Alkalmazott informatika sorozat

Kövesdán Gábor

Szoftverfejlesztés JAVA SE platformon

SZAK KIADÓ

Az Alkinfo sorozat kötetei

Bányász Gábor - Levendovszky Tihamér:
Linux programozás, 2003.

Albert István, Balássy György, Charaf Hassan, Erdélyi Tibor, Horváth Ádám, Levendovszky Tihamér, Péteri Szilárd, Rajacsics Tamás:
A .NET Framework és programozása, 2004.

Charaf Hassan, Csúcs Gergely, Forstner Bertalan, Marossy Kálmán:
Symbian alapú szoftverfejlesztés, 2004.

Benedek Zoltán, Levendovszky Tihamér:
Szoftverfejlesztés C++ nyelven, 2007.

Balogh Péter, Berényi Zsolt, Dévai István, Imre Gábor, Soós István, Tóthfalussy Balázs:
Szoftverfejlesztés Java EE platformon, 2007.

Forstner Bertalan, Ekler Péter, Kelényi Imre:
Bevezetés a mobilprogramozásba. Gyors prototípus-fejlesztés Python és Java nyelven, 2007.

Gál Tibor:
Interfésztechnikák, 2010.

Ekler Péter, Fehér Marcell, Forstner Bertalan, Kelényi Imre:
Android-alapú szoftverfejlesztés, 2012.

Asztalos Márk, Bányász Gábor, Levendovszky Tihamér:
Linux programozás, Második, átdolgozott kiadás, 2012.

Kövesdán Gábor

Szoftverfejlesztés Java SE platformon

2014

Szoftverfejlesztés Java SE platformon
Kövesdán Gábor

Alkalmazott informatika sorozat

Budapesti Műszaki és Gazdaságtudományi Egyetem
Villamosmérnöki és Informatikai Kar
Automatizálási és Alkalmazott Informatikai Tanszék
Alkalmazott Informatika Csoport

A kötet megjelenését a Nemzeti Kulturális Alap támogatta.

Sorozatszerkesztő: Charaf Hassan
Lektor: Goldschmidt Balázs

ISBN 978-963-9863-35-4
ISSN 1785-363X

SZAK Kiadó Kft. ■ Az 1795-ben alapított Magyar Könyvkiadók és Könyvterjesztők Egyesülésének a tagja ■ 2060 Bicske, Diófa u. 3. ■ Tel.: 36-22-565-310 ■ Fax: 36-22-565-311 ■ www.szak.hu ■ e-mail: info@szak.hu ■ http://www.facebook.com/szakkiado ■ Kiadóvezető: Kis Ádám, e-mail: adam.kis@szak.hu ■ Főszerkesztő: Kis Balázs, e-mail: balazs.kis@szak.hu

Öcsémnek, Tamásnak.
Kívánom, hogy hasznodra váljon a programozás tanulásában.

Tartalomjegyzék

Bevezetés

Manapság a Java nyelv a szoftverfejlesztés egyik legnépszerűbb platformja, amelyet számos területen használnak. Sok osztálykönyvtár és keretrendszer készült a nyelvhez, amelyek segítségével az alkalmazások gyorsan kifejleszthetők. Ráadásul a keretrendszerek nagy része nyílt forráskódú, és ez is nagyban hozzájárul a szakmai közösség kialakulásához és a széleskörű elterjedéshez.

A könyv a Java nyelv mély elsajátításához kíván biztos alapot nyújtani. A könyv hiánypótló műnek készült, jelenleg ugyanis a magyar piacon elérhető Java témájú könyvek nem elégítik ki maradéktalanul az olvasói igényeket. Először is, az elérhető könyvek hatalmas terjedelműek, és ez több szempontból is korlátozást jelent. Egyrészt a programozási tankönyveket nagyrészt felsőoktatási hallgatók forgatják. Általában ők véges szabadidővel rendelkeznek, ugyanakkor szeretnék az anyagot gyorsan, alaposan elsajátítani. Tapasztalatból tudom, hogy több száz oldal feldolgozása nem fér bele az időbe, ugyanis más tantárgyakra is készülni kell. Másrészt, a terjedelem a könyvet fizikailag is nehezen kezelhetővé teszi. Nagyméretű könyveket nehéz magunkkal vinni, és utazás közben olvasni. Ezen kívül a nagy terjedelem az árat is megnöveli, ahogyan ez tapasztalható a piacon elérhető könyvek esetén is. Az olvasók többségének ez is fontos szempont.

