú.
- *GridLayout*: a komponenseket egy négyzethálón, rácsszerkezeten helyezi el, ahol minden cella mérete azonos, és minden komponens egy cellát foglal el. A *GridLayout* objektum létrehozásakor meg kell adnunk, hogy hányszor hányas cellarácsba szeretnénk elrendezni a komponenseinket. Ezután a komponensek elhelyezése fentről lefelé történik, balról jobbra, vagy jobbról balra, a *ComponentOrientation* tulajdonságnak megfelelően. Megjegyzendő, hogy a rács méreteinek meghatározásánál az oszlopszámnak csak akkor van jelentősége, ha a sorok száma 0. Ellenkező esetben a szükséges oszlopok száma a megadott sorok számának, és a hozzáadott komponensek számának függvényében lesz meghatározva.
- *GridBagLayou*t: a *GridLayout*-hoz hasonlóan egy cellarácsban helyezi el a komponenseket, de egy komponens több cellát is elfoglalhat. Az AWT által biztosított elrendezésmenedzserek közül ez a típus ad az elrendezés szempontjából legnagyobb szabadságot. Ennek az „ára", hogy a beállításokat *GridBagConstraints* objektumok tulajdonságainak megadásával kell elvégeznünk. Bár ez „manuálisan" kényelmetlen lehet, egy vizuális szerkesztő megoldhatja a „problémát".
- *CardLayout*: a tárolót egy kártyapaklihoz hasonlóan kezeli, ahol mindig a legfelső komponens (kártya) látható. Működése hasonló a SWING esetében használható több fülből álló panel (*tabbed pane*) működéséhez.

Az elrendezésért felelős objektumok az általuk alkalmazott elrendezési stratégia alapján végzik a komponensek méretezését. Ehhez figyelembe veszik a komponensek tulajdonságait, illetve a tároló méreteit, de a döntő szempont az elrendezés típusa.

Például, a *BorderLayout* az északi és déli részre elhelyezett komponensek méreteit horizontálisan, a keleti és nyugati részre elhelyezett komponensek méreteit vertikálisan a tároló méreteihez igazítja, és a középső rész méreteit úgy határozza meg, hogy kitöltse a tároló teljes felületét.

A fenti ténynek következményeként megállapíthatjuk, hogy a komponensek méreteinek manuális meghatározásakor figyelembe kell vennünk az alkalmazott elrendezésmenedzser példány típusát. Egy jó tanács lehet, hogy a fix méretek megadása helyett (ez például a komponensek *setSize*, valamint *setBounds* metódusainak segítségével történhet) inkább alkalmazzuk azokat a metódusokat, amelyekkel a minimum, maximum, illetve kívánt méreteket adhatjuk meg (*setMinimumSize*, *setMaximumSize*, *setPreferredSize* metódusok). Megjegyzendő azonban, hogy így is az elrendezésmenedzser stratégiája a döntő.

Az előző részben említettük, hogy a komponenseknek megfelelő osztályok metódusainak segítségével megadható a komponensek pozíciója és mérete is. Felvetődhet akkor a kérdés, hogy miért van szükség az elrendezésmenedzserre? Miért nem adhatunk meg fix méreteket, pozíciókat? A válasz: megadhatunk (a tárolónak megfelelő elrendezésmenedzser objektumra mutató referenciát állíthatjuk *null* értékre), de ez az esetek többségében nem javasolt. A megértés érdekében gondoljunk egyszerűen arra a tipikus helyzetre, amikor a tárolót újraméretezzük, és szeretnénk, ha az új méreteknek megfelelően a tartalmazott komponensek méretei is változnának, illetve, ha szükséges, az elrendezésük is alkalmazkodjon az új helyzethez. Könnyen belátható, hogy az elrendezésmenedzserek alkalmazása itt nagyon megkönnyítheti a dolgunkat.

A felsorolt elrendezésmenedzserek talán túl „egyszerűnek" tűnhetnek, és felmerülhet a kérdés, hogy akkor miként alkalmazhatóak bonyolultabb felületek létrehozásakor. A megoldást a részekre bontás jelentheti. Tekintsük például a következő helyzetet: egy kereten belül szeretnénk elhelyezni középre egy rajzoló felületet és a felső, illetve alsó részekre egy-egy sor más komponenst (gombokat, címkéket, stb.), amelyek különböző műveletek elvégzését teszik lehetővé (törlés, szín kiválasztása, stb.). Egyik elrendezésmenedzser sem teszi lehetővé (a felsoroltak közül) a keret ilyenszerű felbontását, de könnyen megoldható a probléma. A kereten belül elhelyezünk két panelt az alsó, illetve felső részre, és ezekhez adjuk hozzá a komponenseket. A rajzvászon marad középen. A kereten belül *BorderLayout*-ot, a paneleken belül *FlowLayout*-ot alkalmazunk (maradunk tehát a tárolók alapértelmezett elrendezésénél). Hasonló módszerrel komplex felületek hozhatóak létre ezeknek az egyszerű elrendezésmenedzsereknek a segítségével. Természetesen, ha valami nagyon speciálisat szeretnénk, megvan a lehetőség külső osztályok alkalmazására, illetve új *LayoutManager* megvalósítások létrehozására.

Most, hogy megtanultuk létrehozni, és tárolókon belül rendezve elhelyezni a grafikus komponenseket, semmi akadálya, hogy elkészítsük az első felületünket.

3.1.4 Hello, GUI!

Első példaként készítsünk egy keretet, amelyen belül elhelyezünk két gombot (az alsó, illetve felső részre), és egy címkét (középre). Java-ban ezt a következő formában oldhatjuk meg:

```
import java.awt.Frame;
import java.awt.Button;
import java.awt.Label;
import java.awt.BorderLayout;
public class ExampleFrame extends Frame {
      private Button buttons[];
      private Label label;
      public ExampleFrame() {
            this.setTitle("Example");
            // gombok létrehozása
            buttons = new Button[2];
            buttons[0] = new Button("Button 1");
            this.add(buttons[0], BorderLayout.NORTH);
            buttons[1] = new Button("Button 2");
            this.add(buttons[1], BorderLayout.SOUTH);
            // a címke létrehozása
            label = new Label("Label");
            this.add(label, BorderLayout.CENTER);
      }
      public static void main(String[] args) {
            ExampleFrame f = new ExampleFrame();
            f.setBounds(50, 50, 200, 120);
            f.setVisible(true);
      }
}
```

Az első részben importáljuk a feladathoz szükséges AWT osztályokat: a keret létrehozásához a *Frame* osztályt, a címke és gombok létrehozásához a *Label* és *Button* osztályokat, valamint az elrendezésért felelős *BorderLayout* osztályt. Osztályunk az *ExampleFrame* nevet kapja, és a *Frame* osztálynak lesz leszármazottja. Az osztály attribútumai: egy *Button* típusú objektumokból álló tömb a gombok tárolására, és egy *Label* a címke létrehozásához. Ezeket a konstruktoron belül inicializáljuk. A *Frame* osztály *setTitle* metódusának segítségével egy címet adunk a keretnek, lefoglalunk helyet a két *Button* típusú referenciából álló tömbnek, sorban létrehozzuk a gombokat, példányosítva a *Button* osztályból, majd hozzáadjuk ezeket a tárolóhoz az *add* metódus segítségével. A *Button* konstruktora paraméterként kapja a gomb címkéjének szövegét, azaz azt a szöveget, amelyet a gombon látunk majd. Az *add* metódus első paramétere a tárolóhoz hozzáadandó komponensre mutató referencia, a második paraméter a *BorderLayout* osztályon belül definiált konstansok segítségével jelzi, hogy a tároló melyik részére szeretnénk elhelyezni az illető komponenst. A gombok példájához hasonlóan, a konstruktor utolsó két sorában létrehozzuk a címke objektumunkat, és hozzáadjuk a keret középső részéhez. Ezután a *main* metóduson belül példányosítunk az *ExampleFrame* osztályunkból, tehát tulajdonképpen létrehozzuk a keretünknek megfelelő objektumot. Beállítjuk a pozícióját

és méretét a *setBounds* metódus segítségével (a keret bal felső sarka a képernyőn az első két paraméter által megadott pozíción található majd, a harmadik és negyedik paraméter határozza meg a méreteit), és a *setVisible* metódus meghívásával láthatóvá tesszük a keretünket (a metódus paramétereként kapott *boolean* típusú kifejezés határozza meg az illető komponens láthatóságát).

Megfigyelhetjük, hogy a keretünkön belül nem hívtuk meg a *setLayout* metódust, az alapértelmezett elrendezésmenedzsert alkalmaztuk, amely a *Frame* esetében pontosan a *BorderLayout*. A kóddal kapcsolatban megemlíthetjük még, hogy a *setTitle* és *add* metódusok meghívásánál a *this* referencia használata opcionális. Ha elhagynánk, a metódushívás akkor is az aktuális példányra vonatkozna, a végeredményt nem befolyásolnánk. A példa esetében azért alkalmaztuk mégis ezt az írásmódot, hogy könnyebben érthetővé tegyük a kódot. A későbbiekben nem minden esetben ragaszkodunk ehhez.

Ha a programot Windows Vista alatt futtatjuk a 3.4 ábrán látható eredmény kapjuk (az operációs rendszer említése itt azért fontos, mert, mint ahogyan azt már magyaráztuk, az AWT komponensek megjelenítése platformspecifikus).

3.4 ábra: az *ExampleFrame* osztály futtatásának eredménye

Létrehoztuk az első egyszerű grafikus felületünket az AWT csomag segítségével. Csak az a bökkenő, hogy ebben a felületben egyelőre nem sok örömünk lehet. Egyszerűen nem működnek a dolgok. Hiába kattintunk a gombokra, semmi sem történik, sőt még bezárni sem tudjuk az ablakot (csak ha „kívülről" leállítjuk az egész programot). Szeretnénk orvosolni a problémát, működésre szeretnénk bírni a felületet. Ehhez azonban először meg kell ismernünk az eseménykezelés alapjait.

3.2 Eseménykezelés

A grafikus felületek működése eseménykezelésen (*event handling*) alapszik, de az eseménykezeléssel, vagy általánosabban az *Observer* tervezési mintával nem csak a grafikus komponensek esetében találkozunk. Egy nagyon széles körben alkalmazható modellről van szó.

3.2.1 Megfigyelők és megfigyeltek

A megfigyelt-megfigyelő modell alapja, hogy egy megfigyelt (*observable*) objektum megfigyelő (*observer*) objektumokat tart nyilván, és értesíti ezeket bizonyos állapotváltozásokról (eseményekről). Az események (*events*) tulajdonképpen üzenetek, esemény osztályok példányai, amelyek a megfelelő metódushívásokon keresztül jutnak el a megfigyelő objektumokhoz. Minden eseménynek van egy forrása (*source*), és lehet egy vagy több címzettje (a megfigyelő objektumok, vagy receptorok).

A grafikus komponensek esetében a működés tipikusan eseményalapú. Például, egy gomb „értesíti" az alkalmazást, ha a felhasználó rákattintott. A gomb az esemény forrása. Ő regisztrálja az esemény fogadásában és kezelésében „érdekelt" megfigyelőket. A regisztráció folyamán ezek az objektumok hozzáadódnak egy listához, majd a megfelelő metódushívások segítségével a forrás értesíti őket az illető esemény bekövetkeztéről.

A megfigyelők egy-egy *Listener* interfész megvalósításai. Az interfészekben vannak deklarálva azok a metódusok, amelyeket a forrás meghívhat az esemény bekövetkeztekor. A metódusok paraméterként kapják az esemény típusának megfelelő eseményosztályok egy példányát. Az interfészeket megvalósító figyelő objektumok konkrét implementációt adnak a metódusoknak (pl. meghatározzák, hogy mi történjen, ha a felhasználó kattintott az illető gombra).

Maradjunk a gombra kattintás példájánál, és nézzük meg, hogy miként kezelhető egy ilyen esemény Java-ban. Egyszerűen az egérkattintást is figyelhetnénk (az idekapcsolódó eseményosztályt és figyelő interfészeket a későbbiekben tárgyaljuk), de az AWT egy másik lehetőséget is kínál: az *ActionEvent* eseményosztály alkalmazását, amely esetünkben hasznosabb lehet. Az egyik lehetséges magyarázat a következő: megszoktuk, hogy egy grafikus felületen egy gombra nem csak az egérrel kattinthatunk, hanem, ha az illető gomb „fókuszban van" (lásd az *ExampleFrame* példa esetében „Button 1" gombot, aminek felületén a szaggatott körvonalú keret jelzi a fókuszt), akkor a *space* billentyű lenyomása is azonos hatást eredményez. Az *ActionEvent* eseménytípus alkalmazásával, ezt a lehetőséget is figyelembe vehetjük. Amint azt a későbbiekben még látni fogjuk, ilyen *ActionEvent* típusú esemény lép fel akkor is, amikor *entert* nyomunk egy szövegmezőn belül, illetve, amikor kiválasztjuk egy listának vagy menünek valamelyik elemét.

Azt mondtuk tehát, hogy minden eseménynek van egy forrása. Ez esetünkben egy gomb:

```
Button b = new Button("OK")
```

Az esemény *ActionEvent* típusú, és a megfigyelőknek az *ActionListener* interfészt kell implementálniuk. Ahhoz, hogy a forrás regisztrálhassa a megfigyelőket, egy ennek a célnak megfelelő metódust kell biztosítania. Ezt a *Button* osztály meg is teszi, a következő metóduson keresztül:

```
public void addActionListener(ActionListener listener)
```

Ha a forrás regisztrált egy megfigyelőt, azaz hozzáadta a megfigyelő objektumot a receptorok listájához, arra is lehetőséget kell biztosítania, hogy ez az objektum eltávolítható legyen ebből a listából. Ez akkor lesz fontos, ha az illető címzett egy adott pillanattól kezdve már nem „érdekelt" az események fogadásában.

A *Button* osztály *ActionEvent* típusú eseményeknek megfelelő „figyelő-eltávolító” metódusa a következő:

```
public void removeActionListener(ActionListener listener)
```

A megfigyelőknek meg kell valósítaniuk az *ActionEvent* eseménynek megfelelő *ActionListener* interfészt, és ennek megfelelően implementálniuk kell az *actionPerformed* metódust (ez az *ActionListener* interfészen belüli egyetlen metódus):

```
class Receptor implements ActionListener {
      public void actionPerformed(ActionEvent e) {
            //az esemény felléptekor végrehajtandó kód
            …
      }
}
```

A címzettet a megfelelő metódus meghívásával regisztrálhatjuk:

```
b.addActionListener(new Receptor());
```

Ettől kezdve a *Receptor* osztály példánya értesítést kap a gomb lenyomásakor, és végrehatja az *actionPerformed* metóduson belüli utasításokat.

3.2.2 AWT események

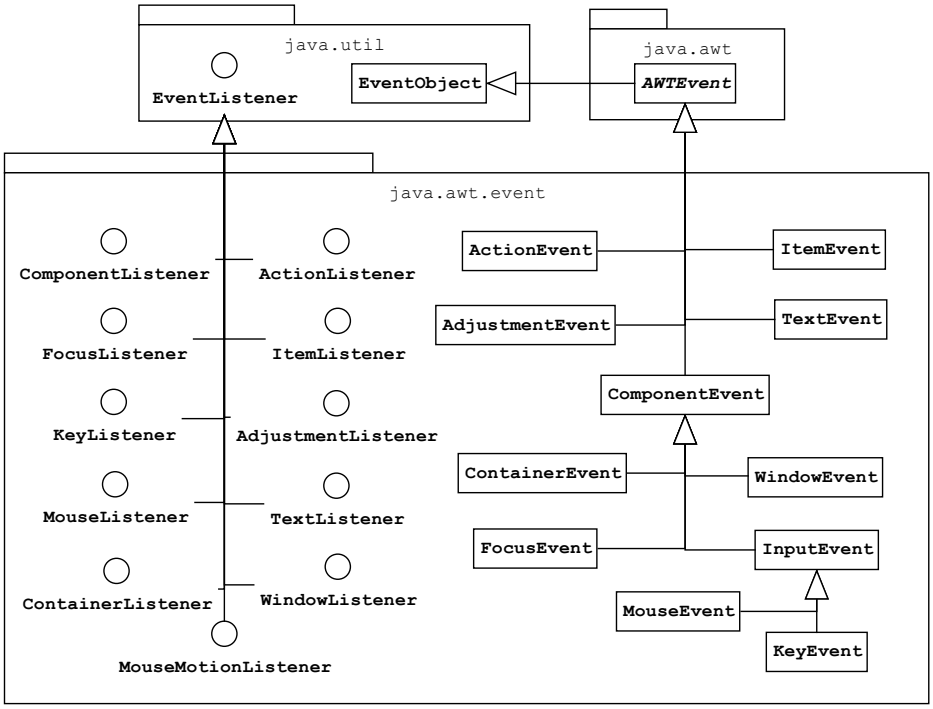
Az AWT komponensekkel kapcsolatos eseményeknek megfelelő osztályok, illetve figyelő interfészek az *awt* csomagon belüli *event* alcsomagban találhatóak.

Az eseményosztályok közös őse az *AWTEvent* absztrakt alaposztály, amely a *java.awt* csomagon belül található, és a *java.util* csomag *EventObject* osztályából származik. Ez a tény is jelzi, hogy az eseménykezelésen alapuló működés nem csak a grafikus komponensek jellemzője, hanem a modell más kontextusban is alkalmazható. A *java.awt.event* csomagban található figyelő interfészeknek szintén van egy *java.util* csomagon belüli közös őse, az *EventListener* interfész.

Az *AWTEvent* eseményből származnak a *java.awt.event* csomag konkrét eseményeknek megfelelő osztályai. A 3.5 ábra néhány gyakori AWT eseményosztályt és figyelő interfészt ábrázol, illetve azok hierarchiáját szemlélteti.

Vannak olyan események, amelyek forrása bármilyen grafikus komponens lehet, és vannak olyanok, amelyek csak bizonyos komponensek esetében léphetnek fel. A 3.1 táblázatban olyan eseményosztályokat sorolunk fel, amelyek bármilyen komponens esetében felléphetnek, és feltüntetjük az eseményeknek megfelelő *Listener* interfészeket, valamint az interfészekben deklarált metódusokat:

A metódusok esetében csak a metódusok neveit tüntettük fel, mivel a visszatérített érték típusa minden esetben *void*, és a paraméterlistában minden esetben a megfelelő eseményosztály egy példánya kap helyet.



3.5 ábra: a *java.awt.event* csomag részleges osztálydiagramja

Láthatjuk, hogy a metódusok nevei elég beszédesek, de azért adjunk egy-egy rövid magyarázatot, a felsorolás sorrendjében. A komponensek megjelenítéssel kapcsolatos tulajdonságainak változását figyelő *ComponentListener* esetében szó lehet a komponens újraméretezéséről, helyének változásáról, megjelenéséről vagy eltűnéséről. A fókuszeseményeket figyelő *FocusListener* interfész esetében, egy komponens fókuszba kerülhet, vagy kikerülhet a fókuszból. A billentyűzettel kapcsolatos eseményeket figyelő *KeyListener* figyeli, ha lenyomunk és felengedünk egy billentyűt, de külön is értesül csak a billentyű lenyomásáról, illetve felengedéséről. Ehhez hasonló a helyzet a *MouseListener* egérműveletekért felelős figyelőnél is: értesítést kapunk arról, ha kattintottunk (egérgomb lenyomása és felengedése a *pointer* mozdítása nélkül) az egérrel, de külön is figyelhetjük az egérgomb lenyomását, illetve felengedését. Ezen kívül a figyelő, még értesítést kap arról is, ha az egér *pointer* egy adott komponens fölé kerül, vagy elhagyja annak felületét. Az egér *pointer* mozgásának figyeléséért egy külön interfész, a *MouseMotionListener* felelős, és különbséget tehetünk aközött, hogy a *pointer* elmozdítása lenyomva tartott gombbal történt-e, vagy sem.

Egy adott esemény felléptekor a forrás mindig a regisztrált címzettek megfelelő metódusát hívja meg, paraméterként átadva az eseményobjektumot. Például, ha az egérrel belépünk egy komponens fölé és kattintunk, akkor először a *mouseEntered* metódus, majd utána sorban a *mousePressed*, *mouseReleased* és *mouseClicked* metódusok kerülnek meghívásra (figyelem: a kattintás esemény csak akkor lép fel, ha a *pointer* a lenyomás és felengedés között mozdulatlan marad).

Esemény	Figyelő interfész	Metódusok nevei
ComponentEvent	*ComponentListener*	*componentResized* *componentMoved* *componentShown* *componentHidden*
FocusEvent	*FocusListener*	*focusGained* *focusLost*
KeyEvent	*KeyListener*	*keyTyped* *keyPressed* *keyReleased*
MouseEvent	*MouseListener*	*mouseClicked* *mousePressed* *mouseReleased* *mouseEntered* *mouseExited*
MouseEvent	*MouseMotionListener*	*mouseDragged* *mouseMoved*

3.1 táblázat: általános AWT események

A címzettek az eseményosztály metódusainak segítségével juthatnak további információkhoz az illető eseményről. Például, egy egérművelet esetében lekérhetjük a pointer koordinátáit:

```
public void mousePressed(MouseEvent e) {
      System.out.println("Position: " + e.getX() + "," + e.getY());
}
```

Hasonlóan az eseményekről más információk is lekérdezhetőek.

Szintén általános eseménytípus a *ContainerEvent*, amely az előző táblázatban azért nem kapott helyet, mert csak tároló komponensek esetében léphet fel. Az eseménynek megfelelő figyelő interfész a *ContainerListener*, a figyelők a *componentAdded* és *componentRemoved* metódusokon keresztül értesülnek arról, ha egy komponens hozzá lett adva az illető tárolóhoz, vagy el lett távolítva belőle.

Az AWT események egy másik kategóriájába tartoznak azok az események, amelyeknek forrásai csak bizonyos komponenstípusok lehetnek. A 3.2 táblázatban ilyen eseményeket tüntettünk fel.

Az *ActionListener* típusú figyelők értesülnek arról, ha a felületen a felhasználó lenyomott egy gombot, *entert* nyomott egy szövegmezőn belül, illetve ha kiválasztotta egy listának vagy menünek valamelyik elemét. Az *ItemListener* típusú figyelők értesítést kapnak, ha egy listán, kiválasztó komponensen, vagy jelölőnégyzeten belül állapotváltozás történt (pl. kijelöltük a négyzetet, vagy kiválasztottunk valamit a listából). A görgetősávokkal kapcsolatos eseményeket az *AdjustmentListener* figyeli, egy szöveges komponens tartalmának változásáról a *TextListener* értesül. Az ablakokkal kapcsolatos eseményeket a *WindowListener* típusú

objektumok figyelik. Értesítést kapnak az ablak megnyitásakor, a bezárás kezdeményezésekor (amikor a felhasználó a bezárás ikonra kattint), a bezárás után (miután a *dispose()* metódushívás felszabadította a komponens által foglalt natív erőforrásokat), az ablak lekicsinyítésekor, vagy méreteinek visszaállításakor, illetve akkor, ha az ablak aktívvá, vagy inaktívvá válik.

Esemény	Előfordulás	Figyelő interfész	Metódusok nevei
ActionEvent	*Button* *TextField* *List* *MenuItem*	*ActionListener*	*actionPerformed*
ItemEvent	*List* *Checkbox* *Choice* *CheckboxMenuItem*	*ItemListener*	*itemStateChanged*
AdjustmentEvent	*Scrollbar* *ScrollPane*	*AdjustmentListener*	*adjustmentValueChanged*
TextEvent	*TextField* *TextArea*	*TextListener*	*textValueChanged*
WindowEvent	*Frame* *Dialog*	*WindowListener*	*windowOpened* *windowClosing* *windowClosed* *windowIconified* *windowDeiconified* *windowActivated* *windowDeactivated*

3.2 táblázat: bizonyos komponensekre jellemző események

Az előző táblázatokban nem kapott helyet, de fontos szerepet játszik az *InputEvent* eseményosztály, amely a *MouseEvent* és *KeyEvent* egérrel, illetve billentyűzettel kapcsolatos eseményosztályok közös őse. Segítségével információkhoz juthatunk a fellépő eseményekről. Az osztály konstansokat definiál, amelyek bináris maszkként alkalmazhatóak a különböző módosítók lekérdezésénél (pl. *SHIFT_MASK, ALT_MASK, BUTTON1_MASK, BUTTON2_MASK*, stb.). Segítségükkel egyszerűen megtudhatjuk például, hogy egy adott billentyű, vagy egérgomb lenyomása közben a felhasználó lenyomott-e valamilyen kontrollbillentyűt:

```
public void mousePressed(MouseEvent e) {
      int mods = e.getModifiers();
      if ((mods & InputEvent.SHIFT_MASK) != 0) {
            // SHIFT lenyomva
            ...
      }
}
```

A fenti ismeretek alapján már ki tudnánk egészíteni az előbbi alfejezetben bemutatott példánkat. Először azonban, tekintsünk át még néhány elvet és alapfogalmat.

3.2.3 Modell, nézet és vezérlő

A Modell-Nézet-Vezérlő (MVC: *Model-View-Controller*) szerkezeti tervezési minta, egy alapelv, amelyet összetettebb, grafikus felhasználói felületekkel rendelkező alkalmazások esetében érdemes alkalmazni. Az ilyen alkalmazásoknál általános cél az adatok, a logika és a megjelenítés szétválasztása. Például, ha valamit változtatnunk kell a felhasználói felületen, az ne vonjon maga után változtatásokat a modellben (az adatok szerkezetében).

A modell az alkalmazás által kezelt adatok ábrázolása. Például, egy gomb esetében a modell egy logikai érték lehet (*true*, ha a gomb le van nyomva, *false* egyébként). Hasonlóan egy fizikai személyek adatait nyilvántartó alkalmazás esetében a *Person* osztály (lásd előző példáinkat) lehetne a modell része.

A megjelenítés (*view*) a modell „képe" a felhasználói felületen. Ha például létre kellene hoznunk egy gomb modellnek megfelelő komponenst, eljárhatnánk a következő módon: egy panelt használnánk, amelyen elhelyeznénk egy címkét. A szürke hátterű panelre egy keretet rajzolnánk. Alaphelyzetben (ha a gomb nincs benyomva, tehát a modellnek megfelelő logikai érték *false*) a felső és baloldali részét a keretnek fehér vonallal, a többi részét fekete vonallal ábrázolnánk, így érve el a térhatást. A gomb lenyomásakor a szélek színezését megcserélnénk. Ezzel az egyszerű eljárással létre tudunk hozni egy egyszerű gomb komponenst, de könnyen belátható, hogy más megközelítés is elképzelhető. Ami lényeges: a megjelenítés hogyanjának nem kell befolyásolnia a modellt. Hasonlóan egy személy adatait megjeleníthetjük egy táblázatban, de ugyanúgy megjeleníthetjük egy szövegmezőket tartalmazó grafikus felületen.

A vezérlő (*controller*) határozza meg a működést: feldolgozza a felülethez érkező eseményeket, és módosításokat végezhet mind a modellben (az értékek az esemény hatására változhatnak), mind a felületen (a felület frissítésre szorulhat, például az értékek változása után). Példánk esetében a vezérlő határozná meg, hogy mi történjen a gomb lenyomásának hatására, milyen utasítások legyenek végrehajtva. Hasonlóan a nyilvántartó esetében, ha a felületen változtatunk és mentjük a változásokat, a vezérlőnek kell gondoskodnia a tárolt személyi adatok frissítéséről, és esetleg a nézet változtatásáról.

A vezérlés tipikusan a következő módon történik: a felhasználó kölcsönhatásba lép a felülettel (pl. lenyom egy gombot), egy figyelő közvetítésével a vezérlő átveszi az eseményt, frissíti a modellt, majd a modell alapján frissíti a felületet, amely ezután újabb eseményekre vár. Fontos kiemelni, hogy a modellnek nem kell „tudnia" a nézetről, vagy vezérlőről.

Már az előzőekben, az interfészekkel kapcsolatos résznél is említettük, hogy milyen fontos egy alkalmazás esetében a fölösleges függőségi viszonyok kiküszöbölése. Ezért fontos az MVC elv betartása is. Természetesen az első példáink annyira egyszerűek lesznek, hogy valószínűleg nem látjuk majd értelmét a „feldarabolásuknak", de a továbbiakban minden esetben fontos az MVC minta betartása. Tekintsünk néhány példát.

A fentebb leírt ismereteket felhasználva már könnyen kiegészíthetnénk az első alfejezetben leírt *ExampleFrame* osztályunkat. Például, ha azt szeretnénk, hogy a gombok lenyomásának hatására a címkén megjelenjen a lenyomott gomb címkéjének szövege, egyszerűen úgy

alakítanánk át az osztályunkat, hogy valósítsa meg az *ActionListener* interfészt, kiegészítve az *actionPerformed* metódussal a következő módon:

```
public void actionPerformed(ActionEvent e) {
      label.setText(e.getActionCommand());
}
```

Az *ActionEvent* osztály *getActionCommand* metódusa a gombok esetében pontosan a gombok címkéjének szövegét téríti vissza. A konstruktoron belül, miután létrehoztuk a gombokat, hozzájuk rendeljük a figyelőt, azaz az osztályunk aktuális példányát:

```
buttons[0].addActionListener(this);
buttons[1].addActionListener(this);
```

Meg is vagyunk: a gombok lenyomásának hatására most már megjelenne a címkén a gombok címkéjének megfelelő "Button 1", vagy "Button 2" szöveg.

Természetesen, valószínűbb, hogy a különböző gombokhoz különböző funkcionalitásokat akarunk hozzárendelni. Példánk esetében bonyolíthatjuk a feladatot azzal, hogy ne egyszerűen a gombok címkéjének szövegét írjuk ki, hanem egy más szöveget, például az első gomb esetében az "első", a második gomb esetében a "második" stringet. Aránylag egyszerűen módosíthatjuk az *actionPerfomed* metódust, a törzsén belül ellenőrizve az esemény forrását:

```
public void actionPerformed(ActionEvent e) {
      if (e.getSource() == buttons[0]) label.setText("elso");
      else label.setText("masodik");
}
```

Megfigyelhetjük, hogy itt már a forrás azonosítását nem a címke szövegének (*getActionCommand* metódus), hanem a forrás objektumra mutató referenciának (*getSource* metódus) segítségével végeztük. Egy felület komponenseinek szövege változhat. Gondoljunk például a többnyelvű alkalmazásokra. Nem jó ötlet az aktuális megjelenítéshez (ide tartozik a címkék szövege is) kötni a vezérlést.

Még mindig könnyű dolgunk volt, de lépjünk tovább. Képzeljük el, hogy a gombok lenyomásának valami sokkal bonyolultabb műveletsor elvégzését kellene eredményeznie. Vagy tételezzük fel, hogy egy bonyolultabb felületről van szó, ahol sokkal több gombunk és egyéb komponensünk van. Ha a különböző források beazonosításának és a kapcsolódó műveletek elvégzésének megfelelő hosszú kódot szintén az *actionPerformed* metóduson, illetve a felület elemeit megjelenítő osztályon belül helyeznénk el, alaposan megsértenénk az MVC alapelvet, és egy rosszul strukturált, nehezen átlátható kódot kapnánk. A megoldás az, hogy a különböző forrásokhoz különböző figyelő objektumokat rendelünk hozzá.

A példánk esetében a két gomb részére készítsünk egy-egy figyelő osztályt:

```
class MyFirstListener implements ActionListener {
      public void actionPerformed(ActionEvent e) {
            // az első gomb lenyomásakor végrehajtandó kód
            …
      }
}
```

Hasonló módon létrehozzuk a második gombnak megfelelő *MySecondListener* osztályt is. Ezután a gombokhoz az osztályok egy-egy példányát rendeljük hozzá figyelőként:

```
buttons[0].addActionListener(new MyFirstListener());
buttons[1].addActionListener(new MySecondListener());
```

Könnyen belátható, hogy a fenti konstrukció arra az esetre is megoldást szolgáltat, amikor egy adott figyelőt több különböző forráshoz is hozzá akarunk rendelni. Például, azt szeretnénk, hogy egy menü valamelyik elemének kiválasztása ugyanazt a hatást váltsa ki, mint a gomb lenyomása. Egyszerűen úgy példányosítunk a megfelelő figyelő osztályból, hogy rendelkezzünk egy rámutató referenciával, és ezt a referenciát adjuk át paraméterként a regisztráló metódusoknak.

Lehetséges, hogy a vezérlést, az MVC elvnek megfelelően egy (vagy több) külön osztályra szeretnénk bízni. A figyelő osztályban deklarálhatunk egy vezérlő típusú referenciát, amelyet a konstruktorban inicializálnánk (átadjuk a vezérlő osztály példányára mutató referenciát), és az *actionPerformed* metóduson belül a referencia segítségével meghívjuk a vezérlő osztály megfelelő metódusait.

3.2.4 Adapter osztályok

Az első példák esetében könnyű dolgunk volt: még akkor is, amikor maga az osztályunk volt a figyelő, az *ActionListener* interfész megvalósítása. Csak egyetlen metódust, az *actionPerformed* metódust kellett implementálnunk, de nem minden esetben ilyen egyszerű a helyzet. Gondoljunk azokra a figyelő interfészekre, amelyek több metódust is deklarálnak (például a *MouseListener* vagy *WindowListener* interfészek). Megtörténhet, hogy adott esetben csak egy-két metódus megvalósításában vagyunk érdekeltek, de, ha a figyelő osztályunk megvalósítja az interfészeket, kötelességünk minden metódust implementálni. Persze megtehetjük, hogy egyszerűen üresen hagyjuk a feladat szempontjából hanyagolható metódusok törzsét, vagy kivételt váltunk ki a metódusokon belül, de ez szükségtelenül bonyolítaná a kódunkat. A megoldást *adapter* osztályok alkalmazása jelentheti.

Általánosan az adapterek olyan osztályok, amelyek megvalósítanak egy vagy több interfészt, de csak makett (*dummy*) implementációt adnak a metódusoknak (például üresen hagyják a metódusok törzsét). Az AWT csomag minden több metódust deklaráló figyelő interfész mellé biztosít adapter osztályt.

Az *ExampleFrame* osztály esetében az előbbiekben működésre bírtuk a gombokat, de még mindig nem tudjuk bezárni az ablakot. Megoldhatnánk a helyzetet úgy, hogy az osztályunk megvalósítja a *WindowListener* interfészt, és a *windowClosing* metóduson belül leállítjuk a program futását (ilyen esetben a felület által foglalt natív erőforrások is felszabadulnak). Kényelmetlenséget okoz viszont, hogy ebben az esetben implementálnunk kellene a *WindowListener* többi metódusát is (ablak megnyitása, kicsinyítése, aktiválódása, stb.), pedig erre a feladat szempontjából semmi szükség. A megoldást a *WindowListener* interfésznek megfelelő *WindowAdapter* osztály alkalmazása jelentheti. A konstruktor végére beszúrhatjuk az alábbi kódrészletet:

```
this.addWindowListener(new WindowAdapter() {
      public void windowClosing(WindowEvent e) {
            System.exit(0);
      }
});
```

Ismerős a kódrészlet. Találkoztunk már vele az előző fejezet névnélküli belső osztályokkal foglalkozó részénél. Remélhetőleg most már érthetőbb a konstrukció. Ismételjük át! A *WindowAdapter* osztályból származtatunk egy osztályt, amelyet névnélküli belső osztályként hozunk létre, és egy példányát regisztráljuk (a *Frame* megfelelő metódusával), mint figyelőt. Az osztályon belül felülírjuk a *windowClosing* metódust, amelyen belül a *System* osztály *exit* statikus metódusát használjuk a program leállítására, 0 kilépési kóddal.

Természetesen nem kötelező névnélküli belső osztályt használni, és „helyben" példányosítani. Ugyanígy megjegyezhető, hogy a belső osztály alkalmazása azokban az esetekben is lehetséges, amikor a figyelő interfész csak egyetlen metódust tartalmaz, és az AWT nem biztosít neki megfelelő adapter osztályt:

```
buttons[0].addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
            //gomb lenyomásakor végrehajtandó kód
            …
      }
});
```

Mint a legtöbb esetben, ezekben a helyzetekben is az aktuális feladat függvényében kell kiválasztanunk a legmegfelelőbb megoldást.

3.2.5 Hello újra, GUI!

Az előzőekben ismertetett eseménykezeléssel kapcsolatos fogalmak összefoglalásaként egészítsük ki, tegyük működőképessé az előző fejezet *ExampleFrame* példáját. Az egyszerűség kedvéért maradjunk annál a változatnál, amikor a címkére a gombok felületén látható szöveget írjuk ki.

```
import java.awt.Frame;
import java.awt.Button;
import java.awt.Label;
import java.awt.BorderLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.awt.event.WindowAdapter;
import java.awt.event.WindowEvent;
public class ExampleFrame extends Frame implements ActionListener {
      private Button buttons[];
      private Label label;
      public ExampleFrame() {
            setTitle("Example");
            // gombok létrehozása
```

```
            buttons = new Button[2];
            buttons[0] = new Button("Button 1");
            buttons[0].addActionListener(this);
            add(buttons[0], BorderLayout.NORTH);
            buttons[1] = new Button("Button 2");
            buttons[1].addActionListener(this);
            add(buttons[1], BorderLayout.SOUTH);
            // a címke létrehozása
            label = new Label("Label");
            add(label, BorderLayout.CENTER);
            addWindowListener(new WindowAdapter() {
                  public void windowClosing(WindowEvent e) {
                        System.exit(0);
                  }
            });
      }
      public void actionPerformed(ActionEvent e) {
            label.setText(e.getActionCommand());
      }
      public static void main(String[] args) {
            ExampleFrame f = new ExampleFrame();
            f.setBounds(50,50, 200, 120);
            f.setVisible(true);
      }
}
```

A forrásállomány első részét kiegészítettük a szükséges importokkal, a *java.awt.event* csomag felhasznált osztályaival és interfészével. Az osztályunk megvalósítja az *ActionListener* interfészt, és ennek megfelelően implementálja az *actionPerformed* metódust, amelyen belül a *getActionCommand* metódushívás segítségével lekérdezi a lenyomott gomb szövegét és kiírja azt a címkére. A konstruktoron belül a gombok létrehozása után hozzájuk rendeljük a figyelő objektumot, azaz az osztályunk aktuális példányát. A konstruktor utolsó részében névnélküli belső osztályként létrehozunk egy *WindowAdapter* osztályból származtatott osztályt, és ezen belül újradefiniáljuk a *windowClosing* metódust, olyan módon, hogy az a *System* osztály *exit* metódusának segítségével kilépjen az alkalmazásból. Megjegyzendő, hogy ezt az utolsó részt a *main* függvényen belül is elhelyezhettük volna (az *f* referencia segítségével hívva meg a regisztráló metódust). Ez a megoldás akkor lenne megfelelőbb, ha nem szeretnénk, hogy az ablakunk minden esetben azonos módon viselkedjen, és lezárásakor leállítsa a teljes programot.

3.3 Javasolt gyakorlatok

1. Hozzunk létre egy keretet (*Frame*), és ezen belül helyezzünk el egy panelt, valamint egy címkét (*Label*). Ha a felhasználó a panelre kattint az egérrel, a címkén jelenítsük meg az egérkattintás koordinátáit. A komponensek elhelyezésére használjunk egy megfelelő *LayoutManager* példányt, ne rögzítsük a pozíciókat és méreteket (ez a javaslat a legutolsó kivételével a következő feladatokra is érvényes).

   Egészítsük ki a programot, olyan módon, hogy ne csak az egérkattintást figyeljük, hanem az egérmutató mozgását is. A címkére az aktuális esemény típusát is írjuk ki a koordináták mellé: amennyiben a felhasználó kattintott a „*clicked*" üzenet, amennyiben csak elmozdította a pointert a „*moved*" üzenet, amennyiben lenyomott gombbal mozdította a pointert a „*dragged*" üzenet jelenjen meg.

2. Hozzunk létre egy keretet, és ezen belül helyezzünk el egy többsoros szöveg megjelenítésére alkalmas komponenst (*TextArea*), egy egysoros szöveg bevitelére alkalmas szövegmezőt (*TextField*), valamint egy gombot. Ha a felhasználó a gombra kattint, vagy a szövegmezőn belül lenyomja az *enter* billentyűt, a szövegmező tartalmát hozzáadjuk a *TextArea* tartalmához, majd töröljük a szövegmezőből (lehetőséget adva egy új szöveg beírására).

3. Egy kereten belül egy címke szövegét változtassuk jelölőnégyzetek (*Checkbox*) segítségével: a címkén mindig az aktuálisan kijelölt jelölőnégyzeteknek megfelelő címkék szövegét jelenítsük meg. Alakítsuk át a programot, olyan módon, hogy egyszerre csak egy jelölőnégyzet legyen kiválasztható (a jelölőnégyzet komponensek „*radio button*" komponensekbe történő alakítása, a *CheckboxGroup* osztály segítségével).

4. Egy kereten belül helyezzünk el egy gombot, véletlenszerűen generált koordinátákra, a „*Push me!*" felirattal. Amikor a felhasználó megpróbál a gombra kattintani (az egérmutató a gomb fölé kerül), a gomb elmozdul (véletlenszerűen újrageneráljuk a koordinátákat, és áthelyezzük a gombot az új koordinátákra). A feladatnak elkészíthetjük egy olyan változatát is, amikor tényleg nem lehetséges a gomb lenyomása: az új koordináták generálásánál kiszűrjük annak a lehetőségét, hogy a gomb újra a mutató alá kerüljön.

   Útmutatás: a koordináták véletlenszerű generálásához a *java.util.Random* osztályt használhatjuk (figyelem: a generátorból egyetlen példány elégséges, ezután ettől több érték is elkérhető a megfelelő metódusok meghívásával, ezért ne hozzunk létre minden változtatáskor egy új *Random* példányt). A feladatot úgy oldhatjuk meg legegyszerűbben, ha ezúttal (kivételesen) nem használunk *LayoutManager* példányt (a *setLayout* metódus *null* paramétert is elfogad), hanem meghatározzuk a gomb méretét és pozícióját (ez utóbbit változtatva a megfelelő esemény felléptekor).

4

Grafika és Applet

A fejezet első részében megismerkedünk az AWT komponensek megjelenítési mechanizmusával, valamint a Java által biztosított grafikával kapcsolatos lehetőségek alapjaival [8]. Megtanulunk rajzolni, és ezáltal speciálisabb külalakkal rendelkező komponenseket létrehozni. A második részben röviden tárgyaljuk a weboldalakba ágyazható egyszerű grafikus alkalmazások készítésének lehetőségét, az Applet osztály használatát [9]. A számítógépes alkalmazásoknál általános követelmény a nemzetköziesítés (*internationalization*, i18n) [10] és lokalizáció (*localization*, L10n), és a Java hatékony eszközöket biztosít a feladat megoldására. Ezeket mutatjuk be a fejezet harmadik részében.

4.1 AWT grafika

Ha grafikus felhasználói felületekről, és általánosan számítógépes grafikáról beszélünk, minden esetben „képbe jön" a renderelés (*rendering*) fogalma. Programozóként, ha grafikus felületeket hozunk létre, akkor mindig a grafikus elemek egy absztrakt reprezentációjával dolgozunk, olyan adatstruktúrákkal, amelyek az illető elemek tulajdonságait írják le. Olyan fogalmakat használunk, mint geometriai alakzatok, felületek, színek, textúrák, fények, pozíciók, méretek, betűtípusok, stb. Amit a felhasználó lát: a kimeneti eszközön (amely tipikusan a képernyő, de lehet például nyomtató is) megjelenő kép. Ez a kép különböző színű pixelekből áll, és pontos „külalakja" (amit látunk) a kimeneti eszköz tulajdonságaitól (például felbontásától) függ. A kép a modell alapján készül, de a konkrét megjelenítését egyéb tényezők is befolyásolják. A folyamatot, amely a belső reprezentációt a megjelenítendő képbe képezi, renderelésnek nevezzük. A renderelés tehát egy adott modell alapján létrehoz egy digitális képet, bittérképet (*bitmap/raster image*).

Gondoljunk például egy egyszerű gombra, amit mi a *Button* osztályból példányosíthatunk, majd a megfelelő metódushívásokkal beállíthatjuk bizonyos tulajdonságait (háttérszín, méret, stb.). Amit a felhasználó lát, az egy egyszerű, színes (háttérszínű) pixelekkel kitöltött kisebb-nagyobb (felbontás függvényében) téglalap, amelyen ír valamit. A modell leképezését a látható bittérképbe egy natív (mivel a megjelenítés platformfüggő) renderelési mechanizmus biztosítja.

Általában elegendő számunkra, ha a grafikus eszköztárak által biztosított standard komponenseket használjuk fel a felszínek létrehozásához. Előfordulhatnak azonban olyan esetek, amikor valamilyen speciálisabb megjelenítésű komponenseket akarunk létrehozni (például grafikus elemeket elhelyezve a komponens felületén), vagy egyszerűen csak „rajzolni" akarunk. Azt már sejthetjük, hogy erre lehetőség van, hiszen az előző fejezetben említettük a rajzvászon (*Canvas*) komponenst. A kérdés, hogy hogyan történik ez a gyakorlatban.

A Java több kiváló lehetőséget is biztosít számunkra grafikus elemek létrehozására. A kétdimenziós grafika támogatására rendelkezésünkre áll a Java 2D API, a háromdimenziós grafika támogatására a Java 3D API, így gyakorlatilag bármilyen grafikus alkalmazást el tudunk készíteni. Ezen kívül természetesen használhatunk külső modulokat, más nyelvekben létrehozott grafikus részeket, és ebben több API is támogat (példaként megemlíthető a Java OpenGL API, a JOGL). Ennek a jegyzetnek nem célja ismertetni az idekapcsolódó részleteket, fontos viszont néhány alapfogalom bemutatása.

4.1.1 A *Graphics* osztály

A Java nyelvben a grafika alapja a *Graphics* absztrakt alaposztály. Ez az osztály teszi lehetővé az alkalmazások számára, hogy különböző komponensek felületére rajzoljanak. Az osztály a Java által támogatott alapvető renderelési műveletek elvégzéséhez szükséges információkat, tulajdonságokat tárolja (forma, szín, betűtípus, méret, pozíció, stb.), valamint metódusokat tartalmaz ezek beállítására, és különböző alakzatok kirajzolására. Néhány példa:

```
void setColor(Color c)
void drawRect(int x, int y, int width, int height)
void fillRect(int x, int y, int width, int height)
```

Az első metódussal beállíthatjuk az aktuális színt, egy *java.awt.Color* típusú objektumot adva át paraméterként. A második metódus egy téglalapot rajzol ki a megadott koordináták és méretek szerint. A harmadik metódus kitölti az aktuális színnel a paraméterek által meghatározott téglalap alakú felületet. Hasonló módon természetesen beállíthatunk más tulajdonságokat, például megadhatjuk a betűtípust (*setFont*), rajzolhatunk más alakzatokat, például ellipszist, vagy kört (*drawOval*), tetszőleges alakzatot (*drawPolygon*), kirajzolhatunk szöveget (pl. *drawString*), vagy képet (*drawImage*). Nem lenne értelme minden lehetőséget felsorolni, de a későbbi példák esetében néhánnyal találkozunk.

A *Graphics* osztály közvetlen leszármazottja a *Graphics2D*, amely további lehetőségeket biztosít, például hatékonyabb módszereket a színkezelésre, transzformációkra, stb.

Természetesen, mint ahogy a *Graphics* osztály (és metódusainak többsége) absztrakt, a *Graphics2D* osztály is absztrakt. Könnyen belátható, hogy miért: a renderelés folyamata platform-specifikus, natív kód segítségével történhet. Ennek megfelelően a különböző platformokra írt JVM-ek különböző implementációkat biztosítanak ezeknek az osztályoknak a megvalósítására, natív mechanizmusokat alkalmazva a platform ablakrendszerével történő „együttműködésre".

4.1.2 AWT komponensek megjelenítése és frissítése

Egy AWT komponens felületének frissítése kétféle módon történhet: a rendszer kezdeményezésére (*system-triggered painting*), vagy az alkalmazás kezdeményezésére (*application-triggered painting*). Az első esetről beszélünk akkor, amikor az ablakrendszer kéri az illető komponens frissítését. Ez akkor fordul elő, amikor a komponens először válik láthatóvá, újraméreteződik, vagy valamilyen okból „sérül" a felülete (pl. föléje helyezünk, majd eltávolítunk egy másik komponenst, vagy lekicsinyítjük, majd visszaállítjuk a komponenst tartalmazó ablakot).

A második eset akkor fordul elő, amikor az alkalmazás valamilyen belső állapotváltozás következményeként úgy dönt, hogy a komponens felülete frissítésre szorul.

A komponensek felületének kirajzolása egy visszahívásos mechanizmuson (*callback*) alapszik: a renderelésért felelős programrészt egy adott metóduson belül kell elhelyezni, és ezt a metódust hívja meg a *toolkit* a komponens felületének kirajzolásakor, vagy frissítésekor. Ez a metódus a *Component* osztály *paint* metódusa, melyet a különböző komponensek a nekik megfelelő módon újradefiniálnak:

```
public void paint(Graphics g)
```

A metódus paraméterként kap egy *Graphics* objektumot, amely a komponens felületének grafikus modellje. A paraméter határozza meg a grafikus megjelenítéssel kapcsolatos tulajdonságokat (szín, betűtípus stb.), illetve a frissítendő felületet (azt a részt, amely újra lesz renderelve) is behatárolja (*clipping*). A tulajdonságok a *paint* metóduson belül természetesen módosíthatóak, a módosítások, illetve a rajzolási műveletek a *g* referencia segítségével történhetnek.

A frissítés szempontjából különbség van a rendszer, illetve alkalmazás általi frissítés között. A rendszer által kezdeményezett frissítés esetében a rendszer behatárolja a „sérült" részt (amely a teljes felület is lehet), és meghívja a *paint* metódust. A rendszer feltételezi, hogy a sérült felület teljes frissítésre szorul, és minden pixelt frissít (tulajdonképpen törli és újrarajzolja a felületet).

Az alkalmazás általi frissítés kezdeményezése a *repaint* metódus meghívásával történik. A *repaint* egy a komponens felületének frissítésére vonatkozó aszinkron kérés a *toolkit*-nek címezve. A metódus meghívása nem vezet azonnali *paint* metódushíváshoz. A kérés teljesítésekor a *toolkit* először meghívja a komponens *update* metódusát:

```
public void update(Graphics g)
```

A metódus alapértelmezett implementációja törli a felületet és meghívja a *paint* metódust, tehát teljesen újrarajzolja az érintett felületet, az „üresen" maradt részeket háttérszínű pixelekkel töltve ki. Az alapvető különbség az előbbi esethez képest abból ered, hogy az *update* újradefiniálható. A mechanizmus lehetőséget ad arra, hogy ne feltétlenül kelljen törölnünk az érintett felületet, és így például egy már kirajzolt részt új grafikus elemekkel egészíthetünk ki (*incremental painting*).

A *repaint* metódusnak különböző változatai vannak: meghívhatjuk paraméterek nélkül (a komponens teljes felületének frissítésekor), paraméterek segítségével behatárolhatjuk a frissítendő részt, illetve megadhatunk egy „határidőt" (mennyi időn belül következzen a *paint* hívás).

Mivel a *repaint* eredménye egy aszinkron kérés, amely nem feltétlenül vezet azonnali *update/paint* híváshoz, megtörténhet, hogy mielőtt az első kérésnek megfelelő *paint* metódushívás megtörténne, további kérések érkeznek. Az ilyen esetekben a több kérés egyetlen *paint* hívásba lesz összevonva. Ez a mechanizmus különösen hasznos lehet akkor, amikor az alkalmazáson belül egymás után (például egy cikluson belül) több apró frissítést szeretnénk elvégezni.

Fontos kihangsúlyozni, hogy az alkalmazás által kezdeményezett frissítéseknél sohasem hívjuk meg közvetlen módon a *paint* metódust. A frissítést minden esetben a *repaint* metódushíváson keresztül kérjük. Ez természetes is, tudva azt, hogy a *paint* paraméterének típusa

Graphics, és absztrakt osztályból nem példányosíthatunk. A *paint* direkt módon történő meghívása legfeljebb az újradefiniált *update* metóduson belül történhet (továbbadva a metódusnak az *update* paraméterét).

A grafikus komponensek mindegyike újradefiniálja a neki megfelelő módon a *paint* metódust. Ha valami speciális megjelenítést szeretnénk, akkor ezt a származtatott osztályainkban mi is megtehetjük. Ha csak ki szeretnénk egészíteni a komponensek grafikus tartalmát, akkor először a *paint*-en belül a *super* referencia segítségével meghívjuk az alaposztály *paint* metódusát, és ezt követhetik a saját utasítások (ez különösen hasznos lehet tárolók megfelelő megjelenítésének esetében, amiről még szó lesz a továbbiakban). Ha egyszerűen csak rajzolni szeretnénk, akkor használhatjuk a rajzvászon (*Canvas*) osztályt, ebből származtatva saját komponensünket, felülírva a *paint* metódust. Ezzel kapcsolatban mutatunk be egy példát.

4.1.3 Kép és vászon

Példaként tekintsük a következő feladatot: egy kereten belül helyezzünk el egy rajzvásznat, és adjunk lehetőséget arra, hogy a felhasználó az egér segítségével kis köröket rajzolhasson ki erre a vászonra.

```
import java.awt.BorderLayout;
import java.awt.Canvas;
import java.awt.Color;
import java.awt.event.MouseAdapter;
import java.awt.event.MouseEvent;
import java.awt.Graphics;
import java.awt.Frame;
import java.awt.event.WindowAdapter;
import java.awt.event.WindowEvent;

public class MyCanvas extends Canvas {

      private int x = 0;
      private int y = 0;

      public MyCanvas() {
            setBackground(new Color(50,100,250));
            addMouseListener(new MouseAdapter() {
                  public void mousePressed(MouseEvent e) {
                        x = e.getX();
                        y = e.getY();
                        repaint();
                  }
            });
      }

      public void paint(Graphics g) {
            g.setColor(Color.red);
            g.fillOval(x,y,20,20);
      }
```

```
        public static void main(String args[]) {
              Frame f = new Frame("Paint");
              f.setBounds(50,50,300,200);
              f.add(new MyCanvas(), BorderLayout.CENTER);
              f.addWindowListener(new WindowAdapter() {
                    public void windowClosing(WindowEvent e) {
                          System.exit(0);
                    }
              });
              f.setVisible(true);
        }
  }
```

A kód első részében importáljuk a szükséges osztályokat. Többnyire már ismerősek: a keret létrehozását, illetve az események kezelését szolgálják. Ami „újdonság" az a rajzvászon létrehozásához szükséges *Canvas* osztály, ebből származtatjuk majd az új osztályunkat, a rajzoláshoz szükséges *Graphics* osztály és a színek beállításához szükséges *Color* osztály.