A másik probléma az elérhető könyvekkel a stílusuk. Nagyrészt angol nyelvű könyvek fordításaival találkozhatunk. Ezeket ugyan magyar nyelven olvashatjuk, stílusukban mégis az angol nyelvű szakirodalom sajátosságait tükrözik. Az angol szakirodalom történetmesélős, terjengős stílusa nem felel meg a magyar olvasók elvárásainak. Az a tapasztalatom, hogy a magyar műszaki irodalomban a komolyabb, tömörebb és lényegre törőbb megfogalmazásokat szeretjük. Ez mellesleg szintén segít a terjedelem kordában tartásában. A fentiek voltak a fő érvek a könyv elkészítése mellett. Egy mellékes szempont még, hogy nem volt olyan könyv, amely a Java legújabb, 7-es verzióját tárgyalná, pedig már több, mint két éve megjelent. Ráadásul a 6-os verzió terméktámogatását az Oracle már megszüntette, tehát a 7-es az egyetlen hivatalosan támogatott Java-verzió. A könyv a 7-es verzióhoz készült, de általános érvényű, nem elévülő ismeretanyagot ad, a későbbi kiadásokat pedig az aktuális Java-verzió szerint frissíteni fogom.

A könyv célja tehát, hogy jól használható, bő ismeretanyagot átadó, ám tömör kötetben mutassa be a Java nyelv használatát. Sok korszerű témát magában foglal, de a terjedelmi korlátozások miatt általában csak a legkönnyebben használható, magas szintű megoldásokat ismertetjük. Például a JDBC-t és a SAX és DOM szabványokat nem tárgyaljuk részletesen, csak a JPA-t és a JAXB-t. A Java nyelv bemutatását az alapoktól kezdjük, ezért a könyv feldolgozásához nem szükséges semmilyen előismeret a nyelvben. A programozás alapjaival azonban nem foglalkozunk, feltételezzük, hogy az olvasó már tud programozni valamilyen más objektumorientált programozási nyelven. A könyv első öt fejezete mutatja be azokat a nyelvi elemeket és technikákat, amelyeket minden Java programozónak ismernie kell. Az 1. fejezet a nyelv általános adottságait ismerteti. A 2. fejezet a nyelvi elemeket tárgyalja részletesen, mint például a típusok, a változók és a vezérlési szerkezetek. A 3. fejezet a Java nyelv objektumorientált eszköztárát mutatja be, azaz hogyan használhatjuk az objektumorientált technikákat a Java nyelv alatt. A 4. fejezet a szabványos Java osztálykönyvtárat ismerteti, amelynek

segítségével rengeteg feladatot meg tudunk valósítani külső osztálykönyvtárak és keretrendszerek telepítése nélkül is. Az 5. fejezet a generikus osztályokat tárgyalja.