Két egész típusú attribútumunk van, az *x* és *y*, az egérkattintás koordinátáinak rögzítésére. A konstruktoron belül a *setBackground* metódussal meghatározzuk a vásznunk háttérszínét. Egy új *Color* objektumot hozunk létre, a paraméterként megadott RGB komponensekkel (a hátterünk kékes színű lesz). Ezután a *MouseAdapter* osztályból származtatva egy figyelő osztályt hozunk létre, amelyen belül újradefiniáljuk a *mousePressed* metódust. Így a felület az egérgomb lenyomására fog reagálni (nem feltétel a kattintás, így az sem, hogy a *pointer* a gomb lenyomása és felengedése között mozdulatlan maradjon). A metóduson belül lekérjük a *pointer* aktuális koordinátáit és eltároljuk azokat az attribútumainkban, majd a *repaint* metódushívással kérjük a felületünk (a teljes vászon) frissítését. Ez a kérés a *paint* metódus meghívásához vezet, és ezen a metóduson belül végezzük el a rajzolással kapcsolatos műveleteket. Beállítjuk az aktuális színt a *setColor* metódus segítségével (szintén a *Color* osztály egy példányát adjuk meg paraméterként, de most az osztályon belül definiált standard színek közül választunk egyet, a piros színt), majd a *fillOval* metódussal rajzolunk. A metódus egy ellipszist fog kitölteni az aktuális színnel (piros). Paraméterként megadjuk az alakzatot magába foglaló téglalap alakú felület bal felső sarkának koordinátáit (az attribútumok aktuális értékei, tehát az egér *pointer* aktuális koordinátái), illetve méreteit. Mivel a méretek megegyeznek, egy négyzet alakú felületet határoztunk meg, így az eredményül kapott alakzat egy kis piros kör lesz.

A keretet a *main* metóduson belül hozzuk létre a *Frame* osztályból példányosítva, beállítjuk a pozícióját és méreteit, hozzáadjuk középre helyezve a rajzvásznat, hozzárendelünk egy figyelő objektumot, amely az ablak lezárása esetén biztosítja az alkalmazásból való kilépést, és végül láthatóvá tesszük.

Ha elindítjuk az alkalmazást, minden működik: megjelenik a keret a kék hátterű vászonnal, és a vászonra az egérrel kirajzolhatjuk a kis piros kört. Ha többször is kattintunk, mindig csak az utolsó kör lesz látható, az előző eltűnik. Ez azért van így, mert a teljes felület frissítését kérjük, és az *update* alapértelmezett implementációja törli a felületet a *paint* meghívása előtt. Ha azt szeretnénk, hogy az előzőekben kirajzolt körök is láthatóak maradjanak, egyszerűen kiegészítjük az osztályunkat, újradefiniálva az *update* metódust:

```
public void update(Graphics g) {
      paint(g);
}
```

Az újradefiniált *update* metóduson belül egyszerűen meghívjuk a *paint* metódust, nem végzünk el törlési műveletet, így az előzőleg kirajzolt grafikus elemek láthatóak maradnak. Kipróbálhatjuk. De még mindig lehet a dologgal egy kis probléma: ha a rendszer kéri a felület újrarajzolását, például, mikor lekicsinyítjük az ablakot, majd visszaállítjuk eredeti méreteit, ismét csak a legutolsó kör lesz látható. Ilyen esetben a rendszer a teljes felület frissítését kéri, és nem tárolja az előzőleg megjelenített körök koordinátáit. Természetesen a koordinátákat rögzíthettük volna például egy listában, eszerint módosítva a *paint* metódust, de van ennél elegánsabb megoldás is.

Egy kép objektumot alkalmazhatunk. A köröcskéket erre a képre rajzoljuk rá, és a *paint* metóduson belül a képet rajzoljuk ki a vászonra, így az előző módosításokat is megőrizhetjük.

Digitális képek (bittérképek) létrehozásában és kezelésében az *Image* absztrakt alaposztály lehet segítségünkre. Minden olyan osztály, amely egy grafikus kép reprezentációja ennek az osztálynak a leszármazottja. Természetesen, lévén absztrakt alaposztályról szó, közvetlen módon nem példányosíthatunk belőle, de több módszer is van kép objektum létrehozására. Például, egyszerűen példányosíthatunk a *BufferedImage* származtatott osztályból.

Esetünkben egy másik lehetőséggel élünk. A *Component* osztály *createImage* metódusát alkalmazzuk. Ez a metódus egy adott méretű képnek megfelelő *Image* típusú objektumot térít vissza. Eredetileg a kettős pufferelés (*double buffering*) mechanizmus támogatásának céljából kapott helyet a *Component* osztályban. A kettős (vagy általánosabban többszörös) pufferelés mechanizmusát a számítógépes grafikában a megjelenítés optimalizálására, a képfrissítés gyorsítására alkalmazzák. A mechanizmus lényege, hogy az új kép létrehozásakor nem közvetlenül a video memóriával dolgozunk. A rajzolási műveletek eredményeit előzőleg a memóriában, egy háttér pufferben tároljuk, majd amikor elkészült a teljes kép, ennek a puffernek a tartalmát egy gyors művelettel a video memóriába másoljuk, lecserélve az aktuálisan látható képet a háttérben elkészített képre. Ilyen módon felgyorsítható a képfrissítés, és elkerülhetőek az olyan kellemetlenségek, mint a kép villogása a folyamatos rajzolási műveletek miatt, vagy a régi és új grafikus elemek keveredése a megjelenített képen belül a frissítés során. A *createImage* metódus által visszatérített *Image* objektum ilyen háttér pufferként szolgálhat, segítségével a háttérben elvégezhetőek a rajzolási műveletek a felület frissítése, az új kép megjelenítése előtt.

Fontos megkötés a metódussal kapcsolatban, hogy a komponensnek láthatónak kell lennie, ellenkező esetben a metódus *null* értéket ad. A megkötésnek egyszerűen megfelelhetünk, ha a képet a *paint* metóduson belül hozzuk létre (a metódus meghívásakor a komponensünk már biztosan látható).

Marad még a kérdés, hogy hogyan rajzolhatunk rá erre a képre? Hasonlóan, mint ahogyan azt a komponens felületének esetében is tettük, a *Graphics* osztály segítségével. Az *Image* osztályokhoz tartozik egy *Graphics* objektum, és a *getGraphics* metódus segítségével kérhető egy erre mutató referencia. A rajzolási műveleteket a referencia segítségével végezhetjük. Magát a képet a komponens felületére fogjuk kirajzolni a *paint* metódus paramétereként kapott *Graphics* típusú objektumra meghívva a *drawImage* metódust.

Az osztályunk import része egyetlen sorral egészül ki, amelyben a *java.awt.Image* osztályt importáljuk. A *main* metódus változatlan marad. Az osztály többi részét módosítjuk:

```
      …   //az import utasítások
public class MyCanvas extends Canvas {
      private Image img;
      private Graphics gr;
      public MyCanvas() {
            setBackground(new Color(50,100,250));
            addMouseListener(new MouseAdapter() {
                  public void mousePressed(MouseEvent e) {
                        gr.fillOval(e.getX(),e.getY(),20,20);
                        repaint();
                  }
            });
      }
      public void paint(Graphics g) {
            if (img == null) {
                  img = createImage(getWidth(),getHeight());
                  gr = img.getGraphics();
                  gr.setColor(Color.red);
            }
            g.drawImage(img,0,0,null);
      }
      public void update(Graphics g) {
            paint(g);
      }
      …   //a main metódus
}
```

A képre az egérgomb lenyomásának pillanatában a figyelő megfelelő metódusán belül rá is rajzoljuk a kört, így nincsen szükségünk a koordináták rögzítésére. Szükségünk van viszont a kép objektumra és az annak megfelelő *Graphics* objektumra mutató referenciákra, az *img* és a *gr* attribútumokra. A kép objektumot a *paint* metóduson belül hozzuk létre a *Component* osztály *createImage* metódusának segítségével. A metódus paraméterként kapja a kép méreteit, esetünkben a rajzvászon méreteit, amelyeket a megfelelő metódusok segítségével kérdezhetünk le. A *paint* természetesen többször meghívódik, és nekünk csak egyetlen *Image* objektumra van szükségünk, így a létrehozás előtt ellenőrizzük, hogy a kép már létezik-e. Csak akkor hozunk létre új képet, ha az *img* referencia értéke még *null*. A kép létrehozása után elkérünk egy, a neki megfelelő *Graphics* objektumra mutató referenciát a *getGraphics* metódus segítségével, és a rajzoláshoz használt színt pirosra állítjuk. A kör kirajzolása a figyelő metódusán belül történik. A *gr* referencia segítségével rajzolunk, tehát a köröket a képre rajzoljuk rá. A frissítéskor, a *paint* metóduson belül a rajzvászon felületére a képet rajzoljuk ki, a *drawImage* metódus segítségével. A metódus paraméterként kapja a képre mutató referenciát, a kép bal felső sarkának koordinátáit, és egy *ImageObserver* típusú objektumot. Erre az utóbbira nekünk nem lesz szükségünk, így *null* értéket adunk paraméterként.

A módosításokkal elértük, hogy az előzőleg kirajzolt körök is megmaradjanak, még akkor is, ha a rendszer kéri a felület frissítését. Természetesen, ha akarunk, még találhatunk néhány „szépséghibát". Például, a kép méreteit a rajzvászon méreteinek segítségével adtuk meg, és a keret újraméretezésénél ezt nem változtattuk. Ha kinagyítjuk a keretet, a rajzvászon bizonyos részeire nem tudunk rajzolni. Persze a kerettel együtt újraméretezhetjük a képet is. Ebben az esetben torzulna a kép, de ez is könnyen kikerülhető (a legegyszerűbb már először egy nagyobb képet létrehozni, de megoldhatjuk például úgy is a problémát, hogy a nagyobb kép létrehozásakor rárajzoljuk az előbbi kisebb képet). Megjegyezhetjük azt is, hogy a kép létrehozásához alkalmazott módszer, nem minden esetben a legmegfelelőbb. A használatával kapcsolatos megkötésről már szóltunk, de megemlíthetjük például azt is, hogy az ilyen módon létrehozott kép objektum nem tartalmazhat átlátszó pixeleket. Szerencsére, mint ahogyan azt már említettük, több módszer is rendelkezésünkre áll.

Természetesen az új kép létrehozása helyett állományból is betölthettünk volna egy képet, és ugyanígy le is menthetjük az „alkotásunkat". Erre is több lehetőség van, de közülük kiemelhetjük a *javax.imageio* csomag, illetve az ezen belül található *ImageIO* osztály használatát, amely kényelmes megoldásokkal szolgálhat.

Láthatjuk, hogy a Java által biztosított grafikával kapcsolatos lehetőségek száma nagy. Nem volt lehetőség itt mindenre részletesen kitérni, de nagyon sok témával kapcsolatos anyag áll rendelkezésünkre, és most az alapok megismerése után ezek megértése nem jelenthet gondot. A következő fejezetben még röviden visszatérünk a grafikával kapcsolatos kérdésekre, mivel a SWING eszközkészlet használata ebből a szempontból is eltéréseket mutat az AWT csomaghoz képest. Most viszont lépjünk tovább és nézzük meg, hogy frissen szerzett, grafikával kapcsolatos tudásunkat hogyan kamatoztathatnánk, például egyszerű kis, weboldalakba beágyazott alkalmazások készítésénél.

4.2 Applet-ek

Az applet egy kis Java program, amelynek futtatásához egy külső programra, egy applet-nézőre (*applet viewer*) van szükség. Az applet-ek tipikusan weboldalakba ágyazott egyszerű kis alkalmazások, és az *applet viewer* a böngésző (*browser/navigator*) része. Tulajdonképpen az applet egy panel, amely néhány plusz funkcionalitást biztosít, többnyire a környezettel (*viewer*/böngésző) történő kommunikáció megvalósítására. Ha visszalapozunk az *awt* csomag osztályainak hierarchiáját szemléltető ábrához, láthatjuk is, hogy az applet-ek létrehozásához használt *Applet* osztály a Panel osztály leszármazottja. Ha saját applet-eket akarunk készíteni, akkor ebből az osztályból továbbszármaztatva tehetjük ezt meg. Nézzük meg, hogyan.

4.2.1 Hello, Web!

Az alábbi egyszerű kis applet a „Hello Web!" üzenetet rajzolja ki egy panelre:

```
public class HelloWeb extends java.applet.Applet {
      public void paint(java.awt.Graphics g) {
            g.drawString("Hello Web!", 50, 50 );
      }
}
```

Az osztályt az Applet osztályból származtatjuk, és egyszerűen újradefiniáljuk a *paint* metódust, amelyen belül a *Graphics* osztály *drawString* metódusának meghívásával kirajzoljuk a "Hello Web!" karakterláncot a paraméterként megadott koordinátákra.

Láthatjuk, hogy a program nem tartalmaz *main* metódust, azonban, mivel az *Applet* osztályból származtattunk, futtatható lesz egy applet-nézőn belül. A fordítás az egyszerű Java alkalmazásokhoz hasonlóan történik (például *javac HelloWeb.java*). A programot egy weboldalba is beágyazhatjuk, erre az esetre hozzuk létre a *HelloWeb.html* állományt:

```
<html>
<body>
      <applet code=HelloWeb width=200 height=200></applet>
</body>
</html>
```

Az applet *tag*-en belül megadjuk az osztály (*.class* állomány) nevét, és a panel méreteit.

Ha betöltjük az oldalt egy olyan böngészőbe, amely Java támogatást biztosít (a legtöbb böngésző ezt megteszi), máris látható lesz a panel a felirattal.

4.2.2 Paraméterek és metódusok

A *html* állományon belül más paraméterek is megadhatóak:

```
< APPLET
      [ CODEBASE = codebaseURL ]
      CODE = appletFile
      [ ARCHIVE = archivesList ]
      [ ALT = alternateText  ]
      [ NAME = appletInstanceName ]
      WIDTH = pixels
      HEIGHT = pixels
      [ ALIGN = alignment ]
      [ VSPACE = pixels ]
      [ HSPACE =pixels ]
>
[<PARAM NAME = appletParameter1 VALUE = value >]
[<PARAM NAME = appletParameter1 VALUE = value >]
...
</APPLET>
```

A szögletes zárójelek között feltűntetett paraméterek opcionálisak:

- CODEBASE: amennyiben az applet nincsen egy könyvtárban (vagy gépen) a megfelelő *html* állománnyal, meg kell adnunk az osztály elérési útvonalát;
- ARCHIVE: csomagolt állományok használatának esetében alkalmazzuk, például, mikor alkalmazásunk több osztályból, csomagból épül fel;
- ALT: ha a böngésző „megérti" ugyan az applet *tag*-et, de nem tudja megjeleníteni az applet-et, nem Java-kompatibilis, paraméterként megadhatunk egy alternatív üzenetet;

- NAME: az applet-nek egy nevet adhatunk, és ez több esetben is fontos lehet, ezt látni fogjuk az applet-ek kommunikációjával foglalkozó résznél is;
- ALIGN, VSPACE, HSPACE: a *html* oldalon belüli elhelyezést határozzák meg;
- PARAM: argumentumok átadására szolgál (a parancssor argumentumaihoz hasonlóan). Nagyon fontosak, az applet és környezet közötti kommunikáció megvalósításában. Példa: <PARAM NAME="meret" VALUE="10">.

Ahogyan azt már írtuk, az applet tulajdonképpen egy panel, amely néhány plusz funkcionalitást biztosít, többnyire az applet és környezete közötti kommunikációt szolgáló metódusok formájában. Néhány példa:

- **`void destroy()`** - a környezet hívja meg, amikor felszabadítja az applet által foglalt erőforrásokat;
- **`AppletContext getAppletContext()`** - a környezetre mutató referenciát térít vissza. Az *AppletContext* osztály metódusai teszik lehetővé, hogy az applet kéréseket továbbítson a környezete felé;
- **`String getAppletInfo()`** - információk az applet-ről. A metódust újradefiniálhatjuk olyan módon, hogy meghívásával hasznos információkat térítsünk vissza (szerző, verziószám, stb.);
- **`URL getCodeBase()`** - az applet osztályának (*.class* állomány) helyét téríti vissza;
- **`URL getDocumentBase()`** - a *html* állomány helyét téríti vissza;
- **`String getParameter(String name)`** - a *name* nevű paraméter értékét adja;
- **`void init()`** - a környezet hívja meg az applet betöltése után, általában a grafikus komponenseket ezen belül adjuk hozzá a panelhez;
- **`boolean isActive()`** - az applet állapotát kérdezi le;
- **`void resize(Dimension d)`** – a panel újraméretezése;
- **`void resize(int width, int height)`** – a panel újraméretezése;
- **`void showStatus(String msg)`** - szöveg kiírása a környezet (böngésző) állapot kijelzőjére (*status bar*);
- **`void start()`** - az *init()* után, valamint az oldal újratöltésekor hívja meg a környezet, ezzel indítva el az applet-et;
- **`void stop()`** - az applet leállításakor, illetve új oldal betöltésekor hívja meg a környezet.

Kiemelt fontossággal bír a *getAppletContext* metódus, amely a visszatérített *AppletContext* típusú referencián keresztül lehetővé teszi, hogy az applet különböző feladatok elvégzését kérje a környezetétől. Például alkalmazhatjuk egy új oldal megnyitására:

```
getAppletContext().showDocument("http://www.java.com");
```

Az *AppletContext* osztály *getApplet* metódusának segítségével egy adott nevű appletre mutató referenciát kaphatunk, amely az applet-ek közötti kommunikáció megvalósításának szempontjából fontos. Ezt a későbbiekben példával is szemléltetjük.

A *getCodeBase* és *getDocumentBase* metódusok egy-egy URL típusú objektumra mutató referenciát térítenek vissza. Az Interneten a különböző erőforrások (pl. oldalak, állományok, képek, stb.) azonosítására szolgál az URI (*Uniform Resource Identifier*). Az URI lehet egyszerűen egy név (URN – *Uniform Resource Name*), vagy lehet egy az illető erőforrás elérési módját

is meghatározó URL (*Uniform Resource Locator*). Az URI és URL fogalmak közötti különbséget egyszerűen úgy fogalmazhatnánk meg, hogy minden URL URI, de nem minden URI URL (tehát az URL-ok tulajdonképpen a URI-ok egy alosztályát képezik). Az URL a következő elemekből áll: az objektum (erőforrás) eléréséhez használható protokoll neve (*http*, *ftp*, stb.), az objektumot tároló számítógép neve, vagy IP címe (ha *localhost*, akkor elhagyható), a hálózati port száma (standard port alkalmazása esetén nem szükséges), és az objektum neve, valamint elérési útvonala.

URL szerkezete: **`protocol://hostname:port/object_name.`**

Példák: **`http://www.ietf.org/rfc/rfc2396.txt, vagy file:///C:/filename.txt`**

Java-ban URL objektumokat a *java.net.URL* osztályból példányosítva hozhatunk létre, a konstruktor paraméterként kaphatja a teljes karakterláncot, vagy külön az alkotó elemeket.

A felsoroltakon kívül az Applet osztály más metódusainál is fontos szerepet játszanak az URL objektumok. Például:

- **`AudioClip getAudioClip(URL url)`** – hangfájl betöltése, a paraméterként kapott URL objektum által azonosított audio állománynak megfelelő *AudioClip* objektumra mutató referenciát térít vissza. Az *AudioClip* osztály *play*, *stop* és *loop* metódusainak segítségével lejátszhatjuk a hanganyagot, leállíthatjuk a lejátszást, illetve ismételt lejátszást kérhetünk;
- **`Image getImage(URL url)`** – kép betöltése, a paraméterként kapott URL objektum által azonosított állománynak megfelelő *Image* objektumra mutató referenciát térít vissza.

Elmondhatjuk tehát, hogy az applet és környezete közötti kommunikáció egyrészt az applet *tag* paraméterein keresztül, másrészt az *Applet* osztály metódusainak segítségével történhet. Az alábbiakban bemutatjuk, a különböző applet-ek közötti kommunikáció megvalósításának lehetőségét.

4.2.3 Applet-ek közötti kommunikáció

Az applet-ek közötti kommunikáció megvalósításának lehetőségét egy egyszerű példával szemléltetjük. Két applet-et hozunk létre. Az első egy szövegmezőt és egy gombot fog tartalmazni. A gomb lenyomására továbbítja a szövegmezőbe írt szöveget a másik applet-nek, és az kirajzolja a szöveget a saját felületére.

Az első osztály forráskódja:

```
import java.applet.Applet;
import java.awt.event.ActionListener;
import java.awt.TextField;
import java.awt.Button;
import java.awt.event.ActionEvent;
public class FirstApplet extends Applet implements ActionListener {
      public TextField tf;
      public SecondApplet a;
      public Button b;
      public FirstApplet() {}
      public void init() {
```

```
            tf = new TextField();
            add(tf);
            b = new Button("Send");
            add(b);
            b.addActionListener(this);
            a = (SecondApplet)
                        getAppletContext().getApplet("APPLET2");
      }
      public void actionPerformed(ActionEvent e) {
            a.showText(tf.getText());
      }
}
```

A második osztály forráskódja:

```
import java.applet.Applet;
import java.awt.Graphics;
public class SecondApplet extends Applet {
      private String string;
      public SecondApplet() {}
      public void paint(Graphics g) {
            if (string != null)
                  g.drawString(string, 10, 10);
      }
      public void showText(String s) {
            string = s;
            repaint();
      }
}
```

A *html* állomány:

```
<html>
<body>
      <applet code=FirstApplet width=200 height=200 />
      <applet code=SecondApplet width=200 height=200 name=APPLET2 />
</body>
</html>
```

Mindkét osztályt az *Applet* osztályból származtatjuk. Az első implementálja az *ActionListener* interfészt, mivel figyelnie kell a gomb lenyomását. A grafikus komponenseket (szövegmező és gomb) ezúttal nem a konstruktoron belül, hanem az *init* metódus törzsében hozzuk létre és adjuk hozzá a panelhez. Applet-ek esetében ez a standard eljárás. Az első osztály *init* metódusának utolsó sorában a *getAppletContext* metódus segítségével kérünk egy környezetre mutató referenciát. Erre meghívjuk a *getApplet* metódust, amely paraméterként kapja a kért applet nevét, és visszatérít egy *Applet* típusú referenciát, amely a név által beazonosított applet-re mutat. A referencia típusát *SecondApplet*-be kell alakítanunk, hogy meghívható legyen a *showText* metódus (egyébként ez nem történhetne meg, mivel ez nem az *Applet* osztály saját metódusa). Az átalakításnak nincs akadálya, mivel az objektum konkrét típusa *SecondApplet.*

Az *actionPerformed* metóduson belül a referencia segítségével meghívjuk a *showText* metódust, paraméterként átadva neki a szövegmező tartalmát, így a gomb lenyomása a metódus meghívását fogja eredményezni, a megfelelő paraméterrel. A második applet-ben a *showText* metódus beállítja a *String* típusú attribútum értékét a paraméter segítségével, majd frissíti a felületet. A *paint* metóduson belül, amennyiben az attribútum értéke nem *null*, kirajzoljuk a felületre. A második applet azonosítása a környezet által a nevének segítségével történik. Ezt a *html* állományban adjuk meg, az applet *tag* paramétereként.

4.2.4 Adatvédelmi és biztonsági szabályok

A programokkal szemben gyakori elvárás, hogy meg kell felelniük bizonyos adatvédelmi és biztonsági követelményeknek, be kell tartaniuk bizonyos szabályokat. Minden programot jellemeznek bizonyos jogosultságok, amelyek meghatározzák például, hogy az illető program milyen állományokat módosíthat, milyen információkhoz férhet hozzá, nyithat-e meg hálózati kapcsolatokat, indíthat-e el más programokat, és így tovább. A Java programok esetében a *SecurityManager* objektumok felelősek azért, hogy a programok betartsák ezeket a szabályokat. A „jogkörök" meghatározása az alkalmazott biztonsági stratégiának, vezérelvnek (*security policy*) megfelelően történik.

A jogok általában *.policy* kiterjesztésű konfigurációs állományokban vannak tárolva, innen olvashatjuk ki őket, és programon belüli reprezentációjuk *java.security.Policy* típusú objektumok formájában adott. Az alapértelmezett *.policy* állomány a JRE *lib/security* alkönyvtárában található. Az állományok jogosultságokat meghatározó bejegyzéseket tartalmaznak, az alábbi szintaxis szerint:

```
grant signedBy "signer_names", codeBase "URL" {
      permission permission_class_name "target_name",
            "action", signedBy "signer_names";
      ....
};
```

A *signedBy* mező a kód szerzőjének meghatározására szolgál: a jogkör olyan kódra vonatkozik, amelyet a megadott szerzők digitális aláírása hitelesít. A Java kódok digitális aláírását itt nem részletezzük. A mező opcionális, elhagyása azt jelenti, hogy a jogok az aláírástól függetlenül érvényesek.

A *codeBase* mező értéke a kód helyét azonosító URL. A jogok az illető helyen található kódokra érvényesek. A mező szintén elhagyható, így a jogok a kód helyétől függetlenül érvényeseknek lesznek tekintve.

A *permission* mezők határozzák meg magukat a jogokat, „engedélyeket". A *permission_class_name* az engedély típusa, például *java.io.FilePermission*. A *target_name* értéke az engedélyezett művelet célpontja, tárgya. Ez egy *FilePermission* típusú engedély esetében például egy állomány lenne. Az *action* mező az olyan engedélytípusok esetében fontos, ahol az engedélyezett műveletnek több formája lehet. Például a *FilePermission* esetében különböző hozzáférési formákról beszélhetünk, az *action* mező értéke lehet például „*read*". A *signedBy* mező itt is digitális aláírással kapcsolatos, de ebben az esetben maga az engedély típusának megfelelő osztály kell hitelesítve legyen a megadott szerző aláírásával, ahhoz, hogy az engedélyt érvényesnek

tekintse a rendszer. A mezők nagyrésze opcionális, például az alábbi módon egy program részére minden jog megadható:

```
grant {
      permission java.security.AllPermissions;
};
```

A *Policy* objektumoknak megfelelő jogok betartásának ellenőrzéséért a *SecurityManager* objektum felelős, és *SecurityException* típusú kivételt generál, ha valamilyen program sérteni próbálja a rá vonatkozó szabályokat. A *System* osztály *getSecurityManager* nevű metódusával kérhetünk egy, az aktuális *SecurityManager* objektumra mutató referenciát. Az osztály *check* előtagú (*checkXXX* formájú) metódusokat biztosít, különböző jogosultságok ellenőrzésére.

Az alapértelmezett *.policy* állománynak megfelelő jogokat kiegészíthetjük, vagy lecserélhetjük. Ez a program futtatásakor történhet a következő módon:

```
java -Djava.security.manager -Djava.security.policy=URL App
```

Az URL a *.policy* állomány helyét határozza meg. A *–D* kapcsoló a rendszer konfigurációs paramétereinek beállítására szolgál. Az első rész biztosítja, hogy a program futtatása során az alapértelmezett *SecurityManager* objektum ellenőrizze a jogok betartását. Erre csak akkor van szükség, ha az alkalmazás (az *App* a *.class* állomány neve) nem biztosít saját *SecurityManager*-t (ezt a *System* osztály *setSecurityManager* metódusának segítségével teheti meg). A második rész az alapértelmezetten használt *.policy* állományban meghatározott jogokat kiegészíti az URL által meghatározott állományban meghatározott jogokkal. A második résznél, az URL előtt „=” helyett „==” is használható, és ebben az esetben csak az URL által beazonosított *.policy* állományban található jogok lesznek érvényesek.

Megjegyzendő, hogy a fenti műveletek elvégzésére, az alapértelmezett beállítások módosítására csak akkor van lehetőség, ha ezt a rendszer biztonsági beállításai megengedik, azaz, ha a *java.security* konfigurációs állományon belül a *policy.allowSystemProperty* tulajdonság értéke *true* (ez az alapértelmezett beállítás).

Az applet-ek tipikusan weboldalakba ágyazott alkalmazások, amelyek futtatása a felhasználó gépén történik. Magától értetődő tehát, hogy nem tehetünk meg bármit egy applet-en belül, a programoknak meg kell felelniük bizonyos adatvédelmi és biztonsági szabályoknak. Senki sem szeretné például, hogy mikor betölt egy weboldalt, egy kis program elkezdjen a fájlrendszerében kutakodni, és esetleg állományokat letörölni. Az applet-ekre vonatkozó fontosabb biztonsági szabályok az alábbi pontokban foglalhatóak össze:

- nem írhatnak, nem olvashatnak, és nem törölhetnek állományokat;
- nem olvashatnak ki rendszerinformációkat;
- nem futtathatnak más programokat;
- nem nyithatnak új hálózati kapcsolatokat;
- nem tölthetnek be natív metódusokat, függvénykönyvtárakat.

Ezek betartásáért a böngésző *SecurityManager*–e felelős, és *SecurityException*-t dob, ha valamelyik szabályt sérteni próbálja a program. A böngészők által alkalmazott biztonsági irányelvek alapértelmezetten úgy vannak meghatározva, hogy a programok betartsák a fenti szabályokat. Igaz ugyan, hogy a beállítások változtathatóak, a felhasználó engedélyt adhat bizonyos műveletekre, de nekünk, programozóként arra kell gondolnunk, hogy ezt nem várhatjuk el tőle. Ne sértsük meg ezeket a szabályokat az applet-eken belül. Ha a helyzet megkövetelné

valamelyik szabály mellőzését, valószínűleg azt jelenti, hogy nem az applet a megfelelő eszköz a probléma megoldására. És az eszközök száma óriási, biztosan megtaláljuk a megfelelőt. Következtetésként elmondhatjuk, hogy az applet-ek csak nagyon egyszerű feladatok esetében alkalmazhatóak, például kis játékok, vagy bemutató jellegű alkalmazások készítésénél.

4.3 Nemzetköziesítés és lokalizáció

A számítógépes alkalmazásoknál általában, de főként a grafikus felhasználói felületek esetében, általános követelmény a nemzetköziesítés és lokalizáció. A nemzetköziesítés többnyelvűséget jelent: a különböző anyanyelvű felhasználók könnyen (a program módosítása nélkül) állíthassák át a felület nyelvét. Fontos továbbá a különböző helyi, a felhasználó országában megszokott megjelenítési módok, konvenciók betartása, például numerikus adatok, dátumok, pénzösszegek megjelenítésének esetén. A már nemzetköziesített, több nyelvet támogató programok esetében a lokalizáció folyamata felelős az adatok megfelelő megjelenítéséért. Az angol nyelvű szakterminológiában a két folyamat neve *internationalization*, illetve *localization*, és gyakran használják az i18n és L10n rövidítéseket, ahol a számok a megnevezések első és utolsó betűi közötti karakterek számát jelölik. Mivel a két műveletet általában együtt alkalmazzák, a szakirodalomban néhol találkozhatunk a *globalization* (g11n) kifejezéssel is, a két művelet együttes megnevezéseként. A Java hatékony eszközöket biztosít mindkét feladat megoldására.

4.3.1 A *Properties* osztály

Elsőként a *Properties* osztályt, és az ennek megfelelő *.properties* állományokat említhetjük meg. Az osztály egy alkalmazás esetében előforduló perzisztens tulajdonságok kezelésére alkalmas. Általános szabály például, hogy a különböző konfigurációs jellemzőket (amelyek változhatnak) ne építsük be a kódunkba (*hardcoding*), mivel ez minden változás esetén az újrafordítás kényszeréhez vezetne. Például, egy hálózati kliens-szerver alkalmazás esetén megtörténhet, hogy a szerver címe, vagy a kliens kiszolgálására használt hálózati port száma változik, de ennek a változásnak nem kellene a kód újrafordítását eredményeznie. Egyszerűen megoldható a probléma, ha az ilyen, és ehhez hasonló paramétereket konfigurációs állományokban tároljuk.

A fenti mechanizmus megvalósítását szolgálja a *Properties* osztály, amely tulajdonképpen egy hasító táblázat, a *Hashtable* osztály (amelyről szó lesz a jegyzet gyűjteményekkel foglalkozó részénél) leszármazottja. A tulajdonságokat azonosító-érték párok formájában állományokban tárolhatjuk, a következő módon:

```
identifier1 = value1
identifier2 = value2
...
```

Ezek az állományok tipikusan *.properties* kiterjesztésűek. A *Properties* osztály megfelelő metódusaival (*load* és *store*) beolvashatjuk, vagy lementhetjük (módosíthatjuk) ezeket a tulajdonságokat. Ezen kívül az osztály lehetőséget biztosít arra, hogy lekérjünk, vagy módosítsunk egy adott azonosítóval rendelkező tulajdonságot:

```
public String getProperty(String key)
public Object setProperty(String key, String value)
```

A második metódus az illető azonosítónak megfelelő előző értéket téríti vissza, ha volt ilyen, egyébként *null* értéket ad.

4.3.2 A *Locale* osztály

A *Locale* objektumok egy-egy specifikus földrajzi, politikai, kulturális régiónak felelnek meg. Egy ilyen objektum létrehozásához két paraméterre van szükségünk: az illető régió (általában ország) kódjára, és a nyelv kódjára.

```
Locale l = new Locale("en","US")
```

Az első paraméter a nyelv kódja (esetünkben angol), a második az ország kódja (esetünkben Amerikai Egyesült Államok). Ezek a kódok az ISO vonatkozó szabványainak felelnek meg. Az alábbi kis program segítségével kilistázhatjuk az érvényes kódokat:

```
import java.util.Locale;
public class LocaleExample {
      public static void main(String[] args) {
            Locale list[] = Locale.getAvailableLocales();
            for (Locale l:list)
                  System.out.println(l.getLanguage() + " "
                                                + l.getCountry());
      }
}
```

Az érvényes kódok listáját tartalmazó tömböt a *Locale* osztály *getAvailableLocales* statikus metódusával kérjük le, majd egy cikluson belül sorban kiírjuk a konzolra a különböző *Locale* objektumoknak megfelelő nyelvek és országok kódjait.

Az alapértelmezett nyelv és ország kódját az általános rendszerkonfigurációkat tartalmazó *Properties* objektum tárolja *user.language* és *user.region* azonosítókkal. Ezek lekérdezhetőek, illetve átállíthatóak. A *System* osztály *getProperties* metódusa visszatéríti a rendszerkonfigurációnak megfelelő *Properties* objektumot, és erre a megfelelő metódusokat alkalmazhatjuk.

A különböző hely-specifikus konvenciók által érintett értékeket (pl. számok, dátumok, stb.) a *Locale* objektumoknak megfelelően formázhatjuk. Példaként tekintsük az alábbi kis programot:

```
import java.text.DateFormat;
import java.text.NumberFormat;
import java.util.Date;
import java.util.Locale;
public class LocaleExample {
      private static void showValues(Locale l) {
            NumberFormat nf;
            DateFormat df;
            Integer i = new Integer(12345);
            Double d = new Double(0.12);
            Date date = new Date();
            nf = NumberFormat.getNumberInstance(l);
            String integerOutput = nf.format(i);
```

```
            String realOutput = nf.format(d);
            nf = NumberFormat.getCurrencyInstance(l);
            String currencyOutput = nf.format(d);
            nf = NumberFormat.getPercentInstance(l);
            String percentOutput = nf.format(d);
            df = DateFormat.getDateInstance(DateFormat.SHORT,l);
            String dateOutput = df.format(date);
            df = DateFormat.getTimeInstance(DateFormat.SHORT,l);
            String timeOutput = df.format(date);
            System.out.println(integerOutput);
            System.out.println(realOutput);
            System.out.println(percentOutput);
            System.out.println(currencyOutput);
            System.out.println(dateOutput);
            System.out.println(timeOutput);
      }
      public static void main(String[] args) {
            showValues(new Locale("hu","HU"));
            showValues(new Locale("en","US"));
      }
}
```

A *showValues* osztályszintű metódus paraméterként kap egy *Locale* objektumot, és ennek megfelelően fog megjeleníteni különböző értékeket a konzolon. A numerikus értékek formázásához egy-egy *NumberFormat* típusú, a dátumok formázásához egy-egy *DateFormat* típusú objektumot fog használni. Ezek az osztályok absztrakt alaposztályok, és tartalmaznak osztályszintű gyártó metódusokat (*factory*) különböző formázásokhoz alkalmazható objektumok példányosítására (a származtatott osztályok példányai). Ennek megfelelően, ha számot akarunk formázni, a *getNumberInstance* gyártó metódust használhatjuk, ha pénzösszeget, akkor a *getCurrencyInstance* metódust, és így tovább. A dátumok esetében hasonló a helyzet, de itt a gyártó metódusok még kapnak egy paramétert, amely a formázás stílusát határozza meg.

A *showValues* metódus példányosít egy egész és egy valós számot burkoló objektumot, valamint egy dátumot, a paraméterként kapott *Locale* objektumnak megfelelően formázza ezeket, és az eredményül kapott karakterláncokat kiírja a konzolra. A *main* metóduson belül különböző *Locale* paraméterekkel meghívhatjuk ezt a metódust. Példánk esetében mi az angol és magyar nyelvek, illetve Magyarország és az Egyesült Államok kódjait használtuk a *Locale* objektumok létrehozásánál. Az értékek kiírása ennek megfelelően történik. Például, a magyar *Locale*-nak megfelelően: a „12 345”, „0,12”, „12%”, „0,12 Ft” kimeneteket kapjuk. Az amerikai *Locale*-nak megfelelően a kimenet a következőképpen módosul: „12,345”, „0.12”, „12%”, „$0.12”. Az aktuális dátumot a konvenciók szerint formázva, a stílusparamétert figyelembe véve kapjuk. Ha a programot 2011. január tízedikén, 13:00 órakor futtatjuk, akkor a magyar *Locale*-nak és az adott stílusparaméternek (*DateFormat.SHORT*) megfelelő kimenetek: „2011.01.10” és „13:00”. Ugyanez az amerikai *Locale*-nak megfelelően: „1/10/11” és „1:00 PM”.

A fenti módszerrel az alkalmazásainkon belül használt értékek megjelenítését könnyedén testreszabhatjuk különböző anyanyelvű, különböző régiókban élő felhasználók számára. Lássuk, hogyan tudnánk megvalósítani a többnyelvűséget.

4.3.3 A *ResourceBundle* osztály

Alkalmazásaink, grafikus felhasználói felületeink többnyelvűsítésében a Java *ResourceBundle* osztálya lehet segítségünkre. A feladat leegyszerűsítve úgy fogalmazható meg, hogy a grafikus komponensek címkéin megjelenő szöveget változtatni tudjuk az adott *Locale*-nak, nyelvnek megfelelően. Természetesen a következőkben bemutatott módszer általánosabban is alkalmazható: a konfigurációs paraméterekhez hasonló módon gyakorlatilag bármilyen szöveges elemet el tudunk választani a kódtól. A módszert (*string externalization*) a legtöbb fejlesztői környezet is támogatja, megkönnyítve munkánkat.

Az eredeti feladatnál maradva, megtehetjük, hogy a különböző nyelvű címkefeliratokat különböző *.properties* állományokban tároljuk. Minden szövegnek lesz egy azonosítója (kulcs), amely a különböző állományokon belül azonos. Csak az érték változik a nyelvnek megfelelően. Az állományok kezelésében a *ResourceBundle* osztály segít. Ahhoz, hogy ezt megtehesse állományainknak be kell tartaniuk bizonyos elnevezési konvenciókat. Az állományok neve két részből fog állni. Az első rész azonos, az utótag a kapcsolódó *Locale*-nak megfelelően változik. Szükséges még egy alap állomány, amelynek neve nem kap utótagot, és ez fogja tartalmazni az alapértelmezett kulcs-elem párokat. Például, ha azt szeretnénk, hogy programunk angol és magyar nyelven működjön, és az angol legyen az alapértelmezett beállítás, akkor a következő állományokat használhatnánk:

```
MessageBundle.properties
MessageBundle_en_US.properties
MessageBundle_hu_HU.properties
```

Az állománynevekben használt közös alap a *MessageBundle* (tetszőlegesen megválasztható), ez az alapértelmezett értékeket tároló állomány neve is, a különböző nyelveknek megfelelő állományok nevei a *Locale* objektumnak megfelelő nyelv és ország kódokat kapják utótagként, egymástól és az alaptól alulvonással elválasztva.

Példaként nézzünk egy egyszerű kis programot, amely a felhasználót az anyanyelvén üdvözli. Az üzeneteket az alkalmazás a konzolra fogja kiírni, de hasonló módon a szövegek grafikus komponensek címkéiben is megjelenhetnének.

Hozzuk létre először a különböző nyelvű üzeneteket tartalmazó állományokat. Egyelőre az angol és a magyar nyelv használatát fogja alkalmazásunk „támogatni".

Az alapértelmezett értékeknek megfelelő állomány (*MessageBundle.properties*) tartalma:

```
greetings = Hello.
farewell = Good bye.
inquiry = How are you?
```

Azonos tartalommal létrehozzuk az angol nyelvnek megfelelő állományt is (*MessageBundle_en_US.properties*). A magyar nyelvnek megfelelő állomány (*MessageBundle_hu_HU.properties*) tartalma a következő lenne:

```
greetings = Szia.
farewell = Viszlát.
inquiry = Hogy vagy?
```

Láthatjuk, hogy a kulcsoknak itt más üzeneteket feleltettünk meg. Most lássuk a program kódját:

```
import java.util.*;
public class LocaleExample {
      public static void main(String[] args) {
            Locale currentLocale;
            ResourceBundle messages;
            try {
                  currentLocale = new Locale(args[0], args[1]);
            } catch (Exception e) {
                  currentLocale = Locale.getDefault();
            }
            messages = ResourceBundle.getBundle(
                        "MessageBundle", currentLocale);
            System.out.println(messages.getString("greetings"));
            System.out.println(messages.getString("inquiry"));
            System.out.println(messages.getString("farewell"));
      }
}
```

A program a parancssor argumentumaként megkap egy nyelv és egy ország kódot. Ezek alapján példányosít egy *Locale* objektumot. A műveletet egy *try-catch* konstrukción belül végezzük el, mivel a program nem megfelelő módon történő indítása kivételhez vezethet. Lehetséges, hogy nincsenek megadva argumentumok (vagy csak egy van), és akkor túllépjük a tömbhatárokat. Az ilyen esetekben az alapértelmezett régiónak és nyelvnek megfelelő *Locale* objektumot fogjuk használni.

A *ResourceBundle* osztály egy absztrakt alaposztály, amely *getBundle* nevű gyártó metódusokat tartalmaz, különböző *Locale* objektumoknak megfelelő *ResourceBundle* típusú objektumok létrehozására. Esetünkben a metódus paramétereként megadjuk az alap állománynevet (*MessageBundle*), és a létrehozott *Locale* objektumot.

A *main* metódus utolsó három sorában történik az üzenetek kiírása. A *ResourceBundle* objektumunk *getString* metódusát alkalmazzuk. A metódus visszatérít egy adott (paraméterként kapott) kulcsnak (azonosítónak) megfelelő értéket. Ezeket az értékeket írjuk ki a konzolra, és így a megadott *Locale* objektumnak megfelelő nyelven üzenhetünk a felhasználónak.

Lehetséges, hogy bár a paraméterek száma megfelelő, és létrehozható a *Locale* objektum, a megadott értékek nem felelnek meg az érvényes régió és nyelv kódoknak. A fordító ilyen esetekben nem jelez hibát, és a problémát megoldja a *ResourceBundle* osztály keresési stratégiája. A gyártómetódus először megpróbálja betölteni a megadott *Locale*-nak megfelelő erőforrás állományt. Amennyiben ez nem sikerül megpróbálja betölteni az alapértelmezett régiónak és nyelvnek megfelelő állományt. Így ugyanazt az eredményt kapjuk érvénytelen kódok megadásakor, mint a kivétel felléptekor. Megjegyezhetjük, hogy a *getBundle* metódus második paramétere el is hagyható, és ilyenkor is az alapértelmezett *Locale*-nak megfelelő állományt próbálja meg betölteni a rendszer.

Az is megtörténhet, hogy az alapértelmezett *Locale*-nak megfelelő állomány sem létezik, de ez sem jelent igazi problémát, mivel a rendszer ilyenkor tovább lépik a keresésben,

és megpróbálja betölteni az alapértelmezettként megadott erőforrás állományt. Megjegyezhetjük, hogy ugyanezt az eredményt elérhetjük úgy is, hogy a *Locale* konstruktorának üres karakterláncokat adunk meg paraméterként. Ilyen esetben mindig az alapértelmezettként megadott állomány, és nem az alapértelmezett *Locale*-nak megfelelő állomány kerül betöltésre.

A fentiekben leírtaknak megfelelően, az erőforrások betöltésekor probléma csak akkor léphet fel, ha az alapértelmezett állomány sem található, mivel ebben az esetben *MissingResourceException* típusú futási idejű kivételt kapunk.

A hibalehetőség kiküszöbölésének érdekében oda kell figyelnünk az erőforrás állományok elhelyezésére, illetve az alap állománynév megadásakor az útvonal helyes megadására. Az erőforrás állományokat, az osztályokhoz hasonlóan, a rendszer az osztálybetöltő (*classloader*) segítségével tölti be (a *ClassLoader.getResource* metódus által). Ennek megfelelően a projekthez tartozó csomagok hierarchiájában keresi ezeket. Amennyiben nem használunk csomagokat, az állományoknak az osztályállományokkal egy könyvtárban kell lenniük. Ha a betöltést végző osztály valamilyen csomagban található, akkor az állományokat a projekt gyökérkönyvtárában kell elhelyeznünk. Általában az erőforrás állományokat egy projekten belül külön könyvtárakba csoportosítjuk. Ebben az esetben az alap állománynév megadásakor az állományt tartalmazó csomagot is meg kell határoznunk. Tételezzük fel, hogy példánk esetében az erőforrás állományokat a projekten belül egy külön „*res*" könyvtárban tároljuk. Ebben az esetben a program megfelelő sora a következőképpen módosul:

```
messages = ResourceBundle.getBundle(
                "res.MessageBundle", currentLocale);
```

A rendszer a megadott "*res.MessageBundle*" karakterláncból felépíti a megfelelő elérési útvonalat (a „." karaktereket „/" karakterekkel helyettesítve, és az állománynevet a *Locale*-nak megfelelő utótagokkal, illetve a *.properties* kiterjesztéssel kiegészítve). Megjegyzendő, hogy a keresés ebben az esetben is a csomaghierarchián belül, a *classpath* gyökerétől kiindulva történik. Amennyiben abszolút elérési útvonalat szeretnénk megadni (bár ez ritkán szükséges, előfordulhat, ha az állományok a projekten kívül találhatóak), akkor egy, a célnak megfelelő osztálybetöltőt (pl. *URLClassLoader*) alkalmazhatunk (a *getBundle* metódus háromparaméteres változatának segítségével adhatunk át a metódusnak egy erre mutató referenciát).

4.4 Javasolt gyakorlatok

1. Egy kereten belül helyezzünk el több különböző színű panelt, a színeket véletlenszerűen generálva. Ha az egérmutató belép egy adott panel fölé, az illető panel véletlenszerűen színt vált.
2. Egy kereten belül helyezzünk el egy rajzvásznat (egy saját osztályt hozunk létre a *Canvas* osztályból származtatva), két *Choice* komponenst, egy jelölőnégyzetet (*Checkbox*), és egy gombot. A felhasználó a két *Choice* komponens segítségével kiválaszthat egy adott alakzattípust (pl. kör, négyzet, stb.), és egy adott színt (pl. kék, piros, stb.). A gomb lenyomásának hatására a vászonra kirajzoljuk a kiválasztott alakzatot a kiválasztott színnel. Amennyiben a jelölőnégyzet be van jelölve, az alakzat felületét is kitöltjük az illető színnel.

   A keretnek megfelelő osztályt (a *Frame* leszármazottja), és a rajzvászonnak megfelelő osztályt (a *Canvas* leszármazottja) külön osztályként, külön állományokban hozzuk létre

(a vásznat ne belső osztályként valósítsuk meg). Figyeljünk arra, hogy a vászon ne függjön a kereten belül alkalmazott komponensektől (pl. ne befolyásolja a vászon osztályt, ha valamelyik *Choice* komponenst listára cseréljük, stb.)

A programnak elkészíthetjük egy olyan változatát is, amelynek esetében nem szükséges a gomb lenyomása: bármelyik másik komponens állapotának változásakor frissítjük a rajzot. Ezen kívül a szín kiválasztására alkalmas *Choice* komponenst helyettesíthetjük olyan módon, hogy a felhasználó tetszőleges R, G, B értékeket meg tudjon határozni (pl. három szövegmező segítségével).

3. Készítsünk egy applet-et, a megfelelő panelen belül elhelyezve egy rajzvásznat, egy *Choice* komponenst, egy szövegmezőt és egy gombot. A rajzvászonra az egér segítségével rajzolhatunk: ha lenyomott gombbal mozdítjuk el az egérmutatót (*mouse dragged*), az „nyomot" hagy maga után (az előző koordinátákat összekötjük az új koordinátákkal). A rajzolást olyan módon valósítsuk meg, hogy a vászon felülete ne törlődjön, még a rendszer által kezdeményezett frissítés esetében sem (*Image* példány használata). Ezen kívül próbáljunk meg arra is figyelni, hogy a vászon átméretezésével a rajz ne torzuljon.

   A *Choice* komponensnek megfelelő listából különböző parancsokat választhatunk ki: vászon törlése, kép megjelenítése, weboldal megnyitása, hanganyag lejátszása. Bizonyos műveletek esetében paramétert is meg kell határoznunk (pl. kép-, vagy hangállomány neve és elérési útvonala, weboldal címe), ezt a szövegmező segítségével tehetjük meg. A gomb lenyomásának hatására végrehajtásra kerül a kiválasztott utasítás: töröljük a vásznat, vagy megjelenítjük rajta a megadott képet, lejátsszuk a megadott hanganyagot, vagy egy új böngészőablakban megnyitjuk a megadott weboldalt.

4. „Nemzetköziesítsük" az előző két program grafikus felhasználói felületét. A program a parancssor argumentumaként megkapja egy adott nyelv és ország kódját, és a felület komponenseinek szövege az ezek alapján létrehozott *Locale* objektumnak megfelelő nyelven jelenik meg (két-három nyelven adjuk meg a szövegeket). A második feladat esetében (alakzat kirajzolása) készítsünk egy olyan változatot is, amelynek esetében a felhasználó a felületről, egy menü segítségével változtathatja a nyelvet.

5

SWING

Az *awt* csomag grafikus komponenseinek használata, az AWT működési mechanizmusából eredően, több esetben hátrányos lehet. A komponensek megjelenítése a platform ablakrendszerének megfelelően történik, és a programozónak nincs lehetősége ezt befolyásolni. Általában ez nem jelent problémát, de vannak helyzetek, amikor ebből a szempontból nagyobb szabadságra, több lehetőségre lenne szükségünk. A *toolkit*-ek bonyolultak, és ez a bonyolultság hibalehetőségek (*bug*-ok) forrása lehet. A különböző platformokhoz különböző *toolkit* implementációkat kell biztosítani, és ez némileg sérti a platformfüggetlenség alapelvét.

A fenti érvek vezettek ahhoz, hogy nagyon hamar megjelent az igény egy alternatív eszközkészletre, és 1996-ban a *Java Foundation Classes* (JFC) részeként megjelent a *javax.swing* csomag [11]. A SWING nagyon hamar népszerűvé vált, és mivel a külső fejlesztők is többnyire ezt a csomagot vették alapul saját komponenseik elkészítéséhez, az AWT-nél sokkal gazdagabbá fejlődött.

A SWING az AWT-re épül, de vele ellentétben teljes egészében Java-ban megvalósított. Az AWT nehézsúlyú (*heavyweight*) komponensekkel dolgozik, ami azt jelenti, hogy minden komponensnek van egy natív párja (*peer*). Ez a SWING esetében változik, a SWING komponensek pehelysúlyúak (*lightweight*), megjelenítésükhöz nincsen szükség natív megfelelőkre. Természetesen a SWING komponensek is egy natív tárolón belül lesznek megjelenítve, de az alapot szolgáltató ablakon, vagy kereten kívül nincsen szükség további natív elemekre.

A *swing* csomag tartalmazza az AWT komponensek pehelysúlyú megfelelőit, de ezeken kívül még nagyon sok komponenst és eszközt biztosít. Az MVC minta is sokkal nagyobb szerepet kap a komponensek esetében. A legtöbb komponenshez tartozik egy modell osztály, és egy megjelenítésért felelős osztály. A modell a komponens belső állapotát, és viselkedését írja le, míg a megjelenítésért felelős osztály feladata az illető komponens grafikus megjelenítése. Például, ha egy táblázatot akarunk megjeleníteni, akkor a *JTable* megjelenítésért felelős osztály használni fog egy *TableModel* típusú objektumot. A *TableModel* interfészt megvalósító osztályok határozzák meg, hogy milyen módon férhetünk hozzá a táblázatban tárolt adatokhoz.

A SWING esetében is megemlíthető néhány hátrány. Amint azt látni fogjuk, azokkal az eszközkészletekkel ellentétben, ahol a komponensek megjelenítése natív API-eken keresztül történik (pl. AWT), a SWING esetében a komponensek felelősek felületük kirajzolásáért. Ez egyrészt a sebesség rovására mehet, másrészt bizonyos ablakrendszerek esetében megjelenítési rendellenességek léphetnek fel. Megjegyzendő viszont, hogy egy nagyon jól megvalósított (és továbbfejlesztett) eszközkészletről van szó, így ezek a hátrányok nem túl jellemzőek, és nem általánosíthatóak. Az eszközkészlet kiválasztásának a feladat függvényében kell történnie. Például, ha csak egy adott ablakrendszerre szeretnénk megírni az alkalmazásunkat, és ki szeretnénk használni az ablakrendszer speciálisabb tulajdonságait, lehetőségeit, hasznosabb lehet egy natív API-t alkalmazó eszközkészlet (például az AWT-hez nagyon sok Windows specifikus kiegészítés létezik). Az esetek többségében viszont valószínűleg a platformfüggetlenség lesz a döntőbb indok.

A SWING kapcsán egy bekezdés erejéig említsük meg az SWT (*Standard Widget Toolkit*) eszközkészletet, amelyet eredetileg az IBM fejlesztett ki, majd az Eclipse projekt részévé vált. Működési háttere inkább az AWT-hez áll közelebb, de egy sokkal gazdagabb eszközkészletről van szó, és az alapelvek szempontjából is lényegesek a különbségek (pl. MVC). Az SWT általános célú, széles körben alkalmazható, de leginkább az Eclipse fejlesztői táborában népszerű. Az SWT-re épülő *JFace* magasabb szintű grafikus komponenseket biztosító eszközkészlet az Eclipse Plug-in, illetve Eclipse RCP fejlesztésekben kap nagyon fontos szerepet. Éppen ezért az SWT bemutatása mellett szükséges lenne ezeknek a „haladóbbnak" számító témáknak is az ismertetése, és ez nem képezheti részét ennek a jegyzetnek, így talán egy következő részben kaphat majd helyet.

Fontos lehet újra kihangsúlyozni, hogy az előző fejezetekben tárgyalt legtöbb téma érvényes marad a SWING esetében is. Az eseménykezeléshez használt *awt.event* csomagot a SWING kiegészíti ugyan, biztosít saját osztályokat és interfészeket specifikus események kezelésére, de az alapvető események esetében továbbra is a már ismert eszközöket alkalmazhatjuk. Hasonlóan, bár rendelkezésünkre áll néhány új elrendezésért felelős osztály (pl. *BoxLayout*), továbbra is alkalmazhatjuk az *awt* csomag *LayoutManager* megvalósításait. A komponensek esetében viszont fontos, hogy egy felületen belül ne keverjük az AWT és SWING komponenseket. A magyarázatra a későbbiekben kitérünk, és ismertetjük a grafikával kapcsolatos különbségeket is [12]. Ezt megelőzően, a következő részben bemutatjuk a SWING felületek elkészítésének alapjait, és az alapvető komponenseket.

5.1 SWING felületek

A SWING komponensek a *javax.swing* csomag osztályainak segítségével hozhatóak létre. A felület alapjának egy natív tárolónak kell lennie (ahhoz, hogy egy adott ablakrendszeren belül megjelenhessen), de a további komponensek megjelenítéséhez már nem szükségesek natív *peer* objektumok. Gyakorlatilag ezek a komponensek kirajzolják önmagukat a natív konténeren belüli tartalom-panel felszínére.

Az alap keretet a *JFrame* osztály segítségével hozhatjuk létre. A *JFrame* osztály a *Frame* osztály kiterjesztése, amely a *Frame*-hez hasonló funkcionalitásokat biztosít. Az alapvető különbség abból ered, hogy a *JFrame* tárolóhoz nem közvetlen módon adjuk hozzá a komponenseinket, hanem a keretnek megfelelő tartalom-panelt használjuk erre a célra. Ezen kívül a *JFrame* még biztosít számunkra néhány plusz funkcionalitást (például alapértelmezett művelet megadása az ablak bezárása esetén, stb.).

5.1.1 Tartalom-panel

A SWING keretekhez (*JFrame* objektumok) nem közvetlen módon adjuk hozzá a komponenseket, hanem a keretnek megfelelő *Container* típusú tartalom-panelt használjuk erre a célra. Minden *JFrame* objektumnak megfelel egy *JRootPane* típusú objektum. A *JRootPane* objektumhoz tartozik egy tartalom-panel (*content pane*) és opcionálisan egy menüsor (*menu bar*), valamint egy „üvegtábla" (*glass pane*). Az első kettő kezeléséért egy *JLayeredPane* típusú objektum felelős. Ez a tároló típus lehetővé teszi azt, hogy rétegeket (*layers*) határoz-

zunk meg a felületén belül, és a hozzáadott komponensek esetében megszabjuk, hogy melyik rétegre szeretnénk helyezni őket, tehát melyik komponens fogja esetlegesen fedni a másikat (*z-ordering*). Az üvegtábla egy átlátszó komponens, amely a többi fölé kerül, és így lehetőséget biztosít például az egérműveletek „elkapására".

A pehelysúlyú grafikus komponenseinket a tartalom-panelhez adjuk hozzá. Az alapértelmezett kerethez rendelt tartalom-panel lekérdezhető a *JFrame getContentPane* nevű metódusával, amely egy *Container* típusú referenciát térít vissza. Azt is megtehetjük, hogy létrehozunk egy tárolót (például *JPanel*), ehhez adjuk hozzá a komponenseket, majd a keret *setContentPane* metódusának meghívásával ezt a tárolót állítjuk be tartalom-panelnek.

Példa:

```
Container contentPane = this.getContentPane();
contentPane.setLayout(new FlowLayout());
contentPane.add(button);
```

Vagy:

```
JPanel panel = new JPanel();
panel.add(button);
this.setContentPane(panel);
```

A fenti példák esetében a keret (*JFrame*) osztályból származtatott osztályon belül vagyunk, tehát a *this* referencia az aktuális keret objektumra mutat. A *button* referencia egy *JButton* típusú objektumra mutat.

5.1.2 Megjelenítési szabvány

A SWING esetében a komponensek megjelenítése már nem platformfüggő, az *UIManager* osztály segítségével befolyásolhatjuk a felületünk kinézetét, úgy, hogy különböző *Look and Feel* (L&F) típust rendelünk hozzá a felületünkhöz. Ezt a *javax.swing.plaf* csomagon, valamint ennek alcsomagjain belüli osztályok és interfészek teszik lehetővé.

A SWING biztosít számunkra néhány standard L&F csomagot, de külső csomagokat, illetve speciális megjelenítéseket biztosító saját megvalósításokat is alkalmazhatunk. A *javax.swing.plaf* csomag absztrakt alaposztályokat biztosít, és a különböző L&F-t megvalósító csomagok osztályai ezekből lesznek származtatva. Az alapértelmezett Java L&F, amelyet még *Metal* L&F-nek, vagy *CrossPlatform* L&F-nek is neveznek a *javax.swing.plaf.metal* osztályait használja. Ha nem állítunk be semmilyen L&F-t, a felületünk ennek megfelelően fog megjelenni. Beállíthatunk adott ablakrendszernek megfelelő megjelenítést (ezeket nevezzük *System* L&F-nek). Erre például a *com.sun.java.swing.plaf* csomag és alcsomagjai (pl. *com.sun.java.swing.plaf.windows*) adnak lehetőséget. Amennyiben saját L&F csomagot szeretnénk létrehozni, a *javax.swing.plaf.basic* csomag osztályaiból indulhatunk ki. Ha úgy szeretnénk speciálisabb megjelenítést, hogy a vonatkozó tulajdonságokat XML állományok segítségével írjuk le, a *javax.swing.plaf.synth* csomag szolgáltathat számunkra alapot. A SWING lehetőséget ad arra is, hogy egy felületen belül különböző L&F csomagokat kombináljunk (*multiplexing*), és ebben a *javax.swing.plaf.multi* csomag osztályai lehetnek segítségünkre. Példaként gondolhatunk egy olyan helyzetre, amikor a vizuális megjelenítést meghatározó L&F csomagot kombinálni szeretnénk a felületekhez, illetve eseményekhez hangjelzéseket társító csomaggal (pl. *text to speech* támogatás), így segítve a látássérült felhasználókat.

A cserélhető L&F működési mechanizmusának alapja, hogy minden komponensnek tulajdonképpen két megjelenítésért felelős osztály felel meg: az egyik a *JComponent* osztály leszármazottja, a másik a megfelelő *ComponentUI* osztály. Például egy *JButton* objektumhoz tartozik egy *ButtonUI* típusú objektum, ahol a *ButtonUI* a *ComponentUI* leszármazottja. A *ComponentUI* osztályok példányai, amelyeket UI *delegate*-eknek neveznek, felelnek a komponensek megjelenítéséért. A programozónak általában nem kell közvetlen módon hozzáférnie ezekhez a *delegate*-ekhez, viszont a *JComponent* osztályok belsejükben felhasználják őket a megjelenítéshez, például hozzájuk „delegálják" a rajzolási műveleteket. Az UI *delegate* konkrét megvalósításának kiválasztása a meghatározott L&F osztálynak megfelelően történik. Például, ha a standard Java megjelenítésnek megfelelő *MetalLookAndFeel* osztályt alkalmazzuk, akkor egy *JButton* típusú komponenshez egy *MetalButtonUI* típusú *delegate* lesz hozzárendelve.

Egy felület esetében több lehetőségünk is van az alkalmazott L&F osztály (és ennek megfelelő csomag) beállítására. A legegyszerűbb a következő módszer:

```
import javax.swing.UIManager;
import javax.swing.UnsupportedLookAndFeelException;
import javax.swing.plaf.metal.MetalLookAndFeel;
…
      try {
            UIManager.setLookAndFeel(new MetalLookAndFeel());
      } catch (UnsupportedLookAndFeelException e) {
            e.printStackTrace();
      }
```

A beállításnak azelőtt kell megtörténnie, mielőtt létrehozzuk a felszín alkotóelemeit. Beállíthatunk különböző megjelenítésre vonatkozó tulajdonságokat is. Például egyes L&F típusok (a *Metal* is) lehetővé teszi, hogy több különböző téma (*theme*) közül választva, egyszerűen módosíthassuk a felület bizonyos tulajdonságait, például színvilágát. Kipróbálhatjuk, ha kiegészítjük a kódunkat, a *try* rész elejére beszúrva az alábbi kódsort:

```
MetalLookAndFeel.setCurrentTheme(new DefaultMetalTheme());
```

Az *UIManager* osztály még biztosít számunkra néhány hasznos metódust, például arra az esetre, amikor be szeretnénk azonosítani valamilyen L&F osztály nevét. Ha azt szeretnénk, hogy a felszínünk mindig az adott ablakrendszernek megfelelően jelenjen meg, akkor az *UIManager getSystemLookAndFeelClassName* metódusával lekérhetjük a megfelelő osztály nevét, és a *setLookAndFeel* metódusnak létezik egy olyan változata is, amely ezt a nevet (egy *String* objektumot) kaphatja paraméterként. Megjegyzendő, hogy a metódus meghívása további kivételek előfordulását eredményezheti, így a használat előtt nézzük meg a vonatkozó specifikációt.

5.1.3 SWING komponensek

A SWING önmagában is egy nagyon gazdag grafikus eszközkészlet, amelyhez még rengeteg külső kiegészítés is létezik, így természetesen nincs lehetőségünk arra, hogy minden komponenst ismertessünk. A teljesség igénye nélkül azért felsorolunk néhány alapvető, gyakran használt grafikus komponenst.

Említettük, hogy minden AWT komponensnek van SWING megfelelője, és a legtöbb esetben ezek néhány plusz lehetőséget is biztosítanak. Általában az alkalmazott elnevezési szabály, hogy nevük egy „J" előtaggal bővül (ahogyan azt már láttuk például a *JFrame* esetében). Néhány példa:

- *JLabel*, *JButton*: ikon rendelhető hozzájuk, és lehetőséget adnak *html* szövegformázásra. Megjegyzendő, hogy ez utóbbi funkcionalitást a legtöbb SWING komponens biztosítja;
- *JTextComponent*: ebből az osztályból lesznek származtatva az alapvető szövegfeldolgozással kapcsolatos grafikus komponensek, a *JTextField*, a *JTextArea*, a *JTextPane*, és a hierarchia egy következő szintjén speciálisabb komponensek, például a *JTextField* osztályból származtatott *JPasswordField* (jelszavak beolvasására, adott *echo* karakterek használatával). Az alaposztály néhány szövegfeldolgozással kapcsolatos hasznos metódust biztosít: *copy, cut, paste, getSelectedText, setSelectionStart, setSelectionEnd, selectAll, replaceSelection, getText, setText, setEditable, setCaretPosition,* stb.;
- *JComboBox*: a *Choice* SWING megfelelője, lehetővé teszi egy szerkeszthető mező használatát;
- *JScrollBar*: a *ScrollBar* SWING változata. Egy hasonló komponens a *JSlider*, amely lehetővé teszi a felhasználónak egy érték kiválasztását egy intervallumból, csúszka segítségével, és különböző beállításokra ad lehetőséget (lépték, megjelenítési tulajdonságok, stb.);
- *JList, JCheckBox, JMenu, JMenuBar, JMenuItem*: a hasonló tulajdonságú AWT komponensek SWING megfelelői.

A tárolók esetében már találkoztunk a *JFrame* osztállyal. Hasonlóan megtalálhatjuk például a *JWindow*, *JDialog* osztályokat is. Megjegyzendő, hogy ezek megjelenítésekor szükségesek natív *peer* objektumok. Kivételt képez a *JInternalFrame* osztály, amely segítségével egy standard keret tulajdonságaival rendelkező belső ablakot hozhatunk létre. A pehelysúlyú tárolók közül néhányat már megemlítettünk. Közülük a leggyakrabban talán a *JPanel* osztályt használjuk. Rendelkezésünkre áll többek közt még a *JTabbedPane* osztály, amely a *CardLayout* által alkalmazott elrendezéshez hasonló megjelenítést tesz lehetővé egy elegánsabb módon, valamint a *JSplitPane* osztály, amely két komponens elválasztására alkalmas, olyan módon, hogy az általuk elfoglalt felület mérete a felhasználó által változtatható legyen.

A SWING számos olyan hasznos komponenst is biztosít számunkra, amelyekkel az AWT esetében még nem találkoztunk. Megemlíthetjük például a *JProgressBar* osztályt, amely időigényesebb folyamatok állapotának megjelenítésére, monitorizálására szolgál (pl. egy ciklus belsejében változtathatjuk az állapotát, információt adva a felhasználónak arról, hogy hogyan halad a műveletsor elvégzése). A táblázatok megjelenítésére használható a *JTable* osztály, hierarchikus adatstruktúrák megjelenítésére alkalmas a *JTree* osztály. Ezeknek esetében nagyon fontos szerepet kapnak a hozzájuk tartozó modell objektumok (*TableModel* és *TreeModel* típusúak), valamint az adatok kezelésére alkalmazható osztályok, amelyek a *javax.swing.table* és *javax.swing.tree* csomagokban találhatóak. Hasonlóan hasznosak lehetnek a *JColorChooser*, illetve *JFileChooser* osztályok, amelyek színek, illetve állományok kiválasztását támogatják, és a *javax.swing.filechooser*, illetve *javax.swing.colorchooser* csomagok számos kapcsolódó lehetőséget biztosítanak.

A SWING nem kizárólag csak grafikus komponenseket biztosít számunkra, hanem több más hasznos eszközt is. Kiemelhető például az időzítésre alkalmazható *Timer* osztály, amelynek példányai megadott időközönként *ActionEvent* típusú eseményeket generálhatnak.

Rendelkezésünkre áll a *javax.swing.undo* csomag, amelynek segítségével *undo/redo* funkcionalitást építhetünk be felszínünkbe (például szövegszerkesztő komponensek esetében). A különböző szövegszerkesztéssel kapcsolatos komponensekkel kapcsolatban segítséget nyújthat a *javax.swing.text* csomag, illetve annak alcsomagjai (például *javax.swing.text.html*, vagy *javax.swing.text.rtf*).

5.1.4 Hello, SWING!

Az előző részben említettük, hogy egyes SWING komponensekhez ikonokat rendelhetünk. Ezek az ikonok olyan osztályok példányai, amelyek megvalósítják az *Icon* interfészt:

```
public interface Icon {
      void paintIcon(Component c, Graphics g, int x, int y);
      int getIconWidth();
      int getIconHeight();
}
```

Példaként hozzunk létre egy kis piros kört ábrázoló ikont:

```
public class RedOval implements Icon {
      public void paintIcon(Component c, Graphics g, int x, int y) {
            g.setColor(Color.red);
            g.drawOval(x, y, getIconWidth(), getIconHeight());
      }
      public int getIconWidth() {
            return 10;
      }
      public int getIconHeight() {
            return 10;
      }
}
```

A *paintIcon* metóduson belül végezzük el a tulajdonképpeni rajzolást. A metódus paraméterként kapja a komponensre mutató referenciát, a rajzoláshoz szükséges *Graphics* objektumra mutató referenciát, valamint az ikon komponensen belüli pozícióját. A második és harmadik metódus a méretek lekérdezésére szolgál.

Arra is lehetőségünk van, hogy az ikonnak megfelelő kis képet állományból olvassuk be. Erre a célra az *ImageIcon* osztályt használhatjuk:

```
Icon iconPicture = new ImageIcon("Apple.gif");
```

Nézzünk most egy egyszerű alkalmazást:

```
import java.awt.Color;
import java.awt.Font;
import java.awt.GridLayout;
import javax.swing.Icon;
```

```
import javax.swing.ImageIcon;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;
public class IconExample extends JFrame {
      public IconExample() {
            this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
            this.setTitle("Hello SWING!");
            JPanel contentPane = new JPanel();
            contentPane.setBackground(Color.white);
            contentPane.setLayout(new GridLayout(3, 1));
            JLabel simpleLabel = new JLabel("Hello SWING!",
                                                JLabel.CENTER);
            contentPane.add(simpleLabel);
            JLabel iconLabel = new JLabel("Hello SWING!");
            Icon icon = new ImageIcon("duke.jpg");
            iconLabel.setIcon(icon);
            Font font = new Font("Serif", Font.BOLD|Font.ITALIC, 20);
            iconLabel.setFont(font);
            contentPane.add(iconLabel);
            JLabel myIconLabel = new JLabel("<html> Hello <br>
                              SWING! </html>", JLabel.CENTER);
            Icon myIcon = new RedOval();
            myIconLabel.setIcon(myIcon);
            contentPane.add(myIconLabel);
            this.setContentPane(contentPane);
      }
      public static void main(String[] args) {
            IconExample ie = new IconExample();
            ie.setBounds(50,50,230,200);
            ie.setVisible(true);
      }
}
```

Az eredmény:

5.1 ábra: Formázott címkék

A szükséges osztályok importálása után a *JFrame* osztályból származtatott *IconExample* osztály konstruktorán belül megadjuk az alapértelmezett műveletet az ablak bezárása esetén (kilépés az alkalmazásból), és a keret címét. Ezután létrehozunk egy *JPanel* típusú komponenst, amely tartalom-panelként fog szolgálni, majd sorban létrehozzuk és hozzáadjuk a komponenseinket. Három címkét hozunk létre, a *„Hello SWING!"* felirattal: egy egyszerű címkét, egy állományból beolvasott, ikonnal ellátott, formázott szövegű címkét, és egy saját ikonnal (piros kör) ellátott címkét, ahol a szövegformázásra *html tag*-eket alkalmazunk. Miután a konstruktor végén beállítjuk tartalom-panelnek az általunk létrehozott panelt, a *main* metóduson belül létrehozzuk, és láthatóvá tesszük a keretet.

Az állományból beolvasott ikon esetében meg kellett adnunk az állomány nevét (a névvel együtt útvonalat is megadhatunk), esetünkben ez "*duke.jpg*", és a kép a James Gosling és társai által Java ikonként javasolt közismert Duke figura egyik változata.

A jegyzet előző részében láthattuk, hogy a SWING által biztosított lehetőségek száma óriási. Egy ilyen rövid jegyzet semmiképpen sem lenne elegendő mindegyiket példával szemléltetni, így a kísérletezés az olvasó egyéni feladata marad. A SWING specifikációja mellett még nagyon sok Internetes (és más) dokumentáció nyújthat ebben segítséget. Az előző rövid példát csak egy nagyon egyszerű ízelítőnek szántuk. A jegyzet következő részében bemutatunk néhány alapelvet és tulajdonságot, amelyek ismerete fontos lehet ahhoz, hogy a kísérletezés igazán szórakoztató legyen.

5.2 SWING tulajdonságok

A fejezet első részében megemlítettük az AWT és SWING működési alapelvei közötti különbségeket. Ezekből a különbségekből adódnak bizonyos következmények, amelyek ismerete szükséges lehet ahhoz, hogy hibamentes programokat írjunk.

5.2.1 Pehelysúlyú és nehézsúlyú komponensek

Ahogyan azt már említettük, míg az AWT komponensek nehézsúlyúak (*heavyweight*), a SWING esetében, az ablakoknak megfelelő alaptárolókat leszámítva, pehelysúlyú (*lightweight*) komponensekről beszélhetünk.

Az egyik fontos különbség a komponensek mélységi rendezésében (*z-ordering*) mutatkozik. A komponensek mélységi sorrendje a tárolóhoz történő hozzáadásuk sorrendjét követi, tehát a később hozzáadott komponensek fedhetik az előzőleg hozzáadottakat. Ezen kívül láttuk, hogy a SWING esetében a *LayeredPane* segítségével meghatározhatjuk a sorrendet.

Probléma akkor adódhat, ha egy felszínen belül keverjük a különböző típusú (AWT és SWING) komponenseket. Minden AWT komponens gyakorlatilag egy-egy megfelelő natív tárolóban jelenik meg, ezzel szemben a SWING komponensek „kirajzolják magukat" a tároló felületére. Következményként, az AWT nehézsúlyú komponensek mindig fedni fogják a SWING pehelysúlyú komponenseit (mivel a nekik megfelelő natív tárolók egy következő szintre, az alaptároló fölé kerülnek, míg a SWING komponensek „öröklik" az őket tartalmazó tároló szintjét).

Bár a Java SE 6.0-ás verziója tartalmaz a problémára vonatkozó javításokat, kiegészítéseket, ezek többnyire platform-specifikusak. Egy jó tanács tehát: ne keverjük egy tárolón belül a különböző komponens típusokat!

Fontos különbség, hogy a nehézsúlyú komponensek nem tartalmazhatnak átlátszó részeket. Ezt úgy is fogalmazhatjuk, hogy a nekik megfelelő téglalap alakú felület minden pixele ki lesz rajzolva, a rendszer az üres részeket háttérszínű pixelekkel tölti ki. A SWING esetében ez nem így van. A komponensek tartalmazhatnak átlátszó részeket, és így különböző alakúak lehetnek (nem feltétlenül téglalap alakúak). Ha a komponensek fedik egymást, átlátszó részeiken az események „áthaladnak", és az alsó komponenshez továbbítódnak (például egérműveletek esetében).

5.2.2 SWING grafika

Az AWT komponensek megjelenítésével kapcsolatban tudjuk, hogy a felület frissítése kétféle módon történhet: a rendszer, vagy az alkalmazás kezdeményezésére. Ez a SWING esetében is így van, de a különbség sokkal kevésbé releváns.

Ennek oka, hogy a SWING komponensek rendszer általi frissítése is a *repaint* metódushíváson keresztül történik. A komponensek felelősek önmaguk kirajzolásáért, a frissítés nem natív API-en keresztül valósul meg. A SWING komponensek frissítése tehát minden esetben a *repaint* metódus meghívását eredményezi, amely *update* metódushíváshoz vezet. A *JComponent update* megvalósítása egyszerűen meghívja a *paint* metódust, tehát nem törli a felületet. Fontos viszont, megjegyezni, hogy a *paint* meghívása adott pillanatban a komponensnek megfelelő *ComponentUI* osztály *update* metódusának meghívását eredményezi. Ennek a metódusnak a *ComponentUI* által biztosított alap megvalósítása, szintén törli a felületet, azaz teljes egészében (az üres részeket is beleértve) kitölti a felületet háttérszínű pixelekkel. Ezt a műveletet azonban csak akkor hajtja végre, ha az illető komponens *opacity* tulajdonságának értéke *true* (erről a későbbiekben még szó esik). Ezen kívül a származtatott osztályok a metódust fölül is írhatják.

További fontos különbség, hogy mivel a komponensek felelősek saját kirajzolásukért, a tárolók frissítésekor be kell járni a hozzájuk tartozó komponensek hierarchiáját, kérve a komponensektől felületük frissítését. Amennyiben a komponensek átlátszó részeket is tartalmaznak, és fedik egymást, a bejárás és frissítés során ezt is figyelembe kell venni.

Ha egy natív tárolón belül pehelysúlyú komponenseket is elhelyezünk, a megfelelő megjelenítéshez a származtatott osztályon belül a *paint* újradefiniálásánál nagyon fontos, hogy meghívjuk az alaposztály *paint* metódusát (*super.paint(Graphics g)*), ellenkező esetben komponenseink nem jelennek meg. A *Container* osztály *paint* metódusa meghívja az általa tartalmazott látható pehelysúlyú komponensek *paint* metódusát, így megjelenítve azokat.

A SWING három részre osztja a *paint* metódust, külön kezelve a komponens felületének frissítését, az általa tartalmazott komponensek frissítését, illetve a komponensnek megfelelő keret frissítését. A keretekkel kapcsolatosan megemlíthető a *BorderFactory* osztály, amely osztályszintű gyártó metódusokat biztosít különböző típusú keretek (*Border* objektumok) létrehozására, és ezeket a *JComponent* osztály *setBorder* metódusának segítségével hozzárendelhetjük komponenseinkhez.

Tulajdonképpen a *paint* metódushívás három különálló metódus meghívását jelenti:

```
protected void paintComponent(Graphics g)
protected void paintBorder(Graphics g)
protected void paintChildren(Graphics g)
```

Nagyon fontos megjegyzés, hogy a legtöbb esetben nem akarjuk a teljes *paint* metódust újradefiniálni. A legtöbbször mi csak a komponens felületére szeretnénk rajzolni, nem akarjuk átalakítani például a tartalmazott komponensek megjelenítési mechanizmusát. Ezért csak akkor definiáljuk újra a *paint* metódust, ha valóban ezt szeretnénk, egyébként a *paintComponent* metódust írjuk fölül!

Amennyiben megfelelően jártunk el, és a *paintComponent* metódust definiáltuk újra, akkor is fontos lehet az alaposztály metódusának meghívása. Például, ha rajzolni szeretnénk és vászonként egy *JPanel*-t használunk (a *swing* csomagon belül nincs *Canvas* osztály, és nem jó ötlet keverni a különböző típusú komponenseket), valószínűleg azt akarjuk, hogy a panel továbbra is rendelkezzen az egyszerű panelektől megszokott tulajdonságokkal. Például szeretnénk, ha egy egyszerű metódushívással beállíthatnánk a háttér színét (*setBackground*). Ezt egy *JPanel* esetében megtehetjük, mivel annak *paintComponent* metódusa (az *opacity* tulajdonság alapértelmezett értékének megfelelően) törli a felületet, és az üres részeket háttérszínű pixelekkel tölti ki. Ha viszont mi újradefiniáljuk ezt a metódust, és nem hívjuk meg az alaposztály *paintComponent* metódusát, ez nem történik meg, a panel felülete átlátszó marad, és így például a háttérszín beállítása sem fog működni.

A SWING komponensek esetében az „átlátszatlanság" (*opacity*) tulajdonság a következő metódussal kérdezhető le:

```
public boolean isOpaque()
```

A metódus igaz (*true*) értéket ad, ha a komponens felületének minden pixele ki lesz rajzolva, tehát a komponens nem tartalmaz átlátszó részeket. A legtöbb komponens esetében ez az alapértelmezett beállítás. A metódus hamis (*false*) értéket ad, ha a komponens nem garantálja, hogy nem tartalmaz átlátszó részeket. Figyelem: ez nem átlátszóságot jelent, csak azt, hogy a komponens nem vállal garanciát arra, hogy nem fog átlátszó részeket tartalmazni. A komponenseknek erre a tulajdonságára inkább úgy tekintsünk, mint egyfajta „szerződésre", de a komponenseknek biztosítaniuk kell a megfelelő implementációt is, és ebben nem mindig lehetünk biztosak. Például egy *JPanel* esetében a metódus alapértelmezetten igaz értéket ad. Ha nem megfelelő módon definiáljuk újra a *paintComponent* metódust, az *opacity* tulajdonságnak megfelelő attribútum értéke változatlan marad, viszont az új komponens típus már nem tartja be a „szerződést". Érdemes erre odafigyelni, hogy elkerüljük a hibás megjelenítést.

Azt már említettük, hogy az átlátszó részeket is tartalmazó komponensek közötti átfedések bonyolulttá teszik a tárolók felületének frissítését. A frissítést végző rendszer optimalizálhatja a hierarchia bejárását. Például, a komponens biztosíthatja a rendszert, hogy nem tartalmaz egymást fedő komponenseket. Ez a tulajdonság a következő metódussal kérdezhető le:

```
public boolean isOptimizedDrawingEnabled()
```

Ha a metódus igaz (*true*) értéket ad, akkor a tároló garantálja, hogy nem tartalmaz egymást fedő komponenseket (persze a megfelelő megjelenítéshez, a hibák elkerüléséhez ezt be is kell tartania). A *false* érték azt jelenti, hogy a tároló nem vállal átfedésekre vonatkozó garanciát.

A SWING grafikával kapcsolatos jellegzetességeinél még megemlíthető, hogy a komponensek alapértelmezetten biztosítanak kettős pufferelés (*double buffering*) támogatást. Ez a tulajdonság a következő metódus segítségével kérdezhető le:

```
public boolean isDoubleBuffered()
```

A legtöbb esetben a metódus igaz értéket fog visszatéríteni, mivel a legtöbb standard komponens biztosítja ezt a lehetőséget.

Remélhetőleg ezzel a nagyon rövid összefoglalóval sikerült meggyőzni az olvasót arról, hogy a SWING valóban egy jól kigondolt és kivitelezett, a lehetőségeknek gazdag tárházát biztosító eszközkészlet. Bár a jelen jegyzet nem ad több lehetőséget arra, hogy ezt a kijelentést további példákkal is alátámasszuk, a leírtak meghozhatják a kedvet a kísérletezésre. Egy jó tanács lehet, hogy a jegyzet további részeinél tárgyalt témák begyakorlása közben ezt az eszközkészletet használjuk a grafikus felhasználói felületek létrehozásához. Könnyen találhatunk vizuális szerkesztőket, amelyek ebben segítségünkre lehetnek (a legtöbb fejlesztési környezet is biztosít ilyeneket), de csak azután folyamodjunk ilyen eszközökhöz, miután megtanultuk „manuálisan" felépíteni felszíneinket, így megismerve a megjelenítéssel és működéssel kapcsolatos részleteket.

5.3 Javasolt gyakorlat

Az előző fejezet végén javasolt második feladat (alakzat kirajzolása) megoldása során elkészített program grafikus felületét írjuk át SWING komponensek felhasználásával. A programot egészítsük ki néhány új funkcionalitással. Egy *JSlider* segítségével legyen változtatható az alakzat mérete, olyan módon, hogy az meg is haladhassa a vászon (esetünkben *JPanel* komponens) aktuális méretét. Amennyiben az alakzat „kilóg" a vászonból, jelenjenek meg görgetősávok, amelyek segítségével változtathatjuk az éppen látható felületet (*JScrollPane* komponenst alkalmazhatunk). A szín kiválasztására ezúttal egy külön grafikus felületet is biztosítsunk, a *JColorChooser* komponens felhasználásával.

6

Végrehajtási szálak

A számítástechnikában a folyamat (*process*) kifejezés egy végrehajtás alatt álló programot jelöl. A folyamatok központi szerepet játszanak a feladatok egyidejű végrehajtását (*concurrent computing*) támogató, *multitasking* lehetőséget biztosító rendszerek esetében. A legtöbb operációs rendszer, és ennek megfelelően a legtöbb programozási nyelv is támogatja ezt a lehetőséget. A mechanizmus alapja, hogy a folyamatok megosztják egymás között a processzszorokat és erőforrásokat, így a rendszer párhuzamosan több művelet elvégzésére képes. Ha egyetlen processzáló egységgel (egymagos processzorral) rendelkező rendszerről van szó, akkor természetesen csak pszeudo-párhuzamosságról lehet szó, a feladatok végrehajtásának egyidejűsége csak látszólagos. A megvalósítás alapja általában a processzoridő megosztása. A folyamatok tulajdonképpen nagyon gyorsan „adogatják át" egymásnak a processzáló egységet, így a felhasználónak úgy tűnik, hogy a programok párhuzamosan futnak, a műveletek elvégzése egyidejűleg történik.

Egy számítógépes alkalmazás végrehajtása egy vagy több (egymással kommunikáló) folyamat elindítását feltételezi. A folyamatok saját végrehajtási környezettel rendelkeznek, és különböző erőforrásokat foglalnak. Egy folyamat adat- és kódszegmensekből áll (a programnak megfelelő gépi kód, valamint specifikus bemeneti és kimeneti adatok), amelyek a memóriában, tipikusan egy virtuális címzési tartományon belül vannak elhelyezve. Tartozik hozzá egy hívási verem (*call stack*), az aktív alprogramok (függvények, metódusok) és események követésére, valamint egy *heap* a futási időben használt adatok tárolására. Ezen kívül hozzátartoznak még a folyamat által foglalt erőforrások (pl. állományok) azonosítói, biztonsági tulajdonságok és állapotinformációk.

A legtöbb esetben a JVM az operációs rendszer egy folyamatán belül fut, és következményként a Java programunk is ezen a folyamaton belül fog futni. Létezik lehetőség arra is, hogy egy Java program új folyamatokat hozzon létre, és léteznek speciális JVM implementációk, valamint megoldások arra, hogy JVM folyamatok adatokat osszanak meg egymás között, de ezek némileg haladóbb témának számítanak, és bemutatásuk nem képezi célját ennek a jegyzetnek.

A végrehajtási szálak (*execution threads*) [13] a folyamat végrehajtási alapegységei. A szálak folyamatokon belül léteznek (minden folyamathoz tartozik legalább egy szál), és megosztják egymás között a folyamathoz rendelt erőforrásokat. Mivel a szálak a folyamat címzési tartományán belül működnek, megosztva a memóriát és más erőforrásokat, bizonyos adatok több szál számára is hozzáférhetőek, ezért nagyon fontos a szálak közötti kommunikáció és a szinkronizálás.

Egy Java program indításakor létrejön egy elsődleges végrehajtási szál, és ez a későbbiekben további szálakat hozhat létre, amelyek a maguk során szintén új szálakat indíthatnak. Természetesen a program futtatásához az elsődleges szálon kívül más szálak háttértevékenységére

(pl. memóriakezelés, eseménykezelés, stb.) is szükség van. A több végrehajtási szálon alapuló programvégrehajtás a Java programozási nyelv és platform alapvető jellegzetessége, és ennek megfelelően a kapcsolódó lehetőségek száma is nagy. A továbbiakban ismerjünk meg néhányat ezek közül.

6.1 Szálak létrehozása

A Java programok esetében minden végrehajtási szálnak megfelel a *Thread* osztálynak egy példánya. Két lehetőségünk van egy új szál létrehozására: osztályunkat a *Thread* osztályból származtatjuk, vagy megvalósítjuk a *Runnable* interfészt. Az első megoldás kényelmes egyszerű programok esetében, de a második megoldás általánosabb, és ezért ez a javasolt. A magyarázat, hogy az interfész megvalósítása során nem öröklési viszonyról van szó, és így az osztályunk lehet egy másik osztály leszármazottja. Mivel a Java nem támogatja a többszörös öröklést, ez az első esetben nem lehetséges. Könnyen előfordulhatnak olyan esetek, amikor azt szeretnénk, hogy az osztályunk valamilyen alaposztály tulajdonságaival rendelkezzen (örököljön attól), és ugyanakkor végrehajtási szálként viselkedjen. Erre csak a második megoldás (az interfész megvalósítása) ad lehetőséget. Megjegyzendő, hogy a *Thread* osztály is megvalósítja a *Runnable* interfészt, és tulajdonképpen ez a tény teszi lehetővé az első, „kényelmesebb" megoldás alkalmazását.

Az alábbiakban mindkét megközelítésre adunk egy-egy példát. Először nézzük a származtatást:

```
public class MyThread extends Thread {
      public MyThread(String name) {
            super(name);
      }
      public void run() {
            while (true)
                  try {
                        System.out.println(getName());
                        sleep(1000);
                  } catch (InterruptedException e) {}
      }
}
```

Az új végrehajtási szálnak megfelelő *MyThread* osztályunkat a *Thread* osztályból származtatjuk. A végrehajtási szálaknak (*Thread* objektumok) nevet adhatunk, ezért létrehozunk egy olyan konstruktort, amely paraméterként kapja ezt a nevet, és meghívja az alaposztály megfelelő konstruktorát. Ez a megoldás opcionális, az alapértelmezett konstruktort is alkalmazhattuk volna. Ami fontosabb a *run* metódus. A szál indítása ennek a metódusnak a meghívásához vezet. Az osztályunk más metódusokat is tartalmazhat, de a *run* metóduson belül kell implementálnunk mindazokat a műveleteket, amelyeket a szálnak futás közben végre kell hajtania.

Esetünkben a *run* metódus egy végtelen *while* ciklust tartalmaz, és így a szálunk futása az elindítás után csak a program leállításakor ér véget. A cikluson belül kiírjuk a konzolra a szálnak

a nevét (a *Thread* osztály *getName* metódusát használjuk a név lekérdezésére), majd a *Thread* osztály *sleep* metódusának segítségével egy másodpercig várunk, „alvó" állapotba helyezve a szálat (a szálak lehetséges állapotairól a későbbiekben még szó esik). A *sleep* metódus paraméterként a várakozás idejét kapja, milliszekundumokban kifejezve. A *try-catch* szerkezetre azért van szükség, mert a *sleep* meghívása csak futó szálak esetében lehetséges. Ez természetes is: nem lenne értelme egy megszakított állapotú végrehajtási szálat alvó állapotba hozni, és ha mégis megpróbálnánk, a próbálkozás kivételt (*InterruptedException*) eredményezne. A későbbiekben azt is meglátjuk majd, hogy a szál futásának megszakításakor (*interrupt*) a *sleep* metódus azonnal visszatér, és *InterruptedException* típusú kivételt kapunk.

Megjegyzendő, hogy a *sleep* metódus meghívása csak annyit jelent, hogy az illető végrehajtási szál megadott ideig felfüggeszti saját futását. Pontos időzítésre nem használható, mivel a valós eltelt időintervallum a rendszer ütemezőjétől is függ. Természetesen az olyan gyakori esetekben, amikor nem szükséges teljes garancia arra, hogy az időzítés valóban valós idejű legyen, szálak is alkalmazhatóak. Hasonló a helyzet a Java által javasolt más alapvető időzítési mechanizmusokkal is, például a *java.util* csomag *Timer* (és a kapcsolódó *TimerTask*), valamint a *javax.swing* csomag *Timer* osztályának esetében. Bár nagyon jól megvalósítottak, és az esetek többségében kiválóan alkalmazhatóak, a valósidejűségre nincs teljes garancia (az időzítés szintén szálalapú). Ahol a precizitás kritikus követelménynek számít, speciálisabb eszközök, csomagok, API-ek használata lehet szükséges (pl. JRTS: *Java Real-Time System*).

Miután létrehoztuk a szálnak megfelelő osztályt, lássuk, hogyan hozhatunk létre végrehajtási szál példányokat, és hogyan indíthatjuk el azokat:

```
public class Control {
      public static void main(String[] args) {
            MyThread[] threads = new MyThread[2];
            threads[0] = new MyThread("First thread");
            threads[1] = new MyThread("Second thread");
            threads[0].start();
            threads[1].start();
      }
}
```

Létrehozunk egy *Control* osztályt egyetlen *main* metódussal. A *main* metóduson belül létrehozunk egy *MyThread* típusú objektumokra mutató referenciákat tartalmazó kételemű tömböt. Sorban létrehozzuk az elemeit, példányosítva a *MyThread* osztályból. A konstruktor paramétereként megadjuk az egyes szálak neveit. Ezután a *Thread* osztály *start* metódusával indítjuk a szálakat. A metódus a szálat futásra kész állapotba hozza, és így az ütemező elindíthatja a szál futását. Ez a *run* metódus meghívásához fog vezetni, és így mindkét szál futási állapotba kerül, másodpercenként kiírva saját nevét a konzolra. Megjegyzendő, hogy a *run* metódust soha nem közvetlen módon hívjuk meg, hanem mindig a *start* metódust használjuk az indításra. Ha egy ideig figyeljük a program futását (esetleg többször lefuttatjuk), azt látjuk, hogy a szálak nevei általában váltakozva lesznek kiírva a konzolra, de valószínűleg előfordul olyan eset is, amikor ez változik. A szálak futása (pszeudo-)párhuzamosan, egymástól függetlenül történik. Bár prioritásuk megegyező (erről a későbbiekben még szó esik), különböző körülmények befolyásolhatják azt, hogy melyik mikor kap processzoridőt.

Ugyanezt a példát tekintsük, ezúttal a *Runnable* interfészt megvalósítva:

```
public class MyRunnable implements Runnable {
      public void run() {
            while (true) {
                  System.out.println(
                        (Thread.currentThread()).getName());
                  try {
                        Thread.sleep(1000);
                  } catch (InterruptedException e) {}
            }
      }
}
```

Osztályunk a *MyRunnable* nevet kapja, és a *Runnable* interfészt valósítja meg. A *Runnable* interfész egyetlen metódusa a *run*, így ezt a metódust kell implementálnunk. Az előző példához hasonlóan a *MyRunnable* példányunknak megfelelő *Thread* objektum indítása ennek a metódusnak a meghívásához vezet. Mivel osztályunk nem a *Thread* osztály leszármazottja, nem használhatjuk közvetlen módon a *Thread* metódusait. Szerencsére ez nem jelent akadályt, mivel a *Thread* osztály statikus metódust (*currentThread*) biztosít az aktuális végrehajtási szálnak (esetünkben a *MyRunnable* példánynak) megfelelő objektumra mutató referencia visszatérítésére. Ennek segítségével a szál neve lekérdezhető (*getName*). A *sleep* a *Thread* osztály statikus metódusa, meghívása az aktuális szál objektumot (esetünkben a *MyRunnable* példánynak megfelelőt) helyezi alvó állapotba.

Lássuk, hogyan hozhatunk létre *MyRunnable* példányokat, ezeknek megfelelő *Thread* objektumokat, és hogyan indíthatjuk el a végrehajtási szálakat:

```
public class Control {
      public static void main(String[] args) {
            MyRunnable[] objects = new MyRunnable[2];
            objects[0] = new MyRunnable();
            objects[1] = new MyRunnable();
            Thread[] threads = new Thread[2];
            threads[0] = new Thread(objects[0], "First");
            threads[1] = new Thread(objects[1], "Second");
            threads[0].start();
            threads[1].start();
      }
}
```

Control osztályunk *main* metódusán belül ezúttal két tömböt hozunk létre. Az első *MyRunnable* típusú objektumokra, a második *Thread* típusú objektumokra mutató referenciákat tartalmaz. Az egyes tömbök létrehozása után létrehozzuk azok elemeit is, példányosítva a *MyRunnable*, illetve *Thread* osztályokból. A *Thread* példányok esetében a konstruktornak átadjuk paraméterként a *MyRunnable* objektumra mutató referenciákat, illetve a szálak neveit. A szálak indítása a *Thread* osztály start metódusával történik. Az eredmény az előző példával azonos.

6.2 Szálak prioritása

A végrehajtási szálak esetében meghatározhatunk egy prioritási szintet. A szinteket a Java szálak esetében egy egész érték jelöli a legalacsonyabb prioritási szintnek megfelelő 1-es értéktől a legmagasabb prioritási szintnek megfelelő 10-es értékig. Az alapértelmezett prioritási szint a közepes prioritásnak megfelelő 5-ös érték. A prioritási szint meghatározására a *Thread* osztály biztosít számunkra egy *setPriority* nevű metódust, amely paraméterként kapja a kívánt szintnek megfelelő értéket. A beállítás megkönnyítésére a *Thread* osztály néhány konstans adattagot is definiál. Ezek a legalacsonyabb szintnek megfelelő *Thread.MIN_PRIORITY* (értéke 1), a közepes szintnek megfelelő *Thread.NORM_PRIORITY* (értéke 5), és a legmagasabb szintnek megfelelő *Thread.MAX_PRIORITY* (értéke 10).

A prioritási szint befolyásolja, hogy a rendszer ütemezője milyen gyakorisággal adjon processzoridőt az adott szálnak. A mechanizmus hasznos lehet sok végrehajtási szálat párhuzamosan futtató rendszerek esetében. A működést könnyen kipróbálhatjuk, ha az első példánkat (szálak létrehozása) módosítjuk. A *MyThread* (vagy *MyRunnable*) osztályon belül csökkentsük le a *sleep* metódus által meghatározott várakozási időt, például egy századmásodpercre. Használjunk egy számlálót, amelynek értékét a cikluson belül minden iterációban növeljük, és kiírjuk a konzolra. A *Control* osztály *main* metódusán belül a szálobjektumok létrehozása után a *setPriority* metódus segítségével határozzunk meg számukra különböző prioritási szinteket (például, az egyiknek adjuk a legalacsonyabb, a másiknak a legmagasabb prioritási értéket). Ha a programot indítása után egy ideig futni hagyjuk (pl. kivárunk kb. 1000 iterációt), majd leállítjuk, akkor különbséget fogunk felfedezni a különböző szálaknak megfelelő számlálók értékei között, természetesen a magasabb prioritású szál javára.

Érdemes ismerni még a démon szálak (*daemon threads*) létrehozásának lehetőségét. Ezek nagyon alacsony prioritású szálak, amelyek általában háttérműveletek végrehajtására szolgálnak. Például a Java szemétgyűjtője (*garbage collector*) is egy ilyen szál. Fontos tulajdonságuk, hogy futásuk véget ér, amennyiben a folyamaton belül már nincsenek más típusú (nem démon) futó szálak. Amennyiben egy programon belül már csak démon szálak futnak, a JVM lezárja a programot. Egy szál „démon tulajdonságát" a *Thread* osztály *setDaemon* metódusával határozhatjuk meg. A metódus egy logikai típusú értéket kap paraméterként. Amennyiben egy adott szálat démon típusúba szeretnénk alakítani, a metódusnak *true* paramétert adunk át.

6.3 Szálak összekapcsolása

Gyakran előfordul, hogy mielőtt egy végrehajtási szál elvégezhetne bizonyos feladatokat, meg kell várnia, hogy egy másik szál befejezze futását. Például, lehetséges, hogy a másik szálnak végre kell hajtania bizonyos előszámításokat, vagy előkészítő műveleteket. Általános esetben az ilyen jellegű helyzetekben a következő, szinkronizálással foglalkozó részben leírt *wait-notify* mechanizmus alkalmazható, de egyszerűbb esetekben megoldást jelenthet a szálak összekapcsolása. Ha a szóban forgó szálak „ismerik" egymást, a *Thread* osztály *join* metódusa alkalmazható.

A *join* metódus meghívása azt eredményezi, hogy a meghívó fél várakozni fog egy adott ideig, vagy mindaddig, amíg a másik fél be nem fejezi futását. Ennek megfelelően a metódusnak több változata létezik:

```
public final void join() throws InterruptedException
public final void join(long millis) throws InterruptedException
```

Az első metódus nem szab meg időkorlátot, így a szál mindaddig várakozik, amíg a másik be nem fejezi futását. A második metódus egy időkorlátot is meghatároz. Ha ennyi idő alatt a másik szál futása nem ér véget, a szál nem vár tovább, folytatja saját tevékenységét. A metódusnak létezik kétparaméteres változata is, amelynek segítségével pontosabban is meghatározhatjuk az időkorlátot. A második paraméter segítségével a nanoszekundumoknak megfelelő értéket adhatjuk át. Megjegyzendő, hogy a *sleep* meghívásához hasonlóan itt sem valós eltelt időről van szó, hanem a várakozás idejéről.

A szálak összekapcsolását egy egyszerű példával szemléltetjük. Két szálat hozunk létre. Az első néhányszor, adott időközönként kiír egy üzenetet a konzolra. A második megvárja, amíg az első befejezi a futását, mielőtt ő is ugyanígy tenne.

Az első szálnak megfelelő osztály forráskódja:

```
public class MyThread1 extends Thread {
      public MyThread1(String name) {
            super(name);
      }
      public void run() {
      System.out.println(getName() + " is running");
            for (int i = 0; i < 5; i++) {
                  try {
                        sleep(500) ;
                  } catch (InterruptedException e) {}
                  System.out.println(getName()+
                              " writes" + Integer.toString(i));
            }
      }
}
```

A *run* metódusba lépve a szál kiír egy üzenetet a konzolra, majd egy *for* cikluson belül ötször egymás után kiír még egy-egy üzenetet, fél másodperces időközönként.

A második szálnak megfelelő osztály forráskódja:

```
public class MyThread2 extends Thread {
      public MyThread1 waitedThread;
      public MyThread2(String name, MyThread1 waitedThread) {
            super(name);
            this.waitedThread = waitedThread;
      }
      public void run() {
            System.out.println(getName() +
                  " waits for " + waitedThread.getName());
```

```
            try {
                  waitedThread.join();
            } catch (InterruptedException e) {}
            System.out.println(waitedThread.getName() +
                                          " has been finished");
            System.out.println(getName() + " has been finished");
      }
}
```

Ahhoz, hogy meghívhassa a *join* metódust, és így megvárhassa az első szálat, a *MyThread2* példánynak ismernie kell a *MyThread1* példányt, így rendelkezik egy ilyen típusú attribútummal. A konstruktor második paramétere a *MyThread1* példányra mutató referencia, és ez lesz az attribútumunk értéke. A *run* metódusba lépve a szál kiír egy üzenetet a konzolra, és megvárja, ameddig az első szál befejezi a futását, a *waitedThread* objektumra meghívva a *join* metódust. Miután ez megtörtént, értesíti erről a felhasználót, kiírva egy üzenetet a konzolra, majd befejezi futását, előbb egy erre vonatkozó üzenetet is megjelenítve.

A kipróbáláshoz hozzunk létre egy harmadik osztályt, egy *main* metódussal:

```
public class JoinControl {
      public static void main(String[] args) {
            MyThread1 thread1 = new MyThread1("First");
            MyThread2 thread2 = new MyThread2("Second", thread1);
            thread1.start();
            thread2.start();
      }
}
```

A *main* metóduson belül létrehozzuk a két szálat, majd egymás után elindítjuk azokat. A konzolon előbb a szálak indulási üzenetei jelennek meg, majd az első szál öt üzenete, és végül a második értesítései arról, hogy mindketten befejezték tevékenységüket.

A *join* metódus paraméteres változatát is egyszerűen kipróbálhatjuk módosítva a megfelelő sort (természetesen ahhoz, hogy különbséget lássunk, két és fél másodpercnél rövidebb várakozási időt kell megszabnunk, és a második szál üzenete, miszerint az első befejezte futását, ebben az esetben valótlan lesz).

6.4 Szálak szinkronizálása

Ahogyan azt a fejezet első részében említettük, a végrehajtási szálak megoszthatnak egymás között különböző erőforrásokat. Amennyiben kritikus erőforrásokról van szó, elengedhetetlenül szükséges a szinkronizálás megvalósítása.

Gondoljunk egy olyan esetre, amikor egy végrehajtási szál kiolvas egy adatstruktúrából egy elemet, hogy elvégezzen vele bizonyos műveleteket, így megváltoztatva annak értékét. Közben egy másik szál is ugyanezt tenné, de amikor kiolvassa az elemet, annak még a régi értékét kapja. A másik szál a műveletek elvégzése után, az eredményeknek megfelelően frissíti az elem értékét, de ezt a frissítést felülírja a második szál saját számításainak végrehajtása után.

A végeredmény egy hibás érték. A megoldás az adat zárolása: a második szálnak nem szabadna hozzáférnie az illető elemhez, amíg az első be nem fejezi számításait. Hasonló helyzetek gyakran előfordulhatnak, ezért feltétlenül szükséges a végrehajtási szálak szinkronizálása. Több közismert feladatot is megfogalmaztak, amelyek olyan helyzeteket szemléltetnek, ahol erre mindenképpen szükség van (a vacsorázó filozófusok problémája, az alvó borbély problémája, a dohányzók problémája, a gyártó-fogyasztó probléma, stb.), és ezek alapján több tervezési mintát is kidolgoztak.

A folyamatok és szálak szinkronizálásának megvalósítására két közismert módszer létezik, az E.W. Dijkstra által bevezetett szemaforok alkalmazása, és a C.A.R. Hoare által bevezetett monitorok alkalmazása. Bár a Java mindkét módszert támogatja, az alapvető szinkronizálási műveletek esetében monitorokat alkalmaz.

A megközelítés alapját képező monitorok olyan objektumok, amelyeket több szál is biztonságosan használhat. Fő jellemzőik, hogy metódusaik meghívása a kölcsönös kizárás elvén (*mutual exclusion*) alapszik, azaz egyszerre csak egyetlen szál férhet hozzá ezekhez a metódusokhoz. Ennek megvalósítása zárak (*lock/mutex*) segítségével történik. A monitorok másik fontos jellemzője, hogy támogatják a várakozás-értesítés (*wait-notify*) mechanizmust, amelyről a későbbiekben még szó esik.

A gyakorlati megvalósítás Java-ban szinkronizált metódusok (*synchronized methods*), illetve szinkronizált programblokkok (*synchronized statements*) alkalmazásával lehetséges.

Azokat a metódusokat, amelyek törzsében kritikus erőforrásokhoz férünk hozzá, szinkronizáltként határozzuk meg. A szinkronizált metódusok deklarálásánál a metódus fejlécében a *synchronized* kulcsszót használjuk:

```
public synchronized void mySynchronizedMethod()
```

Minden objektumhoz hozzárendelődik egy-egy monitor (*monitor, intrinsic lock, monitor lock*). Amennyiben egy szál meghív egy szinkronizált metódust, lefoglalja az objektum monitorát és ettől kezdve az objektum zárolt (*locked*) állapotba kerül. Mindaddig, amíg a monitor fel nem szabadul, más végrehajtási szálak nem férhetnek hozzá az objektum metódusaihoz.