A későbbi fejezetek jelentősen építenek az első öt fejezetre. Ezek sorrendjét úgy választottam meg, hogy folyamatos gondolatmenetet kövessenek, de önállóan is feldolgozhatók legyenek. A 6. fejezet tárgyalja, hogyan menthetjük el a már elkészült alkalmazás állapotát. Megismerkedünk a Properties API-val és a sorosítással is. A 7. fejezet XML-fájlok feldolgozását ismerteti. Az XML lehet az állapotmentés szabványos eszköze, de a ki- és bemenet formátuma is. A 8. fejezet az adatbáziskezelést mutatja be a JPA szabvány segítségével. A 9. fejezet a hálózati kommunikációt mutatja be, főként az RMI protokoll használatával. A 10. fejezet a Swing keretrendszert és a grafikus alkalmazásokat ismerteti. A 11. fejezet a többszálú programozás eszközeit és buktatóit tárgyalja. A 12. fejezet a reflection technikát írja le, amellyel futási időben, dinamikusan deríthetünk fel és érhetünk el osztályokat, objektumokat. A technikával változókat manipulálhatunk, és metódusokat is meghívhatunk, anélkül, hogy az objektum típusát a fejlesztéskor ismernénk. Ez generikus keretrendszerek fejlesztésekor hasznos. A 13. fejezet a naplózás megvalósítását ismerteti. A 14. fejezet leírja, hogyan készíthetünk fel alkalmazásokat különböző nyelvek és kultúrák támogatására. A 15. fejezetben a tesztelést vizsgáljuk meg, végül a 16. fejezet a szoftverek terjesztésével kapcsolatos kérdésekre ad választ. A két függelék a JDK telepítését, valamint az Eclipse fejlesztőkörnyezet használatát ismerteti röviden.

A könyvben olvasható forráskódok esetenként csak az adott részhez szorosan kapcsolódó kódrészeket szemléltetik. A kihagyott kódrészletek helyét három pont (...) jelöli. A könyv formátuma néhol nem engedi meg a hosszú kódrészletek sorhű megjelenítését. Ezekben az esetekben a csupán formai okból beszúrt sortörést a ↵ karakter jelzi. A példaprogramok a kiadó weboldaláról[1] tölthetők le, és könnyen az Eclipse fejlesztőkörnyezetbe importálhatók (lásd B függelék). A parancssoros példákban Windows fájlneveket használok, de természetesen a leírtak érvényesek más operációs rendszerekre is, csupán a megfelelő formába kell átírni a fájlneveket.

A könyv az olvasmányos, élvezhető stílusra törekszik. A hangsúlyt a programozási technikák átadására, és a megfelelő szemlélet kialakítására helyeztem. Ezért igyekeztem kerülni a témák monoton, referenciaszerű ismertetését. Sok esetben a hivatkozott forrás, vagy annak hiányában a Java osztálykönyvtárának Javadoc-dokumentációja[2] szolgál bővebb, referencia jellegű információval. Néhány esetben referenciaszerű részek is olvashatók, de csak ott, ahol ezt szükségesnek ítéltem meg.

A könyv nyelvezetének kialakítása során igyekeztem a gördülékeny magyar hangzást megteremteni, ügyeltem azonban az érthetőségre, és a bevett fordítással nem rendelkező szakkifejezéseket nem fordítottam le. Számos esetben az elterjedt magyar terminológia ellenére is megemlítem zárójelben az angol kifejezést. Véleményem szerint fontos a magyar szakirodalom megteremtése, ugyanis a magyar anyanyelvű olvasók szívesebben olvasnak a saját nyelvükön. Az anyanyelv használata élvezhetőbbé teszi az olvasást, és segíti a jobb megértést, még ha az olvasó jól beszél is angolul. Ennek ellenére úgy gondolom, hogy az informatika nyelve az angol, és minden kedves olvasót bátorítok arra, hogy szakmai nyelvtudását gondozza. A példaprogramok változóneveiben konvencionálisan angol kifejezéseket és rövidítéseket használtam, mert véleményem szerint rossz gyakorlat ettől eltérni. A programokban elhelyezett megjegyzéseket és karakterláncokat azonban a jobb érthetőség kedvért magyarul írtam meg, annak ellenére, hogy a gyakorlatban sosem tennék ilyet.