Példaként tekintsünk egy számlálónak megfelelő osztályt, ahol a számláló aktuális értékét egy privát attribútumban tároljuk és egy szinkronizált metódussal növeljük az értékét:

```
public class Counter {
      private long c = 0;
      public synchronized void inc() {
            c++;
      }
}
```

Mivel az *inc* metódust szinkronizáltnak deklaráltuk, egyszerre csak egyetlen végrehajtási szál hívhatja meg.

Amennyiben osztályszintű (*static*) szinkronizált metódusról van szó, az osztálynak megfelelő *Class* objektum monitorát foglalja le a metódust meghívó szál. Ebben az esetben, amíg a monitor foglalt, más szálak nem férhetnek hozzá az osztály statikus metódusaihoz.

Ha egy osztály tartalmaz egy szinkronizált metódust, és azt a metódust az osztály egy példányára mutató referencia segítségével egy végrehajtási szál meghívja, lefoglalva az objektum

monitorát, akkor a többi szál által egyetlen más metódus sem lesz meghívható a monitor felszabadításáig, beleértve a nem szinkronizált metódusokat is. Ez több esetben hátrányt jelenthet. Például, gondoljunk arra, hogy mi történne, ha az előző példánkat kiegészítenénk olyan módon, hogy két számlálót kezeljen. Bevezetnénk még egy attribútumot, és még egy metódust, amely ennek az értékét növelné. Mivel a metódusok szinkronizáltak, az egyik meghívása az objektum teljes blokkolását eredményezné, így ha egy második szál a másik számlálót szeretné növelni, ezt nem tehetné meg. Mivel a két számláló független egymástól, a teljes blokkolás fölösleges. A megoldást a szinkronizált kódblokkok alkalmazása jelenti:

```
public class Counter {
      private long c1 = 0;
      private long c2 = 0;
      private Object lock1 = new Object();
      private Object lock2 = new Object();
      public void inc1() {
            synchronized (lock1) {
                  c1++;
            }
      }
      public void inc2() {
            synchronized (lock2) {
                  c2++;
            }
      }
}
```

Példánkban ezúttal nem szinkronizált metódusokat alkalmaztunk, hanem a metódusok törzsében használtunk szinkronizált blokkokat. A blokkot határoló első kapcsos zárójel előtt használjuk a *synchronized* kulcsszót, és utána kerek zárójelek között megadunk egy referenciát. Ez a referencia arra az objektumra mutat, amelynek monitorát le szeretnénk foglalni. Esetünkben egyszerűen létrehozunk két különböző példányt az *Object* osztályból, és az ezeknek megfelelő monitorokat foglaljuk le. Így a *Counter* példányunk nem lesz blokkolva, ha egy végrehajtási szál meghívja valamelyik metódust, egy második szál még meghívhatja a másik metódust. Könnyen belátható az is, hogy ha a *lock* referenciák helyére *this*-t írtunk volna, a szinkronizált metódusok használatával azonos hatást értünk volna el (mivel így az aktuális példánynak megfelelő monitort foglaltuk volna le).

A többszálú alkalmazások és a szinkronizálás kapcsán fontos megemlítenünk a hibalehetőségeket, azokat a nem kívánt helyzeteket, amelyek elkerülésére nagyon oda kell figyelnünk. A legvalószínűbb, legtöbbször előforduló hibalehetőség a holtpont (*deadlock*). Ez akkor történik meg, amikor két szál kölcsönösen várakozik a másikra, és emiatt egyik sem haladhat tovább, a program futása holtponthoz érkezik. A helyzetet az a példa illusztrálhatja, amikor két illemtudó személy egyszerre ér oda egy bejárathoz. Ameddig az etikett alapján, a kort és nemet figyelembe véve el tudják dönteni, hogy ki lépjen be elsőnek nincsen probléma. Baj akkor van, ha ilyen jellegű különbségek nincsenek, vagy nem nyilvánvalóak. Mindketten a másik belépésére fognak várni, és így az ajtó előtt, helyben maradnak. Nem annyira gyakori, mint a *deadlock*, de néha előfordul egy másik hibalehetőség is, a *livelock*. Ez olyankor történhet meg, amikor két végrehajtási szál kölcsönösen reagál a másik viselkedésére, „mozdulataira". Amennyiben a másik cselekvése szintén egy ilyen reakció, megtörténhet, hogy olyan

patthelyzetbe kerülnek, amikor egyik sem folytathatja tovább tevékenységét. A helyzetet az a példa illusztrálhatja, amikor két előzékeny személy szembetalálkozik egy szűk folyosón. Mindkettő kitér, hogy elengedje a másikat, de így ismét egymással szembe kerülnek. Ha ezt fogják ismételgetni, megrekednek, egyik sem haladhat keresztül. További problémát jelenthet a „kiéheztetés" (*starvation*), amely akkor fordul elő, mikor egy „mohó" szál hosszú időn keresztül foglalja egy erőforrás monitorát, és így nem enged hozzáférést másoknak.

A szinkronizálás és a monitor fogalom kapcsán fontos megemlítenünk a *wait-notify* mechanizmust. Gyakran előfordulhat olyan eset, amikor egy végrehajtási szál lefoglal egy monitort, de valamilyen okból kifolyólag még nem végezheti el a feladatai. Például, előzőleg szükséges lehet, hogy más szálak még elvégezzenek bizonyos műveleteket. Az említett mechanizmus lényege, hogy az ilyen esetekben a szál átadhatja a monitort más szálnak, és egy várakozási állapotba léphet. A másik szál miután befejezi feladatait, értesíti a várakozó szálakat, átadva a felszabadított monitort.

A Java nyelvben a megvalósításra az *Object* ősosztály *wait, notify* és *notifyAll* metódusai adnak lehetőséget. Ahhoz, hogy egy szál meghívja ezeket a metódusokat egy adott objektumra, birtokolnia kell annak monitorát. A megoldás, hogy a metódusokat csak szinkronizált metódusokon, vagy programblokkokon belül hívjuk meg. Amennyiben egy végrehajtási szál meghívja a *wait* metódust, átengedi a monitort más szálaknak. A döntés meghozatalának alapja általában védett programblokkok (*guarded blocks*) alkalmazása. A megoldás lényege, hogy a szál mielőtt belekezdene saját feladatainak elvégzésébe, ellenőriz egy adott feltételt, például egy adott kontrollváltozó értékét. Ilyen módon információt szerez arról, hogy az illető műveletek végrehajthatóak-e, vagy még várakozni kell, például arra, hogy előzetesen más szálak még elvégezzenek bizonyos feladatokat. Amennyiben várakoznia kell, a szál meghívja a *wait* metódust, és átadja a monitort. Miután az a szál, amelyik az előkészítő műveletek elvégzéséért felelős, elfoglalja a monitort, elvégzi az illető műveleteket, módosítja a megfelelő kontrollváltozó értékét, hogy a feltétel teljesüljön, majd meghívja a *notifyAll* metódust, ezzel értesítve a monitorra várakozó szálakat.

A *notifyAll* metódus hatása, hogy a monitorra várakozó szálak visszatérnek a *wait* metódusból, újra futtatható állapotba kerülnek. Ettől függetlenül nem rendelkeznek prioritással más szálakkal szemben, tulajdonképpen csak újra részeseivé válnak a monitor lefoglalásáért folyó versenynek. Következményként, amikor a későbbiekben megszerzik a monitort, újra kell ellenőrizniük a feltételt, meggyőződve arról, hogy az adott pillanatban elvégezhetőek-e a műveletek. Az újraellenőrzés szükségessége azért is természetes, mert egy adott objektum esetében különböző szálak különböző programrészeken belül, különböző okokból hívhatják meg a *notifyAll* metódust. Az értesítés tulajdonképpen csak azt jelzi, hogy a „feladó" szál elvégzett bizonyos műveleteket, és átengedi a monitort. Nem biztos, hogy pontosan azokat a műveleteket végezte el, amelyekre az értesítés hatására várakozási állapotából visszatért szál várt. A fenti okok miatt a védett programblokkok esetében a feltétel ellenőrzése mindig egy ciklus on belül történik:

```
public synchronized void guardedCode() {
      while (!condition) {
            try {
                  wait();
            } catch (InterruptedException e) {}
      }
      ...
}
```

Mindaddig, amíg értesítés nem érkezik, a szál nem tér vissza a *wait* metódusból. Ennek megfelelően a fenti kód esetében minden értesítésnek egyetlen iteráció felel meg. Más metódusok törzsén belül több szál hívhat meg különböző okokból *notifyAll* metódust, ilyen módon „felébresztve" várakozó szálunkat, de csak a feltétel teljesülése után léphetünk tovább a ciklusból. Tulajdonképpen minket egy meghatározott értesítés érdekel, amelynek feladása előtt a feladó szál olyan módosításokat végzett, melyek következtében a feltétel teljesül:

```
public synchronized void waitedOperation() {
        …
        condition = true;
        notifyAll();
}
```

A *notifyAll* metódus minden várakozó szálat értesít. Általában ezt a megoldást alkalmazzuk, de helyette az *Object* osztály *notify* metódusa is meghívható. Ez a metódus a várakozó szálak közül csak egyetlen szálat fog értesíteni. A „szerencsés" szál kiválasztása nem determinisztikus módon történik, nem tudjuk meghatározni, hogy kit értesítünk. A metódus alkalmazása inkább olyan helyzetekben lehet jó megoldás, amikor nagyon sok végrehajtási szál egymáshoz hasonló műveleteket végez, és tulajdonképpen nem számít, hogy ezek közül melyiket értesítjük.

6.5 Érvénytelenített metódusok

Egy végrehajtási szál futása akkor áll le, amikor a szál befejezi feladatait, visszatér a *run* metódusból. Előfordulhatnak olyan esetek, amikor szeretnénk hamarabb leállítani a szálat, vagy szeretnénk felfüggeszteni a futását, majd a későbbiekben folytatni a végrehajtást a felfüggesztés pillanatától. A *Thread* osztályon belül találhatunk erre a célra használható metódusokat (*stop, suspend* és *resume*). A probléma az, hogy, ha megpróbáljuk ezeket alkalmazni, a fordító figyelmeztetni fog (*warning*), mivel érvénytelennek (*deprecated*) vannak nyilvánítva [14].

A Java fejlődése során gyakran előfordult, hogy egy kiadott verzió valamilyen osztálya biztosított olyan metódusokat, amelyeket a későbbiekben (egy következő verzió megjelenésekor) érvénytelennek nyilvánítottak. A stratégia célja, hogy megőrizzék a kompatibilitást: nem törölhették egyszerűen az adott metódusokat az új kiadásból, mivel akkor az előzőleg megírt programok nem működnének az új platformon. Megoldásként ezeket a problémás metódusokat benne hagyták az új verziókban is, de érvénytelennek nyilvánították, így a fordító csak figyelmeztet, de lefordítja a programot.

Ha egy metódust érvénytelennek nyilvánítottak, arra valószínűleg jó okuk volt. Általában olyan problémás metódusokról van szó, amelyek potenciális hibaforrások lehetnek. Ennek megfelelően az általános szabály az, hogy kerüljük ezeknek a metódusoknak a használatát, mivel biztos létezik jobb, biztonságosabb megoldás. A legtöbb esetben az osztályok specifikációján belül (ahol szintén fel van tűntetve az érvénytelenítés ténye) javaslatokat is találunk az érintett metódusok helyettesítésére, és a modern fejlesztői környezetek szintén útmutatást adhatnak.

Vizsgáljuk meg, hogy a *Thread* osztály esetében miért nyilvánították érvénytelennek a fentebb említett metódusokat, és mi lehet a megoldás a helyettesítésükre.

A *stop* metódus használata azért nem javasolt, mert meghívásának következményeként a szál által foglalt monitorok azonnal felszabadulnak. Ez akkor is megtörténik, ha a szál még nem fejezte be feladatait (például, a blokkolt adatoknak csak egy részét dolgozta fel), és ebben az esetben könnyen előfordulhat, hogy a leállítás hibás adatokat, és hibás működést eredményez. A *suspend* metódus használata holtpont előidézéséhez vezethet. Ha használatával felfüggesztjük egy szál futását, az általa foglalt monitorok nem szabadulnak fel, és más szálak nem férhetnek hozzá a blokkolt objektumokhoz. A *resume* metódus érvénytelenítése ennek következménye. A kérdés, hogy miként helyettesíthetjük ezeket a metódusokat?

Hogyan állítsunk le egy szálat? A megoldást az jelentheti, hogy a szálon belül egy kontrollváltozót alkalmazunk, és a szál futását ennek értékéhez kötjük. Az osztályunkon belül a megfelelő módon implementáljuk a *stop* metódust, úgy, hogy annak meghívása a kontrollváltozó értékének megváltoztatásához vezessen. Mielőtt a *run* metóduson belül használt ciklus következő iterációjába lépnénk, ellenőrizzük ennek a változónak az értékét. Ahhoz, hogy megfelelően működjön a megoldás, szükséges, hogy a szál minden használatkor (hozzáféréskor) ellenőrizze a változó értékét, kivédve azt a lehetőséget, hogy más szálak változtatásai problémához vezessenek. Ez egyszerűen megoldható, a változót *volatile* típusmódosítóval láthatjuk el. Egy másik megoldás az lehetne, ha a változó értékét csak szinkronizált programblokkon belül tennénk megváltoztathatóvá.

Tekintsük a következő példát:

```
public class ThreadExample implements Runnable {
      private volatile Thread control;
      public void start() {
            control = new Thread(this);
            control.start();
      }
      public void stop() {
            control = null;
      }
      public void suspend() {}
      public void resume() {}
      public void run() {
            Thread thisThread = Thread.currentThread();
            while (thisThread == control) {
                  try{
                        Thread.sleep(1000);
                  } catch (InterruptedException ex) {}
                  System.out.println("running");
            }
            System.out.println("finished");
      }
}
```

Osztályunk a *ThreadExample* nevet kapta, és a *Runnable* interfész megvalósítása. A megfelelő *Thread* objektumot az osztályon belül hozzuk létre. Erre szolgál a *control* nevű *Thread* típusú attribútum. Ezt a referenciát fogjuk használni a leállítási feltételben is, ezért a *volatile* típusmódosítóval látjuk el. Az osztályunkon belül a megfelelő módon implementáljuk a *start*, *stop*, *suspend* és *resume* metódusokat. Ezek a szál indítására, leállítására, felfüggesztésére

és tevékenységének folytatására használhatóak. A *suspend* és *resume* metódusok törzsét egyelőre üresen hagytuk, ezekre a későbbiekben még visszatérünk. A *start* metóduson belül létrehozzuk a végrehajtási szálnak megfelelő objektumot, és elindítjuk a végrehajtást. Ez a *run* metódus meghívását eredményezi. A *run* metóduson belül létrehozunk egy, az aktuális szálra mutató referenciát. A *while* ciklus esetében, mielőtt belépnénk egy új iterációba, ellenőrizzük, hogy ez a referencia a *Runnable* példányunknak megfelelő szálra mutat-e. Amennyiben igen, belépünk az iterációba, és egy másodpercnyi várakozás után kiírunk egy üzenetet a konzolra. A *stop* metóduson belül a *control* referenciának *null* értéket adunk. Ha ez megtörténik, a *while* ciklus fejlécében ellenőrzött feltétel többé nem lesz igaz, és így kilépünk a ciklusból, egy megfelelő üzenetet jelenítve meg a konzolon. A szál futása ekkor véget ér.

Ahhoz, hogy könnyen ki tudjuk próbálni a programunkat, hozzunk létre egy egyszerű kis grafikus felhasználói felületet, egy keretet négy gombbal. A gombok lenyomása a szál indítását, leállítását, felfüggesztését, vagy feladatainak folytatását eredményezi.

```
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JPanel;
public class ControlFrame extends JFrame {
      private JButton startButton;
      private JButton stopButton;
      private JButton suspendButton;
      private JButton resumeButton;
      private JPanel contentPane;
      private ThreadExample myThread;
      public ControlFrame() {
            setTitle("Thread example");
            setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
            contentPane = new JPanel();
            startButton = new JButton("Start");
            startButton.addActionListener(new ActionListener() {
                  public void actionPerformed(ActionEvent e) {
                        validateControls(false, true, false, true);
                        myThread = new ThreadExample();
                        myThread.start();
                  }
            });
            contentPane.add(startButton);
            stopButton = new JButton("Stop");
            stopButton.addActionListener(new ActionListener() {
                  public void actionPerformed(ActionEvent e) {
                        validateControls(false, true, false, true);
                        myThread.stop();
                  }
            });
            contentPane.add(stopButton);
```

```
            suspendButton = new JButton("Suspend");
            suspendButton.addActionListener(new ActionListener() {
                  public void actionPerformed(ActionEvent e) {
                        validateControls(false, true, true, false);
                        myThread.suspend();
                  }
            });
            contentPane.add(suspendButton);
            resumeButton = new JButton("Resume");
            resumeButton.addActionListener(new ActionListener() {
                  public void actionPerformed(ActionEvent e) {
                        validateControls(false, true, false, true);
                        myThread.resume();
                  }
            });
            contentPane.add(resumeButton);
            validateControls(true, false, false, false);
            setContentPane(contentPane);
      }
      private void validateControls(boolean start, boolean stop,
                                    boolean resume, boolean suspend) {
            startButton.setEnabled(start);
            stopButton.setEnabled(stop);
            resumeButton.setEnabled(resume);
            suspendButton.setEnabled(suspend);
      }
      public static void main(String[] args) {
            ControlFrame te = new ControlFrame();
            te.setBounds(50,50,400,75);
            te.setVisible(true);
      }
}
```

A gombok létrehozása, hozzáadása a tartalom-panelhez a konstruktoron belül történik. Miután létrehozzuk a gombokat, hozzájuk rendeljük a megfelelő figyelőket. A gombok lenyomása a *ThreadExample* megfelelő metódusainak meghívását eredményezi, a *myThread* referencián keresztül. A *main* metódusban példányosítunk az osztályból, beállítjuk a keret pozícióját és méreteit, majd láthatóvá tesszük.

A szál aktuális állapotának függvényében bizonyos parancsok érvényesek, mások nem. Például, ha még nincs elindítva a szál, akkor nem függeszthetjük fel, vagy folytathatjuk a futását, és nem állíthatjuk le. Mivel a program csak egyetlen végrehajtási szál futásának vezérlését támogatja, ha már elindítottunk egy szálat, nem indíthatunk el még egyet, mielőtt az elsőt leállítanánk. Ha a szál futását felfüggesztettük, nem lenne értelme, hogy újra kiadjuk a felfüggesztési parancsot.

A fentieknek megfelelően, a szál aktuális állapotának függvényében kell beállítanunk azt, hogy a felhasználó milyen parancsokat adhat ki, milyen gombokra kattinthat. Adott pillanatban bizonyos gombok lenyomása engedélyezett, másoké nem. Azt, hogy adott pillanatban melyik

gombok legyenek lenyomhatóak a *validateControls* metóduson belül határozzuk meg. A metódus négy logikai típusú változót kap paraméterként, és ezek alapján állítja be a gombok állapotait. Kezdetben csak a *Start* gomb lenyomható, majd minden gomblenyomásnál a megfelelő módon változtatjuk az állapotokat.

Ha elindítjuk a programot, és a *Start* gombra kattintunk, létrejön és elindul a végrehajtási szál. A konzolon ezután szabályos időközönként megjelenik a szál üzenete. Ha lenyomjuk a *Stop* gombot, a szál kiírja a megfelelő üzenetet, és ezzel futása befejeződik (a *Suspend* és *Resume* gombok lenyomásának esetében egyelőre a gombok állapotának változásán kívül semmi nem történik, de a későbbiekben ezeket is működésre bírjuk).

Végeztünk? Sajnos nem. Próbáljuk ki, mi történik, ha a szál osztályon belül egy nagyobb értéket adunk meg a *sleep* metódus paramétereként (pl. tíz másodpercet). Miután elindítottuk a szálat, és megpróbáltuk leállítani, a tényleges leállás nem fog megtörténni, csak miután a szál visszatért a *sleep* metódusból. Hasonló lenne a helyzet, ha a szál várakozási állapotban lenne. A megoldást az *interrupt* metódus alkalmazása jelenti. A metódus meghívásának hatására a *sleep, wait* vagy *join* metódusok azonnal visszatérnek. A metódus meghívás egy alvó vagy várakozó szál esetében *InterruptedException* típusú kivételt eredményez, és ezt a megfelelő módon kezelnünk kell.

Nézzük, hogyan módosulna a programunk a fentieket figyelembe véve. Először a *stop* metóduson kell változtatnunk:

```
public void stop() {
      Thread tmp = control;
      control = null;
      tmp.interrupt();
}
```

A metóduson belül létrehozunk egy segédváltozót. A referencia a szál objektumra mutat. Ezután a *control* referencia értékét *null*-ra állítjuk, majd a segédreferencia segítségével meghívjuk az *interrupt* metódust. Ha a szál éppen várakozási állapotban van, a metódushívás hatására azonnal visszatér, és *InterruptedException* típusú kivételt kapunk, amelyet a *run* metóduson belül a megfelelő helyen kezelünk, kilépve a *while* ciklusból.

```
public void run() {
      Thread thisThread = Thread.currentThread();
      while (thisThread == control){
            try {
                  Thread.sleep(5000);
            } catch (InterruptedException ex) {
                  break;
            }
            System.out.println("running");
      }
      System.out.println("finished");
}
```

Hasonló módon, ha a *break* helyett *return* utasítást alkalmazunk, azonnal leállíthatjuk a szál futását, visszatérve a *run* metódusból (esetünkben ekkor a „*finished*" üzenet már nem jelent volna meg).

Megjegyzendő, hogy egy futó szál esetében az *interrupt* metódus meghívása nem vezet a szál leállításához. Csak egy állapotváltozó (*flag*) értéke módosul. Az érték lekérdezhető az *isInterrupted* nevű metódus segítségével (egy példány esetében), vagy a *Thread* osztály *interrupted* nevű statikus metódusának segítségével. Míg az első metódus nem változtat a változó értékén, csak visszatéríti azt, a második metódus meghívása *false* értékre módosítja az állapotváltozót (az előző értéket térítve vissza). A feladat, hogy a szál leállítását, futásának megszakítását, a változó értékét figyelembe véve megfelelő módon megvalósítsa, a programozóra hárul.

A fenti megoldás (vagy annak hasonló változata) már a legtöbb esetben alkalmazható szálak leállítására, de még van egy apró „szépséghibája". Amennyiben egy szál blokkolt állapotban van, például, mert egy I/O művelet esetében bejövő adatokra vár, az *interrupt* metódus nem „zökkenti ki" ebből az állapotból. Az ilyen esetekre nincs általános recept, a konkrét helyzetnek megfelelő speciális megoldást kell alkalmaznunk. Például *interrupt* metódushívás helyett lezárhatjuk az I/O kapcsolatot.

Most, miután láttuk, hogy milyen módon állíthatjuk le végrehajtási szálak futását a *Thread* osztály érvénytelenített *stop* metódusának hátrányait kikerülve, nézzük, hogyan tudjuk felfüggeszteni, majd később folytatni a végrehajtást a *Thread* osztály szintén érvénytelenített *suspend* és *resume* metódusainak mellőzésével.

A *wait-notify* mechanizmust alkalmazhatjuk. Bevezetünk még egy kontrollváltozót, amely a *threadSuspended* nevet kapja, és *boolean* típusú. A szál indításakor értékét *false*-ra állítjuk. A *myThread* referenciához hasonlóan ennek a változónak az esetében is *volatile* típusmódosítót alkalmazunk. A *suspend* metóduson belül egyszerűen *true*-ra módosítjuk az értéket. A *while* ciklusunkat kiegészítjük, olyan módon, hogy minden iteráció előtt ellenőrizze a változó értékét, és amennyiben az *true*, egy szinkronizált programblokkon belül mindaddig várakozzon, amíg az érték nem módosul. A várakoztatásra a *wait* metódust használjuk. A *resume* metódus egy szinkronizált programblokkon belül *false*-ra állítja a változó értékét, majd a *notifyAll* metódus segítségével értesíti a várakozó szálakat. Osztályunk módosított kódja:

```
public class ThreadExample implements Runnable {
      private volatile Thread control;
      private volatile boolean threadSuspended;
      public void start() {
            control = new Thread(this);
            threadSuspended = false;
            control.start();
      }
      public void stop() {
            Thread tmp = control;
            control = null;
            tmp.interrupt();
      }
      public void suspend() {
            threadSuspended = true;
      }
      public void resume() {
            synchronized (this) {
```

```
                    threadSuspended = false;
                    notifyAll();
                }
            }
            public void run() {
                Thread thisThread = Thread.currentThread();
                while (thisThread == control) {
                    try {
                        if (threadSuspended) {
                            synchronized (this) {
                                while (threadSuspended)
                                    wait();
                            }
                        }
                        Thread.sleep(1000);
                    } catch (InterruptedException ex) {
                        break;
                    }
                    System.out.println("running");
                }
                System.out.println("finished");
            }
        }
```

Amennyiben egy felfüggesztett szál esetében hívjuk meg a *stop* metódust, az *interrupt* metódus hatására a szál visszatér a várakozási állapotból, és *InterruptedException*-t kapunk, így szintén a megfelelő *catch* ágon belüli *break* utasítás lesz végrehajtva, amely a szál leállítását eredményezi. Megjegyzendő, hogy az ellenőrzést az iteráció elejére tettük, így amennyiben az iteráció közben függesztjük fel a szál futását, az iteráción belüli utasítások még végre lesznek hajtva, csak a következő iterációnál áll le a futás. Ez a megoldás biztonságosabb lehet, mivel így a szál elvégzi az iteráción belüli műveleteket, és nem hagyja félig feldolgozatlanul az adatokat. Ha mégis más viselkedési módot szeretnénk, megtehetjük, hogy a szinkronizált programblokkot a *run* metódusból a *suspend* metódusba helyezzük át. Ebben az esetben, a *suspend* metódus törzsében megfelelő módon kell kezelnünk az esetlegesen fellépő *InterruptedException* típusú kivételt.

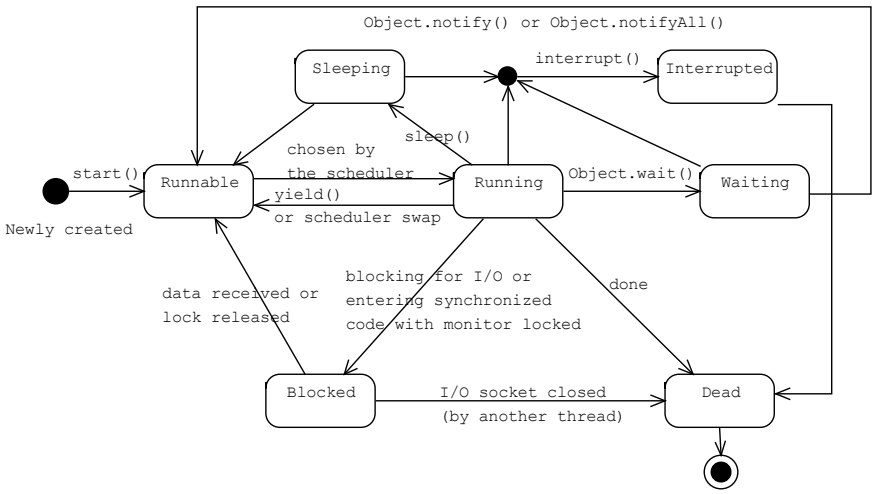
6.6 Szálak állapotai

Amint az előzőekben láthattuk, életciklusa alatt egy végrehajtási szál különböző állapotokba kerülhet. Foglaljuk össze ezeket az állapotokat, és a lehetséges állapotátmeneteket. A 6.1 ábrán látható állapot-átmeneti diagram ezeket szemlélteti.

Miután létrehoztuk az új szál objektumot, a *start* metódussal futásra kész (*runnable*) állapotba hozzuk. Az ütemező ezután elindíthatja a szálat, belépve annak *run* metódusába. A szál így futási állapotba (*running*) kerül. Mivel a processzoridő megoszlik a szálak között, az ütemező adott pillanatban megszakíthatja a szál futását, futó állapotból visszahelyezve a szálat futásra

kész állapotba. Amikor újra „sorra kerül” a szál folytathatja a végrehajtást. Futás közben a szál is dönthet úgy, hogy adott pillanatban átadja a processzort („lemond a kapott processzoridőről”), és ekkor a *yield* metódus segítségével visszahelyezheti magát futásra kész állapotba. Ha a szál futása befejeződött (nincs több feladata, mivel a *run* metódus végére értünk), a „halott” (*dead*) végállapotba jut.

Futási állapotból a szál „alvó” állapotba (*sleeping*) kerülhet a *sleep* metódus meghívásának hatására, és ilyenkor adott ideig felfüggeszti futását. Megtörténhet, hogy a szál egy kritikus erőforráshoz akar hozzáférni, szinkronizált programblokkba lépne be, de a monitor már foglalt, az erőforrást éppen egy másik szál használja. Ebben az esetben a szál blokkolt (*blocked*) állapotba kerül, mindaddig, amíg a monitor fel nem szabadul. Ekkor visszakerül futtatható állapotba, folytathatja a feladat végrehajtását, ezúttal ő foglalva le a kritikus erőforráshoz rendelt monitort. Ugyanez történik, ha a szál I/O műveleteket végez, és bementi adatokat vár. Mindaddig blokkolt állapotban marad, amíg a bemeneti adatok nem állnak rendelkezésre, és csak ezután folytathatja futását. Az is megtörténik, hogy az I/O kapcsolatot egy másik szál lezárja, és ebben az esetben a blokkolt szál futása lezárul, halott állapotba kerül.



6.1 ábra: Végrehajtási szál életciklusa,
a lehetséges állapotokat és állapotátmeneteket ábrázoló diagram

Lehetséges, hogy egy szál, bár hozzáfért egy adott kritikus erőforráshoz (belépve egy szinkronizált programblokkba), és lefoglalhatta az annak megfelelő monitort, de még várakoznia kell. Ahhoz, hogy feladatait elvégezze, először még más szálak által végrehajtott műveletek elvégzésére van szükség. Ilyenkor a szál az *Object* osztály *wait* metódusának segítségével felszabadíthatja a lefoglalt monitort, várakozási (*waiting*) állapotba helyezve magát, így átengedve az erőforrást más szálaknak. Amikor egy másik szál befejezi a kritikus erőforrással kapcsolatos feladatait, az *Object* osztály *notify*, vagy *notifyAll* metódusainak segítségével értesítheti erről a várakozó szálakat. Ezek ebben az esetben visszakerülnek futtatható állapotba, folytathatják a végrehajtást.

Egy szál futási állapotból megszakított (*interrupted*) állapotba kerülhet az *interrupt* metódus meghívása által. Hasonló a helyzet az alvó és várakozó szálak esetében is. Ezek az *interrupt* metódushívás hatására azonnal visszatérnek, és *InterruptedException* típusú kivételt kapunk.

Azt, hogy egy futó szál esetében mi történjen a megszakításkor, a programozó eldöntheti, de általában a szál futása az ilyen esetekben véget ér, és így a szál halott állapotba kerül.

Az alvó (*sleeping*), várakozó (*waiting*) és blokkolt (*blocked*) állapotokat együttes néven nem futtatható (*non-runnable*) állapotoknak nevezhetjük. Ilyenszerű állapotba kerülhet a szál a *join* metódus meghívásának hatására is. Az egyszerűség kedvéért ezt a helyzetet nem tüntettük fel külön állapotként a diagramon.

A fejezet a szálakkal kapcsolatos alapvető fogalmakat tárgyalta. Bonyolultabb többszálú alkalmazások esetében további lehetőségekre és eszközökre is szükségünk lehet. A *java.util.concurrent* csomag további szálkezeléssel kapcsolatos lehetőségeket biztosít, és ezen kívül számos kiegészítő csomag, API is létezik (pl. *Java RTS*). Ezek némileg „haladóbb" témáknak számítanak, és a jegyzetnek nem célja ezek részletezése. A fejezeten belül tárgyalt módszerek alkalmazása az esetek többségében megfelelő megoldást szolgáltathat. Most, hogy alkalmazásainkon belül már bátran használhatunk végrehajtási szálakat, lépjünk tovább, és ismerkedjünk meg az adatfolyamatokkal, a bemeneti és kimeneti műveletek megvalósítási lehetőségeivel, valamint az állománykezeléssel.

6.7 Javasolt gyakorlatok

1. Készítsünk egy egyszerű „autóverseny szimulátort". Egy SWING eszköztár segítségével megvalósított grafikus felületen belül rajzoljunk ki egy versenypályát, és azon helyezzünk el néhány autót (az autókat egyszerű téglalapok segítségével is ábrázolhatjuk, de kis, állományból beolvasott képeket is felhasználhatunk). Arra is lehetőséget adhatunk, hogy a felhasználó meghatározhassa az autók számát (pl. egy szövegmező segítségével). A verseny egy gomb lenyomására indul. Minden autót egy külön végrehajtási szál „irányít", véletlenszerű időközönként változtatva azok pozícióját (a változtatás mértéke is lehet véletlenszerű). A verseny addig tart, ameddig az első autó beér a célvonalba. Készítsünk egy olyan változatot is, amelynek esetében minden autó beér a célba, majd egy párbeszédablak jelenik meg a beérkezés sorrendjével. Egészítsük ki a programot olyan módon, hogy a felületen elhelyezett gombok segítségével a verseny bármelyik pillanatban felfüggeszthető, majd folytatható, vagy leállítható, majd újraindítható legyen (figyelem: ne használjunk érvénytelenített metódusokat).
2. Tetszőlegesen válasszunk ki egy (vagy több) feladatot, a fejezet szinkronizálással kapcsolatos részénél megemlített klasszikus problémák alapján (gyártó-fogyasztó, vacsorázó filozófusok, stb.), és írjunk programot ennek „szimulálására".

   Például, a gyártó-fogyasztó probléma esetében tekinthetjük a következő feladatot: egy irodában több titkárnő dolgozik különböző dokumentumok előállításán, külön számítógépeken. Rendelkezésükre áll néhány nyomtató, amelyekhez mindannyian hozzáférnek. A dokumentumok előállítási ideje különbözik, és a nyomtatókat véletlenszerűen választják ki. Szimuláljuk az iroda „működését".

7

ADATFOLYAMOK

Az adatfolyam (*data stream*) adatok összefüggő szekvenciája, amelyet számítástechnikai eszközök (hardware és software) közötti információ átvitelre használunk. A cél a különböző számítástechnikai rendszereken belüli entitások közötti kommunikáció egységesítése. Ha adatfolyam alapú kommunikációról beszélünk, érdemes megemlíteni Dennis Ritchie nevét, aki 1984-ben elsőként fejlesztett ki *stream* alapú I/O (*input/output*) rendszert egy UNIX operációs rendszerhez, adatfolyamokon keresztül biztosítva a rendszerfolyamatok közötti kommunikációt.

A *stream* tulajdonképpen két entitás közötti kommunikációs csatorna, amely egyirányú információátvitelt tesz lehetővé. Minden esetben van egy forrása, és van egy célja, és ezek sokfélék lehetnek (konzol, rendszerfolyamat, állomány, stb.). Ennek megfelelően beszélhetünk bemeneti (pl. billentyűzethez, vagy más bemeneti eszközhöz rendelt adatfolyam) és kimeneti (pl. monitorhoz, vagy más kimeneti eszközhöz rendelt adatfolyam) adatfolyamokról (*input* és *output stream*-ek). Ezek összekapcsolhatóak, és így valósítható meg a folyamatok, vagy szálak közötti kommunikációnál alkalmazott *pipeline* mechanizmus.

Az adatfolyamokon belüli adatok is többfélék lehetnek, például számértékek, szövegek, vagy tetszőleges objektumok. Két kategóriát különítünk el: szöveges és bináris adatfolyamokat. A Java programozási nyelv esetében az adatfolyamok létrehozására, és a kapcsolódó műveletek elvégzésére a *java.io* csomag interfészeit és osztályait alkalmazhatjuk [15].

7.1 A *java.io* csomag

A bináris adatfolyamokkal kapcsolatos alapfunkcionalitásokat, az alapvető írási és olvasási műveletek elvégzésére alkalmazható metódusokat az *InputStream* és *OutputStream* alaposztályok biztosítják. Hasonlóan, a szöveges adatfolyamok esetében a *Reader* és *Writer* alaposztályok állnak rendelkezésünkre. A speciálisabb adatfolyamoknak megfelelő, speciálisabb műveletek elvégzését támogató osztályok ezekből lesznek származtatva.

A bináris adatok szöveges adatokba konvertálására alkalmazhatjuk az *InputStreamReader* osztályt. Hasonlóan alkalmazható az *OutputStreamWriter*, amely a karaktereket bájtszekvenciába konvertálja.

Általában nem egyszerűen bájtokat vagy karaktereket akarunk írni vagy olvasni, hanem különböző típusú változók értékeit. Erre a primitív adattípusok esetében a *DataInputStream* és *DataOutputStream* osztályokat alkalmazhatjuk. Tetszőleges objektumokat is írhatunk, vagy olvashatunk az *ObjectInputStream*, és *ObjectOutputStream* osztályok segítségével. Ehhez ismernünk kell a Java szerializációs mechanizmusát, amelyet ebben a fejezetben szintén tárgyalunk.

A hatékonyabb hozzáférést az adatokhoz a pufferelés biztosíthatja. Erre a bináris adatfolyamok esetében a *BufferedInputStream* és *BufferedOutputStream* osztályok, a szöveges adatfolyamok esetében a *BufferedReader* és *BufferedWriter* osztályok adnak lehetőséget. Ezek az osztályok puffereket alkalmaznak, hogy csökkentsék az adatforrást érintő műveletek számát, és lehetőséget adjanak hatékonyabb, kényelmesebb feldolgozásra (például soronkénti olvasásra).

Az állományokkal kapcsolatos írási és olvasási műveleteket bináris állományok esetében a *FileInputStream* és *FileOutputStream* osztályok, szöveges állományok esetében a *FileReader* és *FileWriter* osztályok teszik lehetővé. Az állománykezeléssel kapcsolatos fogalmakat a fejezet későbbi részében részletesebben is ismertetjük.

Több esetben szükség lehet a bemeneti és kimeneti adatfolyamok összekapcsolására (például végrehajtási szálak közötti kommunikáció megvalósításakor), amelyet *pipeline* mechanizmusnak nevezünk. Ennek megvalósítására a bináris adatfolyamok esetén a *PipedInputStream* és *PipedOutputStream* osztályok, szöveges adatfolyamok esetén a *PipedReader* és *PipedWriter* osztályok alkalmasak.

Szöveges kimeneti adatfolyamok esetében alkalmazható egy speciális osztály, a *PrintWriter*. A példáinkból már ismert, konzollal kapcsolatos kimeneti műveleteket biztosító *System.out* objektum ilyen típusú. A *BufferedWriter* osztállyal összehasonlítva, egyéb különbségek mellett, fontos kiemelni, hogy a *PrintWriter* által biztosított írási (*print*) metódusok meghívásához nem szükséges *try-catch* szerkezet alkalmazása.

7.2 Írás és olvasás

Egy egyszerű bináris adatfolyamból az olvasás az *InputStream* osztály *read* metódusainak segítségével történhet:

```
public int read() throws IOException
public int read(byte[] b) throws IOException
public int read(byte[] b, int off, int len) throws IOException
```

Az első metódus egy bájtot olvas be, egy *int* típusú változóba térítve vissza annak értékét (0-255). Az adatfolyam végére érve -1 értéket térít vissza.

A második metódus beolvassa az adatfolyamon belüli adatokat (bájtokat), és a paraméterként kapott *byte* típusú tömbbe másolja azokat. A visszatérített érték a beolvasott bájtok száma, amelynek maximuma a *b* tömb mérete. A metódus harmadik változata ehhez hasonló, de paraméterként még megkapja a beolvasandó bájt-szekvencia hosszát és egy *offset* értéket, amely azt határozza meg, hogy a beolvasott adatok tömbbe történő másolása hányadik indextől kezdődjön. A visszatérített érték szintén a beolvasott bájtok száma. Az adatok tárolása a tömbön belül a *b[off]* elemtől kezdődik. A metódus kivételt (*ArrayIndexOutOfBoundsException*) vált ki olyan esetekben, amikor a két egész típusú paraméter közül valamelyik negatív, illetve mikor azok összege nagyobb a tömb méreténél. Megjegyezhetjük, hogy a *read(b, 0, b.length)* tulajdonképpen a *read* metódus másodikként bemutatott változatának meghívásával egyenértékű, és mindkét metódus esetében gyakorlatilag az első változat többszöri meghívásáról van szó.

Láthatjuk, hogy a metódusok fejlécében megjelenik a *throws* kulcsszó, illetve az *IOException* kivételtípus. A részletesebb magyarázatra még visszatérünk a kivételkezeléssel kapcsolatos

kiegészítő résznél. Egyelőre elegendő, ha tudjuk, hogy következményként a metódusokat kötelező módon *try-catch* konstrukción belül kell meghívni, mivel meghívásuk *IOException* típusú kivételt eredményezhet. Példát a fejezet következő részében mutatunk be, ahol a standard adatfolyamokról lesz szó.

Az *InputStream* osztály esetében fontos még megemlítenünk a következő metódusokat:

- **`int available()`** – az adatfolyamból kiolvasható bájtok számát téríti vissza (példa a következő részben);
- **`void close()`** – az adatfolyamok esetében az írási vagy olvasási műveletek elvégzése után nagyon fontos az adatfolyam bezárása. Ez tipikusan a *try-catch* szerkezet *finally* ágán belül történik (a magyarázatról már szó esett a kivételkezeléssel foglalkozó résznél);
- **`void mark(int readlimit)`** – megjelöli az adatfolyamon belül az aktuális pozíciót, így a kurzor a későbbiekben visszaállítható erre a pozícióra. A mechanizmus alapja, hogy az adatfolyam rögzíti a megjelölt pozíció után kiolvasott bájtokat. Ez csak a mechanizmust támogató adatfolyamok esetében történik meg, így a *mark* metódus alkalmazása csak ezeknek esetében lehetséges. A paraméterként kapott egész érték azt határozza meg, hogy a jelölés hány bájt beolvasása után tekintődjék érvénytelennek;
- **`void reset()`** – a kurzort a megjelölt pozícióra állítja vissza. Amennyiben nem volt megjelölt pozíció, vagy a jelölés már nem érvényes, akkor a metódus meghívása általában kivételt eredményez (*IOException*), de az adatfolyam konkrét típusának megfelelően megtörténhet az is, hogy a metódus az utolsó megjelölt pozícióra, vagy az adatfolyam elejére pozicionálja a kurzort. Hasonlóan, ha az adatfolyam nem támogatja a jelölési mechanizmust, a *reset* meghívása általában kivételt eredményez, de ez szintén változhat a konkrét típusnak megfelelően;
- **`boolean markSupported()`** – segítségével lekérdezhető, hogy az illető adatfolyam támogatja-e a jelölési mechanizmust (meghívhatóak-e a *mark* és *reset* metódusok). Az *InputStream* alaposztály esetében a metódus hamis (*false*) értéket térít vissza, így látható, hogy a jelöléssel kapcsolatos metódusok inkább a származtatott osztályok kedvéért kaptak helyet az *InputStream* osztályban;
- **`long skip(long n)`** – *n* bájt kihagyása (átugrása). A standard implementáció ismételten meghívja *n*-szer a *read* metódust, egy tömbbe helyezve az adatokat, de a származtatott osztályok hatékonyabb implementációkat biztosíthatnak. Megtörténhet, hogy nem lehetséges *n* bájt kihagyása (például, mert nincsen *n* kiolvasható bájt az adatfolyamon belül), ezért a metódus visszatéríti a ténylegesen kihagyott bájtok számát.

Az írásra az *OutputStream* metódusait alkalmazhatjuk:

```
public void write(int b) throws IOException
public void write(byte[] b) throws IOException
public void write(byte[] b, int off, int len) throws IOException
```

A metódusok az *InputStream read* metódusainak írásra használható megfelelői. Nincsenek visszatérített értékek, és kivételt generálhatnak. Ahhoz, hogy a *read* metódus -1 értéket téríthessen vissza az adatfolyam végére érve, a visszatérített érték típusának *int*-nek kellett lennie (*byte* helyett). Következményként a *write* metódus is *int* típusú értéket kap paraméterként,

és a 8 alacsonyabb helyiértékű bitnek megfelelő *byte* értéket írja ki az adatfolyamba (a 24 magasabb helyiértékű bitet nem veszi figyelembe). A második metódus meghívása tulajdonképpen egy *write(b, 0, b.length)* metódushívásnak felelne meg. A három paraméterrel rendelkező metódus esetében a kiírás az *off* indexű elemtől kezdődik és az *off+len-1* indexű elemig tart. A *read* megfelelő változatához hasonlóan a tömbindexek határának átlépése itt is kivételt eredményez.

A kimeneti adatfolyamok bezárása is fontos lehet az írási műveletek elvégzése után, így az *OutputStream* osztály is biztosít számunkra *close* metódust. A kimeneti adatfolyamok puffereket alkalmazhatnak, ezért az *OutputStream* osztály egy *flush* nevű metódust is biztosít a puffer kiürítésére.

Természetesen a származtatott osztályok hatékonyabb írási és olvasási metódusokat is biztosítanak. Például a *DataInputStream* és *DataOutputStream* osztályok esetében egy-egy olvasási, vagy írási művelettel különböző típusú adatok értékeit írhatjuk, vagy olvashatjuk. A következő részben erre is adunk példát.

7.3 Standard bemenet és kimenet

A standard bemeneti és kimeneti műveletek megvalósítására (konzol I/O) a *System* osztály publikus adattagjait használhatjuk:

- *InputStream in*: a standard bemenetnek (alapértelmezetten billentyűzet) megfelelő adatfolyam
- *PrintWriter out*: a standard kimenetnek (alapértelmezetten monitor) megfelelő adatfolyam
- *PrintWriter err*: a standard hibakimenetnek (alapértelmezetten monitor) megfelelő adatfolyam

Ezeket az adatfolyamokat a rendszer automatikusan megnyitja a program indításakor, és a kilépés után bezárásuk is automatikusan megtörténik.

Említettük, hogy az olvasásnak *try-catch* szerkezeten belül kell megtörténnie:

```
try {
      int val = System.in.read();
      …
} catch (IOException ex) {…}
```

Vagy:

```
byte b[1024];
try {
      int noBytes = System.in.read(b);
} catch (IOException ex) {…}
```

Szó volt arról is, hogy egy adatfolyam esetében lekérdezhetjük a kiolvasható bájtok számát az *available* metódus segítségével:

```
try {
```

```
        int available = System.in.available();
        if (available > 0) {
              byte b[] = new byte[available];
              System.in.read( b );
        }
  } catch (IOException e) {...}
```

Az alapvető írási és olvasási műveletek megértéséhez tekintsük a következő példaprogramot, amely a standard bemenetet a standard kimenetre másolja:

```
  import java.io.IOException;
  import java.io.InputStream;
  import java.io.OutputStream;
  public class StreamExample {
        public static void copy(InputStream sin, OutputStream sout)
                                                    throws IOException {
              int b;
              while ((b = sin.read()) != -1) sout.write(b);
        }
        public static void main(String[] args) {
              try {
                    copy(System.in, System.out);
              } catch (IOException e) {
                    System.out.println("I/O error");
              }
        }
  }
```

A *StreamExample* osztályon belül a másolást a *copy* metódus belsejében végezzük el. A bemeneti adatfolyamból bájtonként olvassuk be az adatokat, mindaddig, amíg rendelkezésünkre állnak adatok. A standard bemenetnek megfelelő adatfolyam nem lesz bezárva a program futása során, nem érünk a végére, így folyamatosan olvashatunk. A beolvasás után azonnal kiírjuk az adatot a kimeneti adatfolyamba. Ez egy puffert alkalmazó adatfolyam esetében (amilyen a standard kimenetnek megfelelő *PrintWriter* adatfolyam is) nem vezet azonnal kiíráshoz a célban (az adatok nem jelennek meg a konzolon). Ez csak a puffer kiürítése után történik meg, amely a *flush* metódus meghívásával valósítható meg. Léteznek olyan adatfolyamok, amelyek bizonyos események hatására automatikusan ürítik a puffert. Ilyen a *PrintWriter* is, amely újsor karakter esetében kiüríti a puffert. Ez esetünkben azt jelenti, hogy amikor lenyomjuk az *enter* billentyűt az addig beírt karakterek megjelennek a konzolon. Megfigyelhetjük, hogy, bár az olvasás kivételt eredményezhet, nem kezeljük helyben (az olvasás pillanatában) ezt a kivételt, hanem a *copy* metódusunk továbbítja azt az őt meghívó metódushoz (esetünkben a *main*), és ezen a metóduson belül kap helyet a kivételkezelés (erre a későbbiekben részletesebb magyarázattal is szolgálunk). Fontos megjegyezni még, hogy az adatfolyamok esetében (ahogyan ezt a későbbiekben látni is fogjuk) nagyon fontos az írási és olvasási műveletek elvégzése után az adatfolyam bezárása (*close* metódushívás, tipikusan a *try-catch* konstrukció *finally* ágán belül). Ez példánk esetében azért nem volt szükséges, mivel a standard bemeneti és kimeneti adatfolyamok a program futása során végig nyitva maradnak.

7.4 Állománykezelés

Az adatfolyamok forrása, vagy célja állomány is lehet. A bináris állományokkal kapcsolatos alapvető olvasási és írási műveletekre a *FileInputStream* és *FileOutputStream* osztályok, szöveges állományok esetében a *FileReader* és *FileWriter* osztályok alkalmazhatóak. Használatukat példákon keresztül szemléltetjük.

Az alábbi példa egy szöveges állomány tartalmát jeleníti meg a konzolon.

```
import java.io.FileReader;
import java.io.FileNotFoundException;
import java.io.IOException;
public class Cat {
      public static void main(String[] args) {
            FileReader fr = null;
            int b;
            if (args.length != 1) {
                  System.out.println(
                        "Inditas: java Cat <fajlnev_text>");
                  System.exit(1);
            }
            try {
                  fr = new FileReader(args[0]);
                  while((b = fr.read()) != -1 )
                        System.out.print((char) b);
            } catch (FileNotFoundException e1) {
                  System.out.println("File not found");
                  System.exit(2);
            } catch (IOException e2) {
                  System.out.println("I/O error");
                  System.exit(3);
            } finally {
                  try {
                        if (fr != null) fr.close();
                  } catch (IOException e3) {
                        System.out.println("I/O error");
                  }
            }
      }
}
```

A *Cat* nevű osztályunk egyetlen metódusa a *main*. A program a parancssor argumentumaként kapja meg az állomány nevét. Először ellenőrzi az argumentumok számát, és, amennyiben ez különbözik egytől, hibaüzenetet ír ki a konzolra, majd 1-es kilépési kóddal zárja a futását. Amennyiben egy paramétert kapott, annak segítségével egy *try-catch* szerkezeten belül létrehoz egy *FileReader* objektumot. Bájtonként olvas a szöveges állományból, mindaddig, amíg nem ér az állomány végére (ebben az esetben a *read* metódus -1 értékkel tér vissza).

A beolvasás után azonnal karakterbe alakítja a beolvasott bájtokat, és kiírja a karaktereket a konzolra. Amennyiben nem található az állomány *FileNotFoundException* típusú kivételt kapunk, amit az első *catch* ágon belül kezelünk le, hibaüzenetet írva ki a konzolra, majd 2-es kilépési kóddal zárva az alkalmazást. Hasonlóan az olvasási hiba esetén kapott *IOException* típusú kivételt, a második *catch* ágon belül kezeljük, a hibaüzenet megjelenítése után 3-as kilépési kóddal zárva a programot. Mivel a *FileNotFoundException* az *IOException* leszármazottja, fontos a *catch* ágak sorrendje (a magyarázatról szó esett a kivételkezeléssel kapcsolatos résznél). A műveletek elvégzése után zárjuk az adatfolyamot, és ezt a *finally* ágon belül tesszük meg. A *close* metódus meghívása a maga során szintén *IOException* típusú kivételt eredményezhet, így egy külön *try-catch* szerkezeten belül hívjuk meg.

Ugyanezzel a példával szemléltethetjük a puffereket alkalmazó adatfolyamok előnyeit. A *BufferedReader* osztály importálása után, a *main* metódus elején a *FileReader* típusú referenciához hasonlóan deklaráljunk egy *br* nevű *BufferedReader* típusú referenciát, majd a *try-catch* szerkezet kódját módosítsuk az alábbi módon:

```
try {
      fr = new FileReader(args[0]);
      br = new BufferedReader(fr);
      String line;
      while ((line = br.readLine()) != null)
            System.out.println(line);
} catch (FileNotFoundException e) {…
} catch (IOException e) {…
} finally {
      try {
            if (br != null) br.close();
      } catch (IOException e3) {
            System.out.println("I/O error");
      }
}
```

Az előbbi példához hasonlóan itt is létrehozzuk a *FileReader* objektumot, de most ezután létrehozunk egy *BufferedReader* objektumot is. A konstruktor paraméterként kapja a *FileReader* objektumunkra mutató referenciát. A puffert alkalmazó új adatfolyam a *FileReader* objektumunkat fogja használni. A *BufferedReader* biztosít számunkra egy *readLine* nevű metódust, amelynek segítségével soronként olvashatjuk az adatokat. Az adatfolyam végére érve a metódus egyszerűen egy *null* értékű *String* objektumot térít vissza. A *while* cikluson belül eddig a pillanatig olvasunk, és a beolvasott sorokat kiírjuk a konzolra. A kivételek kezelése az előbbi változattal azonos módon történik. Az egyetlen különbség, hogy most a *finally* ágon belül a *BufferedReader* objektumunkra hívjuk meg a *close* metódust, így kiürítve a puffert, és zárva az adatfolyamot.

Említettük, hogy az *InputStream* és *OutputStream* osztályok leszármazottjai hatékonyabb írási és olvasási műveleteket is biztosítanak. Tekintsük például a *DataInputStream* és *DataOutputStream* osztályokat, amelyek az alaposztályokból származtatott *FilterInputStream* és *FilterOutputStream* leszármazottjai (mivel tulajdonképpen szűrési műveletekre adnak lehetőséget), és megvalósítják a *DataInput*, illetve *DataOutput* interfészeket. Segítségükkel különböző primitív típusoknak megfelelő értékeket, illetve karakterláncokat olvashatunk be adatfolyamokból.

Használatukat az alábbi példával szemléltetjük:

```
import java.io.*;
public class DataFileExample {
      private static DataInputStream din;
      private static DataOutputStream dout;
      private static void readData(String fileName)
                  throws FileNotFoundException, IOException {
            FileInputStream fin = new FileInputStream(fileName);
            din = new DataInputStream(fin);
            System.out.println(din.readInt());
            System.out.println(din.readFloat());
      }
      private static void writeData(String fileName)
                  throws FileNotFoundException, IOException {
            FileOutputStream fout = new FileOutputStream(fileName);
            dout = new DataOutputStream(fout);
            dout.writeInt(42);
            dout.writeFloat(42.42f);
      }
      public static void main(String[] args) {
            try {
                  writeData("data.dat");
                  readData("data.dat");
            } catch (FileNotFoundException ex1) {
                  System.out.println("File not found");
            } catch (IOException ex2) {
                  System.out.println("I/O error");
            } finally {
                  try {
                        if (din != null) din.close();
                        if (dout != null) dout.close();
                  } catch (IOException ex) {
                        System.out.println("I/O error");
                  }
            }
      }
}
```

A *DataFileExample* osztályunk tartalmaz két osztályszintű metódust az írásra, illetve olvasásra. Az írást elvégző *writeData* metódus a paraméterként kapott állománynév segítségével létrehoz egy *FileOutputStream* objektumot, majd az erre mutató referenciát átadva a *DataOutputStream* konstruktorának létrehoz egy speciális írási műveletekre alkalmas példányt. A *DataOutputStream* megfelelő metódusainak segítségével az adatfolyamba ír egy egész majd egy valós értéket. A *readData* metódus létrehoz egy *FileInputStream* objektumot, majd ezt felhasználva egy *DataInputStream* objektumot, és a megfelelő metódusok segítségével kiolvas az adatfolyamból egy egész, majd egy valós értéket, és kiírja ezeket a konzolra. A két említett metódust hívja meg sorrendben a *main*, kiírva az értékeket, majd visszaolvasva azokat. A metóduson belül kezeljük a megfelelő kivételeket, és végül zárjuk az adatfolyamokat.

A *DataInputStream* és *DataOutputStream* osztályok a példánk esetében alkalmazott metódusokhoz hasonló *read* és *write* metódusokat biztosítanak más primitív típusok, illetve karakterláncok olvasására és írására. A metódusok neve a *read*, illetve *write* előtagokkal kezdődik, és a típusnak megfelelő utótaggal végződik (*readXXX* és *writeXXX* metódusok, például *readDouble, writeDouble, readBoolean, writeBoolean* stb.)

Az írás és olvasás „szabályait" a megvalósított interfészek specifikációja írja le, és a metódusokat az osztályok ennek megfelelően valósítják meg. Például, egy egész érték esetében 4 bájt lesz kiírva, vagy beolvasva, egy *boolean* érték írásánál 1, vagy 0 érték kerül kiírásra, de a beolvasást elvégző metódus már *boolean* típust térít vissza, stb. A protokoll ismerete elengedhetetlen a kód hibamentességének szempontjából. Megjegyzendő, hogy ezeket az osztályokat a gyakorlatban „együtt" érdemes használni, tehát a *DataInputStream* metódusainak segítségével általában olyan adatokat olvasunk, amelyek írása *DataOutputStream* metódusokon keresztül történt.

Visszatérve az állománykezeléssel kapcsolatos lehetőségekhez, fontos megemlítenünk a központi szereppel bíró *File* osztályt.

7.4.1 A File osztály

Az előző példáink esetében az állományokat egyszerűen nevük segítségével azonosítottuk, így rendelve hozzájuk a megfelelő adatfolyam objektumokat. A Java biztosít számunkra egy *File* osztályt is, melynek példányai állományok és könyvtárak beazonosítására szolgálnak. Az osztály számos állománykezeléssel kapcsolatos hasznos metódust biztosít, ezek közül sorolunk fel néhányat:

- **`boolean canRead(), boolean canWrite()`**: ellenőrzik, hogy az adott állomány esetében engedélyezett-e az olvasás, illetve az írás;
- **`boolean delete()`**: az állomány vagy (üres) könyvtár törlése, *true* értéket térít vissza, ha a művelet végrehajtása sikeres volt, *false* értéket egyébként;
- **`boolean exists()`**: ellenőrzi, hogy létezik-e az adott állomány (vagy könyvtár);
- **`String getPath()`**: az elérési út (relatív) szöveges formában történő visszatérítése;
- **`String getAbsolutePath(), String getCanonicalPath()`**: a teljes útvonal visszatérítése. A pontos meghatározás platformfüggő;
- **`String getName(), String getParent()`**: az állomány (vagy könyvtár) nevének, illetve az állományt (könyvtárat) tartalmazó könyvtár nevének lekérdezése;
- **`boolean isFile(), boolean isDirectory()`**: segítségükkel eldönthető, hogy a *File* objektum létrehozásánál használt név (és útvonal) állománynak vagy könyvtárnak felel meg;
- **`long length()`**: az állomány hosszának lekérdezése (bájtokban megadva);
- **`String[] list(), String[] list(FilenameFilter filter)`**: a könyvtárban található állományok neveinek listája. A második változat esetében egy állománynév-szűrőt adhatunk át paraméterként;
- **`File[] listFiles(), File[] listFiles(FileFilter filter), File[] listFiles(FilenameFilter filter)`**: a könyvtárban található állományok listájának lekérdezése. A második és harmadik változat esetében a paraméterek segítségével szűrőket határozhatunk meg;

- **`boolean mkdir(), boolean mkdirs()`**: könyvtár létrehozása. A második metódus az útvonal által meghatározott szükséges főkönyvtárakat is létrehozza;
- **`URL toURL()`**: átalakítás URL típusú objektumba.

A felsoroltak közül néhány metódus használatát szemléltetjük az alábbi példával. A program a parancssor argumentumaként megkapja egy állomány vagy könyvtár nevét, ennek segítségével létrehoz egy *File* objektumot, a megfelelő metódusok segítségével ellenőrzi, hogy létezik-e, és olvasható-e az illető állomány, majd eldönti, hogy állományról vagy könyvtárról van-e szó. Könyvtárak esetében kilistázza a könyvtárban található állományok neveit, állományok esetében kilistázza azok tartalmát (szöveges állománynak tekintve azokat).

```
import java.io.File;
import java.io.FileReader;
import java.io.BufferedReader;
import java.io.IOException;
public class FileExample {
      public static void main(String[] args) {
            File file;
            if (args.length != 1) {
                  System.out.println("Specify a file name");
                  return;
            }
            file = new File(args[0]);
            if (!file.exists() || !file.canRead()) {
                  System.out.println("Can't read: "+args[0]);
                  return;
            }
            if (file.isDirectory()) {
                  String files[] = file.list();
                  for(int i = 0; i < files.length; i++)
                        System.out.println(files[i]);
            } else {
                  try {
                        FileReader fr = new FileReader(file);
                        BufferedReader in = new BufferedReader(fr);
                        String line;
                        while ((line = in.readLine()) != null)
                              System.out.println(line);
                  } catch (IOException e) {
                        System.out.println("Read error");
                  }
            }
      }
}
```

Miután ellenőriztük az argumentumok számát, létrehozzuk a *File* objektumot, majd az *exists* és *canRead* metódusok segítségével elvégezzük a megfelelő ellenőrzéseket. Az *isDirectory* metódus segítségével ellenőrizzük, hogy könyvtárról van-e szó, és amennyiben igen, a *list*

metódus segítségével lekérdezzük a könyvtárban található állományok neveit, és kiírjuk azokat a konzolra. Amennyiben állományról van szó, a tartalom listázása az előző példákhoz hasonlóan történik, *FileReader* és *BufferedReader* objektumok segítségével. A különbség annyi, hogy a *FileReader* konstruktora ezúttal egy *File* típusú referenciát kap paraméterként, és nem kezeljük külön a *FileNotFoundException* kivételt, mivel az állomány létezéséről már előzőleg meggyőződtünk.

7.4.2 Közvetlen elérésű állományok

Az eddigi példáinkban az adatfolyamokhoz történő hozzáférés sorosan történt, azaz az adatfolyam elejéről a vége felé haladva sorban kiolvastunk minden bájtot. Könnyen belátható, hogy egy nagyobb méretű állomány esetében, amennyiben annak csak bizonyos részeihez szeretnénk hozzáférni, ez óriási hátrányt jelentene. Az is kényelmetlenséget jelent, hogy az olvasásra, illetve írásra külön objektumokat kell létrehozni.

A Java mindkét problémára szolgáltat megoldást. A *RandomAccessFile* osztály segítségével közvetlen módon férhetünk hozzá állományainkhoz, tetszőlegesen pozicionálhatjuk a kurzort, és mind olvasási, mind írási műveletekre lehetőségünk van. Az osztály megvalósítja a *DataInput* és *DataOutput* interfészeket, így a *DataInputStream* és *DataOutputStream* osztályokhoz hasonlóan lehetőséget ad különböző adattípusok olvasására, illetve írására. Megjegyzendő, hogy bár megvalósítja az említett interfészeket, az osztálynak nincs kapcsolata az eddig megismert adatfolyam típusokkal, nem leszármazottja egyik említett osztálynak sem. Használatát egy egyszerű példával szemléltetjük.

Az alábbi kis program véletlenszerűen generált valós számokat ír egy állományba, majd szintén véletlenszerűen generál pozíciókat, és az adott pozíciókon található értékeket meghatározott valós számértékre cseréli.

```
import java.io.IOException;
import java.io.RandomAccessFile;
import java.util.Random;
public class RAFExample {
      private RandomAccessFile f;
      private String name;
      private long dim;
      private static Random r = new Random();
      public RAFExample(String name, long dim) throws IOException {
            this.name = name;
            this.dim = dim;
            f = new RandomAccessFile(name, "rw");
            for (int i = 0; i < dim; i++) {
                  double d = r.nextDouble();
                  f.writeDouble(d);
                  System.out.print(d+"\t");
            }
            f.close();
      }
      public double getDouble(long poz) throws IOException {
```

```
            try {
                  f = new RandomAccessFile(name, "r");
                  f.seek(poz*8);
                  return(f.readDouble());
            } finally {
                  f.close();
            }
      }
      public void putDouble(long poz, double d) throws IOException {
            try {
                  f = new RandomAccessFile(name, "rw");
                  f.seek(poz*8);
                  f.writeDouble(d);
            } finally {
                  f.close();
            }
      }
      public long getDim() {
            return dim;
      }
      public static void main(String[] args) {
            try {
                  RAFExample c = new RAFExample("doublefile", 10);
                  long n = c.getDim();
                  for (int i = 0; i < 5; i++) {
                        long poz = Math.abs(r.nextLong());
                        poz = poz % n;
                        System.out.println("Poz: "+poz);
                        c.putDouble(poz, 1.0);
                  }
                  System.out.println("File changed: ");
                  for (long i = 0; i < n; i++)
                        System.out.print(c.getDouble(i)+ "\t");
            } catch (IOException e) {
                  System.out.println("Error");
            }
      }
}
```

A konstruktor paraméterként megkapja az állomány nevét, valamint a tárolandó valós értékek számát, és értéket ad a megfelelő attribútumoknak. Ezután létrehoz egy *RandomAccessFile* példányt, a konstruktor paramétereként átadva az állomány nevét és a megnyitás módját. Ez utóbbi esetünkben „*rw*", ami azt jelenti, hogy olvasásra és írásra nyitjuk az állományt (külön írásra történő megnyitásra nincs lehetőség). A konstruktoron belül feltöltjük az állományt véletlenszerűen generált valós számokkal. A generálásra a *Random* osztály példányát használjuk, amelyet osztályszintű attribútumként deklaráltunk és inicializáltuk. Ez azért történt így, mert ugyanezt a generátor példányt a későbbiekben a *main* metóduson belül is használni fogjuk. A *Random* osztály *nextDouble* metódusa az objektumnak megfelelő szekvenciából

a következő egyenletes eloszlású, 0.0 és 1.0 közötti *double* típusú értéket adja. Az állományba történő írás a *RandomAccessFile writeDouble* metódusával történik, amely a valós érték *long* típusú megfelelőjét egy 8 bájton tárolt érték formájában menti le. A konstruktor utolsó sorában zárjuk az állománynak megfelelő adatfolyamot.

A *getDouble* és *putDouble* metódusok egy-egy valós érték olvasására, illetve felülírására alkalmasak. Az olvasás esetében paraméterként megkapjuk a pozíciót, és visszatérítjük a pozíción található valós értéket. Az írás esetében szintén a pozíciót kapjuk paraméterként, valamint a beírandó valós értéket. Mindkét metódus elején megnyitjuk az állománynak megfelelő adatfolyamot, az egyik esetben csak olvasásra, a másik esetben írásra és olvasásra. A kurzort a *seek* metódus segítségével a megfelelő helyre pozicionáljuk. Az értékek 8 bájton vannak tárolva, így a paraméterként kapott pozíciót meg kell szoroznunk 8-al. A műveleteket *try-catch* szerkezeten belül végezzük, és a *finally* ágon belül zárjuk az adatfolyamot.

A *getDim* metódus az állományban tárolt valós értékek számát adja. A *main* metóduson belül az osztályunkból történő példányosítás után ezt a metódust használjuk, és a visszatérített értéket az *n* változóban tároljuk. Ezután egy *for* cikluson belül véletlenszerűen kiválasztunk öt pozíciót. A generátorunk *nextLong* metódusát használjuk, amely az objektumnak megfelelő szekvenciából mindig a következő egyenletes eloszlású *long* típusú értéket adja. Ez negatív is lehet, így a *Math* osztály *abs* statikus metódusát hívjuk segítségül, és a szám abszolút értékének *n*-el való osztási maradékával dolgozunk tovább, hogy biztosan az állományon „belül" maradjunk. Az ilyen módon meghatározott pozíción tárolt értéket 1.0 –ra módosítjuk, majd egy másik *for* cikluson belül kiírjuk a konzolra a megváltoztatott állomány tartalmát.

Természetesen a konkrét példa esetében nem túl optimális megoldás, hogy minden egyes olvasási, illetve írási műveletnél új objektumot hozunk létre, de egy általánosabb esetben fontos az állománynak megfelelő adatfolyam lezárása. Esetünkben a problémát „áthidalhatnánk", ha külön metódusokat vezetnénk be a megnyitásra, illetve bezárásra.

7.5 Szerializáció

Az objektumorientált rendszerek esetében gyakori elvárás a perzisztencia megvalósítása, az objektumok állapotának elmentése (archiválása), a későbbi felhasználás céljából. Ez többféle módon történhet, például elmenthetjük az állapottal kapcsolatos információkat állományokba, vagy használhatunk különböző adatbázis rendszereket. A perzisztencia megvalósításának egy sajátos formája a *marshalling* mechanizmus, amikor az objektumok állapotával együtt a kód (*codebase*) is elmentésre kerül. Ez fontos lehet például osztott rendszerek esetében, ahhoz, hogy a lementett állapotinformációk alapján egy távoli rendszeren is újra felépíthetőek legyenek az objektumok.

A szerializáció [16] egy standard eljárás objektumok állapotának adatfolyamba történő kimentésére, illetve adatfolyamból történő betöltésére (az objektumok felépítésére az adatfolyamokban bájt-szekvenciaként tárolt állapotinformációk alapján).

A Java biztosít számunkra egy standard protokollt az objektumok szerializálására. Ez a protokoll módosítható, és saját protokoll is alkalmazható.

7.5.1 A standard protokoll

Az objektumok adatfolyamokba történő kiírása az *ObjectOutputStream* osztály segítségével történhet, állományba történő mentés esetében a következő módon:

```
fouts = new FileOutputStream(filename);
out = new ObjectOutputStream(fouts);
out.writeObject(new MyObject());              //serialization
out.close();
```

A beolvasás az *ObjectInputStream* segítségével valósítható meg:

```
fins = new FileInputStream(filename);
in = new ObjectInputStream(fins);
time = (MyObject) in.readObject(); //live object, exact replica
in.close();
```

A fenti kódrészletek *try* programblokkon belül alkalmazhatóak, mivel különböző kivételeket válthatnak ki: olvasási vagy írási hiba esetében *IOException* típusú, nem létező állomány esetében *FileNotFoundException* típusú, nem szerializálható objektumok esetében *NotSerializableException* típusú kivételt kaphatunk.

Ahhoz, hogy az objektumok szerializálhatóak legyenek, az osztálynak meg kell valósítania a *Serializable* interfészt, vagy örökölnie kell annak megvalósítását az alaposztálytól. Az *Object* ősosztály nem implementálja ezt az interfészt, így nem minden objektum szerializálható. Ez természetes is, mivel bizonyos objektumtípusok esetében értelmetlen lenne perzisztens állapotról, szerializálhatóságról beszélnünk (pl. végrehajtási szálak, hálózati kapcsolatok stb.). A legtöbb standard Java osztály viszont szerializálható (gyűjtemények, grafikus komponensek, stb.).

Példa a *Serializable* interfészt megvalósító osztály létrehozására:

```
import java.io.Serializable;
import java.util.Date;
import java.util.Calendar;
public class PersistentTime implements Serializable {
      private Date time;
      public PersistentTime() {
            time = Calendar.getInstance().getTime();
      }
      public Date getTime() {
            return time;
      }
}
```

A *PersistentTime* osztály konstruktora lekéri az aktuális időt, és eltárolja a *time* attribútumban. Mivel az osztály implementálja a *Serializable* interfészt, példányai kiírhatóak adatfolyamokba, állapota (a *time* adattag értéke) elmenthető.

Ha egy osztály tartalmaz nem szerializálható részeket (pl. egyik attribútuma egy végrehajtási szál), a többi része még lehet szerializálható. Azoknak az adattagoknak az esetében, amelyek nem szerializálhatóak, vagy nem szeretnénk, hogy részét képezzék az objektum perzisztens állapotának (tehát a szerializáció során nem kerülnek kiírásra) a *transient* típusmódosítót használjuk. Például:

```
import java.io.Serializable;
public class PersistentAnimation implements Serializable, Runnable {
      transient private Thread animator;
      private int animationSpeed;
      public PersistentAnimation(int animationSpeed) {
            this.animationSpeed = animationSpeed;
            animator = new Thread(this);
            animator.start();
      }
      public void run() {
            while (true) {
                  // do animation here
            }
      }
}
```

A fenti osztály példányai egy tetszőleges időzítésű animáció lejátszását valósítanák meg. Magáért az animálásért egy végrehajtási szál felelős. A szál nem képezheti részét az objektum perzisztens állapotának (ezért a megfelelő adattagot *transient* típusmódosítóval látjuk el), de ettől függetlenül az időzítés beállítása elmenthető.

A fenti példával az a probléma, hogy miután visszatöltjük az elmentett objektumot, a konstruktor nem kerül meghívásra, így az animáció elindítása sem történik meg. Azt szeretnénk, hogy az elmentett állapotinformációkból felépített objektumaink azonos módon viselkedjenek az egyszerű példányosítással létrehozott megfelelőikkel, de ez ebben az esetben nem így történik. Megoldást szolgáltathatna, ha kiegészítenénk az osztályt egy metódussal, amely elindítaná az animációt, de akkor a felhasználónak tudnia kellene erről a metódusról. Egy elegánsabb megoldás a protokoll módosítása.

7.5.2 A standard protokoll módosítása

A szerializálási protokoll módosítása a következő metódusok segítségével lehetséges:

```
private void writeObject(ObjectOutputStream out)
                        throws IOException;
private void readObject(ObjectInputStream in)
                        throws IOException, ClassNotFoundException;
```

Nem újradefiniálásról, vagy interfész megvalósításáról van szó, hanem a Java szerializálási mechanizmusa által nyújtott lehetőségről: a JVM ellenőrzi, hogy az osztályunk implementálja-e ezeket a metódusokat, és ha igen, akkor ezeket hívja meg akkor, amikor írni, vagy olvasni szeretnénk (ami egyébként az adatfolyam osztályok metódusainak segítségével történne).

Megjegyzendő még, hogy bizonyos biztonsági meggondolásokból a metódusok láthatósága *private.* A JVM meg tudja hívni őket (erre a *private* metódusok esetében is lehetősége van), de más külső osztályok nem férhetnek hozzájuk. A JVM ellenőrzi, hogy az osztályunk megvalósítja-e a *Serializable* interfészt, amennyiben ez megtörténik, ellenőrzi, hogy biztosítja-e a fenti privát metódusokat, és ha igen meghívja ezeket, paraméterként átadva nekik az adatfolyamnak megfelelő objektumra mutató referenciát. Ha ez történik, akkor az adatfolyam osztályok azonos nevű metódusait nem hívja meg a rendszer. Általában nem akarjuk implementálni a teljes szerializálási mechanizmust, csak ki szeretnénk terjeszteni azt. Ezt úgy tehetjük meg, hogy a *writeObject* és *readObject* metódusokon belül meghívjuk az alapértelmezett protokollnak megfelelő metódusokat:

```
out.defaultWriteObject();
in.defaultReadObject();
```

A példaként megadott osztályunk a következő módon alakulna:

```
import java.io.Serializable;
public class PersistentAnimation implements Serializable, Runnable {
      transient private Thread animator;
      private int animationSpeed;
      public PersistentAnimation(int animationSpeed) {
            this.animationSpeed = animationSpeed;
            startAnimation();
      }
      public void run() {
            while (true) {
                  // do animation here
            }
      }
      private void writeObject(ObjectOutputStream out)
                              throws IOException {
            out.defaultWriteObject();
      }
      private void readObject(ObjectInputStream in)
                           throws IOException, ClassNotFoundException {
            in.defaultReadObject();
            startAnimation();
      }
      private void startAnimation() {
            animator = new Thread(this);
            animator.start();
      }
}
```

Az osztályunkat kiegészítettük egy új metódussal, amely az animáció elindításáért felelős. Ezt a metódust a konstruktor végén, és a *readObject* metódus végén is meghívjuk. Ilyen módon az animáció indítása mind a példányosítás által „frissen" létrehozott, mind a deszerializáció során felépített objektumok esetében megtörténik, anélkül, hogy a felhasználónak bármit is tudnia kellene az új metódus létezéséről.

Amennyiben nem szeretnénk, hogy az osztályunk példányai szerializálhatóak legyenek, de az alaposztály megvalósítja a *Serializable* interfészt, megtehetjük, hogy a *writeObject* és *readObject* metódusokon belül egyszerűen kivételt generálunk (erre a *throw* kulcsszó alkalmazható, használatára a későbbiekben még visszatérünk):

```
private void writeObject(ObjectOutputStream out)
                            throws IOException {
      throw new NotSerializableException("Not today!");
}
private void readObject(ObjectInputStream in)
                         throws IOException, ClassNotFoundException {
      throw new NotSerializableException("Not today!");
}
```

7.5.3 Egyéni protokoll

Arra is lehetőségünk van, hogy teljes egészében implementáljuk a szerializálási mechanizmust, egy teljesen új, egyéni protokollt határozva meg. Ezt az *Externalizable* interfész megvalósításával tehetjük meg, amelynek metódusai:

```
public void writeExternal(ObjectOutput out)
                       throws IOException;
public void readExternal(ObjectInput in)
                       throws IOException, ClassNotFoundException;
```

Amennyiben az osztályunk megvalósítja az *Externalizable* interfészt, a JVM ezeket a metódusokat hívja meg a *readObject* és *writeObject* metódusok helyett. Fontos kihangsúlyozni, hogy ebben az esetben az alap protokoll által meghatározott műveletek nem lesznek végrehajtva, így a szerializálási mechanizmus teljes megvalósítása reánk hárul. Az eljárás hasznos lehet például speciális fájlformátumoknak (pl. *pdf*) megfelelő Java objektumok szerializálása esetén.

7.5.4 Verziókövetés

A szerializációval kapcsolatban nagyon fontos a verziókövetés. Gondoljunk arra az esetre, amikor lementjük egy objektum állapotát, módosítjuk az osztályt, majd vissza akarjuk tölteni az objektumot. Eredményül egy *InvalidClassException* típusú kivételt kapunk, mivel minden szerializálható osztályhoz egy egyedi azonosító rendelődik hozzá (amely minden objektum állapotával együtt elmentődik), és ha betöltéskor az azonosító nem talál (a módosított osztály azonosítója is módosul), nem megy végbe a deszerializáció.

Sok esetben hasznos lenne, ha ez nem így történne. Például, ha az osztályt csak egy új adattaggal egészítettük ki, valószínűleg azt szeretnénk, hogy az előzőleg elmentett objektumok beolvasásakor ezt az adattagot egy alapértelmezett értékkel inicializáljuk, a többi adattag esetében pedig használjuk az elmentett állapotinformációkat. Természetesen létezik megoldás.

Az azonosító egy *serialVersionUID* nevű mezőben van tárolva, és ezt az értéket manuálisan is beállíthatjuk. A JDK biztosít is számunkra egy eszközt (*serialver*), amely képes automatikusan

generálni ilyen azonosítót, alapértelmezetten a *hash* kód alapján. Amennyiben rögzítjük az azonosító értékét, és ezt a módosításkor nem változtatjuk, a deszerializáció megtörténhet. Fontos megjegyezni viszont, hogy ez csak kompatibilis változtatások esetében lehetséges (például attribútum vagy metódus törlése vagy hozzáadása). Inkompatibilis változtatásnak számít például a hierarchia megváltoztatása, interfész (pl. *Serializable*) megvalósításának eltávolítása stb. (a különböző típusú változtatások teljes listája megtalálható a *Java Serialization Specification* dokumentumban). Ilyen változtatások esetében a deszerializáció nem lehetséges.

7.5.5 *Object cache*

A szerializáció gyakorlati megvalósításával kapcsolatos kellemetlen élmények elkerüléséhez fontos lehet még megemlíteni az *ObjectOutputStream* által alkalmazott *object cache* mechanizmust. Ennek lényege, hogy az osztály nyilvántart egy-egy referenciát a beleírt objektumokról. Amikor egy objektumot szeretnénk kiírni az adatfolyamba, akkor ellenőrzi, hogy rendelkezik-e már erre az objektumra mutató referenciával, és amennyiben igen, az állapot kiírása nem fog újra megtörténni. Ez nagyon hasznos lehet bizonyos teljesítménnyel kapcsolatos meggondolásokból, például hálózati alkalmazások esetén, de hibalehetőségekhez is vezethet. Nézzük a következő példát:

```
ObjectOutputStream out = new ObjectOutputStream(...);
MyObject obj = new MyObject(); // must be Serializable
obj.setState(100);
out.writeObject(obj); // saves object with state = 100
obj.setState(200);
out.writeObject(obj); // does not save new object state
```

A második kiírásnál az objektum állapota nem lesz újra elmentve, így a célhoz érkező objektum állapota változatlan marad (továbbra is 100).

Természetesen, megtehetnénk, hogy minden kiírásnál új objektumot hozunk létre, de ez nem a legelegánsabb megoldás, és memóriapazarláshoz (esetenként túlcsorduláshoz) vezet. Amit megtehetünk viszont, hogy minden írás után zárjuk az adatfolyamot (az *ObjectOutputStream* objektumra meghívva a *close* metódust). A másik lehetőség, hogy a *reset* metódust alkalmazzuk, amely kiüríti a *cache*-t, de ennek a módszernek is van hátránya: a metódus meghívása a *cache* teljes kiürítését eredményezi.

7.6 Pipeline mechanizmus

A *pipeline* (csővezeték) mechanizmus alapja két adatfolyam összekapcsolása, olyan módon, hogy az egyik kimenete a másik bemenetét képezze. Tipikusan többszálú alkalmazások esetében, végrehajtási szálak közötti kommunikáció megvalósítására alkalmazzuk. A Java megvalósítására a *PipedInputStream* és *PipedOutputStream* osztályokat biztosítja, bináris adatfolyamok esetében, illetve a *PipedReader* és *PipedWriter* osztályokat szöveges adatfolyamok esetében.

Ahhoz, hogy egy vezetéket létrehozhassunk, két adatfolyamot, így ennek megfelelően két objektumot kell használnunk, és ezeknek kapcsolatban kell állniuk egymással. Példányosításkor az egyik konstruktornak át kell adnunk egy, a másik objektumra mutató referenciát:

```
PipedInputStream pis = new PipedInputStream();
PipedOutputStream pos = new PipedOutputStream(pis);
```

Vagy:

```
PipedOutputStream pos = new PipedOutputStream();
PipedInputStream pis = new PipedInputStream(pos);
```

Hasonlóan járhatunk el a szöveges adatok esetében alkalmazható *PipedReader* és *PipedWriter* típusú objektumok létrehozásánál is.

A mechanizmus működésének szemléltetésére tekintsük a következő példát. Két végrehajtási szálat hozunk létre, és „összekötjük" őket egy „csővezetékkel". Az egyik szál másodpercenként beírja a *pipe*-ba az aktuális dátumot és időt. A másik szál kiolvassa ezt, és kiírja a konzolra.

Az első szálnak megfelelő osztály forráskódja:

```
import java.io.PipedWriter;
import java.io.PipedReader;
import java.io.PrintWriter;
import java.util.Date;
public class FirstThread extends Thread {
      private PipedWriter pipewrite;
      private PipedReader piperead;
      public FirstThread(String name) {
            super(name);
            try {
                  pipewrite = new PipedWriter();
                  piperead = new PipedReader(pipewrite);
            } catch (java.io.IOException e) {}
      }
      public PipedReader getReader() {
            return piperead;
      }
      public void run() {
            PrintWriter ptw = new PrintWriter(pipewrite);
            while (true) {
                  String s = (new Date()).toString();
                  ptw.println(s);
                  System.out.println(getName() + ": " + s +
                              " has been written in pipe");
                  try {
                              sleep(1000);
                  } catch (InterruptedException e) {}
            }
      }
}
```

A konstruktoron belül meghívjuk az alaposztály (*Thread*) konstruktorát, és létrehozzuk a *pipeline* mechanizmus megvalósítására alkalmazott *PipedWriter* és *PipedReader* objektumokat. Az osztály biztosít egy *getReader* metódust, amely a csővezetéknek megfelelő bemeneti adatfolyamra mutató referenciát térít vissza. Ezt használjuk majd fel a másik szál létrehozásánál, így létrehozva a kommunikációs csatornát. A *run* metóduson belül a *PipedWriter* objektumunk segítségével létrehozunk egy *PrintWriter* objektumot, majd egy végtelen *while* ciklusson belül minden lépésben lekérjük és *String*-be alakítjuk az aktuális dátumot. Az így kapott *String* objektumot a *PrintWriter* példányunk segítségével beleírjuk a *pipeline*-nak megfelelő kimeneti adatfolyamba, majd egy másodpercig várakozunk.

A második szálnak megfelelő osztály forráskódja:

```
import java.io.PipedReader;
import java.io.BufferedReader;
import java.io.IOException;
public class SecondThread extends Thread {
      private PipedReader piperead;
      public SecondThread(String name, PipedReader pr) {
            super(name);
            piperead = pr;
      }
      public void run() {
            String s = null;
            BufferedReader br = new BufferedReader(piperead);
            while (true) {
                  try {
                        s = br.readLine();
                  } catch (IOException e ) {
                        System.out.println("Error reading");
                  }
                  System.out.println(getName() + ": " + s +
                                          " read from pipe");
                  try {
                        sleep(1000);
                  } catch (InterruptedException e) {}
            }
      }
}
```

A konstruktor paraméterként kapja a csővezetékhez tartozó bemeneti adatfolyamnak megfelelő *PipedReader* típusú objektumra mutató referenciát. A *run* metóduson belül ennek segítségével létrehozunk egy *BufferedReader* objektumot, majd egy végtelen *while* cikluson belül soronként olvasunk az adatfolyamból, kiírjuk a beolvasott sort a konzolra, és egy másodpercig várakozunk.

A program működésének szemléltetéséhez még létrehozunk egy *Control* osztályt:

```
public class Control {
      public static void main(String[] args) {
            FirstThread ft = new FirstThread("Writer");
```

```
            SecondThread st = new SecondThread("Reader",
                                                ft.getReader());
            ft.start();
            st.start();
        }
    }
```

A *main* metóduson belül létrehozzuk a két szálobjektumot, a másodiknak átadva az első *PipedReader* típusú attribútumára mutató referenciát, majd sorban elindítjuk a szálakat.

7.7 Javasolt gyakorlat

Az ötödik fejezet végén javasolt feladatot (alakzatok kirajzolás SWING felületen) egészítsük ki mentési és betöltési lehetőséggel. A megfelelő menük segítségével a felhasználó kérheti egy adott konfiguráció (alakzat típusa, színe, mérete) elmentését, illetve betöltését. A cél- és forrásállományok kiválasztásában *JFileChooser* komponensek szolgáltatnak segítséget. A betöltés esetében az alakzat megjelenítésén kívül a komponensek állapotát is változtassuk.

8

GYŰJTEMÉNY KERETRENDSZER

A gyűjtemény (*collection*) olyan objektum, amely más típusú objektumok egységben történő, összefoglaló jellegű tárolását szolgálja, és lehetőséget biztosít elemek hozzáadására, a tárolt elemek keresésére (a gyűjtemény bejárására), kinyerésére és manipulálására (változtatás, rendezés, stb.).

A szakirodalomban a gyűjteményeket (kollekciókat) tárolóknak (*containers*) is szokták nevezni (a jegyzet a gyűjtemény kifejezést használja, hogy kihangsúlyozza a különbséget a grafikus tároló komponensek és gyűjtemények között). Bár nem jellemző, a tárolt objektumok lehetnek különböző típusúak, de tipikusan egy természetes csoportosulásukat (*natural grouping*) tároljuk egy gyűjteményen belül.

Mivel az általánosan használt adatszerkezetek (listák, halmazok, hasító táblázatok, stb.) is ebbe a kategóriába tartoznak, programjainkban nagyon gyakran használunk gyűjteményeket. Munkánk megkönnyítésére a Java egy teljes keretrendszert biztosít, amelynek neve *Java Collections Framework* (JCF) [17].

A JCF által biztosított eszközöket a következő kategóriákba sorolhatjuk:

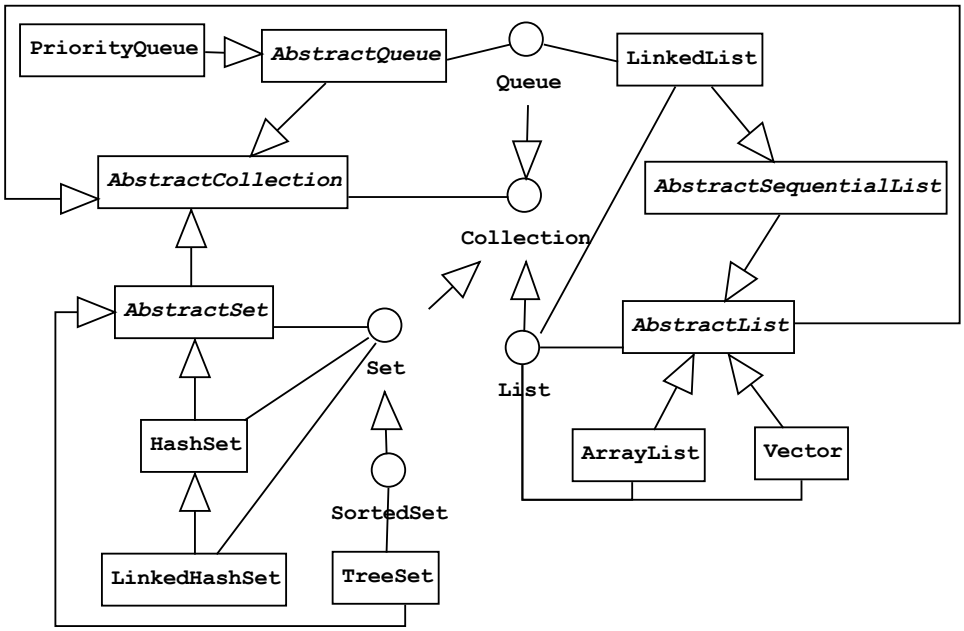
- interfészek: absztrakt adattípusok, amelyek gyűjtemények megvalósítástól független ábrázolását szolgálják, ezek szolgáltatásait, publikus interfészeit határozzák meg;
- megvalósítások: a gyűjtemény interfészek konkrét implementációi;
- algoritmusok: gyűjteményekkel kapcsolatos hasznos és gyakran szükséges műveletek megvalósításai (keresés, rendezés, stb.). A gyűjtemény interfészeket implementáló osztályok példányaira alkalmazhatjuk ezeket. A keretrendszer szerkezetének köszönhetően, egy adott algoritmust implementáló metódus különböző gyűjtemény-megvalósítások esetében alkalmazható.

A következő részben a JCF keretrendszer szerkezetét, az interfészek és osztályok hierarchiáját ismertetjük

8.1 JCF típushierarchia

A fontosabb JCF gyűjtemény interfészeknek és megvalósításoknak a hierarchiája a 8.1 ábrán látható. Az osztálydiagram csak az interfészek és osztályok neveit, valamint a közöttük lévő kapcsolatokat jeleníti meg, az attribútumokat és metódusokat mellőzi.

A keretrendszer részét képező legtöbb interfész és osztály *a java.util* csomagban kapott helyet. A diagram által ábrázolt hierarchia csúcsán a *Collection* interfész található. A különböző speciálisabb gyűjteménytípusoknak (listák, halmazok, stb.) megfelelő interfészek a *Collection* interfészből vannak származtatva.



8.1 ábra: JCF gyűjtemények. A *Collection* interfész leszármazottjai és megvalósításai. Részleges, leegyszerűsített osztálydiagram.

Megjegyzendő, hogy az osztálydiagram csak részleges, és leegyszerűsített. Közel sem tartalmazza a JCF összes megvalósítását, és nem tüntet fel bizonyos interfészeket, kapcsolatokat. Például, minden konkrét gyűjtemény megvalósításnak megfelelő osztály klónozható és szerializálható (implementálják a *Cloneable* és *Serializable* interfészeket). Minden gyűjtemény bejárható, és ennek megfelelően a *Collection* interfész az *Iterable* interfész (*java.lang* csomag) leszármazottja (az egyszerűség kedvéért ezt az interfészt is kihagytuk a diagramról). Ezen kívül a hasító táblázatoknak megfelelő interfészek és osztályok a keretrendszeren belül egy külön hierarchiának a részei (8.2 ábra).

A *Collection* interfész, az *Iterable* interfész leszármazottja, a gyűjteményhierarchia központi eleme, a legáltalánosabb interfész. Akkor alkalmazzuk, ha maximális általánosságra törekszünk, például paraméterátadások esetében. A *Collection* interfésznek nincs konkrét megvalósítása a JCF keretrendszeren belül. Az *AbstractCollection* absztrakt osztály megvalósítja az interfészt, de nem ad konkrét implementációt minden metódusára, csak alaposztályként szolgál más, konkrét gyűjteménytípusoknak megfelelő osztályok számára.

A *Collection* interfészből származtatott *Set* interfész a halmazoknak felel meg, amelyek duplikált elemeket nem tartalmazó gyűjtemények. A hierarchia következő szintjén található a rendezett halmazoknak megfelelő *SortedSet* interfész. A *Set* interfészt valósítja meg az *AbstractCollection* osztályból származtatott *AbstractSet* alaposztály. Ennek leszármazottja a *HashSet* osztály, és a hierarchia következő szintjén található *LinkedHashSet* osztály. A *SortedSet* interfészt megvalósítja a *TreeSet* osztály, amely szintén az *AbstractSet* osztály leszármazottja. A konkrét megvalósításokról a későbbiekben még szó esik.

A *List* interfész szintén a *Collection* alapinterfész leszármazottja, olyan szekvenciáknak is nevezett gyűjteményeknek felel meg, amelyek esetében minden elem meghatározott pozícióval (a pozíciónak megfelelő index egy pozitív egész érték) rendelkezik, és az elemekre pozíciójuk segítségével hivatkozhatunk. A *List* interfészt valósítja meg az *AbstractCollection* osztályból származtatott *AbstractList* alaposztály. Ennek leszármazottjai az *ArrayList*, valamint a *Vector* osztályok. Az *AbstractSequentialList* absztrakt osztály is az *AbstractList* leszármazottja, és ebből származik a *LinkedList* osztály.

A *Queue* várakozási soroknak megfelelő interfész, amelyet tipikusan feldolgozásra váró elemeket tároló gyűjtemények esetében alkalmazunk, és ezekkel kapcsolatos speciálisabb műveleteket biztosít. Az elemek a legtöbb megvalósítás esetében a FIFO elv szerint lesznek rendezve, de ez nem kötelező (kivétel például a prioritási sor). A *Queue* interfész absztrakt megvalósítása az *AbstractQueue* osztály, és ebből származik a *PriorityQueue* osztály. Az interfészt a *LinkedList* osztály is megvalósítja.

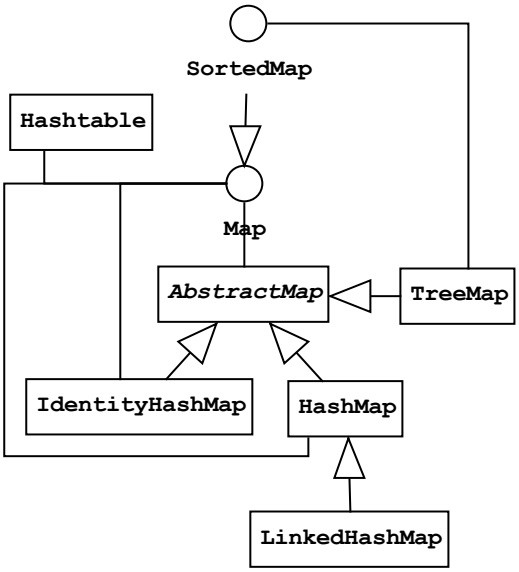
A hasító táblázatoknak megfelelő alapinterfész a *Map*, és az idekapcsolódó interfészek és osztályok a JCF hierarchia egy különálló ágát képezik. A *Map* típusú tárolók speciális gyűjtemények, rögzített szabályoknak megfelelő adatszerkezetek. A hierarchiát a 8.2 ábrán látható osztálydiagram szemlélteti. Az előzőhöz hasonlóan ez a diagram is részleges és leegyszerűsített.

A hasító táblázatoknak megfelelő *Map* interfészt valósítja meg az *AbstractMap* absztrakt osztály, és ebből származnak többek között a *HashMap* és *IdentityHashMap* osztályok. A hierarchia következő szintjén megtalálhatjuk például a *LinkedHashMap* osztályt. A *Map* interfész leszármazottja a *SortedMap*, amely rendezett táblázatok esetében használható, és megvalósítása az *AbstractMap* osztályból származtatott *TreeMap* osztály. A *Map* interfészt a *Hashtable* osztály is megvalósítja.

Ha megnézzük a *Collection* interfész deklarációját, a következő fejlécet láthatjuk:

```
public interface Collection<E> extends Iterable<E>
```

Az olvasó számára még furcsa lehet az interfészek nevei után kisebb jel és nagyobb jel között megjelenő *E* betű. A megértéshez a generikus típusok ismerete szükséges, és ezért a következő résznek ez lesz a témája.



8.2 ábra: JCF hasító táblázatok. A *Map* interfész leszármazottjai és megvalósításai. Részleges, leegyszerűsített osztálydiagram.

8.2 Generikus típusok

Amikor létrehozunk egy adott gyűjteménytípusnak megfelelő osztályt, ezt úgy kell megtennünk, hogy az általános legyen, a későbbiekben fel lehessen használni különböző típusú elemek tárolására. Az osztálynak a tárolt elemek típusától függetlenül alkalmazhatónak kell lennie. A Java esetében minden osztály az *Object* ősosztály leszármazottja, így megoldást jelenthet, ha *Object* típusú elemeket tárolunk a gyűjteményen belül. Például, egy *Object* típusú elemekből álló tömböt, vagy láncolt listát alkalmazhatunk, és így a későbbiekben bármilyen típusú elemet hozzáadhatunk a gyűjteményhez. A JCF első verzióiban ez pontosan így is történt. A megközelítésnek van azonban néhány hátránya.

Bár kijelentettük, hogy a gyűjtemények elemeinek nem feltétlenül kell teljesen megegyező típusúaknak lenniük, a gyűjtemények csak egy „természetes csoportosulás" egységben történő tárolását szolgálják, ennek ellenére általában legalább az interfészek, vagy ősosztályok szintjén megegyező típusú elemeket szeretnénk egy gyűjteményen belül tárolni. A magyarázat kézenfekvő: az elemek kinyerésénél szükséges, hogy egységesen tudjuk feldolgozni őket, ismerjük tulajdonságaikat. Ha egyszerűen *Object* típusúként tároljuk őket, akkor a kinyerésnél, ahhoz, hogy tovább tudjunk dolgozni velük (például, meghívjuk a tárolt objektumok valamelyik metódusát, amely nem az *Object* osztály metódusa), mindenképpen át kellene őket alakítanunk egy megfelelő típusba (*cast*). Ez egyrészt kényelmetlen, másrészt nem szűri ki a hibalehetőségeket. Mivel gyakorlatilag a gyűjteményhez bármilyen típusú elem hozzáadható, a kinyerés pillanatában helytelenül feltételezhetjük, hogy az illető elem valamilyen típusba

tartozik, és, amikor megpróbáljuk átalakítani, futási idejű hibát kapunk. A hátrányokat hívatottak megszüntetni a generikus típusok.

A generikus típusok támogatásának bevezetése (*generics*) a Java 5.0 egyik legfontosabb újítása volt. Ha azt mondanánk, hogy erre a fentiekben leírt gyűjteményekkel kapcsolatos problémák jelentették a motivációt, és a gyűjtemények jelentik a legfőbb alkalmazási területet, valószínűleg nem tévednénk, de nem csak ennyiről van szó. A generikus típusok bevezetésének más előnyei is vannak, és ezek nem feltétlenül csak a gyűjteményekkel kapcsolatosak. Ezt alátámaszthatja az a tény is, hogy a *Class* osztály is generikus, típusparaméterrel rendelkezik. A megközelítésnek több más témakörben is jelentősége van (pl. *reflection*), de mivel ezek haladóbb témáknak számítanak, a jegyzet inkább a gyűjteményekkel kapcsolatos részeket tárgyalja.

A generikus típusok egy új absztrakciós szint bevezetésére adnak lehetőséget a típusok tekintetében. A megközelítés megengedi, hogy az interfészek és osztályok típusparaméterekkel rendelkezzenek. Ilyen módon, létre tudunk hozni generikus interfészeket és osztályokat, amelyek esetében a típusparaméterek a későbbiekben konkrét típusokkal lesznek helyettesítve. A módszer emlékeztetheti az olvasót a C++ nyelvből ismert osztálysablonok (*template classes*) alkalmazására, de a későbbiekben látni fogjuk, hogy a két megközelítés között lényeges eltérések vannak a megvalósítás tekintetében.

8.2.1 Típusparaméterek

Az előző részben láthattuk, hogy a *Collection* interfész generikus, egy típusparaméterrel (*E*) rendelkezik. Ez azt jelenti, hogy a gyűjtemények „valamilyen" típusú elemeket tárolnak, de az elemek konkrét típusát csak egy adott gyűjtemény létrehozásakor fogjuk meghatározni. Hasonlóan a JCF minden gyűjtemény interfésze (és osztálya) generikus.

Egy generikus interfészen, vagy osztályon belül a típusparamétert ugyanúgy használjuk, mintha egy konkrét típussal dolgoznánk, és több típusparamétert is megadhatunk. Ebben az esetben a megadott példához hasonlóan az osztály (vagy interfész) neve után kisebb és nagyobb jelek között meg kell adnunk ezeknek a paramétereknek a neveit (az elnevezési konvenció szerint egy nagybetűvel jelölve őket), egymástól vesszőkkel elválasztva.

Ahhoz, hogy a generikus típusok jelentőségét jobban megértsük, először tekintsünk egy egyszerű példát gyűjtemény létrehozására a generikus típusok alkalmazása nélkül:

```
List myList = new ArrayList();
myList.add(new Integer(42));
Integer i = (Integer) myList.get(0);
```

Létrehoztunk egy *ArrayList* példányt, hozzáadtunk egy *Integer* objektumot, amely a 42 egész értéket burkolja. A burkoló osztályok leírásánál említett automatikus dobozolási mechanizmus egyszerűbb írásmódot is megengedett volna, de így a példa kihangsúlyozza, hogy a JCF gyűjteményekben csak objektumokat tárolhatunk. A *get* metódus segítségével kiolvastuk a lista első elemét. Ahhoz, hogy az *i* referenciának megfeleltessük a kiolvasás során visszatérített objektumot, először *Integer* típusba kell azt alakítanunk (a *get* metódus *Object* típusú referenciát térít vissza). Az átalakítás kényelmetlen, és hibaforrás is lehet. A listához *Integer* típusú objektum helyett például egy *String* példányt is hozzáadhattunk volna, és az átalakítás

ebben az esetben futási idejű kivételt eredményezne (feltételezésünk, miszerint a lista első elemének típusa *Integer*, hibás lehetne).

Generikus típus alkalmazásával kódunkat a következő formában írhatnánk:

```
List<Integer> myList = new ArrayList<Integer>();
myList.add(new Integer(42));
Integer i = myList.get(0);
```

Az első sorban a deklarációnál és példányosításnál a *List* interfész, illetve *ArrayList* osztály típusparamétereit (*E*) az *Integer* konkrét típussal helyettesítjük. Az elem hozzáadása hasonló módon történik, de a kiolvasáskor már nem kell átalakítanunk a visszatérített értéket. Ha a hozzáadáskor megpróbálnánk nem *Integer* típusú (pl. *String*) példányt hozzáadni a gyűjteményhez, fordítási hibát kapnánk, programunk nem lenne lefordítható.

Megjegyzendő, hogy az első változata a kódnak az új Java verziókban is érvényes, lefordítható. Ennek legfőbb oka, hogy szükséges volt a kompatibilitás megőrzése az előző verziókkal. A fordító jelezni fogja a veszélyt, figyelmeztet (*warning*), de figyelmeztetése figyelmen kívül hagyható. Természetesen, ez nem ajánlott.

A generikus típusok alkalmazásánál nagyon fontos odafigyelnünk egy dologra. Tekintsük a következő példát:

```
List<String> ls = new ArrayList<String>();
List<Object> lo = ls;
```

Helyes a fenti kód? Lehetséges, hogy az intuíció azt mondatná velünk, hogy igen: egy *String* objektumokból álló lista *Object* típusú elemekből álló listának is tekinthető, mivel, mint minden Java osztály, a *String* is az *Object* leszármazottja (*String is Object*). De nézzük az esetleges folytatást:

```
lo.add(new Object());
String s = ls.get(0);
```

Az *ls* és *lo* referenciák ugyanarra a gyűjtemény objektumra mutatnának. A *lo* referencián keresztül a gyűjteményhez bármilyen objektumot hozzáadhatnánk, és ebben az esetben az elemek *ls* referencián keresztüli kinyerése hibás lehetne. Természetesen ezt a rendszer nem engedheti meg. A fenti kódrészlet hibás, fordítási hibát eredményez. Érdemes megjegyeznünk tehát, hogy, ha a *B* osztály az *A* osztálynak leszármazottja, és *C* egy generikus típus, *C<B>* nem tekinthető a *C<A>* leszármazottjának.

8.2.2 Jokerek

Ahogyan azt az előző részben láthattuk, a szabályok rögzítettek, de a megkötéseket némileg feloldja a „jokerek" (*wildcards*) használatának lehetősége. Tekintsük a következő metódust:

```
public void myMethod(Collection c) {…}
```

A metódus általános, bármilyen gyűjteményt kaphat paraméterként, de „régi" stílusú, „elavult" kódnak számít, mivel nem használtunk generikus típust. Nem is biztonságos, mivel a metódus belsejében tetszőleges típusú objektumokat adhatnánk hozzá a gyűjteményhez. Nézzük, megoldást jelentene-e a következő átalakítás:

```
public void myMethod(Collection<Object> c) {…}
```

A válasz: nem. A két metódus között lényeges a különbség. Az első bármilyen gyűjteményt kaphat paraméterként, a második viszont csak kizárólag *Object* típusú objektumokat tartalmazó gyűjteményt. Amint azt már láttuk, az ilyen gyűjtemények nem ősei a más típusú objektumokat tartalmazó gyűjteményeknek, így nem helyettesíthetőek be azokkal. A megoldást jokerek alkalmazása jelentheti:

```
public void myMethod(Collection<?> c) {…}
```

A „*?*" egy „joker", azt jelenti, hogy helyére bármilyen konkrét típus behelyettesíthető. A *Collection<?>* úgy értelmezhető, hogy „valamilyen típusú objektumok gyűjteménye". Használatával hasonló hatást értünk el, mint az első kód esetében, de úgy, hogy kódunk már megfelel az új verziónak, és elkerültük a fordító figyelmeztetését. A metódusunk most már bármilyen gyűjtemény esetében meghívható. Azért még maradt egy lényeges különbség az első verzióhoz képest, és ettől tekinthető a megoldás biztonságosabbnak. A metódus törzsében nem használhatunk a következőhöz hasonló megoldásokat (az első verzió esetében ez lehetséges lett volna):

```
c.add(new Object());
```

A gyűjteményről nem tudjuk, hogy milyen típusú elemeket tartalmaz, csak azt, hogy „valamilyen" típusúakat, így nem adhatunk hozzá *Object* típusú elemeket. Az *add* metódus *E* típusú argumentumot vár (az *E* a generikus interfész típusparamétere, ilyen típusúak a gyűjtemény elemei). Esetünkben az *E* típus „*?*", ismeretlen. Bármilyen argumentumot adunk át a metódusnak, annak legalábbis az *E* leszármazottjának kellene lennie, és például az *Object* típusú objektum nem a „*?*" leszármazottja. Mivel nem ismerjük az *E* típusát, egyszerűen nem hívhatjuk meg a metódust. Természetesen a *get* metódust nyugodtan használhatjuk, mert a visszatérített referencia biztosan olyan objektumra mutat, amely az *Object* osztálynak, vagy valamelyik leszármazottjának példánya.