[1] http://szak.hu/java/peldak.zip

[2] http://docs.oracle.com/javase/7/docs/api/

Szeretném megköszönni kedvesemnek, Cristinának, hogy lelkesített a könyvírás során, és türelemmel viselte az írással töltött estéket. Fuyurnak is köszönöm, hogy dorombolással motiválta a munkát. Köszönet illeti Goldschmidt Balázst a rengeteg hasznos észrevételért, amelyet a könyv szakmai lektorálása során összegyűjtött. Mártonfi Attila, a könyv olvasószerkesztője hasonlóan sok értékes visszajelzést adott a könyv nyelvi megformálásával kapcsolatban. Hálával tartozom Kis Ádámnak, és a SZAK Kiadó munkatársainak, hogy gondoskodtak a könyv kiadásáról és az azzal kapcsolatos adminisztratív teendőkről. Köszönöm továbbá a Budapesti Műszaki és Gazdaságtudományi Egyetem Automatizálási és Alkalmazott Informatikai Tanszékéről Charaf Hassannak és Forstner Bertalannak, hogy helyet adtak a könyvnek az Alkalmazott Informatika sorozatban, valamint Lengyel Lászlónak, hogy biztosította a könyvíráshoz szükséges körülményeket. Végül, de nem utolsó sorban köszönettel tartozom hallgatóimnak, hiszen az oktatás során magam is tanulok, és nélkülük ez a könyv nem jöhetett volna létre.

Remélem, hogy az Olvasó örömét leli a könyv olvasásában, és az elsajátítottak alkalmazásában. Bízom benne, hogy a könyvben közölt ismeretanyag a gyakorlatban is hasznosnak bizonyul.

Budapest, 2014. február

A szerző

ELSŐ FEJEZET
A Java nyelv bemutatása

Mielőtt új programozási nyelv elsajátításába kezdünk, érdemes tisztában lennünk azzal, hogy milyen alapvető sajátosságokkal rendelkezik, és milyen célokra használható. A fejezet célja, hogy röviden ismertesse a Java nyelv legfontosabb ismérveit és gyakori alkalmazási területeit, és ezzel kedvcsinálóként is szolgáljon az Olvasó számára.

1.1. A Java nyelv jellemzői

A Java *általános célú* programozási nyelv. Ez azt jelenti, hogy a nyelv utasításai és a könyvtári komponensek bármilyen algoritmikusan megoldható problémához jól használhatók. A Java ezen kívül *objektumorientált,* azaz az adatokat és a rajtuk végrehajtandó műveleteket osztályok és objektumok segítségével egységbe zárja. A nyelv *típusrendszere statikus és erős,* tehát a változódeklarációknak köszönhetően már a programok lefordításakor könnyen ellenőrizhető a típusbiztosság.

A Java nyelv legjellemzőbb tulajdonsága, hogy a lefordított kódot nem közvetlenül az operációs rendszer, hanem egy *futtatókörnyezet* futtatja. Ezt a speciális programot *virtuális gépnek (virtual machine)* nevezzük. Ez a fogalom különbözik a hétköznapi értelemben használt virtuális géptől. Ez ugyanis nem egy emulációs szoftver, amelyben teljes operációs rendszert futtathatunk, hanem egy szoftverréteg, amely a hordozhatóságot és a biztonságos futást valósítja meg. A Java virtuális gép saját gépi utasításokkal rendelkezik. A Java fordító a Java-programokat nem a célplatform, hanem a virtuális gép utasításkészletére fordítja le. A program futtatásakor a virtuális gép képezi le ezeket az utasításokat a tényleges hardveres utasításkészletre. Ebből következik a programok *hordozhatósága.* Ahhoz, hogy a Java nyelvű programokat más platformon is futtatni tudjuk, csupán a virtuális gép átültetésére van szükség. A Java készítője, az Oracle jelenleg Linux, Windows és Solaris operációs rendszerekhez kínál virtuálisgép-megvalósítást. Mindhárom operációs rendszeren elérhető virtuális gép a szokásos x86-os és x64-es PC-architektúrákhoz, illetve Solaris rendszeren SPARC számítógépeket is támogat az Oracle. A Java-programok hordozhatóságát ezért a *Write Once, Run Anywhere (Egyszer megírni, bárhol futtatni!)* szlogennel szokták jellemezni.

A virtuális gép használatának másik előnye, hogy a programok nem közvetlen a futtató számítógépen hajtódnak végre, ezért nehezebben tudnak kárt okozni. A virtuális gép több különféle védelmi mechanizmust támogat. Ezek segítenek a támadások elleni védekezésben. Mivel a virtuális gép kezeli a futtatandó kódot, az ilyen elven működő programokra sokszor a *felügyelt kód (managed code)* kifejezéssel hivatkoznak.

A Java nyelvre jellemző még az elérhető *osztálykönyvtárak (class library)* széles tárháza. Már az alapértelmezésben feltelepülő könyvtári komponensek is sok problémára kínálnak megoldást, de sok más, nyílt forrású és ingyenesen használható osztálykönyvtár is létezik. Ezek számos gyakori problémára nyújtanak kész meg-

oldást, tehát jelentősen megkönnyítik és felgyorsítják új, összetett alkalmazások kifejlesztését. A magas szintű osztálykönyvtárak és a Java átgondolt kialakítása elősegítik a stabil és hibamentes programok írását. Sokszor ez a szempont fontosabb, mint a program puszta gyorsasága, bár a Java virtuális gép jól finomhangolható, ezért a kész program általában megfelelő teljesítményt nyújt.

1.2. A Java nyelv felhasználási területei

Ugyan a Java általános célú programozási nyelv, mégis kiemelhetünk néhány tipikus, gyakori felhasználási területet. Ehhez először a Java nyelv három változatát tekintjük át. Az általános alkalmazásoknál a *Java Standard Edition* (SE) változatot használjuk. Ez magában foglalja a virtuális gépet és az alapvető funkcionalitásokat megvalósító osztálykönyvtárat. A könyv ezt a változatot ismerteti, a másik kettőről csak röviden lesz szó.

A *Java Enterprise Edition (EE)* a Java SE olyan kibővítése, amely a háromrétegű architektúra kialakítását támogatja. A Java EE speciális komponenseit az alkalmazásszerver kezeli. Ez a Java virtuális gépre épülő, annál több szolgáltatást kínáló futtatókörnyezet. A legalsó réteget a szabadon választott, Java-környezettől független adatbázisszerver képviseli. Erre épül az üzletilogika-réteg, amely megvalósítja az adatbázison elvégezhető műveleteket, és ezeket a legfelső, megjelenítési réteg számára elosztott Java-objektumokon keresztül elérhetővé teszi. Az elosztott objektumokat a Java EE által támogatott *Enterprise JavaBeans (EJB)* szabvány segítségével valósíthatjuk meg. Az alkalmazásszerver az elosztott üzleti komponensek számára middleware-szolgáltatásokat nyújt. A middleware-szolgáltatások az üzleti alkalmazásokban gyakran felbukkanó problémákat oldják meg, ilyen probléma például az adatok perzisztenciája, az aszinkron üzenetkezelés és a munkafolyamatok kezelése. A megjelenítési réteg lehet egy asztali Java SE-alkalmazás vagy a Java EE webes technológiáival megvalósított vékonykliens. A *Servlet* technológia HTTP-kéréseket tud programból kezelni. Ez a gyakorlatban azt jelenti, hogy a felhasználó a böngészőn keresztül lekér egy weboldalt, amelyre a választ a meghívott szervlet állítja elő. A böngészőben visszaadott HTML-oldal azonban nem írható le könnyen Java-kóddal. A Java EE ezért további szabványokat is bevezetett, ilyen a *JavaServer Pages (JSP)*, a *Facelets* és a *JavaServer Faces (JSF)*. Ezek közelebb viszik a fejlesztést a HTML-programozáshoz, és a háttérben a Servlet technológiára épülnek. A Java EE magában foglal még más technológiákat is, itt csak a legfontosabbakat említettük. A Java EE változatot [1] mutatja be részletesen.