A jokerek esetében megkötéseket, „határokat" alkalmazhatunk, ezt a következő példák szemléltetik:

```
public void myMethod(Collection<? extends BaseType>) {…}
public void myMethod(Collection<? super ExtendedType>) {…}
```

Az első esetben a *Collection<? extends BaseType>* egy olyan gyűjteményt jelöl, amelynek az elemei *BaseType* típusúak, vagy a *BaseType* leszármazottjai. A „*?*" szintén ismeretlen típust jelöl, de ezúttal egy megkötést is megszabtunk: az ismeretlen típus vagy *BaseType*, vagy annak leszármazottja. Amint láthatjuk, az „*extends*" kulcsszó ebben az esetben nem szó szerint értendő, a konkrét típus maga a *BaseType* is lehet, nem kell feltétlenül ebből származtatottnak lennie. Tulajdonképpen, egy „alsó korlátot" szabtunk meg a típushierarchiában. A megkötés fordítottját alkalmaztuk a második esetben. Itt a *Collection<? super ExtendedType>* egy olyan gyűjteményt jelöl, amelynek az elemei vagy *ExtendedType* típusúak, vagy az *ExtendedType* példányok öröklik a típusuknak megfelelő tulajdonságokat (az *ExtendedType* az ismeretlen típus leszármazottja). Ebben az esetben egy „felső korlátot" szabtunk meg a típushierarchiában.

Egy kis kiegészítésként megjegyezhetjük még, hogy a határok esetében több típust is alkalmazhatunk. Bár a Java nem támogatja a többszörös öröklést, egy osztály több interfészt is megvalósíthat, így több típusnak is lehet „leszármazottja" (altípusa). Ennek megfelelően az *extends* kulcsszó után felsorolhatjuk ezeket a típusokat, egymástól „*&*" jellel elválasztva.

8.2.3 Generikus metódusok

A jokerek használata mellett a Java lehetőséget ad generikus metódusok alkalmazására is. Példaként tekintsük a *Collection* interfész egyik metódusát, amely a gyűjteményt tömbbe alakítja:

```
public <T> T[] toArray(T[] a);
```

A metódus a gyűjtemény elemeit beteszi az *A* tömbbe, és visszatérít egy *A* tömbre mutató referenciát. Amennyiben az *A* tömb mérete ezt a műveletet nem teszi lehetővé, létrehoz egy megfelelő méretű, az *A* tömb típusával megegyező típusú tömböt (a tömb típusát futási időben ellenőrzi), és egy erre mutató referenciát térít vissza.

A *Collection* generikus interfész típusparamétere *E*, elemei ilyen típusúak. A metódus esetében egy másik típusról, és ennek megfelelően új típusparaméterről (*T*) van szó. Ezért beszélhetünk generikus metódusról. Természetesen ahhoz, hogy a művelet elvégezhető legyen, az *E* típusnak a *T* típus leszármazottjának kell lennie, ellenkező esetben kivételt kapunk.

Megjegyezhetjük azt is, hogy azoknak a metódusoknak az esetében, ahol jokereket alkalmaztunk, generikus metódusokat is alkalmazhattunk volna. Példaként szintén tekinthetjük a *Collection* interfész metódusait:

```
public boolean containsAll(Collection<?> c);
public boolean addAll(Collection<? extends E> c);
```

Az első metódus ellenőrzi, hogy a gyűjtemény tartalmazza-e a paraméterként kapott gyűjtemény minden elemét. A második metódus hozzáadja a gyűjteményhez a paraméterként kapott gyűjtemény minden elemét. Ez az utóbbi művelet természetesen csak akkor lehetséges, ha a paraméterként kapott gyűjtemény elemeinek típusa az *E* leszármazottja.

A fenti metódusokat a következő formában is írhatnánk:

```
public <T> boolean containsAll(Collection<T> c);
public <T extends E> boolean addAll(Collection<T> c);
```

Miért alkalmaztak a fejlesztők mégis jokereket? Mi a szabály? Ha összehasonlítjuk ezeket a példákat a fentebb tárgyalt generikus metódussal, azt láthatjuk, hogy a fentivel ellentétben, ezeknek a metódusoknak az esetében nincsen függőség a visszatérített érték és a paraméterek típusa között. A típusparaméterek alkalmazásának a célja kizárólag az, hogy az argumentumok különböző típusúak lehessenek. Ha csak ennyiről van szó, akkor a joker a megfelelő megoldás. Ha azonban függőség van a paraméterek típusai között, vagy a paramétertípusok és a visszatérített érték típusa között, akkor ez a függőség már nem írható le jokerek segítségével. Ilyenkor a generikus metódusok alkalmazása jelenti a megoldást.

Amikor nem feltétlenül szükséges generikus metódust alkalmaznunk (nincsenek függőségek), használjunk inkább jokereket, így átláthatóbb kódot kapunk eredményül. A jokerek előnyeként felhozható az a tény is, hogy metódusokon kívül is használhatóak, például attribútumok típusának meghatározásakor. Természetesen arra is van lehetőségünk, hogy generikus metódusokkal együtt alkalmazzuk őket. Példaként tekintsük a JCF *Collections* osztályának (nem összetévesztendő a *Collection* interfésszel) következő metódusát (a *Collections* osztályról a későbbiekben még szó esik):

```
public static <T> void copy(List<? super T> dest,
                            List<? extends T> src);
```

A metódus két listát kap paraméterként, és a második elemeit az elsőbe másolja. Tetszőleges típusú elemeket tartalmazó listákat kaphat paraméterként, de a két lista elemeinek típusa között függőségi viszony áll fent: ha az első lista elemeinek típusa *T*, vagy annak egy őse, akkor a második lista elemeinek is *T* típusúaknak kell lenniük, vagy a *T* leszármazottjainak, ahhoz, hogy a másolás megtörténhessen.

8.2.4 Elavult örökség

A generikus típusok bevezetésénél fontos szempontnak kellett lennie a kompatibilitás megőrzésének a régi Java verziókkal. A régi programoknak az új platformokon is fordíthatóaknak, futtathatóknak kellett maradniuk. Már szó volt arról, hogy, ha például a gyűjtemények esetében nem használjuk ki a generikus típusok lehetőségeit, „régi stílusú" kódot írunk, a fordító csak figyelmeztet, de az illető kód lefordítható. Az ilyen esetekben oda kell figyelnünk az ebből adódó hibalehetőségek elkerülésére. A „régi stílusú" kód neve az angol szakterminológiába *legacy code* („örökölt kód"), és ha használni akarjuk, vagy programunkba olyan modulokat integrálunk, amelyek használják, a megfelelő óvatossággal kell eljárnunk. Szerencsére a fordító figyelmeztetései segítenek ebben.

Tekintsük a következő példát:

```
List<String> ls = new ArrayList<String>();
List l = ls;
l.add(x);
```

A kód lefordítható, a fordító csak figyelmeztet, hogy nem tudja elvégezni a megfelelő ellenőrzéseket, így a kód nem biztonságos (*unchecked warning*). A példában régi stílusú kódot akartunk alkalmazni az új stílusú kódon belül, és ez, bár lehetséges, nem biztonságos. Néhány speciális probléma szükségessé tehet hasonló megoldásokat, de ez nem jellemző, és ha lehet, kerüljük az ilyen helyzeteket. Az eset fordítottja gyakrabban előfordul, amikor például egy régebbi modult akarunk integrálni programunkba. Bár mi kihasználjuk a generikus típusok lehetőségeit, a régi kód még a *generics* bevezetése előtt lehetett megírva, és ezt figyelembe kell vennünk. Tekintsük a következő példát:

```
class LegacyCode {
      public static List getItems() {
            List list = new ArrayList();
            list.add(new Integer(1));
            list.add("two");
            return list;
      }
}
```

Az osztály nem használja a generikus típusokat, sőt, a listához különböző típusú objektumokat ad hozza. Egyszerűen egy *Object* típusú elemeket tartalmazó gyűjteményt hoz létre és egy erre mutató referenciát térít vissza. Tételezzük fel, hogy az osztályt fel kell használnunk saját kódunkban:

```
class NaiveClient {
      public void processItems() {
            List<Integer> list = LegacyCode.getItems();
            int s = 0;
            for (int i : list) s += i;
      }
}
```

Mi a kódunkban már használtunk generikus típust, de tévesen feltételeztük a *LegacyCode* osztály metódusáról, hogy milyen típusú elemeket tartalmaz. Azt feltételeztük, hogy egész értékeket burkoló objektumokat tartalmaz, és ennek megfelelően dolgoztuk fel az elemeket, de a hibás feltételezésünk futási idejű hibát eredményezne.

Az átalakítások esetében is a megfelelő óvatossággal kell eljárnunk. Például, a következő kód lefordítható, csak figyelmeztetést kapunk (*unchecked warning*), mivel az átalakítás nem minden esetben működne:

```
Collection cs = new ArrayList<String>();
Collection<String> cstr = (Collection<String>) cs;
```

Hasonló a helyzet a típusparaméterek esetében is:

```
<T> T badCast(T t, Object o) {return (T) o;}
```

Az átalakítás nem feltétlenül működőképes, a program lefordítható, de a fordító figyelmeztet a nem ellenőrzött és ennek megfelelően nem biztonságos kódra.

Ha hibamentes programokat szeretnénk, a fenti lehetőségeket szem előtt kell tartanunk. Az „elavult örökséggel együtt kell élnünk", mivel a kompatibilitás csak így volt megőrizhető. Hogyan volt ez megvalósítható a gyakorlatban? Erre próbál rövid választ adni a következő rész.

8.2.5 A színfalak mögött

A előzőekben leírt módszerek emlékeztethetik az olvasót a C++ nyelvből ismert osztálysablonok alkalmazására. Bár a cél azonos, és az eredmény hasonló, a Java és C++ megközelítései között lényeges különbségek is vannak. A legfontosabb különbség, hogy a C++ esetében a típusparamétereknek konkrét típusokkal történő behelyettesítésekor valóban „elkészül" az osztálynak egy megfelelő változata, azaz minden új konkrét típus megjelenésekor a fordító létrehozza (lefordítja) az osztálynak egy megfelelő változatát. Bár vannak lényeges különbségek, tulajdonképpen a C++ által alkalmazott sablon metaprogramozás hasonlóságokat mutat a preprocesszor direktívák és makrók használatával. A Java megközelítése merőben eltérő.

A Java módszere a típusinformációk törlése (*type erasure*), és ennek megfelelően nem jön létre minden konkrét típus számára egy külön osztály. A fordító a típusparamétereket fordításkor *Object* típussal helyettesíti, vagy, amennyiben ezek esetében meg van határozva egy származtatási viszony (*extends*), akkor az ősosztály típusával. A módszer legfőbb motivációja, hogy segítségével megőrizhető volt a kompatibilitás a régi (a generikus típusok bevezetése előtt megírt) programokkal.

A Java esetében a generikus osztálynak, a konkrét típusbehelyettesítések számától függetlenül, csak egyetlen lefordított változata fog létezni. Úgy is értelmezhetjük, hogy a lefordított kód független a megadott konkrét típusoktól, a fordító a kódból „törli" a (kisebb és nagyobb jelek között megadott) konkrét típusok neveit. Ettől függetlenül a fordítás közben felhasználhatja ezeket a megfelelő ellenőrzések elvégzésére, és a kényelmetlen „manuális" átalakítási műveletek szükségtelenné tételére.

A fentiekben leírtak fontos kiegészítéseként megjegyzendő, hogy a *type erasure* nem jelenti azt, hogy a típusparaméterek helyettesítésével kapcsolatos információk teljesen elvesztődnek a lefordított kódban. Ez azt jelentené, hogy futási időben semmiképpen sem szerezhetünk információt a típusbehelyettesítésekről. Ez csak részben igaz. Ha például egy listának megfelelő osztályból példányosítunk, megadva az elemek konkrét típusát, akkor a lista objektumról később (futási időben) valóban nem tudjuk eldönteni, hogy milyen típusú elemeket tartalmaz. A listát megvalósító generikus osztálynak csak egy lefordított, általános változata fog létezni (amely *Object* típusú elemeket tartalmaz). Ennek ellenére a konkrét típusok behelyettesítésével kapcsolatos információk elmentődnek a helyettesítést (példányosítást) megvalósító osztály definícióját tároló objektumban. Esetünkben a lista objektumot létrehozó osztálynak megfelelő *Class* objektumban. A Java *reflection* mechanizmusa lehetőséget ad arra, hogy ebből az objektumból futási időben kinyerjük az információkat. Java-val kapcsolatos tanulmányaink jelen fázisában ezek a módszerek számunkra még nem túl fontosak. Ami fontosabb, hogy a *generics* alkalmazása a gyűjtemények esetében megkímél az átalakításoktól, és fordítási időben történő ellenőrzéseket téve lehetővé, csökkenti a hibalehetőségeket. Térjünk vissza a gyűjteményekre.

8.3 JCF interfészek

A JCF interfészek a különböző típusú gyűjtemények absztrakt, megvalósítástól független reprezentációi. Tulajdonképpen (mint az interfészek általában) szerződéseknek tekinthetőek, amelyeket a megvalósító osztályoknak be kell tartaniuk. Az általános javaslat az, hogy amennyiben lehetséges „interfészekben gondolkodjunk", interfészeket használjunk. A legtöbb esetben nem fontosak az implementációs részletek. Például, ha egy metódus egy gyűjteményt vár paraméterként, a legtöbbször elég azt tudnia, hogy az illető gyűjtemény betartja a rávonatkozó szerződést, biztosítja a szükséges metódusok megvalósításait. Elegendő lehet, ha tudjuk, hogy az illető gyűjtemény bejárható, elemeket adhatunk hozzá, és módosíthatjuk, vagy eltávolíthatjuk ezeket. Szükségtelen lenne a metódusunkat használó osztály fejlesztőjének „kezét megkötnünk" azzal, hogy egy konkrét megvalósítást várunk paraméterként. A metódusok által visszatérített értékek esetében hasonló a helyzet. Természetesen a gyűjtemény objektum létrehozásánál (a példányosításnál) választanunk kell egy adott megvalósítást, és ezen kívül bizonyos megkötések (például lehetséges, hogy a hozzáférésnek szinkronizáltnak kell lennie) megkövetelhetik tőlünk egy adott implementáció alkalmazását, de amennyire lehetséges, az általánosságra kell törekednünk.

8.3.1 Gyűjtemények bejárása

A *Collection* interfész az *Iterable* interfész leszármazottja. Tulajdonképpen ez utóbbi képezi a gyűjteményhierarchia csúcsát, mivel minden gyűjteménynek bejárhatónak kell lennie. Az *Iterable* interfész egyetlen metódust deklarál:

```
public interface Iterable<T> {
      Iterator<T> iterator();
}
```

Az *iterator* metódus egy *T* típusú gyűjteménynek megfelelő *Iterator* típusú objektumra mutató referenciát térít vissza. Ennek az iterátor objektumnak a segítségével tudunk bejárni egy adott, *T* típusú elemeket tartalmazó gyűjteményt. Az *Iterator* szintén egy interfész, amely három metódust deklarál:

```
public interface Iterator<E> {
      boolean hasNext();
      E next();
      void remove();           //optional
}
```

A bejárás (iterálás) során az iterátorral a gyűjtemény elejéről indulunk, és lépésenként haladunk, így az iterátor mindig a gyűjtemény egy adott eleménél „tart". Az első metódus segítségével lekérdezhető, hogy a gyűjtemény tartalmaz-e egy következő elemet. A második metódus a következő elemre mutató referenciát téríti vissza. Az utolsó metódus az iterátor által utoljára visszatérített elemet törli a gyűjteményből (tehát minden *next* metódushívás után csak egyszer hívható meg). A metódus opcionális, abban az értelemben, hogy nem minden megvalósításnak kell feltétlenül a törlés műveletet támogatnia. Lehetnek olyan megvalósítások, amelyek egyszerűen kivételt dobnak a metóduson belül.

A gyűjtemények bejárására az iterátorok alkalmazásán kívül, a Java 5.0 verziójától kezdve van egy másik lehetőségünk is, a *for-each* ciklus használata:

```
for (Object o : collection) System.out.println(o);
```

A cikluson belül az *o* referencia mindig a *collection* nevű gyűjtemény adott iterációnak megfelelő elemére mutat. Ha egyszerű bejárást szeretnénk megvalósítani, a cikluson belül kinyerve a lista elemeit, akkor ez a lehetőség egyszerűbb írásmódot, elegánsabb, áttekinthetőbb, kényelmesebb megoldást jelent. Ha nem egy egyszerű bejárásról van szó, hanem például törölni is szeretnénk elemeket, iterátort kell alkalmaznunk. A *for-each* ciklus is iterátort alkalmaz a háttérben, de ezt „elrejti" előlünk, így törlési műveletre nincs lehetőségünk. Természetesen, iterátor alkalmazásával ez is lehetséges:

```
static void filter(Collection<?> c) {
      for (Iterator<?> it = c.iterator(); it.hasNext();)
            if (cond(it.next())) it.remove();
}
```

A példa egy gyűjtemények szűrésére alkalmazható metódust szemléltet, ahol a bejárás közben bizonyos feltétel teljesülése esetén törölnünk is kell, így iterátort kell használnunk

a bejáráshoz. Ha egyidejűleg több gyűjtemény bejárását akarjuk megvalósítani, szintén az iterátorok alkalmazása jelent számunkra jobb megoldást.

8.3.2 Alapvető metódusok

Az előző részekben már részben megismerkedtünk az *Iterable* interfészből származó, a gyűjteményhierarchia központi elemének tekinthető *Collection* interfésszel. Összefoglalóként tekintsük a teljes interfészt:

```
public interface Collection<E> extends Iterable<E> {
      // Basic operations
      int size();
      boolean isEmpty();
      boolean contains(Object element);
      boolean add(E element);                          //optional
      boolean remove(Object element);                  //optional
      Iterator<E> iterator();
      // Bulk operations (tömeges metódusok)
      boolean containsAll(Collection<?> c);
      boolean addAll(Collection<? extends E> c);       //optional
      boolean removeAll(Collection<?> c);              //optional
      boolean retainAll(Collection<?> c);              //optional
      void clear();                                    //optional
      // Array operations
      Object[] toArray();
      <T> T[] toArray(T[] a);
}
```

A deklarált metódusok három kategóriába sorolhatóak: alapvető metódusok, több elemet érintő, „tömeges" (*bulk*) metódusok és tömbökkel kapcsolatos metódusok. Az első kategóriába sorolhatjuk a *size* metódust, amelyet a gyűjtemény méretének lekérdezésére használhatunk, az *isEmpty* metódust, amelynek segítségével eldönthető, hogy a gyűjtemény tartalmaz-e elemeket, a *contains* metódust, amely egy paraméterként átadott *Object* típusú elemről eldönti, hogy része-e a gyűjteménynek. Ugyanebbe a kategóriába sorolhatjuk az *add* és *remove* metódusokat. Az *add* metódus egy *E* típusú elemet vár paraméterként, és hozzáadja ezt a gyűjteményhez, a *remove* metódus törli a paraméterként kapott *Object* típusú elemet a gyűjteményből. A művelet sikerességéről a visszatérített *boolean* érték ad információt (a metódusok *true* értéket térítenek vissza, ha a művelet végrehajtása sikeres volt, és ennek megfelelően a gyűjtemény tartalma változott). Látható, hogy a törlésnél nem fontos a paraméter típusa, mivel a gyűjtemény elemei biztosan az *Object* ősosztály leszármazottjai. A gyűjteményhez viszont nem tudunk hozzáadni bármilyen objektumot, csak a típusparaméternek megfelelő *E* típusú objektumokat. Az *iterator* metódus segítségével, egy a gyűjteménynek megfelelő *Iterator* objektumra mutató referenciát kérhetünk, és ennek segítségével végezhetjük el a gyűjtemény bejárását.

Megjegyzendő, hogy a metódusok esetében az „*optional*" megjegyzés nem azt jelenti, hogy ezeket a metódusokat a megvalósító osztályoknak nem kell implementálniuk. Természetesen, ha egy osztály „vállalja", hogy megvalósít egy interfészt, akkor annak minden metódusát

implementálnia kell. Az opcionális jelölés, csak arra vonatkozik, hogy az interfészt megvalósító osztályoktól nem kell feltétlenül elvárni, hogy „támogassák" az illető műveleteket, konkrét megvalósítást adjanak ezekre. Lehetséges, hogy egy olyan gyűjteményt szeretnénk létrehozni, amelyből például nem törölhetőek az elemek. Az osztályoknak megvan a lehetősége arra, hogy az opcionális metódusok törzsén belül egyszerűen kivételt dobjanak (pl. *UnsupportedOperationException*), vagy üresen hagyják valamelyik metódus törzsét. Az opcionális megjelölés még azt is jelenti, hogy a keretrendszeren belül is lesznek olyan megvalósítások, amelyek nem támogatják az illető műveleteket, vagy azoknak egy részét.

A második kategóriába tartoznak azok az opcionális műveletek, amelyek lehetővé teszik egyszerre több elem manipulálását. Ezek például halmazokkal kapcsolatos műveletek esetében lehetnek segítségünkre. A *containsAll* metódus segítségével lekérdezhetjük, hogy a gyűjtemény tartalmazza-e egy másik gyűjtemény minden elemét, az *addAll* metódus hozzáadja a gyűjteményhez egy másik gyűjtemény elemeit, a *removeAll* törli belőle azokat (amennyiben ezek az elemek a gyűjteményben is megtalálhatóak), a *retainAll* minden más elemet töröl. A *clear* metódus a gyűjtemény minden elemét törli, „kiürítve" azt. A hozzáadásra és törlésre használt metódusok egy *boolean* érték visszatérítésével szolgáltatnak információt arról, hogy sikeres volt-e az illető művelet. Érdemes továbbá megfigyelnünk a paraméterlistákban a jokerek használatát.

A harmadik kategóriát a tömbökkel kapcsolatos műveletek képezik, és két ilyen metódust láthatunk. Az első egy *Object* típusú tömbbe alakítja a gyűjtemény elemeit, és egy erre a tömbre mutató referenciát térít vissza. A második metódust az előző rész példái között már láthattuk. Megpróbálja a gyűjtemény elemeit belemásolni a paraméterként kapott *T* típusú tömbbe, és egy erre mutató referenciát térít vissza. Amennyiben a művelet nem lehetséges, mert a tömb mérete nem teszi lehetővé, a metódus létrehoz egy új *T* típusú tömböt, belemásolja az elemeket, és egy, erre az új tömbre mutató referenciát térít vissza. A tömb típusát futási időben fogja ellenőrizni, így fontos, hogy paraméterként is megkapja a tömbre mutató referenciát. A gyűjtemény elemeinek konkrét típusáról nem tudna futási időben használható információhoz jutni. Megjegyzendő továbbá, hogy a művelet természetesen csak akkor lehet sikeres, ha a gyűjtemény elemeinek típusa kompatibilis a tömb típusával, ellenkező esetben futási idejű kivételt kapunk (*ArrayStoreException*).

A gyűjtemények általános tulajdonsága, hogy a feldolgozás közben bizonyos műveletek kivételeket válthatnak ki. A leggyakrabban előforduló kivételtípusok a következők:

- *UnsupportedOperationException* – az opcionális metódusok esetében fordulhat elő, ha az adott megvalósítás nem támogatja az illető műveletet, és ennek megfelelően egyszerűen egy ilyen típusú kivételt dob a metóduson belül (erre az esetre vonatkozott az „*optional*" kommentár);
- *NullPointerException* – például akkor fordulhat elő, ha egy adott megvalósítás nem ad lehetőséget *null* elem beillesztésére;
- *IllegalArgumentException* – akkor fordulhat elő, amikor egy adott implementáció esetében egy elem valamilyen tulajdonsága nem teszi lehetővé egy művelet végrehajtását;
- *IllegalStateException* – az adott implementációnak megfelelően az illető művelet nem engedélyezett a gyűjtemény aktuális állapotában. Például előfordulhat, hogy egy szinkronizált gyűjtemény adott metódusa ilyen típusú kivételt dob, ha a gyűjteményobjektumnak megfelelő monitor egy másik végrehajtási szál által foglalt;

- *ClassCastException* – az elem konkrét típusa miatt nem történhet meg egy adott átalakítási művelet végrehajtása;
- *ArrayStoreException* – a típusok közötti inkompatibilitás miatt nem történhet meg a tömbbe alakítás.

A továbbiakban a különböző speciálisabb gyűjteménytípusoknak megfelelő interfészeket tárgyaljuk.

8.3.3 Halmazok

A *Collection* interfészből származik a *Set* interfész, amely halmaz típusú gyűjteményeknek felel meg. Az interfész a deklarált metódusok szempontjából teljesen azonos a *Collection* interfésszel. Különbség csak a megkötések szintjén van, amelyek a *Set* interfész esetében szigorúbbak. A legfontosabb megkötés, hogy az ilyen típusú gyűjtemények nem tartalmazhatnak duplikált elemeket. Természetesen interfészről lévén szó, a megkötések csak „szerződésként" értelmezhetőek, a megvalósító osztályoknak kell felelniük azért, hogy be is tartják ezeket. A JCF megvalósításai ennek megfelelnek.

A *Set* interfész kapcsán megemlíthetjük az alapvető halmazokkal kapcsolatos műveletek támogatását. Mint mondtuk, a több elem manipulálására lehetőséget adó metódusok kitűnően alkalmazhatóak erre a célra. Nézzünk néhány példát.

Halmazok egyesítése:

```
Set<Type> union = new HashSet<Type>(s1);
union.addAll(s2);
```

Halmazok metszete:

```
Set<Type> intersection = new HashSet<Type>(s1);
intersection.retainAll(s2);
```

Halmazok különbsége:

```
Set<Type> difference = new HashSet<Type>(s1);
difference.removeAll(s2);
```

A következő kis példaprogram a parancssor argumentumait különválogatja két halmazba, különválasztva a csak egyszer előforduló elemeket a többször előforduló elemektől:

```
import java.util.Set;
import java.util.HashSet;
public class FindDups {
      public static void main(String[] args) {
            Set<String> uniques = new HashSet<String>();
            Set<String> dups = new HashSet<String>();
            for (String a : args)
                  if (!uniques.add(a)) dups.add(a);
            uniques.removeAll(dups);
            System.out.println("Unique words: " + uniques);
            System.out.println("Duplicate words: " + dups);
      }
}
```

A program a *Set* interfész *HashSet* megvalósítását használja az elemek tárolására. Egy halmaz esetében az *add* metódus *false* értékkel tér vissza, amennyiben az illető elem már részét képezi a halmaznak, és így nem adható újra hozzá. Ezt a tényt használjuk fel az *if* feltételében. Először minden elemet megpróbálunk hozzáadni az *uniques* nevű halmazhoz. Amennyiben valamelyik esetében ez nem lehetséges (mert az elem már része a halmaznak), akkor az illető elemet a duplikált elemeket tartalmazó *dups* halmazhoz adjuk hozzá. Ezután a *removeAll* metódus segítségével töröljük a duplikált elemeket tartalmazó halmaz elemeit az *unique* halmazból (az első megjelenésükkor, a hozzáadás sikeres volt, így ebben a halmazban is megvan egy-egy példányuk), és végül kiírjuk a konzolra a halmazok elemeit.

8.3.4 Listák

Láthattuk, hogy a *Collection* interfészhez képest a *Set* interfész csak a megkötések szintjén jelent többet. A *List* interfész esetében már egy kicsit másabb a helyzet, az interfész több új metódust is deklarál. Ezeket soroljuk fel az alábbi példában:

```
public interface List<E> extends Collection<E> {
      // Positional access
      E get(int index);
      E set(int index, E element);                    //optional
      void add(int index, E element);                 //optional
      E remove(int index);                            //optional
      boolean addAll(int index,
      Collection<? extends E> c);                     //optional
      // Search
      int indexOf(Object o);
      int lastIndexOf(Object o);
      // Iteration
      ListIterator<E> listIterator();
      ListIterator<E> listIterator(int index);
      // Range-view
      List<E> subList(int from, int to);
}
```

A *List* interfészt megvalósító gyűjtemények rendezettnek tekinthetőek abból a szempontból, hogy a gyűjteményen belül minden elem meghatározott pozícióval rendelkezik, és így indexek segítségével hivatkozhatunk az elemekre. Ennek megfelelően, az interfész a *Collection* interfészhez képest néhány új metódust is biztosít. A *get* metódus visszatérít egy megadott indexű elemre mutató referenciát, a *set* metódus segítségével felülírhatunk egy adott pozíciót a metódus paramétereként átadott elemre. Az *add* metódusnak szintén van egy olyan változata, amely egy adott pozícióra szúr be egy elemet, és a *remove* metódusnak is van egy pozíció szerinti törlésnek megfelelő változata. Az *addAll* metódus esetében meghatározhatjuk, hogy a paraméterként kapott gyűjtemény elemeinek beszúrása milyen pozíciótól kezdődjön. Meg is kereshetjük egy adott elem pozícióját: az *indexOf* metódus az első előfordulás, a *lastIndexOf* metódus az utolsó előfordulás pozícióját téríti vissza. Ezen kívül a listának egy megadott részét is különválaszthatjuk: a *subList* metódus a paraméterként megadott pozíciók közötti elemeket tartalmazó listára mutató referenciát térít vissza.

A bejárás szempontjából is különbséget fedezhetünk fel. A *List* interfész két új metódust deklarál iterátorok visszatérítésére, és ezek esetében a visszatérített referencia már nem egyszerűen *Iterator*, hanem *ListIterator* típusú. A *ListIterator* interfész az *Iterator* interfész kiterjesztése, amely lista-specifikus metódusokat is deklarál:

```
public interface ListIterator<E> extends Iterator<E> {
      boolean hasNext();
      E next();
      boolean hasPrevious();
      E previous();
      int nextIndex();
      int previousIndex();
      void remove();                                    //optional
      void set(E e);                                    //optional
      void add(E e);                                    //optional
}
```

A *hasNext*, *next* és *remove* metódusok már ismerősek. A *ListIterator* lehetőséget ad fordított irányú bejárásra is, így egyes metódusoknak létezik egy ennek megfelelő változata is. A *hasPrevious* metódus ellenőrzi, hogy létezik-e előző elem, a *previous* metódus visszatéríti azt. Az elemek esetében a pozíció is lekérdezhető: a *nextIndex* metódus a következő elem, a *previousIndex* metódus az előző elem indexét téríti vissza. Ezeken kívül az interfész még deklarál két opcionális metódust: a *set* metódus segítségével felülírhatunk egy adott elemet, az *add* metódus segítségével beszúrhatunk egy elemet a listába.

A következő példa a fordított irányú bejárás lehetőségét szemlélteti:

```
for (ListIterator<Type> it = list.listIterator(list.size());
                                              it.hasPrevious();) {
            Type t = it.previous();
            ...
}
```

A *List* interfész *listIterator* metódusa a listának megfelelő *ListIterator* típusú objektumokra mutató referenciát térít vissza. A metódusnak van egy paraméteres változata is, amelynek segítségével meghatározható, hogy a bejárás melyik pozíciótól kezdődjön (ez a példa esetében a lista mérete, vagyis, mivel az indexek kezdőértéke a tömbökhöz hasonlóan 0, az „utolsó utáni" elem pozíciója).

8.3.5 Hasító táblázatok

A hasító táblázatok kulcs-elem párokat tároló adatszerkezetek, amelyek esetében a kulcsoknak egyedieknek kell lenniük, tehát kulcs-elem párok halmazáról beszélhetünk. Egy ilyen adatszerkezetnek megfeleltethetünk két gyűjteményt: a kulcsok halmazát, és az elemek gyűjteményét. A hasító táblázat típusú gyűjteményeknek megfelelő JCF interfész a *Map*:

```
public interface Map<K, V> {
      // Basic operations
      V put(K key, V value);
      V get(Object key);
```

```
        V remove(Object key);
        boolean containsKey(Object key);
        boolean containsValue(Object value);
        int size();
        boolean isEmpty();
        // Bulk operations
        void putAll(Map<? extends K, ? extends V> m);
        void clear();
        // Collection Views
        public Set<K> keySet();
        public Collection<V> values();
        public Set<Map.Entry<K, V>> entrySet();
        // Interface for entrySet elements
        public interface Entry {
              K getKey();
              V getValue();
              V setValue(V value);
        }
  }
```

A generikus interfész két típusparaméterrel rendelkezik: a *K* felel meg a kulcsok típusának, a *V* az értékek típusának. A táblázatba egy kulcs-elem pár beszúrása a *put* metódus segítségével történik. Az értékek beazonosítása az egyedi kulcsok segítségével lehetséges. Egy adott kulcsnak megfelelő érték kinyerésére a *get*, törlésére a *remove* metódus alkalmazható. A *containsKey* metódus segítségével lekérdezhető, hogy a táblázat tartalmaz-e egy adott kulcsot, a *containsValue* metódus segítségével, hogy tartalmaz-e egy adott értéket. A *size, isEmpty* és *clear* metódusok az előzőekben tárgyalt megfelelőikhez hasonlóan működnek. A *putAll* metódus egy paraméterként kapott hasító táblázat elemeit szúrja be az aktuális táblázatba. Ahhoz, hogy ez megtörténhessen, a kulcsok és értékek típusainak meg kell egyezniük, vagy a paraméterként kapott táblázat kulcsainak és elemeinek a *K*, illetve *V* típusok leszármazottjainak kell lenniük.

Az interfész lehetőséget ad arra is, hogy egy külön halmazt hozzunk létre a kulcsok tárolására, vagy egy külön gyűjteményt az értékek tárolására. Erre adnak lehetőséget a *keySet* és *values* metódusok. A kulcs-elem párok halmaza is lekérdezhető az *entrySet* metódus segítségével. Ebben az esetben egy olyan halmaz jön létre, amelynek elemei *Entry* típusúak. Minden elem egy kulcs-érték párnak felel meg, és a tárolásukra használt objektum osztályának meg kell valósítania az *Entry* interfészt (a *Map* interfész belső interfésze). Az interfész három metódust deklarál: az elemnek megfelelő kulcs, illetve érték lekérdezésére, valamint az érték felülírására.

Az eddig bemutatottakon kívül a keretrendszer még tartalmaz fontos interfészeket. Ezek egy része rendezett gyűjtemények létrehozására biztosít alapot. A következő részben bemutatunk néhányat ezek közül.

8.4 Rendezett gyűjtemények

A fejlesztések során sokszor előfordul, hogy olyan speciális gyűjtemények létrehozására van szükség, amelyek elemei valamilyen szempont szerint rendezettek. Ahhoz, hogy rendezni tudjuk egy gyűjtemény elemeit, természetesen azoknak összehasonlíthatóaknak kell lenniük. Említettük, hogy a gyűjtemény keretrendszer fontos részét képezik a gyűjtemények manipulálására alkalmazható algoritmusok. Ezek többnyire a *Collections* osztály statikus metódusain belül vannak implementálva, és a későbbiekben még szó esik róluk. Közülük most csak a rendezésre alkalmazható *sort* metódust emelnénk ki:

```
static <T extends Comparable<? super T>> void sort (List<T> list)
static <T> void sort(List<T> list, Comparator<? super T> c)
```

Amint azt láthatjuk, generikus metódusról van szó, amelynek két változata van. Mindkét változat a paraméterként kapott listát rendezi. Mivel a paraméter típusa *List*, annak bármilyen megvalósítását rendezhetjük, de vannak megkötések. Az első esetben a listának olyan elemeket kell tartalmaznia, amelyek összehasonlíthatóak, azaz a *Comparable* típus kiterjesztései. A második esetben át kell adnunk egy *Comparator* objektumot is paraméterként, és a rendezés ennek megfelelően történik majd. Az alábbiakban röviden bemutatjuk a *Comparable* és *Comparator* interfészeket.

8.4.1 A *Comparable* interfész

Az elemek akkor rendezhetőek, ha összehasonlíthatóak, azaz megvalósítják a *Comparable* interfészt:

```
public interface Comparable<T> {
      public int compareTo(T o);
}
```

Egy generikus interfészről van szó, amely egyetlen metódust deklarál. A *compareTo* metódus összehasonlítja az aktuális objektumot a paraméterként kapott objektummal, és egy egész értéket térít vissza. Ha az aktuális objektum kisebb, mint az argumentum által meghatározott, akkor negatív értéket, ha nagyobb, pozitív értéket térít vissza. Amennyiben az objektumok egyenlőek, a visszatérített érték 0.
Az interfész az elemek természetes rendezésének megvalósítását célozza, így a megvalósító osztályoknak be kell tartaniuk a megfelelő szabályokat. Ha *sgn*-el jelöljük az előjel (szignum) függvényt, amelynek értékészlete {-1, 0, 1}, akkor igaznak kell lennie, hogy

sgn(x.compareTo(y)) == -sgn(y.compareTo(x)).

A relációnak tranzitívnak kell lennie, tehát, ha

x.compareTo(y) > 0 && y.compareTo(z) > 0, akkor ***x.compareTo(z) > 0.***

A megvalósításnak azt is biztosítania kell, hogy, ha

x.compareTo(y) == 0, akkor ***sgn(x.compareTo(z)) == sgn(y.compareTo(z)).***

Ezen kívül ajánlott, de nem kötelező, hogy teljesüljön a

(x.compareTo(y) == 0) == x.equals(y)

feltétel is, és, amennyiben ezt a megvalósítás mégsem teljesíti, azt a specifikációban lehetőleg jeleznie kell.

Példaként készítsünk egy személyek adatainak tárolására alkalmazható *Person* osztályt, a vezetéknév és keresztnév attribútumokkal, és valósítsuk meg a *Comparable* interfészt, olyan módon, hogy lehetőséget adjunk a személyek névsor szerinti rendezésére:

```
public class Person implements Comparable<Person> {
      private String firstName;
      private String lastName;
      …
      public int compareTo(Person p) {
            int lastCmp = lastName.compareTo(p.lastName);
            return lastCmp!=0 ? lastCmp : firstName.compareTo(p.firstName);
      }
}
```

A *Person* osztály megvalósítja a *Comparable* interfészt, és ennek megfelelően implementálja a *compareTo* metódust. A személy objektumokat más személyekkel szeretnénk összehasonlítani, így az interfész típusparaméterének a *Person* konkrét típust feleltetjük meg, ennek megfelelően a *compareTo* metódus paramétere is *Person* típusú. A metódus első sorában összehasonlítjuk a vezetékneveket, és az eredményt a *lastCmp* változóban tároljuk. Az összehasonlítást a *String* osztály *compareTo* metódusának segítségével végezzük el. A *String* osztály szintén megvalósítja a *Comparable* interfészt, és az összehasonlítás az alfabetikus sorrend szerint történik. A második sorban megvizsgáljuk, hogy az eredményt tároló *lastCmp* értéke különbözik-e 0-tól. Amennyiben igen, az azt jelenti, hogy a vezetéknevek között különbség van, ezért azok sorrendje adja meg a személyek sorrendjét a névsorban, így a *lastCmp* értéke lesz a metódus visszatérítési értéke. Amennyiben a vezetéknevek azonosak, a keresztnevek sorrendje fog dönteni, ezért ebben az esetben azokra is meghívjuk a *String compareTo* metódusát, és ennek eredményét térítjük vissza.

Az ilyen módon megvalósított *Person* osztály példányai összehasonlíthatóak, és így egy személy objektumokat tartalmazó lista rendezhető a *Collections* osztály *sort* metódusával. Amennyiben a természetes sorrendtől (amely esetünkben a névsornak felel meg) eltérő rendezést szeretnénk, akkor más megoldást kell választanunk.

8.4.2 A *Comparator* interfész

Tételezzük fel, hogy a személy osztályból származtatunk egy alkalmazottakkal kapcsolatos információk tárolására alkalmas *Employee* osztályt. Az alkalmazottak is személyek, de az *Employee* osztály még kiterjesztheti a *Person* osztályt néhány specifikus, csak alkalmazottakra jellemző attribútummal és metódussal. Például, rögzíthetjük az alkalmazott felvételének időpontját egy *Date* típusú *hireDate* attribútumban. Ha rendezni akarjuk az alkalmazottak listáját, általában a névsor szerinti rendezés lesz a megfelelő, de megtörténhet, hogy másfajta rendezés is szükségessé válik. Például, tételezzük fel, hogy adott pillanatban szükséges lesz a régiség szerinti rendezés. Ebben az esetben a *Person* osztály természetes sorrend szerinti rendezést lehetővé tevő *compareTo* metódusa már nem lenne megfelelő az összehasonlításra. Megtehetnénk, hogy az *Employee* osztályban újradefiniáljuk ezt a metódust, de ebben az esetben nehézkessé tennénk az alkalmazottak névsor szerinti rendezését, és a megoldás azért sem lenne javasolt, mert nem oldaná meg azt a helyzetet, ha egy legközelebbi alkalommal például a fizetések

szerint szeretnénk sorba rendezni az alkalmazottakat. A megoldást a *Comparator* interfész megvalósítása jelentheti:

```
public interface Comparator<T> {
      int compare(T o1, T o2);
}
```

Hasonlóan a *Comparable* interfészhez, a *Comparator* interfész is generikus, és egyetlen metódust deklarál, amely a *compareTo* metódushoz hasonlóan működik, és azonos megkötéseknek kell megfelelnie. Ezúttal a metódus a *compare* nevet kapja, és két paraméterrel rendelkezik, az argumentumokat hasonlítja össze. Ha az első kisebb, mint a második, negatív, ellenkező esetben pozitív értéket térít vissza. Ha egyenlők, a visszatérített érték 0.

A *Comparator* interfészt megvalósító osztályok példányait olyan esetekben használhatjuk, amikor az elemek rendezését nem a természetes rendezés szerint szeretnénk megvalósítani, vagy amikor olyan elemeket akarunk rendezni, amelyek egyébként nem összehasonlítóak, a megfelelő osztály nem valósítja meg a *Comparable* interfészt. Az előbbiekben leírt példa esetében az jelentene megoldást, ha létrehoznánk egy olyan *Comparator* osztályt, amely az alkalmazottak régiségét figyelembe vevő rendezést biztosítana:

```
…
static final Comparator<Employee> SENIORITY_ORDER =
                        new Comparator<Employee>() {
      public int compare(Employee e1, Employee e2) {
            return e2.hireDate().compareTo(e1.hireDate());
      }
};
…
```

Az osztályt névnélküli belső osztályként hoztuk létre, és a *SENIORITY_ORDER* nevű, *Comparator* típusú, osztályszintű konstans attribútum az osztály egy példányára mutat. A generikus interfész típusparaméterét az *Employee* konkrét típussal helyettesítjük, mivel alkalmazottak összehasonlítását szeretnénk megvalósítani. Feltételezzük, hogy az alkalmazottak esetében a felvétel dátuma a *hireDate* metódussal kérdezhető le. A *compare* metódus két személy típusú paramétert kap. Az első referenciára meghívjuk a *hireDate* metódust, és a visszatérített referenciára a *Date* osztály *compareTo* metódusát, annak paraméterként átadva a második *Employee* típusú argumentumot. A *Date* osztály *compareTo* metódusa elvégzi a dátumok összehasonlítását, és a visszatérített érték lesz a mi *compare* metódusunk által visszatérített érték is. Ilyen módon lehetőséget teremtettünk arra, hogy egy alkalmazott lista régiség szerint is rendezhető legyen. Erre a *Collections* osztály *sort* metódusának második változatát alkalmazhatjuk:

```
List<Employee> e = new ArrayList<Employee>();
…     // inserting elemenets into e;
Collections.sort(e, SENIORITY_ORDER);
System.out.println(e);
```

A *Comparable* és *Comparator* interfészek segítségével osztályaink példányait összehasonlíthatóvá tehetjük, ilyen módon gyűjteményeinket rendezhetjük, és rendezett gyűjteménytípusokkal is dolgozhatunk. A JCF biztosít számunkra olyan interfészeket, amelyek rendezett gyűjtemények alapjául szolgálnak.

8.4.3 Várakozási sorok

A *Queue* interfész várakozási sor típusú gyűjteményeknek felel meg. Az ilyen gyűjteményeket tipikusan feldolgozásra váró elemek tárolására használjuk. A legtöbb megvalósítás a FIFO elvnek megfelelően működik, de ez nem kötelező, kivételek is vannak (pl. *PriorityQueue*). A kivételek esetében is érvényes viszont, hogy az elemek valamilyen összehasonlítási kritérium szerint rendezettek (például prioritási szint).

A *Queue* interfész a *Collection* interfészből származik, és néhány új metódust deklarál a sorokkal kapcsolatos speciálisabb műveletek támogatására:

```
public interface Queue<E> extends Collection<E> {
      E element();
      boolean offer(E e);
      E peek();
      E poll();
      E remove();
}
```

Az alkalmazott rendezéstől függetlenül a soroknak mindig van egy első eleme (*head*), és a *remove*, valamint *poll* metódusok a *head* elemnek a kinyerésére és törlésére szolgálnak. Ugyanígy beszélhetünk utolsó elemről is (*tail*). A FIFO elv szerint rendezett sorok esetében a beszúrás mindig a sor végére történik. A *remove* és *poll* metódusok annyiban különböznek egymástól, hogy míg a *remove* üres lista esetén kivételt dob, addig a *poll null* elemet térít vissza. Az elemek hozzáadása a sorhoz az *add* vagy *offer* metódusok segítségével történhet. A két metódus abban különbözik egymástól, hogy az *offer* egyszerűen *false* értéket térít vissza, ha a hozzáadás sikertelen, míg az *add* kivételt dobhat. Ennek megfelelően az *offer* metódust inkább olyan esetekben használjuk, amikor valószínűbb az, hogy nem lesz sikeres a hozzáadás (amikor ez inkább normális, és nem kivételes viselkedésnek számít), például korlátozott elemszámú sorok esetében (*bounded queue*). Az *element* és *peek* metódusok egyaránt a *head* elemre mutató referencia visszatérítésére szolgálnak, de a *remove*, vagy *poll* metódusokkal ellentétben ezek nem törlik az elemet a sorból. A különbség közöttük az, hogy míg a *peek null* értéket térít vissza üres lista esetében, az *element NoSuchElementException* típusú kivételt dob.

8.4.4 Rendezett halmazok és hasító táblázatok

Rendezett halmazok létrehozásának a *SortedSet* interfész lehet az alapja, amely a *Set* interfész leszármazottja. A *SortedSet* típusú halmazok rendezettek, és az interfész biztosít néhány speciálisabb metódust, amelyek ilyen halmazok esetében alkalmazhatóak:

```
public interface SortedSet<E> extends Set<E> {
      // Range-view
      SortedSet<E> subSet(E fromElement, E toElement);
      SortedSet<E> headSet(E toElement);
      SortedSet<E> tailSet(E fromElement);
      // Endpoints
```

```
        E first();
        E last();
        // Comparator access (returns null for natural order)
        Comparator<? super E> comparator();
}
```

A *subSet*, *headSet* és *tailSet* metódusok egy-egy részhalmazt térítenek vissza. A *headSet* a halmaz elején található elemek részhalmazát, egy adott elemig, a *tailSet* a halmaz végén található elemek részhalmazát egy adott elemtől. A *subSet* egy tetszőleges részhalmazt, egy adott elemtől egy másik megadott elemig. Rendezett halmazról lévén szó, lekérdezhetjük a „végpontokat", az első és utolsó elemet, a *first* és *last* metódusok segítségével. A *comparator* metódus az elemek rendezésére használt *Comparator* példányra mutató referenciát térít vissza, vagy *null* értéket, amennyiben a halmaz rendezése a természetes rendezési módnak megfelelően történt.

A *Map* interfész kiterjesztése, a *SortedMap* olyan hasító táblázatok létrehozásának alapját teremti meg, amelyek esetében a kulcsok rendezettek. A *SortedSet* interfészhez hasonlóan néhány speciális metódust biztosít:

```
public interface SortedMap<K, V> extends Map<K, V> {
      SortedMap<K, V> subMap(K fromKey, K toKey);
      SortedMap<K, V> headMap(K toKey);
      SortedMap<K, V> tailMap(K fromKey);
      K firstKey();
      K lastKey();
      Comparator<? super K> comparator();
}
```

Az első három metódus (*subMap, headMap, tailMap*) a *SortedSet* megfelelő metódusaihoz hasonlóan egy-egy „részтáblázat" visszatérítésére szolgál. A *firstKey* és *lastKey* metódusok az első, illetve utolsó kulcsot térítik vissza. Amennyiben a kulcsok rendezése nem a természetes rendezési mód szerint történt, a *comparator* metódus segítségével lekérdezhetjük a rendezéshez használt *Comparator* objektumra mutató referenciát.

A JCF fontosabb interfészeinek áttekintése után nézzük, ezeknek milyen megvalósításai állnak rendelkezésünkre a keretrendszeren belül.

8.5 JCF megvalósítások

A JCF interfészek keretrendszer által biztosított megvalósításait a következő kategóriákba sorolhatjuk:

- **absztrakt megvalósítások**: konkrét implementációt nem minden metódus esetében biztosító megvalósítások, absztrakt osztályok. Ezekből származnak a keretrendszer konkrét gyűjteménytípusoknak megfelelő megvalósításai, és ezen kívül alapul szolgálhatnak új megvalósítások létrehozásához;
- **általános célú megvalósítások**: ezekkel találkozunk leggyakrabban munkánk során. Alapvető, gyakran használt gyűjteménytípusoknak megfelelő osztályok;

- **speciális megvalósítások**: speciális feladatok megoldásánál használható, speciális funkcionalitásokat biztosító, vagy különböző megkötéseknek megfelelő gyűjtemények létrehozására alkalmazható osztályok;
- **konkurens megvalósítások**: több végrehajtási szálat használó alkalmazások számára tervezett, szinkronizált hozzáférést támogató gyűjteményeknek megfelelő osztályok (tulajdonképpen a speciális megvalósítások egy részhalmazának tekinthetőek);
- **burkoló (*wrapper*) megvalósítások**: más, többnyire általános célú megvalósításokkal együtt használhatóak. Ezekhez plusz funkcionalitásokat adnak hozzá, vagy korlátozásokat vezetnek be, és így speciális megvalósításokat (például szinkronizált hozzáférést támogató gyűjteményeket) kapunk eredményül;
- **kényelmi megvalósítások**: tipikusan gyártó metódusokon keresztül elérhető megvalósítások, amelyeket az általános célúak helyett alkalmazhatunk bizonyos speciális esetekben, így leegyszerűsítve, kényelmesebbé téve a munkánkat, és átláthatóbb kódot kapva eredményül.

A továbbiakban tekintsünk néhány példát ezekből a kategóriákból.

8.5.1 Absztrakt megvalósítások

Amint azt már írtuk, és a fejezet első részében található osztálydiagramok segítségével is szemléltettük, ezekből az osztályokból származnak a JCF különböző megvalósításai. A legtöbb interfésznek van ilyen absztrakt megvalósítása, így a következő absztrakt alaposztályokról beszélhetünk: *AbstractCollection, AbstractSet, AbstractList, AbstractSequentialList, AbstractQueue* és *AbstractMap*.

Ezeket abban az esetben is felhasználhatjuk, ha új, speciális gyűjteményosztályokat szeretnénk létrehozni. Példaként tekintsük a következő osztályt, amely segítségével nem módosítható listákat hozhatunk létre:

```
import java.util.AbstractList;
public class MyArrayList<T> extends AbstractList<T> {
      private final T[] a;
      MyArrayList(T[] array) {
            a = array;
      }
      public T get(int index) {
            return a[index];
      }
      public T set(int index, T element) {
            T oldValue = a[index];
            a[index] = element;
            return oldValue;
      }
      public int size() {
            return a.length;
      }
}
```

Létrehoztunk egy saját lista típusnak megfelelő *MyArrayList* osztályt, amely az elemeket egy tömbben tárolja, és a konstruktor paraméterként kapja meg a tárolandó tömböt. Az osztály az *AbstractList* leszármazottja, implementálja a *get* metódust (amely az *AbstractList* absztrakt metódusa), visszatérítve a tömb adott indexű elemét, és a *size* metódust (az *AbstractCollection* osztály absztrakt metódusa), visszatérítve a tömb méretét. Újradefiniáljuk a *set* metódust, amely ellenkező esetben, az *AbstractList* implementációja szerint egyszerűen kivételt (*UnsupportedOperationException*) dobna. Mivel az *add* és *remove* metódusokat nem definiáljuk újra, ezek meghívása egy *MyArrayList* objektum esetében egyszerűen kivételt dobna. Ettől függetlenül a listánk esetében működnének azok az alapvető, általános jellegű műveletek, amelyekre egy lista típusú gyűjtemény esetében szükségünk lehet (iterátorok kérése, pozíciók lekérdezése stb.), mivel ezekre a műveletekre az *AbstractList* osztály konkrét implementációt biztosít, és azokat osztályunk örökli tőle.

8.5.2 Általános célú megvalósítások

Az általános célú megvalósítások közül elsőként a *List* interfészt megvalósító *ArrayList*, illetve *LinkedList* osztályokat említhetjük. Az utóbbi a *Queue* interfészt is megvalósítja.

Az *ArrayList* típusú listák az elemeket változó méretű tömbben tárolják. Ez azt jelenti, hogy rendelkeznek egy olyan konstruktorral, amelynek segítségével megadhatunk egy kezdeti kapacitást. Amennyiben az alapértelmezett konstruktort alkalmazzuk, akkor egy, az alapértelmezett kapacitásnak megfelelő, 10 elemű tömb lesz létrehozva. Ha kifogytunk a helyből, és egy beillesztés már nem lenne lehetséges, az elemek rögzítésére használt tömböt a megvalósítás újraméretezi. Ez a gyakorlatban azt jelenti, hogy létrehoz egy, a régi tömbre mutató referenciát, majd létrehoz egy új tömböt megnövelt elemszámmal. Ha a régi méret n, akkor az új tömb mérete minimum $3n/2+1$ (vagy egy megadott, ennél nagyobb minimális méret). Ezután a régi tömb elemeit az új, megnövelt kapacitású tömbbe másolja. A másolásra az *Arrays* osztály *copyOf* nevű metódusának egyik változatát használja, amely a maga során a *System* osztály *arraycopy* nevű natív metódusát hívja meg. Maga a másolás tehát natív műveleten keresztül, viszonylag gyorsan végbemegy, de egyértelmű, hogy a művelet szükségessége így is a komplexitás növekedéséhez vezet. Ha viszont kezdőkapacitásnak nagy értéket határozunk meg, akkor fölöslegesen foglaljuk a memóriát. Éppen ezért, inkább olyankor érdemes *ArrayList* objektumokat használni, amikor legalább „van egy elképzelésünk" a lista méretéről, és így a konstruktorban egy megközelítőleg helyes kezdőkapacitást tudunk meghatározni, lecsökkentve az újraméretezési művelet szükségességének valószínűségét, de ugyanakkor elkerülve a fölösleges memóriafoglalást.