A harmadik Java-változat a *Java Micro Edition (ME)*. Ez mobiltelefonokon és egyéb mobil eszközökön használható. Míg a Java SE a Java EE részhalmaza, a Java ME alapvetően különbözik a Java SE-től. A Java ME korlátozottsága miatt manapság már kevésbé használatos. Megemlítjük azonban, hogy az Android operációs rendszerrel rendelkező eszközök programozási nyelve a Javán alapul, de valójában annak egy módosított változata. Az Android által használt Dalvik virtuális gép eltér a szabványos Java virtuális géptől, ezért ezeken a rendszereken a szabványos Java bájtkód nem futtatható. Az Android programozásáról [2] kínál bővebb információt.

1.3. A Java SE-alkalmazások típusai

A három közül a Java SE változata szolgál az általános alkalmazások kifejlesztésére, de ennek segítségével is többféle alkalmazást készíthetünk. Először a normál *asztali alkalmazásokat* érdemes megemlíteni, amelyek ugyanúgy a saját gépünkön futnak, mint a többi felhasználói program. Általában grafikus felhasználói felülettel rendelkeznek. Idetartoznak a szövegszerkesztő programok, a programozási fejlesztőkörnyezetek, a torrentkliensek stb. De olyan program is készült már Javában, amellyel a Nap aktivitását tanulmányozhatjuk.[1] Az is elképzelhető, hogy az alkalmazás nem rendelkezik grafikus felhasználói felülettel, hanem parancssorból futtatható. Ilyenek lehetnek például konverziós programok vagy egyszerű segédprogramok, amelyek nem igényelnek bonyolult felhasználói interakciót. A Java SE-vel együtt települő keytool segédprogram is ilyen. Programok aláírásához használt kulcsokat kezelhetünk vele.

A Java SE-alkalmazások speciális fajtája a *Java-applet*. A Java-applet a böngésző által indított virtuális gépen futó webes alkalmazás. Ehhez a böngészőnek egy kiegészítésre, a Java-pluginra van szüksége. Mivel az applet kódja a Webről érkezik, ezért a Java-plugin ezt alapértelmezésben biztonsági korlátozásokkal futtatja, például tiltja a fájlműveleteket és a távoli géphez való kapcsolódást (kivéve azt a webszervert, amelyről az applet érkezett). A korlátozások miatt a normál Java-appletekkel csak korlátozottabb feladatokat lehet megvalósítani. Ha olyan műveletet szeretnénk elvégeztetni az applettel, amely alapértelmezésben tilos, akkor az appletet digitális aláírással kell ellátni. A hiteles aláírás azt jelzi, hogy az applet készítője vállalja a személyazonosságát, ezért az ilyen appletben biztonsági szempontból megbízhatunk. A Java-plugin az aláírt appleteket biztonsági korlátozások nélkül futtatja. Ha az aláírást olyan kulccsal végezték, amelyet tanúsító hatóság nem hitelesített, akkor nincs garancia a készítő személyazonosságára. Ilyenkor a Java-plugin felugró ablakban kéri a felhasználót az aláírás elfogadására vagy elutasítására.

Az appletek kezdetben többre voltak képesek, mint a natív webes technológiák. Manapság azonban sok webalkalmazás JavaScript- és AJAX-technológiákkal is gazdag funkcionalitást valósít meg. Ezeket a technológiákat a böngésző natívan támogatja, ezért nincs szükség kiegészítő plugin telepítésére, sem digitális aláírásra. Az appletek tehát mára visszaszorultak. Térvesztésük másik oka a *Java WebStart* alkalmazások megjelenése. Az appleteknél szintén nehézséget jelent, hogy ezek implementációs osztályára is megkötések vonatkoznak. A fejlesztésnél az `Applet` osztály leszármazott osztályát kell létrehoznunk, tehát az asztali alkalmazásokat nem tudjuk közvetlenül appletté alakítani. A WebStart technológia ezzel szemben lehetővé teszi, hogy általános Java SE-alkalmazásokat indítsunk el böngészőből. A hálózati indítást a Java Network Launching protokoll (JNLP) valósítja meg, ez `.jnlp` kiterjesztésű fájlból olvassa be a konfigurációs adatokat. A konfigurációs fájl tartalmazza az alkalmazás rövid leírását, a gyártó nevét, a szükséges Java SE-környezet verziószámát, a futtatáshoz szükséges Java-csomagot vagy csomagokat, illetve az alkalmazás belépési pontját. Szintén engedélyezhetjük az alkalmazás automatikus frissítését. A WebStart-alkalmazás ugyanis első futtatáskor települ a számítógépre, de a kapcsolódó adatok, mint az alkalmazás kódjának letöltési helye, eltárolódnak a számítógépen. Ezért, ha az automatikus frissítés engedélyezve van, a WebStart futtatókörnyezet az alkalmazás

[1] JHelioViewer Project: `http://jhelioviewer.org/`

indításakor képes a frissítéseket megtalálni és letölteni. A WebStart-alkalmazások ezeknek a mechanizmusoknak köszönhetően könnyebben kifejleszthetők és használhatók, mint az appletek, ugyanakkor a biztonságra vonatkozó megkötések rájuk is érvényesek. A megkötések feloldásában itt is a digitális aláírás segíthet. Jelenleg a magyarországi elektronikus adóbevalláshoz használható nyomtatványkitöltő program is Java-alkalmazás, amelyet a Nemzeti Adó- és Vámhivatal honlapjáról a WebStart technológiával érhetünk el. A 16.4. alfejezetben bemutatjuk, hogyan tehetjük elérhetővé saját Java-alkalmazásainkat a WebStarton keresztül.

1.4. A Java verziói

A Java nyelvet folyamatosan fejlesztik, hogy lépést tartson az újabb programozási trendekkel és ipari elvárásokkal. A Java SE 7-es, aktuális verziója 2011 júliusában jelent meg. A könyv ezt a változatot mutatja be, de a verziók közti eligazodást segítendő, röviden összefoglaljuk azok számozási rendszerét.

A Java-verziók számozása 1.0-tól indult. Az 1.2-es verzió olyan sok újdonságot hozott, hogy utólag 2-es verziónak nevezték át, ugyanakkor az 1.2-es verziószám is használatban maradt. Ennél a verziónál vált szét a Java a három változatra, és a 2-es verziószám miatt ezekre a *J2SE, J2EE* és *J2ME* rövidítésekkel is hivatkoztak. Ezt követően J2SE 1.3-ról és J2SE 1.4-ről beszélünk, bármilyen meglepő is, hogy egyszerre jelen van a 2-es és az 1.3-as és 1.4-es verziószám. A következő verzió ugyanakkor J2SE 5.0 lett, tükrözve a bevezetett újítások jelentőségét, meghagyva viszont a verziószámok közti zavart. Végül rendezték a verziószámok kétértelműségét, és a következő verziót már Java SE 6 névvel adták ki, majd ezt követte a Java SE 7. Ennek ellenére néhol, például a telepítési mappák nevében, az 1.6 és 1.7 verziószámokkal is találkozhatunk, de ezek az imént említett két verziót jelölik.

A Java EE számozása hasonlóan történik, azaz sokáig a J2EE rövidítést használták a mögé írt, megtévesztő verziószámmal, majd a Java EE 6 megjelenésével rendeződött a helyzet. A Java EE megjelenése mindig az azonos verziójú SE változatot követi jelentős eltéréssel. Míg a Java SE 7 az írás pillanatakor már két éve elérhető, a Java EE 7 éppen csak megjelent.

Fel kell még hívnunk a figyelmet arra, hogy minden Java SE-változat két disztribúcióban érhető el. A *Java Runtime Environment (JRE)* csupán a lefordított Java-programok használatához szükséges futtatókörnyezetet és a lefordított osztálykönyvtárakat tartalmazza, a fejlesztéshez szükséges eszközöket nem. Fejlesztéshez a *Java Development Kit (JDK)* telepítése szükséges, amely a futtatókörnyezeten kívül tartalmazza a fordítót és más fejlesztői segédeszközöket is.