A *LinkedList* osztály az elemeket duplán láncolt listában tárolja. Az elemekhez történő hozzáférés szempontjából az *ArrayList* alkalmazása természetesen gyorsabb, mivel a tömbök esetében a hozzáférés konstans időben történik, míg a listák esetében lineáris. Ha viszont a listát többször teljes egészében be kell járnunk, és közben törlési vagy beillesztési műveleteket kell elvégeznünk, akkor a *LinkedList* lehet a jobb megoldás. A láncolt listák esetében az aktuális pozíciónak megfelelő elem törlése konstans idejű, és hasonlóan konstans időben történik egy új elem beillesztése is az aktuális pozícióba, míg a tömbök esetében ugyanezek a műveletek lineáris idejűek (az elemek elmozgatásával járnak). A *LinkedList* alkalmazása jobb ötlet lehet olyankor is, amikor „nincs elképzelésünk" a lista méretéről (az előbbiekben már magyaráztuk, hogy milyen hátrányokkal járhat ilyen esetben az *ArrayList* alkalmazása).

A *LinkedList* osztállyal kapcsolatban érdemes lehet még újra kihangsúlyozni, hogy megvalósítja a várakozási soroknak megfelelő *Queue* interfészt is, a FIFO elvnek megfelelő műveleteket téve lehetővé.

A hasító táblázatoknak megfelelő *Map* interfésznek általános célú megvalósítása a *HashMap* osztály, amely az *ArrayList*-hez hasonlóan változó méretű tömbök segítségével implementált. Ennek kiterjesztése a *LinkedHashMap*, amely nyilvántartja az elemek egy duplán láncolt listáját is (*LinkedList* példány segítségével), és ez a lista határozza meg az elemek bejárásának sorrendjét (általában a kulcsok beillesztésének sorrendje). A *Map* interfészből származtatott *SortedMap* interfész megvalósítása a *TreeMap*, rendezett táblázat, amely piros-fekete fa típusú adatszerkezetet használ az elemek tárolására.

Halmazok létrehozására a *Set* interfészt megvalósító *HashSet* osztályt használhatjuk, amely egy hasító táblázat alapú megvalósítás, az elemek tárolására *HashMap* példányt használ. Leszármazottja a *LinkedHashSet* osztály a *LinkedHashMap* osztályhoz hasonlóan nyilvántartja az elemeket egy duplán láncolt listában is, így a bejárás adott sorrendben (általában a beillesztés sorrendjében) történhet. A *Set* interfész leszármazottját, a *SortedSet* interfészt valósítja meg a rendezett halmazok létrehozására alkalmazható *TreeSet* osztály, amely *TreeMap* példányt használ az elemek tárolására.

A *Queue* interfésznek a *LinkedList* osztályon kívül talán a *PriorityQueue* osztály a leggyakrabban használt megvalósítása, amelyen belül az elemek érték szerint (tipikusan prioritási szint szerint) vannak rendezve.

Az általános célú implementációk közös tulajdonságai:

- megengedik *null* elemek beillesztését;
- megvalósítják a *Cloneable* és *Serializable* interfészeket;
- a műveletek nem szinkronizáltak.

Az utolsó tulajdonság egyfajta „szakítás a múlttal". Míg a JCF megjelenése előtt használt általános célú gyűjteményosztályok (*Hashtable, Vector*) szinkronizáltak voltak, és ennek megfelelően a keretrendszerhez „igazított", felújított változataik is azok, a JCF keretein belül a szinkronizált megvalósítások speciális implementációknak számítanak. A *java.util.concurrent* csomag biztosít megfelelő interfészeket és osztályokat ilyen gyűjtemények létrehozására. Ezen kívül az általános célú megvalósítások példányai szinkronizált gyűjteményekbe alakíthatóak megfelelő burkoló (*wrapper*) megvalósítások segítségével.

Az általános célú megvalósítások közül egy konkrét gyűjteménytípus kiválasztásának mindig az adott feladatnak megfelelően, az egyes típusok tulajdonságait figyelembe véve kell megtörténnie. Ebben a tekintetben, az előbbiekben leírt *ArrayList* és *LinkedList* összehasonlítás szolgáltathat egy alapvető példát. Hasonlóan, például csak akkor érdemes szinkronizált megvalósításokat alkalmazni (ezzel csökkentve az általánosságot, és növelve a komplexitást), ha arra valóban szükség van. A legfontosabb, hogy amennyiben lehetséges interfészeket használjunk, „interfészekben gondolkodjunk". Ne vezessünk be szükségtelen megkötéseket, törekedjünk az általánosságra.

8.5.3 Burkoló megvalósítások

A burkoló (*wrapper*) megvalósítások a *Decorator* tervezési minta alapján vannak implementálva. Az alapvető feldolgozási műveleteket más megvalósításokhoz, általában általános célú implementációkhoz továbbítják, de ezeket plusz funkcionalitással egészítik ki, és így gyakorlatilag speciális megvalósításoknak felelnek meg.

Tipikus plusz funkcionalitások: szinkronizált hozzáférés, módosítások letiltása, típusellenőrzés (a statikus típusellenőrzések megkerülésének megelőzése, például régi stílusú, generikus típusokat nem alkalmazó kódok esetén), stb. A burkoló megvalósítások létrehozása általában gyártó metódusok (*Factory* minta) segítségével történik. Példaként tekintsük a *Collections* osztály szinkronizált gyűjtemények létrehozására használható gyártó metódusait:

```
public static <T> Collection<T>
                    synchronizedCollection(Collection<T> c);
public static <T> Set<T> synchronizedSet(Set<T> s);
public static <T> List<T> synchronizedList(List<T> list);
public static <K, V> Map<K, V> synchronizedMap(Map<K, V> m);
public static <T> SortedSet<T> synchronizedSortedSet(SortedSet<T> s);
public static <K, V> SortedMap<K, V>
                    synchronizedSortedMap(SortedMap<K, V> m);
```

Láthatjuk, hogy a metódusok paraméterként kapnak egy gyűjtemény típusú referenciát (interfészek alkalmazása), és az illető gyűjteménynek egy szinkronizált változatára mutató referenciát térítenek vissza. Használatukkal bármilyen (például egy általános célú) gyűjtemény burkolható olyan módon, hogy eredményül annak szinkronizált hozzáférést támogató változatát kapjuk:

```
List<Type> list = Collections.synchronizedList(new ArrayList<Type>());
```

Megjegyzendő, hogy az iterálás szinkronizálását „manuálisan" kell biztosítani, ha azt szeretnénk, hogy több lépés egyetlen atomi műveletként legyen végrehajtva:

```
Collection<Type> c = Collections.synchronizedCollection(myCollection);
synchronized (c) {
      for (Type e : c) someMethod(e);
}
```

A szinkronizált változatokon kívül más típusú burkoló megvalósítások is léteznek, és a *Collections* osztály biztosít más gyártó metódusokat is, például nem módosítható gyűjtemények (pl. *unmodifiableCollection*, stb.), és korlátozásokat, megkötéseket bevezető (pl. *singletonList*) gyűjteményeknek megfelelő burkolt példányok létrehozására.

8.5.4 Speciális megvalósítások

A JCF több speciális gyűjteménytípusnak megfelelő osztályt is biztosít számunkra. Megemlíthető például a *Set* interfészt megvalósító *EnumSet*, amely *Enum* típusú elemek tárolására alkalmas. Az elemek belső reprezentációja egy bitsorozat segítségével történik, így a megoldás kompakt tárolásra és hatékony feldolgozásra ad lehetőséget. Kisméretű gyűjteményeket és több végrehajtási szálat használó alkalmazások esetében hasznosak lehetnek

a *CopyOnWriteArraySet* és *CopyOnWriteArrayList* osztályok (a *java.util.concurrent* csomagban találhatóak). Ezeknek esetében nincsen szükség szinkronizálásra, és mégis biztonságosan dolgozhatunk velük, mert minden írási művelet esetében az eredeti gyűjteményről egy másolat készül. Természetesen nagyobb méretű gyűjtemények esetében használatuk nem tanácsos.

A hasító táblázatok esetében a speciális megvalósítások közül talán az *IdentityHashMap* osztály a legközismertebb. A kulcsok (és értékek) összehasonlításának esetében az általános célú implementációk az objektumok tartalmának egyezését vizsgálják (*equals*), ezzel szemben ez a speciális megvalósítás a referenciák azonosságát (==) ellenőrzi. Megemlíthető még az *EnumSet*-hez hasonlóan *Enum* típusú elemek tárolására alkalmas, kompakt tárolási módot és hatékony hozzáférést biztosító *EnumMap* osztály. Bizonyos esetekben hasznos lehet a *WeakHashMap* osztály, amelynek esetében egy kulcs-érték megfeleltetés létezése nem gátolja meg a Java szemétgyűjtőjét a kulcs objektum által foglalt memória felszabadításában. Gyakorlatilag, amennyiben nincsenek a kulcsra mutató külső referenciák, a szemétgyűjtő törölheti a tárolt elemet (kulcs-elem pár), felszabadítva a memóriát (a megvalósításban a *WeakReference* osztály játszik szerepet).

A speciális megvalósítások közé sorolhatjuk a szinkronizált feldolgozást támogató osztályokat is. Ezek részére a *java.util.concurrent* csomag biztosít interfészeket, és a megvalósítások is itt találhatóak. Megemlíthető a *Map* interfészből származtatott *ConcurrentMap* interfészt megvalósító *ConcurrentHashMap* osztály, és a *Queue* interfészből származtatott *BlockingQueue* interfészt megvalósító osztályok (*LinkedBlockingQueue, ArrayBlockingQueue, PriorityBlockingQueue* stb.). Természetesen, az esetek többségében a szinkronizált gyűjtemények létrehozására a burkoló megvalósításokat és a *Collections* osztály gyártó metódusait alkalmazzuk, vagy a *Vector* és *Hashtable* szinkronizált gyűjteményeket használjuk.

8.5.5 Kényelmi megvalósítások

A kényelmi (*convenience*) megvalósítások néhány speciális feladat esetében egyszerű, elegáns, kompakt írásmódot tesznek lehetővé. Tipikusan gyártó metódusokon keresztül érhetőek el. Tekintsünk néhány példát.

Az *Arrays* osztály *asList* metódusának segítségével létrehozhatjuk egy tömb lista (*List*) típusú megfelelőjét:

```
List<String> list = Arrays.asList(new String[size]);
```

A *Collections* osztály *nCopies* metódusával létrehozhatunk több azonos elemet tartalmazó listát. Például egy 1000 darab *null* elemet tartalmazó listát az alábbi metódushívással hozhatunk létre:

```
List<Type> list = new ArrayList<Type>(
                      Collections.nCopies(1000, (Type) null));
```

Ha egy olyan gyűjteményt szeretnénk létrehozni, amely csak egyetlen meghatározott elemet tartalmaz, a *Collection* osztály *singleton* gyártómetódusát alkalmazhatjuk. A megoldás több esetben hasznos lehet, például, ha egy gyűjteményből el szeretnénk távolítani egy adott elem minden előfordulását:

```
myCollection.removeAll(Collections.singleton(objectToRemove));
```

Vannak esetek, amikor egy metódus gyűjteményt vár paraméterként, de „nincs mit átadnunk neki”, és ilyenkor hasznos lehet a *Collections* osztály üres halmazt „legyártó” *emptySet* metódusa:

```
myObject.myMethod(Collections.emptySet());
```

Láthatjuk tehát, hogy a kényelem, a kompakt kód és az áttekinthetőség szempontjából több esetben hasznosak lehetnek a JCF által biztosított kényelmi megvalósítások.

8.6 Algoritmusok

A JCF biztosítja számunkra a gyűjteményekkel kapcsolatos közismert, gyakran használt algoritmusok megvalósításait. Ezeket a *Collections* osztály statikus metódusainak meghívásával vehetjük igénybe. Az alábbiakban tekintsünk ezek közül néhány hasznos metódust:

```
public static void shuffle(List<?> l);
public static void sort(List<T> l);                        //merge sort
public static void sort(List<T> l, Comparator<? super T> c);
public static void swap(List<?> l, int i, int j);
public static void rotate(List<?> l, int distance);
public static void reverse(List<?> l);
public static <T extends Object & Comparable<? super T>> T
                                        min(Collection<? extends T> c);
public static <T extends Object & Comparable<? super T>> T
                                        max(Collection<? extends T> c);
public static <T> int binarySearch(List<? extends Comparable<? super T>>
                                                list, T key);
public static <T> void copy(List<? super T> dest,
                                                List<? extends T> src);
public static <T> void fill(List<? super T> list, T obj);
```

A *shuffle* metódus segítségével véletlenszerűen összekeverhetjük egy lista elemeit. A metódusnak létezik egy olyan változata is, amely esetében második paraméterként egy véletlenszám-generátor objektumra (*Random*) mutató referencia is átadható, és a „keverés” ennek megfelelően történik. A *sort* metódusokról már szó esett, ezek listák rendezésénél alkalmazhatóak, az összefésülő rendezési *(merge sort*) algoritmust implementálják. A *swap* metódus két elem pozícióját cseréli fel egy listában. A *rotate* metódus „elforgatja” a listát egy meghatározott lépésszámmal. Minden lépésben az aktuális lista utolsó eleme az első helyre kerül, és a többi elem egy pozícióval jobbra tolódik. A *reverse* metódus megfordítja a lista elemeinek sorrendjét. A *min* és *max* metódusok a legkisebb és legnagyobb elem visszatérítésére szolgálnak. Mindkettőnek van olyan változata is, amely második paraméterként egy *Comparator* objektumot is kaphat, ennek megfelelően végezve el az összehasonlításokat. A *binarySearch* a bináris keresési algoritmus megvalósítása, szintén rendelkezik egy *Comparator* típusú második paraméter megadását is lehetővé tevő változattal. A *copy* metódus egy lista elemeit egy másik listába másolja, a *fill* metódus feltölt egy adott listát a paraméterként megadott elemmel.

A felsorolás a teljesség igénye nélkül történt. Ha megnézzük a *Collections* osztály specifikációját, meggyőződhetünk arról, hogy még számos hasznos metódus (elemek előfordulásának számolása, fordított rendezést lehetővé tevő *Comparator* objektum kérése, különböző gyártó metódusok stb.) áll rendelkezésünkre. Remélhetőleg, a fejezetben tárgyalt fogalmak megismerése után a gyűjtemények, és a megfelelő algoritmusok alkalmazásának szükségessége már nem jelenthet problémát alkalmazásaink elkészítésénél.

8.7 Javasolt gyakorlatok

1. Egy szöveges állomány soraiban személyek (*Person* objektumok) adatait rögzítjük (név, életkor). Készítsünk egy osztályt (pl. *DataManagerImpl* névvel) az adatok beolvasására. Az osztály tartalmaz egy metódust, amelyik beolvassa az állományból az adatokat, azok alapján felépít egy személy objektumokból álló listát, és visszatérít egy, a listára mutató referenciát. Ugyanakkor az osztály a mentésért is felelős, tartalmaz egy metódust, amely paraméterként egy lista típusú referenciát kap, és a lista elemeit kiírja az állományba. Az állomány nevét és elérési útvonalát egy *.properties* állományból olvassuk be.

   Osztályunk egy megadott interfész (pl. *DataManager*) megvalósítása legyen. Az interfész deklarálja az előzőekben említett olvasásra és írásra alkalmas metódusokat. A metódusok visszatérítési értékének, illetve paraméterének esetében a lista típusa interfész segítségével (*List<Person>*) van meghatározva. A megvalósító osztályon belül a lista létrehozására tetszőleges implementációt alkalmazhatunk (pl. *ArrayList, LinkedList*).

   A SWING eszköztár segítségével készítsünk egy grafikus felületet, amelyik az interfészen keresztül kommunikál az adatok beolvasását és mentését végző objektummal. Ezt úgy valósíthatjuk meg, hogy a felületünknek megfelelő osztály konstruktora paraméterként kap egy *DataManager* típusú referenciát. Az interfész megfelelő metódusának segítségével olvassuk be az adatokat, és jelenítsük meg azokat a felületen (erre a célra pl. *JTable* komponenst használhatunk). Adjunk lehetőséget az adatok módosítására, és a módosított adatok mentésére (szintén az interfész megfelelő metódusának segítségével).

   Egy központi vezérlő osztályban hozzunk létre egy *DataManager* objektumot (példányosítva a *DataManagerImpl* osztályból), majd ennek segítségével egy grafikus felületet.

   Figyeljük meg, hogy a felületünk ilyen módon független a *DataManager* implementációtól. Ezt ki is próbálhatjuk olyan módon, hogy egy második megvalósítást is készítünk, amelyen belül más típusú gyűjteménnyel dolgozunk.
2. A JCF általános célú megvalósításai közül tetszőleges módon válasszunk ki két (vagy több) implementációt, amelyek összehasonlítása releváns és érdekes lehet (pl. *ArrayList, LinkedList*). Hasonlítsuk össze ezeket, kiválasztva egy-két olyan műveletet, amelyek szempontjából az összehasonlítás eltéréseket mutathat (pl. beszúrás/törlés a lista elejéről/végéről/közepéről), majd többször végrehajtva ezeket a műveleteket különböző méretű gyűjteményekre (pl. 100-100 végrehajtás, 50, …, 50000, …, 5000000, … elemű gyűjteményekre) rögzítsük a végrehajtási idők átlagát. A végrehajtási idők rögzítését végezhetjük olyan módon, hogy a művelet végrehajtása előtt és után is lekérjük az aktuális időt (pl. a *java.util.Date* osztály felhasználásával), majd kiszámoljuk az ezek közötti időintervallumot. A rögzített adatokat grafikusan ábrázolhatjuk, tetszőleges grafikonszerkesztő program segítségével.

9

Kiegészítések

A jegyzet előző fejezeteiben egyes témákat csak nagyon röviden tárgyal. A szöveg folytonosságának megőrzését célozva, néhány téma esetében nem ment bele részletekbe, csak a gyakorlati tevékenység megalapozására szolgáló alapfogalmakat mutatta be. Természetesen, egy ilyen rövid jegyzeten belül semmiképpen nem lenne lehetőség az összes részletre kitérni. Napjainkban a Java a szoftverfejlesztés majdnem minden területén jelen van, a kapcsolódó programozási felületek és technológiák száma óriási. Már az alap Java platform részletes bemutatása is egy jóval terjedelmesebb könyvet igényelne, de a jelen jegyzetnek nem is ez volt a célja. Csak az „alapok alapjainak" áttekintését célozta, megpróbálva ezeket egy olyan formában összegezni, hogy egyrészt alapot biztosítson a gyakorlati munka elkezdéséhez, másrészt megmutassa a továbbtanulási lehetőségeket, irányvonalakat. Néhány felszínesebben bemutatott téma esetében mégis szükséges lehet egy-két kiegészítés leírása. Ilyen kiegészítések kapnak helyet ebben a fejezetben.

Már a jegyzet elején hangsúlyoztuk, hogy a kivételkezelésnek [18] Java-ban rendkívül fontos szerepe van. Mégis, csak egy nagyon rövid részben volt lehetőségünk a kapcsolódó fogalmakkal foglalkozni, csak az első feladatok megoldásához szükséges alapokat ismertetve. A későbbiekben még találkoztunk néhány példával, de különösebb magyarázatok nélkül. Ezért szükséges lehet talán ennek a résznek a kicsivel részletesebb újratárgyalása.

Hasonló a helyzet az eseménykezeléssel is. Bár említettük, hogy az eseménykezelés, és általánosabban az *Observer* tervezési minta, a megfigyelt-megfigyelő modell egy általánosan alkalmazható módszer, amely nagyon sok helyzetben szolgáltathat számunkra kényelmes és hatékony megoldásokat, a jegyzet előző részeiben kizárólag a grafikus felhasználói felületekkel kapcsolatban beszéltünk erről a témáról. A minta általánosságát talán érdemes lenne alátámasztani konkrét példákkal. Egy ilyen példa kap helyet a fejezet második felében.

Az előző részekben láthattuk, hogy az alap Java fejlesztési platform nagyon sok eszközt, több csomagot és ezeken belül számos hasznos interfészt és osztályt biztosít, megkönnyítve ezzel fejlesztői munkánkat. Példáinkban több interfészt és osztályt fel is használtunk, de nagyon sok alapvető, fontos eszköz és lehetőség kimaradt. Természetesen, ebben a fejezetben sem lenne lehetőség mindegyikre részletesen kitérni. Hasznos lehet viszont a teljesség igénye nélkül a csomaghierarchiát bemutatni, és néhány fontosabb interfészt és osztályt megemlíteni, ezzel egy alapvető útmutatást adva az olvasónak a platform dokumentációjának további tanulmányozásához. Ez a rövid áttekintés kap helyet a fejezet harmadik részében.

Talán érdemes újra kihangsúlyoznunk, hogy a jelen fejezet összeállítása a teljesség igénye nélkül történt. Hosszasan lehetne értekezni még például a jegyzetben nagyon röviden tárgyalt szerializálási mechanizmus hátteréről, a párhuzamos programozást támogató lehetőségekről, és így tovább. Ugyanakkor, ahogyan azt már említettük, az alapvető nyelvi elemek részletes bemutatása sem képezte a jegyzet célját, nem is beszélve az alap platform részét képező haladóbb fejlesztési módszereket támogató csomagokról, vagy a kapcsolódó programozási

interfészekről, keretrendszerekről, technológiákról. Szerencsére a nyomtatott és internetes források, dokumentációk száma óriási, így az olvasónak remélhetőleg könnyű dolga lesz az ismeretek elmélyítésének tekintetében, és remélhetőleg ehhez a jelen jegyzet áttanulmányozása is hozzájárul.

9.1 Kivételkezelésről, részletesebben

A jegyzet első fejezetében már röviden ismertettük a kivételkezelés alapjait, és a későbbi példák esetében alkalmaztuk is ezeket az ismereteket. Néhány részlet és magyarázat azonban kimaradt, itt ezeket próbáljuk meg áttekinteni.

A kivételeknek megfelelő Java osztályhierarchia csúcsán a *Throwable* osztály található, minden kivételosztály valamilyen ágon ennek az osztálynak leszármazottja. A *Throwable* objektumok a kivételek közös tulajdonságainak rögzítésére és lekérdezésére szolgálnak. Minden objektum tartalmaz egy „pillanatképet" a neki megfelelő végrehajtási szálhoz tartozó hívási veremről, amelyet létrejöttének pillanatában rögzít. Ezek az információk a *getStackTrace* metódus segítségével lekérdezhetőek (vagy a *printStackTrace* metódusok segítségével kiírhatóak). A *getStackTrace* metódus egy *StackTraceElement* típusú elemekből álló tömböt térít vissza. Ezektől az objektumoktól különböző információkat kérdezhetünk le: a metódus és a megfelelő osztály nevét, a forráskódot tartalmazó állomány nevét, a forráskód megfelelő sorának sorszámát stb. Ezeken az információkon kívül a kivételobjektum tartalmazhat szöveges információkat az illető kivételről. Néhány esetben a kivétel egy másik kivételnek a következménye (láncolt kivételek, *chained exceptions*), és ezekben az esetekben a kivétel „oka" (a másik kivételobjektumra mutató referencia) lekérdezhető a *Throwable* osztály *getCause* metódusával.

Java-ban a kivételeknek három kategóriáját különítjük el: ellenőrzött kivételek (*checked exceptions*), futási idejű kivételek (*runtime exceptions*) és hibák (*errors*). Az utóbbi két kategóriát együttesen nem ellenőrzött kivételeknek (*unchecked exceptions*) is nevezik.

Az első kategóriába azok a kivételtípusok tartoznak, amelyek olyan kivételes körülmények között léphetnek fel, amelyek előfordulása előrelátható, így egy alkalmazástól elvárhatjuk, hogy a megfelelő módon kezelje ezeket. Példaként gondolhatunk az állománykezeléssel kapcsolatos műveletekre, amikor első lépésként a felhasználótól egy állomány nevét várjuk. Valószínűleg a legtöbb esetben a felhasználó egy létező és hozzáférhető állomány nevét adja meg, de könnyen előfordulhat, hogy ez mégsem így történik. A hibalehetőségre számíthatunk, így elvárható, hogy a kivételes helyzetet a megfelelő módon kezeljük. Ennek megfelelően a rendszer is ezt várja el tőlünk, következésképpen az olyan metódusok esetében, amelyek meghívása ellenőrzött kivételeket eredményezhet (pl. olvasás egy állományból, amely *FileNotFoundException*, vagy *IOException* típusú kivételekhez vezethet) kötelező módon kezelnünk kell ezeket a kivételeket. Erre két lehetőség van: a metódust egy *try-catch* szerkezeten belül hívjuk meg, és helyben kezeljük a kivételeket, vagy a meghívó metódusból tovább „dobjuk" az illető kivételeket (ezt az illető metódus fejlécében is jelezve), és az alkalmazás egy következő szintjén kezeljük ezeket egy *try-catch* szerkezeten belül. A két lehetőség valamelyikét alkalmaznunk kell (és következésképpen valahol egy *try-catch* szerkezetet kell használnunk), ellenkező esetben fordítási idejű hibát kapunk, a programunk nem lesz lefordítható.

Az ellenőrzött kivételek a *Throwable* osztályból származó *Exception* ősosztály leszármazottjai. A legtöbb kivételosztály ebbe a kategóriába tartozik.

A második kategóriába tartozó futási idejű kivételek szintén az alkalmazással kapcsolatos belső körülmények eredményei, kivételes esetekben léphetnek fel, de ezek az esetek nem, vagy nehezebben előreláthatóak, így kezelésük nem minden esetben lehetséges. Sokszor az ilyen kivételek programozási hibák, *bug*-ok eredményei, és a helyes megoldás ezeknek a hibáknak a kiszűrése. Példaként gondoljunk arra az esetre, amikor az előbb említett állománykezeléssel kapcsolatos műveletek meghívásakor egy *null* értékű állománynevet adunk át paraméterként, és így *NullPointerException* típusú kivételt kapunk. Az alkalmazáson belül lehetőségünk van kezelni az illető kivételt, de a legtöbb ilyen esetben biztosabb megoldást jelenthet a hiba forrásának kiküszöbölése. Hasonló helyzetekkel találkozhatunk például a szöveges adatok numerikus értékekbe történő átalakításakor is, vagy amikor közvetlen módon hivatkozunk egy tömb valamelyik elemére, anélkül, hogy ellenőriznénk a tömbhatárok túllépését (lehetséges, hogy *ArrayIndexOutOfBounds* típusú kivételt kapunk), és így tovább. Természetesen, a rendszer nem várhatja el tőlünk, hogy minden olyan művelet esetében, amely potenciálisan futási idejű kivételt eredményez, kezeljük a lehetséges kivételeket. Ez például azt is jelentené, hogy *try-catch* szerkezetet kellene alkalmaznunk minden olyan esetben, amikor egy tömb elemeire hivatkozunk. Azokat a metódusokat amelyek futási idejű kivételek forrásai lehetnek, a kivételek kezelése nélkül is meghívhatjuk, nem kapunk fordítási hibát. Következményként az ilyen jellegű kivételes helyzetek nehezebben azonosíthatóak, így potenciálisan futási idejű hibák forrásai lehetnek, és erre oda kell figyelnünk.

A futási idejű kivételeknek megfelelő osztályok az *Exception* osztályból származtatott *RuntimeException* ősosztály leszármazottjai.

A harmadik kategóriát a hibák képezik, amelyek az alkalmazástól független külső körülmények hatására léphetnek fel, előrejelzésük nem lehetséges, és általában komoly következményekkel járnak. Ezeknek a hibáknak a kezelése általában értelmetlen, az alkalmazások egy-egy ilyen típusú hiba fellépésekor olyan állapotba kerülnek, amelyből már nem térhetnek vissza a normál működéshez, és ezért általában a program lezárása lehet a helyes megoldás. Ilyen eset lehet például, amikor a virtuális gép meghibásodásáról, vagy az erőforráskeret túllépéséről van szó, és ez *VirtualMachineError* típusú hibát eredményez, vagy amikor egy szükséges osztály definíciója nem betölthető és *NoClassDefFoundError* típusú hibát kapunk, stb. Szerencsére, az ilyen jellegű, komolyabb problémákat okozó hibatípusoknak a száma a másik két kategóriához viszonyítva kicsi.

A hiba típusú kivételeknek megfelelő osztályok a *Throwable* osztályból származtatott *Error* ősosztály leszármazottjai.

A kivételek kezelésére szolgáló alapmechanizmusról, a *try-catch-finally* szerkezet alkalmazásáról már az előbbiekben szóltunk. Azt is említettük, hogy az ellenőrzött kivételek esetében, amennyiben a kivétel kezelése nem történik meg helyben, a problémás kódot tartalmazó metódus fejlécében jeleznünk kell, hogy az illető metódus kivételt dobhat. Ilyen metódusokkal már találkoztunk az előző példáink esetében, de ismétlésként tekintsünk itt is egy példát. Tételezzük fel, hogy egy állományokkal kapcsolatos műveleteket végrehajtó metódus esetében nem akarjuk helyben kezelni a lehetséges kivételeket, hanem továbbítani szeretnénk ezeket a program egy másik részéhez:

```
public void readFile(String filename)
            throws FileNotFoundException, IOException {
      FileReader fr = new FileReader(filename);
      BufferedReader br = new BufferedReader(fr);
```

```
        String line;
        while ((line = br.readLine()) != null)
              System.out.println(line);
  }
```

A példánk esetében a metódus egy szöveges állomány sorait írja ki a konzolra. Az állomány nevét paraméterként adjuk át a metódusnak. Ilyen jellegű példákkal már találkoztunk a jegyzet adatfolyamokkal kapcsolatos részénél, és láthattuk, hogy az állományokkal kapcsolatos műveletek végrehajtását *try-catch* konstrukción belül kell elvégeznünk, mivel különböző kivételek léphetnek fel. Előfordulhat, hogy az illető állomány nem létezik, és *FileNotFoundException* típusú kivételt kapunk, vagy írási/olvasási hiba léphet fel, és ekkor *IOException* típusú kivételt kapunk. Ez érvényes a fenti példánk esetében is, de úgy dönthetünk, hogy ezeket a kivételeket nem kezeljük helyben, hanem továbbítjuk a rendszer más részeihez, azokhoz, amelyek a *readFile* metódusunkat használni fogják. Ebben az esetben a metódus fejlécében jeleznünk kell, hogy a metódus meghívása kivételt eredményezhet, a metódus „kivételt dobhat". Ezt a *throws* kulcsszó segítségével tehetjük meg, felsorolva (egymástól vesszővel elválasztva) a lehetséges kivételtípusokat. Ezeket a metódust meghívó kódon belül egy *try-catch* szerkezet segítségével kötelező módon kezelnünk kell, vagy hasonló módon továbbítanunk az alkalmazás egy felsőbb részéhez. Természetesen ez csak az ellenőrzött kivételekre vonatkozik, a nem ellenőrzött kivételek lehetséges előfordulásait nem kell jeleznünk az érintett metódusok fejlécében, és a metódusokat meghívó komponensek esetében sem feltétlenül szükséges ezek kezelése.

Az alkalmazások komponensei általában több rétegbe szerveződnek. Az általánosan elfogadott minta szerint a kivételeket az alsóbb rétegektől a felsőbb rétegek felé kell továbbítani, és a lekezelés tipikusan a legfelső, általában a kliensalkalmazás grafikus felhasználói felületének megfelelő prezentációs rétegen belül történik. A javaslat alátámasztására tekinthetünk egy egyszerű példát. Tételezzük fel, hogy egy adatkezelésre alkalmas, kliens-szerver architektúrára épülő többrétegű alkalmazást szeretnénk készíteni. Az adatokat a szerveren tároljuk, például egy relációs adatbázisban. A szerver oldalon található adathozzáférési réteg felelős az adatok eléréséért. Ezzel a réteggel a közvetlenül fölötte elhelyezkedő, szintén a szerver oldalon található szolgáltatási réteg kommunikál a megfelelő interfészeken keresztül. A kliens oldalon a felhasználók egy grafikus kezelői felület segítségével kérhetik bizonyos műveletek végrehajtását, és ezek a kérések a szolgáltatási réteg megfelelő interfészein keresztül továbbítódnak a szerver oldali komponensekhez. Ha egy kivétel lép fel a szerveroldalon, például az adatok elérése közben, akkor nem lenne jó megoldás azt csak helyben kezelni. Itt legfeljebb a hibák naplózására (*logging*) lenne lehetőségünk, a felhasználóhoz viszont így nem jut el visszajelzés (hiába írunk ki például egy üzenetet a konzolra, mert a szerver oldali konzolt a felhasználó nem láthatja). A megoldás az lehet, ha az illető hibát továbbítjuk a felsőbb rétegekhez, így esetünkben a kivétel az adathozzáférési rétegtől továbbítódna a szolgáltatási réteghez, majd innen a kliens oldali prezentációs réteghez. A legfelső kliens oldali rétegen belül megtörténhetne a kivételek végső lekezelése, és itt már lehetőség lenne figyelmeztetni a felhasználót, például egy hibát jelző párbeszédablak megjelenítésével. Természetesen, a kivétel rétegek közötti továbbításával párhuzamosan helyben is elvégezhetünk bizonyos kivételkezeléssel kapcsolatos műveleteket. Például, az előzőekben említett naplózás is fontos lehet, hogy megkönnyítsük a fejlesztők hibaelhárítási és javítási tevékenységét.

Az előző példával kapcsolatban az is megemlíthető, hogy a végső felhasználók nem feltétlenül érdekeltek a hiba pontos azonosításában. Például, lehetséges, hogy a szerver oldalon az adatokat

állományokban tároljuk, de erről a felhasználónak nem kell feltétlenül tudnia (az is lehetséges, hogy az adathozzáférési réteget a későbbiekben valamilyen meggondolásból lecseréljük, és egy jól megtervezett és felépített rendszer esetében az ilyen változtatásoknak nem kell befolyásolniuk a felsőbb rétegek működését). Természetesen, a szerver oldalon, például a hibák naplózásánál, különbséget tehetünk a lehetséges kivételtípusok között, külön rögzítve például a nem található állományokkal kapcsolatos kivételeket, és az egyéb írási/olvasási hibákkal kapcsolatos kivételeket, de a felhasználó számára ezek a részletkérdések nem feltétlenül fontosak. Valószínűleg a felhasználó megelégedne azzal, ha tudná, hogy valamilyen adathozzáféréssel kapcsolatos hiba lépett fel, a további részletek ismerete nem fontos számára. Az ilyen esetekre megoldást szolgáltathat a saját kivételtípusok bevezetése. Ezt úgy tehetjük meg, hogy az *Exception* ősosztályból, vagy annak egy számunkra megfelelőbb leszármazottjából származtatunk. Tekintsük az alábbi példát:

```
public class MyException extends Exception {
      String info;
      public MyException(String info) {
            super(info);
            this.info = info;
      }
      public String getInfo() {
            return info;
      }
}
```

A saját kivételtípusnak megfelelő *MyException* osztályunkat az *Exception* osztályból származtatjuk. A kivételosztály példányainak segítségével rögzíthetünk az új típusnak megfelelő speciális tulajdonságokat. Ennek szemléltetésére osztályunkban bevezettünk egy szöveges adattagot, és egy ennek lekérésére szolgáló metódust (természetesen a kivétellel kapcsolatos szöveges információk tárolására a *Throwable* osztály is lehetőséget ad, de az új kivételosztályon belül ugyanilyen módon más információkat is rögzíthetnénk). Az osztály konstruktora paraméterként kapja ezt a szöveget, és miután meghívta az alaposztály konstruktorát beállítja a megfelelő adattag értékét.

Nézzük, hogyan alkalmazhatnánk ezt az új kivételtípust az előzőekben említett feladat megoldására:

```
public void readFile(String filename) throws MyException {
      try {
            FileReader fr = new FileReader(filename);
            BufferedReader br = new BufferedReader(fr);
            String line;
            while ((line = br.readLine()) != null)
                  System.out.println(line);
      } catch (FileNotFoundException ex1) {
            //...      -          log this error
            throw new MyException("data access error");
      } catch(IOException ex2) {
            //...      -          log this error
            throw new MyException("data access error");
      }
}
```

A *readFile* metódusunkat módosítottuk. Ezúttal helyben végeztük el a kivételek kezelését. Miután rögzítettük a meghibásodás pontos okát a rendszernaplóban, egy új példányt hoztunk létre a saját kivételtípusunknak megfelelő *MyException* osztályból, és a *throw* kulcsszó segítségével „továbbdobtuk" ezt a kivételt a metódusunkat meghívó felsőbb rétegnek. A *readFile* metódus fejlécében a *throws* kulcsszó után ezúttal csak a *MyException* típust tüntettük fel, így a felsőbb rétegeknek csak ezt kell kezelniük. A felhasználó ilyen módon értesítést kaphat a meghibásodásról, és nem terheljük a fölösleges részletek ismertetésével. Megjegyezhetjük, hogy a naplózási műveletek végrehajtására a Java esetében több eszköz is rendelkezésünkre áll (az alapmechanizmus támogatására szolgáló interfészek és osztályok a *java.util.logging* csomagban találhatóak), de ezek részletesebb ismertetése nem képezi célját a jelen jegyzetnek.

Az előbbi példánkat úgy is értelmezhetjük, hogy a *MyException* típusú kivétel létrejötte más kivételtípusok előfordulásának volt az eredménye (példánk esetében konkrétan a *FileNotFoundException* vagy *IOException* típusú kivételek megjelenésének következményeként jött létre a kivételobjektum). Az ilyen helyzetekben, bizonyos esetekben hasznos lehet, ha vissza tudjuk követni a kivételek okozati láncolatát. A Java erre a célra a láncolt kivételek mechanizmusát (*chained exceptions*) biztosítja. A megoldás nagyon egyszerű: a *Throwable* osztály rendelkezik olyan konstruktorokkal, amelyeknek átadható egy másik *Throwable* példányra, a kivételt okozó másik kivételobjektumra mutató referencia, és ez a *getCause* metódus segítségével le is kérdezhető. A *Throwable* osztály megfelelő konstruktorai:

```
Throwable(String message, Throwable cause)
Throwable(Throwable cause)
```

Természetesen, ezeknek megfelelő konstruktorokat a származtatott osztályok (pl. *Exception*) is biztosítanak. Példánk esetében egyszerűen kiegészíthetnénk a *MyException* osztályunk konstruktorát egy második - *Throwable* típusú - paraméterrel, és a példányosításnál átadhatnánk az *ex1*, illetve *ex2* referenciákat.

Remélhetőleg, ez a rövid kiegészítés meggyőzte az olvasót a kivételkezelés hasznáról, és a Java által biztosított kapcsolódó lehetőségek hatékonyságáról, eleganciájáról. A továbbiakban egy példán keresztül hasonló módon megpróbáljuk kiegészíteni az eseménykezeléssel kapcsolatos ismereteink tárházát is.

9.2 Eseménykezelésről, általánosabban

Láthattuk, hogy az eseménykezelés a grafikus felhasználói felületek működésének kulcsfontosságú összetevője, de a módszer hátterében álló *Observer* tervezési minta, a megfigyelt-megfigyelő modell ennél jóval általánosabb jellegű, a szoftverfejlesztés nagyon sok területén alkalmazható. Magát a mintát nem mutatjuk be részletesen (mivel ez egy más célú, például tervezési mintákkal foglalkozó anyag esetében lenne alátámasztott), de egy konkrét példán keresztül szemléltetjük, hogy a Java nyelv esetében hogyan alkalmazhatjuk az eseménykezelést különböző (nem feltétlenül grafikus felületekkel kapcsolatos) feladatok elegáns és hatékony megoldására.

Tekintsük a következő egyszerű példát: egy számlálót szeretnénk létrehozni, amely értékének változásaikor speciális eseményeket generál, és lehetőséget ad az események fogadásában

érdekelt receptorok regisztrálására. Ezek a megfigyelők a számláló változásainak hatására különböző műveleteket hajthatnak végre. Megjegyezhetjük, hogy a program által biztosított funkcionalitás hasonló a *swing* csomagba beépített *Timer* osztály által biztosított funkcionalitáshoz, de esetünkben egy teljesen egyéni megvalósításról lesz szó.

Először egy speciális eseménytípust hozunk létre, az ennek megfelelő osztályt a *java.util* csomag *EventObject* osztályából származtatva:

```
import java.util.EventObject
class CounterEvent extends EventObject {
      private int count;
      CounterEvent(Object source, int count) {
            super(source);
            this.count = count;
      }
      public int getCount() {
            return count;
      }
}
```

Egy privát adattag segítségével rögzítjük a számláló aktuális értékét. A *CounterEvent* osztályunk konstruktora paraméterként kapja ezt az értéket, és ezen kívül az esemény forrását. Ez utóbbit paraméterként továbbadja az *EventObject* alaposztály konstruktorának. Az *EventObject getSource* metódusának segítségével ezután az esemény forrása lekérdezhető. A számláló értékének lekérdezésére a *CounterEvent* osztályunk *getCount* metódusát alkalmazhatjuk. Hasonló módszerrel tetszőleges eseménytípusoknak megfelelő osztályokat hozhatunk létre, az eseményobjektumokban rögzítve az események különböző tulajdonságait, amelyeket a figyelők lekérdezhetnek, és a műveletek végrehajtása során felhasználhatnak.

A következő lépésben hozzunk létre egy figyelő interfészt. Ezt az interfészt kell majd implementálniuk azoknak az osztályoknak, amelyek a számlálóval kapcsolatos események fogadásában érdekeltek.

```
import java.util.EventListener;
interface CounterChangeListener extends EventListener {
      void counterChange(CounterEvent e);
}
```

CounterChangeListener nevű interfészünk a *java.util* csomag *EventListener* interfészének kiterjesztése. Egyetlen metódust deklarál, amelynek neve *counterChange*, és paramétere egy *CounterEvent* típusú esemény. Az interfészt megvalósító osztályok ezen a metóduson belül implementálhatják a számláló változásaikor végrehajtandó műveleteket.

Hátra van még magának a számlálónak a létrehozása, annak az osztálynak a megírása, amelynek példányai a speciális *CounterEvent* típusú események forrásai lehetnek.

```
import java.util.LinkedList;
public class Counter extends Thread {
      private int count;
      private int delay;
      private LinkedList<CounterChangeListener> listeners =
                              new LinkedList<CounterChangeListener>();
```

```
        public Counter(String name, int count, int delay) {
              super(name);
              this.count = count;
              this.delay = delay;
        }
        public Counter(int count, int delay) {
              this.count = count;
              this.delay = delay;
        }
        public Counter() {
              this(0, 1000);
        }
        public void run() {
               while (true) {
                    try {
                          sleep(delay);
                    } catch (InterruptedException e) {}
                    count++;
                    notifyCounterChange(count);
              }
        }
        public void startCounting() {
              this.start();
        }
        public synchronized void notifyCounterChange(int count) {
              CounterEvent ce = new CounterEvent(this, count);
              for(CounterChangeListener ccl:listeners) {
                    ccl.counterChange(ce);
              }
        }
        public synchronized void addCounterChangeListener(
                              CounterChangeListener ccl) {
              listeners.add(ccl);
        }
        public synchronized void removeCounterChangeListener(
                              CounterChangeListener ccl) {
              listeners.remove(ccl);
        }
  }
```

A számlálót egy végrehajtási szálként valósítjuk meg, így a neki megfelelő *Counter* osztályt a *Thread* osztályból származtatjuk. A *count* nevű adattagban rögzítjük a számláló aktuális értékét, a *delay* adattagban a változások közötti időközök értékét. A regisztrált figyelők rögzítésére egy láncolt listát használunk. A *LinkedList* megvalósítás kiválasztását azok a tények motiválhatják, miszerint nem tudhatjuk előre a figyelők hozzávetőleges számát, egy esemény felléptekor minden esetben be kell járnunk a teljes listát, és ezen kívül csak adott elem hozzáadását vagy törlését szolgáló műveletekre van szükségünk. Osztályunk három konstruktorral rendelkezik. Az első paraméterként kapja a végrehajtási szál nevét, a késleltetés értékét és a számláló kezdőértékét. A második esetében csak a késleltetést és a kezdőértéket határozzuk meg.

Ezeken kívül egy alapértelmezett konstruktort is meghatározunk, amely alapértelmezett értékeket rendel a késleltetés (egy másodperc) és kezdőérték (0) attribútumokhoz.

A számláló értékét a *run* metóduson belül módosítjuk, és minden két módosítás között a megadott ideig várakoztatjuk a szálat. A módosításokról értesítjük a regisztrált figyelőket. Ezt a *notifyCounterChange* metóduson belül valósítjuk meg. Létrehozzuk az eseménynek megfelelő *CounterEvent* objektumot, paraméterként átadva neki a forrásra mutató referenciát (esetünkben *this*), és a számláló aktuális értékét. Ezután egy *for-each* ciklus segítségével bejárjuk a figyelők listáját, és minden regisztrált figyelő objektumra meghívjuk a *counterChange* metódust, paraméterként átadva a létrehozott eseményobjektumra mutató referenciát.

Azt szeretnénk, hogy az esemény előfordulását követően, a lista bejárása folyamán a figyelők listája ne változhasson (a bejárás közben ne lehessen új figyelőt regisztrálni, vagy regisztrált figyelőket törölni), ezért a bejárást szinkronizált metóduson belül végezzük el. Amennyiben valószínűsíthetően sok figyelőt kell értesítenünk, és nem akarjuk a teljes bejárás idejére blokkolni a számláló objektumunkat, megtehetjük azt is, hogy nem a metódust szinkronizáljuk, hanem a metódus belsejében egy szinkronizált programblokkon belül lemásoljuk az aktuális listát (egy segédváltozót, és a *clone* metódust felhasználva), majd a szinkronizált programblokkon kívül végezzük el a konkrét bejárást.

A fentieken kívül további két metódusra van szükségünk a figyelők regisztrációjához (a paraméterként kapott figyelőobjektum hozzáadása a figyelők listájához), illetve adott regisztrált figyelő eltávolításához (a figyelő törlése a regisztrált figyelők listájából). Mivel azt szeretnénk, hogy ezeknek a műveleteknek a végrehajtásával párhuzamosan ne lehessen más metódusokat meghívni (például az éppen módosítás alatt álló lista bejárása a receptorok értesítésének céljából), a művelet végrehajtásának idejére lefoglaljuk a számláló objektumunk monitorát, szinkronizált metódusokon belül hajtjuk végre az elemek hozzáadását vagy törlését.

Ahhoz, hogy a számlálónk működését szemléltessük, létrehozunk egy *CounterControl* osztályt. Az egyszerűség kedvéért ez az osztály fogja megvalósítani a *CounterChangeListener* interfészt, és a *main* metódus is ezen az osztályon belül kap helyet.

```
public class CounterControl implements CounterChangeListener {
      public CounterControl() {
            Counter c = new Counter();
            c.addCounterChangeListener(this);
            c.startCounting();
      }
      public void counterChange(CounterEvent e) {
            System.out.println("Counter value has changed: "
                                          + e.getCount());
      }
      public static void main(String[] args) {
            CounterControl cc = new CounterControl();
      }
}
```

A konstruktoron belül létrehozunk egy számláló objektumot, hozzárendelünk egy figyelőt (esetünkben az osztályunk aktuális példányát), és elindítjuk a számlálót. A számláló értékváltozásaikor generált események hatására a *counterChange* metóduson belül kiírunk

a konzolra egy üzenetet, és a számláló aktuális értékét. A *main* metóduson belül példányosítunk az osztályunkból.

Hasonlóan más figyelő osztályokat is létrehozhatunk, és ezeken belül a számláló értékváltozásaira különböző műveletek végrehajtásával reagálhatunk. Láthatjuk tehát, hogy az előzőekben csak grafikus felületek esetében alkalmazott eseménykezelés egy általánosan alkalmazható módszer, amely nagyon sok esetben hasznosnak és hatékonynak bizonyulhat. Remélhetőleg, ezt a tényt példánk segítségével sikerült a megfelelő módon alátámasztanunk.

9.3 Alapvető Java csomagokról, röviden

Az alap Java fejlesztési platform nagyon gazdag a biztosított eszközök, csomagok, interfészek és osztályok szempontjából. Ezen kívül, a rendkívül népes fejlesztői közösségnek és a szerteágazó alkalmazási lehetőségeknek köszönhetően, nagyon sok kapcsolódó API, keretrendszer és technológia áll rendelkezésünkre. Természetesen, már az alapcsomag részletes bemutatására sem lenne lehetőségünk a jelen jegyzet keretei között, de talán a további dokumentálódás szempontjából hasznos lehet egy rövid összefoglaló jellegű áttekintés.

A fejlesztői alapcsomagok hierarchiáján belül legalapvetőbb interfészek és osztályok, amelyek a legtöbb alkalmazás esetében szükségesek a *java.lang* csomagban kaptak helyet. Ezek közül már többekkel megismerkedtünk. Itt található az *Object* ősosztály, a *Class* osztály, a primitív adattípusokat burkoló osztályok, a karaktersorok tárolására alkalmas *String* osztály, a rendszertulajdonságok lekérdezésére, és különböző alapműveletek elvégzésére (konzol I/O, program lezárása, tömbök másolása, biztonsági beállítások, stb.) alkalmas *System* osztály, az osztályok betöltéséért felelős *ClassLoader*, és a biztonságért felelős *SecurityManager* osztályok. A végrehajtási szálak létrehozásához szükséges *Runnable* interfész és *Thread* osztály is ebben a csomagban kapott helyett, és rendszerfolyamatokat is létrehozhatunk a *ProcessBuilder* és *Process* osztályok segítségével. A kivételtípusok őseként szolgáló *Throwable* osztály, és az ebből származó *Exception*, *RuntimeException* és *Error* osztályok mellett nagyon sok konkrét kivételtípusnak megfelelő osztályt is találhatunk ebben a csomagban (*ArrayIndexOutOfBoundsException, NumberFormatException, NullPointerException, InterruptedException, SecurityException, VirtualMachineError*, stb.). Továbbá, itt találhatjuk meg a *Cloneable*, *Iterable* és *Comparable* interfészeket, és még nagyon sok alapvető hasznos osztályt és interfészt.

A *String* osztály mellett kiemelhetőek a *StringBuilder* és *StringBuffer* osztályok. A *String* objektumok konstans karakterláncok rögzítésére alkalmasak, ezek a karakterláncok a későbbiekben nem módosíthatóak. A *StringBuilder* osztály szintén karakterláncok rögzítésére alkalmas, de lehetőséget ad ezek változtatására is. Nagyon sok metódust biztosít számunkra, amelyek lehetővé teszik az illető karakterlánc kiegészítését, bizonyos részeinek cseréjét, vagy törlését, stb. Hasonló célt szolgál a *StringBuffer* osztály is, de ennek esetében a metódusok többsége szinkronizált, így az osztály példányai többszálú alkalmazások esetében is biztonságosan használhatóak.

Valószínűleg, nagyon gyakran fogjuk használni a *Math* osztályt is. Ez az osztály az alapvető matematikai függvényeknek megfelelő statikus metódusokat biztosít: abszolút érték (*abs*), hatványozás (*pow*), négyzetgyök (*sqrt*), exponenciális függvény (*exp*), természetes (*log*), vagy 10-es alapú (*log10*) logaritmus, trigonometriai függvények (*sin, cos, tan, asin, acos, atan,*

stb.), és így tovább. Ezeken kívül még számos más hasznos metódus is rendelkezésünkre áll: minimum- és maximumszámítás (*min* és *max* metódusok), kerekítés (*round*), átalakítások (*toRadians, toDegrees*), véletlen szám generálása (*random*), stb.

A *java.lang* csomag néhány nagyon fontos alcsomagot is tartalmaz. Bár a kapcsolódó fogalmakat a jegyzeten belül nem volt lehetőségünk tárgyalni, ezek közül talán érdemes kiemelnünk a Java annotációs mechanizmusát támogató *java.lang.annotation*, a *reflection* mechanizmust támogató *java.lang.reflect*, és a különböző referenciatípusokat biztosító, a memóriakezelés hatékonyabbá tételét szolgáló *java.lang.ref* csomagokat.

A Java alapcsomagok közül fontossági sorrendben a *java.util* csomagot kell másodikként megemlítenünk. Ebben a csomagban találhatóak a Java gyűjtemény keretrendszerének (JCF) interfészei és osztályai, a tömbökkel kapcsolatos műveleteket támogató *Arrays* osztály, a szöveges adatok formázását támogató *Formatter* osztály, az adatok beolvasásának hatékonyabbá tételét célzó *Scanner* osztály, a dátum és idő lekérdezését és manipulációját, valamint a kapcsolódó műveleteket támogató *Date, Calendar, GregorianCalendar, TimeZone, SimpleTimeZone* osztályok. Az előző részben meggyőződhettünk az eseménykezelés alapjául szolgáló *EventObject* osztály és *EventListener* interfész fontosságáról. Ezek szintén ebben a csomagban találhatóak, ugyanúgy, mint az említett *Observer* tervezési mintát támogató *Observer* interfész és *Observable* osztály. A tulajdonságok meghatározását (*Property*) és a nemzetköziesítést támogató (*Locale, ResourceBundle, Currency*) osztályok is itt találhatóak. Az alapvető időzítési műveletekre a *Timer* és *TimerTask* osztályokat, karakterláncok feldarabolására a *StringTokenizer* osztályt alkalmazhatjuk. Példáinkban gyakran használtunk fel véletlen számokat, az ezek generálására alkalmas *Random* osztály szintén ebben a csomagban található. Ezeken kívül a csomag tartalmazza az osztályokkal kapcsolatos műveletek esetében előforduló kivételeknek megfelelő osztályokat, és a felsoroltakon kívül még számos hasznos eszközt biztosít.

A *java.util* nagyon fontos alcsomagokat is tartalmaz. Itt található meg a párhuzamos programozást támogató *java.util.concurrent* csomag, az alapvető naplózási mechanizmusnak megfelelő *java.util.logging* csomag, a *zip* és *jar* csomagolási formátumokat támogató *java.util.jar* és *java.util.zip* csomagok.

Az adatfolyamokkal, bemeneti és kimeneti műveletekkel, állománykezeléssel kapcsolatos interfészeket és osztályokat a *java.io* csomagon belül találhatjuk meg. Ezek közül többet megismertünk a jegyzet kapcsolódó fejezetében, de, ha megnézzük a dokumentációt, látni fogjuk, hogy a felsorolás itt is a teljesség igénye nélkül történt.

Hasonló a helyzet a grafikus kezelői felületekkel kapcsolatos *java.awt* és *javax.swing*, valamint az applet-ek létrehozását támogató *java.applet* csomagokkal. Bár áttekintettük a fontosabb komponenseket és kapcsolódó osztályokat, valamint a grafikus felhasználói felületekkel kapcsolatos eseménykezelés alapjait (*java.awt.event* csomag), sőt az alcsomagok által biztosított eszközök közül is felhasználtunk néhányat, nagyon sok eszközre és lehetőségre nem tudtunk kitérni. Meggyőződhetünk erről, ha egy pillantást vetve a specifikációra, látni fogjuk az említett csomagokon belüli alcsomagok számát. Ezek közül az *awt* esetében megemlíthetjük például a Java2D API osztályait tartalmazó *awt.geom* csomagot, a színekkel és betűtípusokkal kapcsolatos *awt.color* és *awt.font* csomagokat, a képfeldolgozást támogató *awt.image* csomagot, a nyomtatási műveleteket támogató *awt.print* csomagot, valamint a *drag and drop* mechanizmus megvalósítását szolgáló *awt.dnd* csomagot. A swing esetében érdemes talán megemlíteni a speciális swing eseményekkel kapcsolatos *swing.event*, a keretek létrehozására alkalmas *swing.border*, a színek és állományok kiválasztását támogató *swing.colorchooser* és *swing.*

filechooser csomagokat. A táblázatok és hierarchikus adatok megjelenítését a *swing.table* és *swing.tree* csomagok, a műveletek visszavonását a *swing.undo* csomag támogatja. A szövegfeldolgozással kapcsolatos műveletek támogatását célozzák a *swing.text* csomag és alcsomagjai (pl. *text.html, text.rtf*).

Az alapcsomagok közül már nagyrészt fel is soroltuk mindazokat, amelyekkel a jegyzetben tárgyalt témakörök és példák esetében találkoztunk. Természetesen, a sornak itt még nincsen vége.

Egyszerű hálózati alkalmazások létrehozásakor a *java.net* csomag lehet segítségünkre. A bemeneti és kimeneti műveletek hatékonyabb végrehajtását, és kapcsolódó haladóbb módszerek alkalmazását támogatja a *java.nio* csomag, amely kiváló alapot szolgáltathat például komplexebb hálózati alkalmazások (pl. nagyon sok klienst párhuzamosan kiszolgáló szerverek), valamint kapcsolódó keretrendszerek létrehozására. Osztott alkalmazások készítését, távoli metódushívások megvalósítását támogatják az OMG (Object Managment Group) által létrehozott CORBA (Common Object Request Broker Architecture) szabványnak megfelelő *org.omg* csomag alcsomagjai, valamint a Java RMI API-nek (Remote Method Invocation) megfelelő *java.rmi* csomag, és alcsomagjai. A Java biztonsági keretrendszerének alapjául szolgál a *java.security* csomag. A szövegfeldolgozással kapcsolatos, és különböző formázási műveleteket támogató *java.text* csomag egyes osztályaival már megismerkedhettünk. Az adatfeldolgozással kapcsolatos műveleteket (általában relációs adatbázisban tárolt adatok) a *java.sql* csomag támogatja. Ha nagyon nagy számokkal szeretnénk dolgozni, a *java.math* csomag osztályai lehetnek segítségünkre (a *BigDecimal* és *BigInteger* osztályok). Ha XML állományokkal dolgozunk, az *org.xml.sax* csomag lehet hasznos a számunkra.

A swing csomag esetében láthattuk, hogy a csomag neve a *javax* előtaggal kezdődik. Ez arra a tényre utal, hogy az illető csomag kiterjesztésként (*Java extension*) kapott helyet a platformon belül (az ilyen csomagokat a szakirodalomban néhány helyen opcionális csomagokként is említik). A swing-hez hasonlóan még számos más csomag lett hasonlóan része a Java-nak. A kiterjesztések közül érdemes talán megemlíteni a képek írását és olvasását támogató *javax.imageio* csomagot, a hangfeldolgozással kapcsolatos *javax.sound* csomagot, a kriptográfiai műveleteket támogató *javax.crypto* csomagot, a speciális felhasználói felületek készítését (pl. látáskárosultak részére) támogató *javax.accessibility* csomagot, az XML dokumentumok feldolgozását támogató *javax.xml* csomagot, valamint a nyomtatási műveleteket támogató *javax.print* csomagot. A kiterjesztések között helyet kaptak olyan csomagok is, amelyek az azonos célú alapcsomagok kiterjesztését célozzák (pl. *javax.net, javax.sql*).

Láthatjuk, hogy az eszközök és lehetőségek száma óriási. Tudva azt, hogy ez a rövid felsorolás a teljesség legkisebb igénye nélkül készült, és számításba véve, hogy csak az alap platformról volt szó, nem tárgyaltuk azt a rengeteg külső API-t, keretrendszert és technológiát, amelyeknek már az egyszerű felsorolása is egy nagyobb terjedelmű könyvet venne igénybe, nyilvánvalóvá válhatott számunkra, hogy amennyiben a Java-t választjuk, nem lesz nehéz eszközöket találni a fejlesztési feladataink elvégzéséhez. Inkább talán az lehet kihívás, hogy ne vesztődjünk el a „bőség zavarában". Természetesen, senkitől sem várható el, hogy az összes eszközt és technológiát részleteiben ismerje. Már az sem várható el, hogy az alap platformon belül tökéletesen ismerjünk minden csomagot, és ezeken belül minden osztályt és interfészt, kívülről tudva azok attribútumait és metódusait. Lehet viszont egy jó általános rálátásunk a rendszerre, és akkor könnyen meg tudjuk keresni a számunkra éppen legmegfelelőbb eszközt, és ha rendelkezünk a megfelelő alapismeretekkel, nem fog nehézséget jelenteni ennek megértése

és alkalmazása. Ehhez, mint a programozás és általánosabban az informatika területein általában, elsősorban rengeteg gyakorlásra van szükség. Ezt a tevékenységet valószínűleg már el is kezdtük a jegyzetben javasolt feladatok megoldása során. A következő lépés talán már egy egyszerű kis projekt megvalósítása is lehet. A tízedik fejezet ennek a gyakorlati munkának az elkezdéséhez kíván alapvető útmutatást nyújtani.

9.4 Javasolt gyakorlatok

1. Egészítsük ki az előző fejezet végén javasolt első feladat (személyek adatainak beolvasása, megjelenítése és módosítása) megoldásakor elkészített programot. Hozzunk létre egy saját kivételtípust (pl. *DataAccessException*), és amennyiben az adathozzáférés során kivétel lép fel (pl. *FileNotFoundException*, vagy *IOException*), hozzunk létre egy példányt ebből a kivételtípusból, és azt továbbítsuk a felület fele (az interfészben jelezzük, hogy a metódusok ilyen típusú kivételt dobhatnak). Amennyiben kivétel lép fel, egy megfelelő párbeszédablak megjelenítésével figyelmeztessük a felhasználót.
2. Alakítsuk át a hatodik fejezet végén javasolt „autóversennyel" kapcsolatos feladat megoldása során elkészített programot. Hozzunk létre egy saját eseménytípust, és megfelelő figyelő interfészt. Mindenik versenyautó ilyen esemény forrása lehet: amennyiben beér a célba, létrejön egy esemény objektum. A felület figyeli az eseményeket, és ezeknek alapján építi fel az eredménylistát.

10

UML ALAPOK

Az előző fejezetekben leírt fogalmak áttanulmányozása, megértése, és a kapcsolódó gyakorlatok elvégzése után, szakmailag felkészülteknek kell lennünk az első egyszerű Java projektünk megvalósítására. Természetesen, még nem egy olyan bonyolultabb projektről, van szó, amely egy valós gyakorlati helyzetben, például vállalati felhasználásban is megállja a helyét. Egyelőre gondoljunk egy kisebb projektre, például egy egyszerű asztali (*desktop*) alkalmazásra, vagy kis játékprogramra. Egy ilyen jellegű projekt megvalósításához már rendelkezünk a szükséges ismeretekkel. A kérdés, hogy hogyan kezdjük el a munkát? A válasz: a követelmények pontos megfogalmazása és elemzése után a terv elkészítésével.

Nem lenne jó ötlet a kódolással kezdeni. Ha a programozó rendelkezik a megfelelő alapismeretekkel és némi tapasztalattal, követi a tervet, betartja a szabályokat és konvenciókat, akkor magának a kódolásnak egy jó terv esetén jó eredményhez kell vezetnie. A projekt „lelke" a terv: egy jó tervezőnek tapasztaltnak kell lennie (programozóként is), nem elég ismernie az alapokat, hanem tudnia kell alkalmazni a megfelelő tervezési mintákat, és át kell látnia a teljes projektet. Egy hibás terv nagy valószínűséggel a projekt bukását jelenti, de mindenképpen súlyos következményekkel járhat. Természetesen, egy egyszerű projekt esetében, a terv is egyszerű, nem kell szembenéznünk bizonyos nagyobb rendszerekre jellemző kihívásokkal, de az elemzési és tervezési fázis így is fontos, nem elhanyagolható. Éppen ezért, mi is ezekkel kezdjünk. Ebben a kezdésben lehet segítségünkre a jegyzet utolsó fejezete.

A fejezet egy egyszerű, vázlatos esettanulmányon keresztül mutatja be az elemzés és objektumorientált szoftvertervezés, valamint a Unified Modelling Language (UML) alapjait. A fejlesztés lépéseiről, és ezen belül az elemzési és tervezési szakaszról már röviden szó volt az objektumorientált alapfogalmakkal foglalkozó második fejezetben. Azt is említettük, hogy a jelen jegyzetnek nem célja részleteiben foglalkozni ezzel a témakörrel, és erre nem is lenne lehetőség (egy különálló területről lévén szó). Az előbbiekben leírtaknak megfelelően, szükséges lehet viszont néhány alapfogalomnak (az „alapok alapjainak") a bemutatása.

Tételezzük fel, hogy egy egyszerű nyilvántartó alkalmazást szeretnénk elkészíteni az egyetemi könyvtár részére. A lépések bemutatásánál hanyagolni fogjuk a különböző implementációs részletek bemutatását. Például, elégségesnek tekintjük, ha tudjuk, hogy az adatokat valamilyen módon tároljuk (valószínűleg relációs adatbázisban), de nem részletezzük az adathozzáférési réteg megvalósítását. Ugyanúgy a szoftver szerver részét úgy kellene megterveznünk, hogy a későbbiekben különböző típusú kliensalkalmazásokat szolgálhasson ki. Például, a könyvtár alkalmazottjainak szükségük van egy asztali alkalmazásra, amelynek segítségével adminisztrálhatják az adatokat. A későbbiekben megjelenhet az igény, hogy ugyanezt egy webes felület segítségével is megtehessék. A kliensek számára szintén szükséges lehetne egy webes felület, amelynek segítségével információkhoz juthatnak az aktuálisan kikölcsönözhető példányokról, és esetleg foglalásokat eszközölhetnek. Példánk esetében csak a könyvtáros által használt asztali adminisztrációs felületre koncentrálunk, tehát csak az egyik lehetséges

kliensalkalmazást tárgyaljuk, és itt is eltekintünk bizonyos részletektől. Esetünkben a klienseknek csak közvetett módon, a könyvtároson keresztül lehet hozzáférése a rendszerhez.

Egy valós helyzetben a szerver oldali résznek, az adathozzáférési rétegnek és a fölé rendelt szolgáltatási rétegnek lenne nagyon fontos a szerepe. Mivel még nem tárgyaltunk bizonyos idekapcsolódó tervezési mintákat és technológiákat, a szerver oldali részre a példa bemutatásának során tulajdonképpen „fekete dobozként" tekintünk majd (a részletek bemutatása talán egy következő részben kaphat majd helyet). Célunk csupán az elemzési és tervezési folyamat lépéseinek, és a kapcsolódó fogalmaknak, diagramoknak a vázlatos bemutatása. Ennek megfelelően kezdjük az első lépéssel a követelmény specifikáció elkészítésével.

10.1 Követelményspecifikáció

Az első lépés a felhasználó (kliens, *end user*) közreműködésével összeállítani a követelmények listáját. Ezek a nyilvántartó alkalmazásunk esetében a következő pontokban foglalhatóak össze:

- a program célja egy könyvtár adatainak nyilvántartása és menedzsmentje, a könyvtár működésének támogatása;
- a könyvtár regisztrált felhasználóknak kölcsönöz könyveket és folyóiratokat. A regisztrált felhasználók, valamint könyvek, és folyóiratok adatait a rendszer nyilvántartja;
- a könyvtár különböző könyveket és folyóiratokat szerez be. Egy-egy címnek több példánya is lehet. Az egyes régebbi példányok elhasználódhatnak, a rossz minőségű példányok kikerülnek a forgalomból. Az új példányok beszerzésével, és a régiek törlésével kapcsolatos adatokat a rendszernek nyilván kell tartania;
- a könyvtáros a könyvtár alkalmazottja, aki közvetlen hozzáféréssel rendelkezik a rendszerben tárolt adatokhoz. A rendszer első verziójában a kölcsönzők (kliensek) a könyvtároson keresztül juthatnak információkhoz (közvetett kapcsolat);
- a kliensek (a könyvtáros segítségével) információkat szerezhetnek a különböző címekről: rendelkezésre álló példányok, foglalási lehetőségek. Amennyiben egy adott címből van szabad példány, azt kikölcsönözhetik. Ellenkező esetben foglalásokat eszközölhetnek. A foglalások lemondhatóak;
- a könyvtáros egy grafikus felhasználói felület segítségével egyszerűen menedzselheti a rendszerben nyilvántartott adatokat (címek, példányok, kliensek, kölcsönzések, foglalások). Új adatokat vezethet be, módosíthat adatokat, vagy törölheti azokat;
- a könyvtáros által használt kliensalkalmazásnak az első verzió esetében egy egyszerű asztali alkalmazásnak kell lennie, amely a népszerű operációs rendszerek közül bármelyiken futtatható (platformfüggetlen);
- a rendszernek kliens-szerver architektúrára kell épülnie, és kiterjeszthetőnek kell lennie. Az első verziónak egyetlen asztali kliensalkalmazást kell tartalmaznia (a könyvtáros által használt adminisztrációs programot), de a rendszernek a későbbiekben kiegészíthetőnek kell lennie más modulokkal (kliensalkalmazásokkal).

A követelményspecifikáció elkészítése után következhet az elemzés fázisa.

10.2 Elemzés

Az elemzési fázis (analízis) első szakaszában a követelményspecifikáció alapján azonosítjuk a rendszerrel kapcsolatos használati eseteket (*use cases*), és elkészítjük ezek leírását, valamint a megfelelő diagramot (*use case diagram*).

10.2.1 Használati esetek

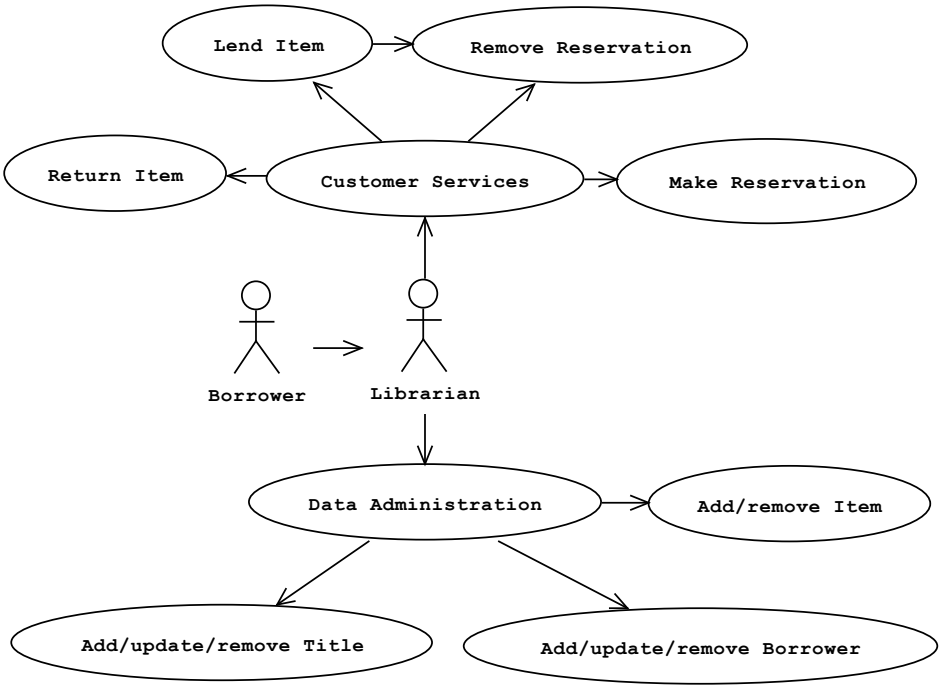
A használati esetek a rendszer által biztosított funkcionalitásokat írják le: kik és milyen módon használhatják a rendszert. Az első lépésben a felhasználókat (aktorokat) kell azonosítanunk. Esetünkben a rendszerrel csak a könyvtáros kerül közvetlen kapcsolatba, de mivel közvetett módon a kliensek is hozzáférhetnek információkhoz, őket is aktoroknak tekinthetjük. Megjegyzendő még, hogy az aktor (*actor*) fogalom általános: a rendszer funkcionalitásait igénybe vevő felhasználó nem feltétlenül egy konkrét személy, hanem egy szoftverkomponens is lehet.

A követelményspecifikáció alapján a nyilvántartó programunk esetében, a következő használati eseteket (funkcionalitások) határozhatjuk meg:

- példány kölcsönzése (*lend item*);
- kikölcsönzött példány visszaszolgáltatása (*return item*);
- foglalás (*make reservation*);
- foglalás törlése/lemondása (*remove reservation*);
- cím hozzáadása, módosítása, törlése (*add/update/remove title*);
- példány hozzáadása, törlése (*add/remove item*);
- kliens regisztrációja, vonatkozó adatok módosítása, törlése (*add/update/ remove borrower*).

A használati esetek beazonosítása után egy-egy leírást készítünk ezekhez. Az egyszerűség kedvéért tekintsük csak a kölcsönzés leírását:

- kliens beazonosítása
- cím beazonosítása
- foglalás ellenőrzése
- ha nem volt foglalás
    - szabad példány keresése
    - kölcsönzés regisztrálása
- ha volt foglalás
    - szabad példány keresése
    - foglalás törlése
    - kölcsönzés regisztrálása



10.1 ábra: Használati eset diagram (use case diagram)

Hasonlóan a többi használati esethez is elkészítünk egy-egy leírást. Ezután megszerkesztjük a használati eset diagramot (*use case diagram*) (10.1 ábra).

A használati esetek beazonosítása, leírása és a megfelelő diagram elkészítése után az elemzés következő lépése a környezeti elemzés (*domain analysis*).

10.2.2 Környezeti elemzés

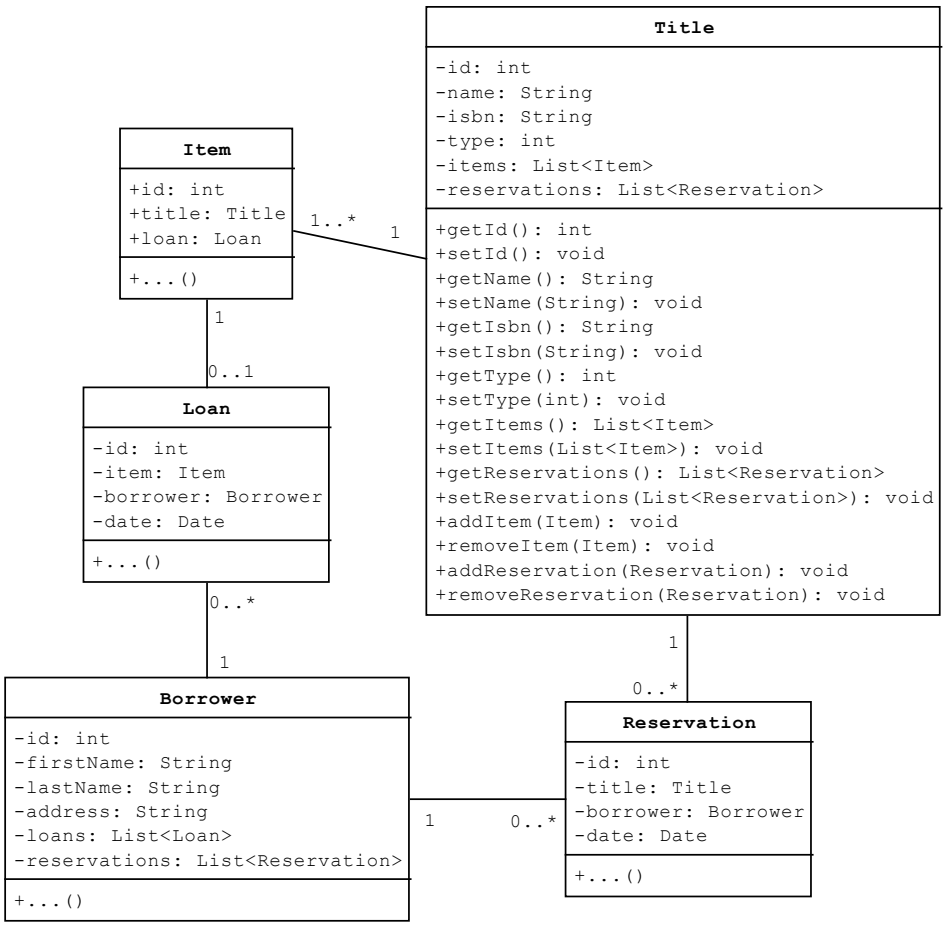
Az elemzésnek ebben a szakaszában a rendszer magját (*core*), a központi osztályokat, valamint ezek függőségeit azonosítjuk be, és elkészítjük a megfelelő osztálydiagramot. Ezek az osztályok a rendszer minden komponense számára fontosak, minden alrendszer használni fogja őket, szerver és kliens oldalon egyaránt, legyen szó az adathozzáférési rétegről, a vezérlésért felelős modulról, vagy a grafikus felületről. Az idetartozó osztályokat *domain* osztályoknak (*domain classes*), példányaikat *business* objektumoknak (*business objects*), vagy modell objektumoknak nevezik. A rendszer központi entitásait leíró osztályok, amelyek tipikusan csak a tulajdonságok leírására, az adatok rendszeren belüli reprezentációjára, és a rétegek közötti információáramlás megvalósítására (adathordozás) szolgálnak. A Java esetében többnyire egyszerű POJO-k, vagy Java Bean-ek, amelyek privát adattagokat és publikus *getter/setter* metódusokat tartalmaznak, és esetenként a megfelelő módon újradefiniálhatják az *Object* ősosztály bizonyos metódusait. Bonyolultabb funkcionalitásokat nem valósítanak meg.

A rendszerünk esetében ezeket az osztályokat és kapcsolataikat a 10.2 ábrán látható osztálydiagram szemlélteti. Az egyszerűség kedvéért csak egy osztály esetében tüntettük fel a *getter* és *setter* metódusokat. A többi osztálynak ezek természetesen ugyanúgy részei (a tényt, miszerint a diagram esetében hanyagoltuk ezek feltüntetését a +...() jelzi).

A környezeti elemzés elvégzése, és a megfelelő osztálydiagram elkészítése után továbbléphetünk a következő fázisra, a tervezésre.

10.3 Tervezés

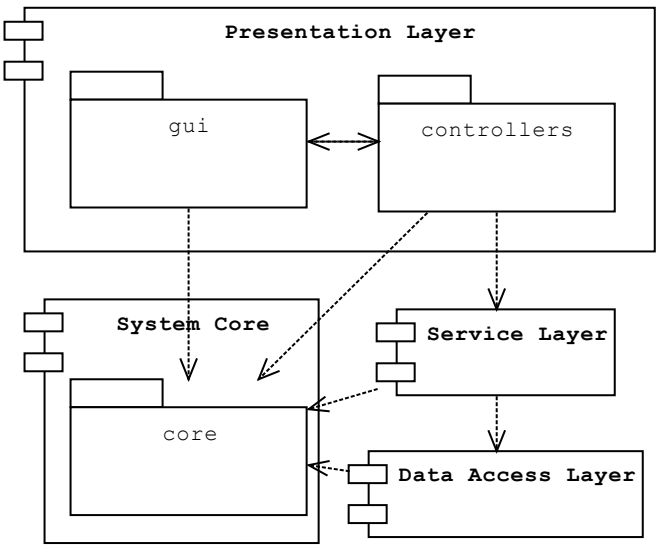
A tervezés (*design*) fázis első szakasza az architektúra megtervezése (*architecture design*). Ebben a szakaszban azonosítjuk be az alrendszereket, az ezeket alkotó csomagokat, és ezek kapcsolatait.



10.2 ábra: a Környezeti elemzés során elkészített osztálydiagram (*domain* osztályok)

10.3.1 Architektúra megtervezése

A teljes alkalmazást több rétegre osztjuk: a legalsó adathozzáférési réteg felelős az adatok menedzsmentjéért. Ezzel a réteggel közvetlen módon a második szinten elhelyezkedő szolgáltatási réteg kommunikál a megfelelő interfészeken keresztül. Ez a két réteg felelne meg a szerver oldali komponenseknek. Ahogyan azt már említettük, erre a részre a jelen példa esetében „fekete dobozként" tekintünk. A harmadik réteg, a prezentációs réteg, a különböző kliensalkalmazásoknak felelne meg, amelyek a szolgáltatási réteggel a megfelelő interfészeken keresztül kommunikálnak. Esetünkben egyetlen kliens program, a könyvtáros által használt asztali alkalmazás tartozik ehhez a réteghez, és ennek csomagjait fogjuk röviden tárgyalni.



10.3 ábra: a rendszer architektúrája

A rendszer magját egy külön komponensként tüntettük fel. A *core* csomag tartalmazza a környezeti elemzés során beazonosított interfészeket és osztályokat. Ezeket a rendszer minden komponense használni fogja. Az MVC elvnek megfelelően a prezentációs rétegen belüli kliensalkalmazást két csomagra bontottuk: az egyikbe a grafikus kezelői felület interfészei és osztályai kerülnek, a másikba a vezérlésért felelős osztályok. Az utóbbiak a szolgáltatási réteg interfészein keresztül továbbítják a felületről érkező kéréseket a szerver oldali komponensekhez.

A rendszer architektúrájának megtervezése, a rétegek, modulok és csomagok, valamint az ezek közötti függőségek beazonosítása után következhet a részletes terv elkészítése (*detailed design*). A részletes terv elkészítésének első fázisában a különböző csomagoknak megfelelő részletes osztálydiagramok készülnek el. Ezután következik az úgynevezett dinamikus UML diagramok elkészítése. Ide tartoznak a szekvencia diagramok, a kollaborációs és állapotátmenet diagramok.

10.3.2 Osztálydiagramok

A részletes terv elkészítésének első lépése az egyes csomagokon belüli interfészek és osztályok, valamint ezek kapcsolatainak beazonosítása, és a megfelelő osztálydiagramok elkészítése.

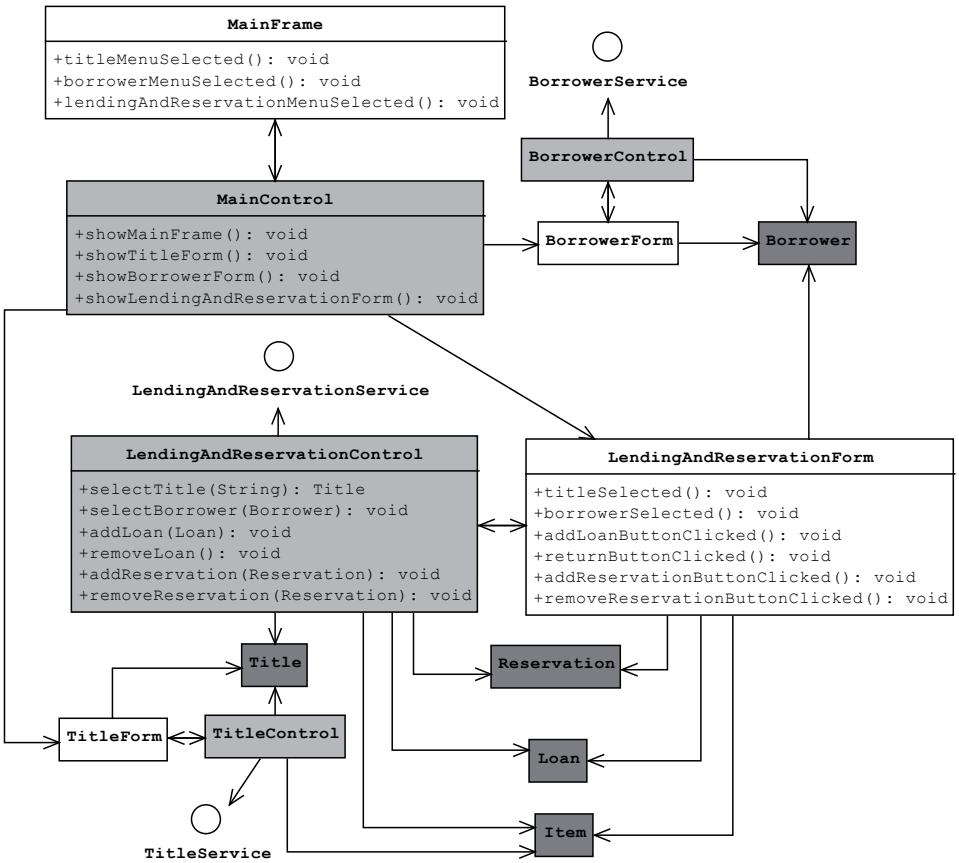
A tervezés során „lentről" érdemes elindulni. Ennek megfelelően, a legelső lépésben a környezeti elemzés során elkészített osztálydiagramot vizsgáljuk felül. A tervezés már a felhasznált technológiák és implementációval kapcsolatos információk ismeretében történik, és ennek megfelelően a *domain* osztályok terve módosulhat, kiegészülhet. Lehetséges, hogy további információkkal, adattagokkal kívánjuk kiegészíteni az osztályokat, és új elemeket, kapcsolatokat is bevezethetünk. Esetünkben például lehetséges, hogy miután további eltérések mutatkoznak a könyvek és folyóiratok között, úgy döntünk, hogy a típust meghatározó adattag helyett egy származtatási viszonyt szükséges bevezetnünk (pl. A *Title* osztályból származtatva a *BookTitle* és *JournalTitle* osztályokat). Az adattárolásra és menedzsmentre használt technológia figyelembevételével felülbírálhatjuk az azonosítók láthatóságával kapcsolatos döntéseket (pl. bizonyos keretrendszerek esetében nem szükségesek *setter* metódusok az egyedi azonosítókhoz). Dönthetünk továbbá arról, hogy osztályaink újradefiniálják az *Object* ősosztály bizonyos metódusait, vagy megvalósítsanak bizonyos interfészeket (például szeretnénk, ha a foglalások időpontjuk szerint összehasonlíthatóak lennének, stb.). A felsoroltakon kívül természetesen más hasonló módosításokat, kiegészítéseket is bevezethetünk a tervbe. Ezeknek megfelelően szükséges a *domain* osztályoknak megfelelő osztálydiagram módosításokat és kiegészítéseket tartalmazó új verziójának az elkészítése. Az egyszerűség kedvéért ezt kihagyjuk a jegyzetből. Hasonlóan mellőzük a következő két lépést. Az adathozzáférési réteg, majd a szolgáltatási réteg csomagjait, alcsomagjait, valamint az ezeken belüli interfészeket és osztályokat, illetve ezek kapcsolatait kellene beazonosítanunk, elkészítve a megfelelő osztálydiagramokat. Mivel a rendszernek ezekre a részeire „fekete dobozként" tekintünk, bemutatásukat is mellőzük.

A prezentációs rétegen belüli kliensalkalmazás esetünkben két csomagból áll. Az ezeknek megfelelő részleges osztálydiagramokat a 10.4 ábra szemlélteti. Az egyszerűség kedvéért a két csomag osztályait egy diagramon belül tüntettük fel, csak a háttér színe változik (fehér a grafikus kezelői felület osztályainak esetében, szürke a vezérlésért felelős osztályok esetében). A csomagok osztályai egyrészt használják a *domain* osztályokat, másrészt a vezérlésért felelős osztályok ismerik a szolgáltatási réteg interfészeit, így az ábrán ezeket is feltüntettük, de természetesen az adattagokat és metódusokat ezúttal mellőztük.

A grafikus felhasználói felület osztályainak esetében célszerű lehet először egy vázlatos felülettervet készíteni valamilyen grafikus szerkesztő programmal. Egy ilyen terv alapján egyszerűbb beazonosítani az osztályokat, illetve azok adattagjait és metódusait. Esetünkben ezt a lépést is kihagyjuk, a grafikus komponenseknek megfelelő adattagokat nem adjuk meg, és a lehetséges metódusokra is csak néhány példát adunk.

A vázlatos terv szerint az alkalmazás egy központi ablak (*MainFrame*) megjelenítésével indul, és innen valamilyen menü segítségével kérhetjük az egyes funkcionalitásoknak megfelelő ablakok megjelenítését: a kliensekkel kapcsolatos információk kezelését a *BorrowerForm* nevű ablak, a címekkel kapcsolatos adatok menedzsmentjét (beleértve a rendelkezésre álló

példányszámok módosítását) a *TitleForm* teszi lehetővé. A kölcsönzésekkel és foglalásokkal kapcsolatos információk menedzsmentje a *LendingAndReservationForm* ablak által biztosított felület segítségével történik. A felületeken elhelyezett komponensekhez hozzárendeljük a megfelelő figyelőket. A figyelők egy adott esemény felléptekor a vezérlést a megfelelő vezérlésért felelős osztálynak továbbítják.



10.4 ábra: a *gui* és *controllers* csomagoknak megfelelő osztálydiagram

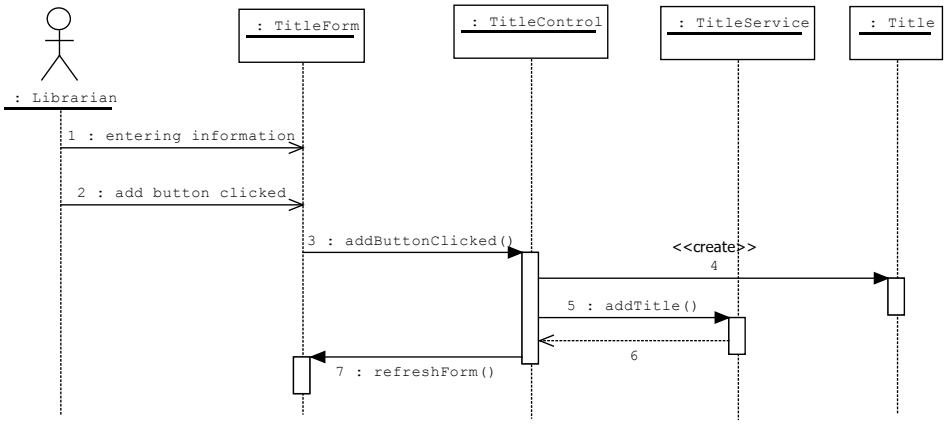
A vezérlésért felelős osztályokat tartalmazó csomag központi tagja a *MainControl,* hozzá továbbítódnak a központi menüvel kapcsolatos események, és ennek megfelelően ez az osztály felelős a különböző ablakok megjelenítéséért. A kölcsönzések és foglalások menedzsmentjével kapcsolatos felületről az események a *LendingAndReservationControl* osztályhoz továbbítódnak. A vázlat példaként megad néhány lehetséges metódust. A felhasználó a felületről kiválaszthat egy adott klienst és egy adott címet, a program ennek megfelelően megjeleníti a felületen a kikölcsönzött és rendelkezésre álló példányokkal, valamint a foglalásokkal kapcsolatos információkat. A felhasználónak (könyvtárosnak) lehetősége van új kölcsönzést létrehozni, vagy egy régit törölni, és hasonlóan foglalásokat hozzáadni vagy törölni. Ezeket a műveleteket a felület megfelelő komponenseinek segítségével kezdeményezheti. Az események

hatására meghívódnak a vezérlésért felelős osztály megfelelő metódusai. Ezek a szolgáltatási réteg interfészén (*LendingAndReservationService*) keresztül a végrehajtásért felelős szerver oldali komponensekhez továbbítják a kéréseket, majd az adatbázis frissítése után, a szerver oldaltól kapott információk alapján frissítik a felületet. Hasonlóan történnek a dolgok a címek és kliensinformációk kezeléséért felelős felületek és megfelelő vezérlők esetében. A címek menedzsmentjét megvalósító felülettől az események a *TitleControl* osztályhoz továbbítódnak, a kliensinformációk esetében a *BorrowerControl* osztály felelős a vezérlésért. Ezektől az osztályoktól a kérések szintén a szolgáltatási réteg megfelelő interfészein keresztül továbbítódnak a szerver oldali komponensekhez, és a vezérlők a visszatérített eredmény alapján végzik a felület frissítését. Az utóbbi felületeknek és vezérlőknek az esetében nem adtunk példákat metódusokra, és a példák kiválasztása az előző esetben is a teljesség igénye nélkül, orientatív jelleggel történt.

10.3.3 Szekvencia diagramok

A szekvencia diagramok (*sequence diagrams*) egy-egy használati esetnek felelnek meg, és a használati esetnek megfelelő folyamatok, műveletek időbeli sorrendjét reprezentálják. Vázlatos elkészítésükre már az elemzés fázisában lehetőségünk van, de a részletes terv elkészítésekor, a komponensek pontosabb beazonosítása után mindenképpen frissítenünk kell őket.

Példánk esetében a címek hozzáadásának megfelelő használati eset szekvencia diagramját szemléltetjük, a 10.5 ábrán.



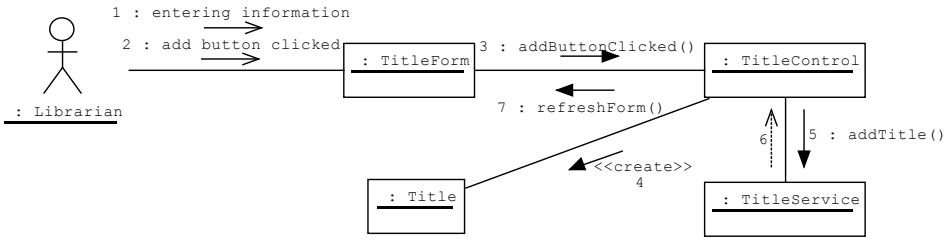
10.5 ábra: a cím hozzáadása (add title) használati esetnek megfelelő szekvencia diagram

A könyvtáros miután megnyitotta a címek beírására alkalmas ablakot (*TitleForm*), kitölti adatokkal a megfelelő mezőket, majd a hozzáadás gombra kattint. Az eseményt a felület a vezérlésért felelős osztályhoz (*TitleControl*) továbbítja, amelyik az adatok (kitöltött mezők) alapján felépít egy cím (*Title*) objektumot, és a szolgáltatási réteg megfelelő interfészén (*TitleService*) keresztül kéri ennek az objektumnak az elmentését. A művelet által visszatérített érték alapján (pl. sikeres beszúrás, vagy hiba) frissíti a felületet.

10.3.4 Együttműködési diagramok

Az együttműködési, vagy kollaborációs diagramokat (*collaboration diagrams*) a szekvencia diagramok alternatívájaként (vagy azokkal együtt) alkalmazhatjuk, azokban az esetekben, amikor kevésbé vagyunk érdekeltek a műveletek időrendiségének ábrázolásában, csak a komponensek együttműködését szeretnénk szemléltetni. Opcionálisan a műveletek sorrendje az együttműködési diagramok esetében is feltüntethető. A legtöbb szerkesztő lehetőséget kínál a diagram automatikus legenerálására egy meglévő szekvencia diagram alapján.

Az előző példánkban bemutatott szekvencia diagramnak megfelelő kollaborációs diagramot a 10.6 ábra szemlélteti.

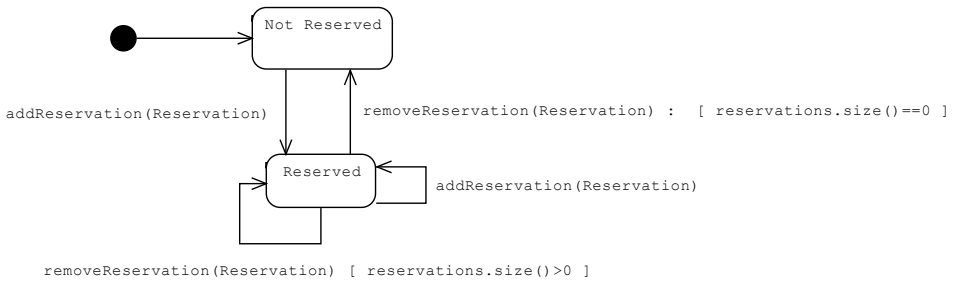


10.6 ábra: a cím hozzáadása használati esetnek megfelelő együttműködési diagram

10.3.5 Állapotátmenet diagramok

Az előzőekben bemutatott szekvencia és együttműködési diagramokhoz hasonlóan az állapotátmenet diagramok (*state diagrams*) szintén a rendszer dinamizmusát, a változást írják le, de az előzőekkel ellentétben egy „passzív" szempontból. A rendszer központi entitásainak külső események hatására bekövetkező reakcióit, állapotváltozásait szemléltetik.

Példánk esetében tekintsük a cím (*Title*) objektumok lehetséges állapotait. Az állapotokat, és a külső események hatására bekövetkező állapotátmeneteket a 10.7 ábra szemlélteti.



10.7 ábra: a cím objektumoknak megfelelő állapotátmenet diagram

Ha egy adott címből nincsenek rendelkezésre álló példányok (mert előzőleg mindegyiket kikölcsönözték), a kliens kérheti a könyvtárostól foglalás bejegyzését. Kezdetben minden cím „szabad", még nincsenek kapcsolatos foglalások, a cím objektumok életciklusa „nem foglalt" (*not reserved*) állapotból indul. Egy foglalás hozzáadásának hatására az állapot megváltozik, a cím „foglalt" (*reserved*) állapotba kerül. Ezután további foglalások hozzáadása is lehetséges, amely ugyanebbe az állapotba viszi a címobjektumokat (*self transition*). Foglalások törlése is lehetséges. Ennek hatására egy cím foglalt állapotból visszakerülhet nem foglalt állapotba (amennyiben már nincsen több kapcsolatos foglalás, a foglalásokat tartalmazó lista mérete 0). Ellenkező esetben (ha még vannak foglalások) a cím megmarad foglalt állapotban.

10.4 Megvalósítás, ellenőrzés és utómunkálatok

A tervezési fázis után következhet a megvalósítás szakasza, melynek során a tervek alapján az előzőleg meghatározott programozási nyelvben, a kiválasztott programozási felületek és technológiák alkalmazásával megtörténik a konkrét implementáció (kódolás), és elkészül a végtermék, a tervezett szoftverrendszer.

Az elemzési, tervezési és megvalósítási fázis mellett a fejlesztési folyamat fontos összetevője a verifikációs és validációs (V&V) folyamat. A verifikáció során azt ellenőrizzük, hogy a szoftver megfelel-e a specifikációjának, eleget tesz-e a funkcionális és nem funkcionális (pl. teljesítménnyel, hordozhatósággal, biztonsággal kapcsolatos) követelményeknek. Azt ellenőrizzük tehát, hogy a terméket jól készítettük-e el (Boehm megfogalmazása szerint: „Are we building the product right?"). A validáció során azt ellenőrizzük, hogy a termék kielégíti-e a kliens (valós) igényeit, megfelel-e az elvárásoknak. Azt ellenőrizzük tehát, hogy a megfelelő terméket készítettük-e el (Boehm megfogalmazása szerint: „Are we building the right product?").

A V&V folyamat a fejlesztés teljes folyamatát felöleli, tulajdonképpen az elemzéssel egyszerre kezdődik, és a végtermék átadásáig (vagy esetenként azt követően is) tart. Nagyon sokféle tevékenység, módszer és eszköz tartozik ehhez a folyamathoz, amelyek két alapvető kategóriába sorolhatóak: statikus módszerek (átvizsgálás) és dinamikus módszerek (tesztelés). Mivel a második kategóriával ellentétben az első kategóriához tartozó módszerek alkalmazásához nincs szükség működő modulokra, ezeket a módszereket már a fejlesztés korai szakaszában is alkalmazhatjuk: a forráskódon kívül, a terveket is átvizsgálhatjuk, formális matematikai módszerekkel bizonyíthatjuk az algoritmusok helyességét, automatikus kódellenőrzőket alkalmazhatunk. Az általános célú szoftverek esetében (amikor nem kritikus rendszerekről van szó) általában a dinamikus módszerek dominálnak. A tesztelés esetében is nagyon sokféle módszer létezik: beszélhetünk a hibák felderítését célzó hiányosságtesztelésről, vagy a rendszer megfelelő működését alátámasztó validációs tesztelésről; az egyes különálló komponensek esetében alkalmazható komponens tesztelésről, vagy a komponensek együttműködését vizsgáló integrációs tesztelésről; az implementációs részleteket figyelembe vevő „üvegdoboz" megközelítésről, vagy a rendszert „fekete dobozként" kezelő stratégiáról, és így tovább.

Miután elkészült a termék, és a V&V folyamat igazolta, hogy megfelel az adott elfogadási szintnek, megtörténhet annak átadása, „üzembe helyezése", majd a visszajelzések alapján az esetleges további javítások, módosítások elvégzése, a folyamatos karbantartás, a felhasználók segítése, és bizonyos esetekben ezzel párhuzamosan új (kiegészítéseket, további komponenseket tartalmazó) verziók fejlesztése.

Már a jegyzet második fejezetében is említettük, hogy a modern fejlesztési stratégiák esetében a fejlesztés említett alapszakaszai nem egyszerűen, szekvenciálisan követik egymást. Többféle megközelítés létezik, a megfelelő kiválasztásának a vállalati struktúrának, a fejlesztői tapasztalatoknak és a termék típusának figyelembevételével kell megtörténnie. Ugyanúgy, ahogy a UML esetében csak néhány nagyon alapvető témát tárgyalhattunk, a V&V folyamat és a szoftvertervezés rendkívül gazdag módszer- és eszköztárainak esetében sincsen lehetőségünk kitérni a részletekre, de ez nem is képezte a jelen jegyzet célját. A cél az volt, hogy egy alap rálátást biztosítson az olvasónak a szoftverfejlesztés folyamatára, megnyitva az utat a kapcsolódó témakörök továbbtanulmányozására, és ugyanakkor mutasson be néhány alapvető módszert és eszközt, megteremtve az alapot az első kis Java projekt elkészítéséhez.

Remélhetőleg a jegyzet segítségére volt az olvasónak a Java alapismeretek elsajátításában, és az alapvető szoftverfejlesztéssel kapcsolatos tevékenységek megismerésében. Amennyiben így van, semmi akadálya az említett Java projekt megvalósításának, legyen az egy kis játékprogram, vagy egy más egyszerű asztali alkalmazás. Ezt követheti majd a haladóbb témakörök áttanulmányozása: a különböző fejlesztési módszerek és stratégiák, tervezési minták, verifikációs és validációs eljárások, és természetesen nem utolsó sorban a haladóbb Java technológiák megismerése. Talán lesz lehetőség arra, hogy ezekben a jegyzet következő részei is segítséget nyújtsanak. Addig is jó munkát, jó szórakozást, tartalmas időtöltést kíván a szerző.

10.5 Javasolt gyakorlat

A fejlesztési folyamat alapszakaszain belüli lépések betartásával készítsünk egy egyszerű Java projektet. Lehet ez egy kis asztali alkalmazás, vagy egy egyszerű játékprogram, tetszőlegesen választhatunk. Útmutatásként néhány ötlet:

- egyszerű, állománykezelésen alapuló nyilvántartó program adataink kezelésére (pl. könyvek, zenék, filmek, stb.), vagy más számunkra hasznos asztali alkalmazás (pl. határidőnapló, stb.)
- egyszerű rajzolóprogram, vagy valamilyen grafikus szerkesztő (pl. kottaszerkesztő és zenelejátszó)
- egyszerű játékprogram, például kígyó (*snake*), pasziánsz (*solitaire*), aknakereső (*minesweeper*), *shokoban*, *mastermind*, memória (kártyával, képpel, vagy zenével), vagy bármelyik klasszikus játék valamilyen (lehetőleg minél egyénibb) változata. Ötletekkel szolgálhatnak a különböző egyszerű mobiltelefonra írt, vagy *Flash* játékok.

Internetes hivatkozások

	Leírás	**Cím**
1.	A Java hivatalos weboldala	`http://java.sun.com/`
2.	Java fejlesztői dokumentáció	`http://java.sun.com/javase/7/docs/`
3.	Java nyelvi specifikáció (*The Java Language Specification*)	`http://java.sun.com/docs/books/jls/`
4.	Sun által kiadott Java oktatási anyagok	`http://java.sun.com/docs/books/tutorial/`
5.	Sun oktatási anyag: Java nyelvi alapok	`http://java.sun.com/docs/books/tutorial/java/`
6.	Sun által meghatározott Java elnevezési és kódolási konvenciók	`http://java.sun.com/docs/codeconv/`
7.	Sun oktatási anyag: AWT felületek és eseménykezelés	`http://java.sun.com/developer/onlineTraining/awt/contents.html`
8.	Sun oktatási anyag: Java 2D grafika	`http://java.sun.com/docs/books/tutorial/2d/overview/`
9.	Sun oktatási anyag: Applet-ek	`http://java.sun.com/docs/books/tutorial/deployment/applet/`
10.	Sun oktatási anyag: nemzetköziesítés	`http://java.sun.com/docs/books/tutorial/i18n/`
11.	Sun oktatási anyag: a SWING eszköztár	`http://java.sun.com/docs/books/tutorial/uiswing/`
12.	Sun által kiadott cikk: az AWT és SWING grafika összehasonlítása. Cím: *Painting in AWT and SWING*. Szerző: Amy Fowler	`http://java.sun.com/products/jfc/tsc/articles/painting/`
13.	Sun oktatási anyag: végrehajtási szálak	`http://java.sun.com/docs/books/tutorial/essential/concurrency/`
14.	Sun oktatási anyag: a *Thread* osztály érvénytelenített metódusai	`http://java.sun.com/j2se/1.4.2/docs/guide/misc/threadPrimitiveDeprecation.html`

	Leírás	Cím
15.	Sun oktatási anyag: adatfolyamok	`http://java.sun.com/docs/books/tutorial/essential/io/`
16.	Sun által kiadott cikk a szerializációról. Cím: *Discover the secrets of the Java Serialization API* Szerző: Todd Greanier	`http://java.sun.com/developer/technicalArticles/Programming/serialization/`
17.	Sun által kiadott oktatási anyag a gyűjtemény keretrendszerről. Szerző: Josh Bloch	`http://java.sun.com/docs/books/tutorial/collections/`
18.	Sun oktatási anyag: kivételkezelés	`http://java.sun.com/docs/books/tutorial/essential/exceptions/`
19.	Az Eclipse weboldala	`http://www.eclipse.org/`
20.	A NetBeans weboldala	`http://netbeans.org/`
21.	Az OSGi weboldala	`http://www.osgi.org/`
22.	Programozási nyelvek gyűjteménye	`http://99-bottles-of-beer.net/`
23.	Az Excelsior Jet weboldala	`http://www.excelsior-usa.com/`
24.	A UML hivatalos weboldala	`http://www.uml.org/`
25.	A StarUML oldala	`http://staruml.sourceforge.net/en/`
26.	James Gosling internetes naplója	`http://blogs.sun.com/jag/`
27.	A Groovy weboldala	`http://groovy.codehaus.org/`
28.	A JRuby weboldala	`http://jruby.org/`
29.	A Jython weboldala	`http://www.jython.org/`
30.	A JUnit weboldala	`http://www.junit.org/`
31.	Wikipédia szócikk a vízesés modellről	`http://en.wikipedia.org/wiki/Waterfall_model`
32.	Wikipédia szócikk, szoftverfejlesztési stratégiák listája	`http://en.wikipedia.org/wiki/List_of_software_development_philosophies`
33.	Wikipédia szócikk az Agile szoftverfejlesztési stratégiáról	`http://en.wikipedia.org/wiki/Agile_software_development`
34.	Wikipédia szócikk a spirális modellről	`http://en.wikipedia.org/wiki/Spiral_model`
35.	Wikipédia szócikk a RUP szoftverfejlesztési folyamatról	`http://en.wikipedia.org/wiki/IBM_Rational_Unified_Process`

Ajánlott szakirodalom

ANGSTER E., *Objektumorientált tervezés és programozás. Java*, I. és II. kötet, 4KÖR Bt., 2003, 2004.

ANTAL M., *Objektumorientált programozás*, Scientia, 2007.

ARNOLD K., **GOSLING J.**, **HOLMES D.**, *The Java Programming Language*, 4th edition, Addison-Wesley Professional, 2005.

BOOCH G., **RUMBAUGH J.**, **JACOBSON I.**, *The Unified Modeling Language User Guide*, 2nd edition, Addison-Wesley, 2005.

ECKEL B., *Thinking in Java*, 4th edition, Prentice Hall, 2006.

FLANAGAN D., *Java in a Nutshell. A Desktop Quick Reference*, 5th edition, O'Reilly Media, 2005.

GOSLING J., **JOY B.**, **STEELE G.**, **BRACHA G.**, *The Java Language Specification*, 3rd edition, Addison-Wesley Professional, 2005.