1.5. Termék és szabvány

A Java nyelvet a Sun Microsystems vállalat hozta létre, amelyet később az Oracle felvásárolt. Jelenleg a Java nyelvet tehát az Oracle gondozza. Ő fejleszti és adja ki a nyelv újabb verzióit, a Java nyelvhez terméktámogatást nyújt, tanfolyamokat szervez, illetve vizsgákkal megszerezhető fejlesztői minősítéseket ad. A Java nyelv tehát az Oracle kereskedelmi terméke.

A Java nyelv ugyanakkor szabvány is. A nyelv és az osztálykönyvtár, valamint a virtuális gép specifikációja is teljesen nyilvános, ráadásul ezek nagy része nyílt forrású szoftver is. A kiegészítő keretrendszerek is szabványként vannak specifikálva, és azokat nyílt, közösségi szabványosítási folyamat során dolgozzák ki. Ez a *Java Community Process (JCP)*. Az egyes szabványokat a *Java Specification Request (JSR)* számukkal azonosítjuk. Például a JPA 2.1 szabvány száma JSR 338. A szabványosításnak köszönhetően a Java nyelvből vagy annak részeiből bárki készíthet saját megvalósítást. A hivatalos kiadáson kívül nincs teljes megvalósítás, de egyes Java-szabványokhoz több implementáció is létezik, ilyen például a Java Persistence API (JPA) szabvány (lásd 8.2. alfejezet). Ezek között az alternatív keretrendszerek között is nagyrészt nyílt forrású szoftvereket találunk. A szabványosítás lehetővé teszi, hogy a nyelv egyes részeit alternatív megvalósításokkal lecseréljünk. Ezek a nem szabványosított pontokban el is térhetnek, illetve a szabványon felül extra funkcionalitást is biztosíthatnak. A programozónak ezért lehetőséget adnak a választásra, és a fejlesztett terméket nem teszik függővé egy adott gyártótól.

MÁSODIK FEJEZET
A Java nyelv felépítése

A fejezet a Java nyelv alapelemeit (változók, literálok, oprátorok, kifejezések és utasítások) mutatja be tömören, lényegre törően. A könyv feltételezi, hogy az olvasó már rendelkezik programozási ismeretekkel, ezért a fejezet nem tér ki a fogalmak jelentésére, csak a Java nyelven történő használatukat ismerteti. Szintén nem tárgyaljuk a Java nyelv objektumorientált eszköztárát, mert azt a következő fejezet ismerteti részletesen.

2.1. Pár szó az objektumorientált programozásról

A könyv nem foglalkozik a programozás alapjaival, sem az objektumorientált programozással. Ehhez [3] vagy az alapozó programozási kurzusok adhatnak segítséget. A továbbiakban feltételezzük, hogy az olvasó már rendelkezik alapvető programozási ismeretekkel, és tud objektumorientáltan programozni. Ismétlésként megemlítjük, hogy az objektumorientált nyelveken az adatokat és a rajtuk végzett műveleteket az objektum fogalma zárja egységbe. Az osztály objektumok típusát definiálja, az objektum pedig valamely osztálynak egy példánya. Az osztályokból leszármazott osztályokat is létrehozhatunk. Ekkor a leszármazott osztály állapotot és viselkedést örököl a szülőjétől, a funkcionalitását kiterjeszti, specifikusabbá teszi. Például egy járművet reprezentáló osztályból készíthetünk leszármazott osztályokat, amelyek a járművek közös jellemzőit és viselkedését kiterjesztik, specializálják az egyes konkrét járműtípusoknak megfelelően. Az objektumok tulajdonságain és metódusain kívül létezhetnek az egész osztályra jellemző, ún. statikus tulajdonságok és metódusok is. Ezeket a példányoktól függetlenül is elérhetjük.

A Java nyelvben az `Object` osztály minden osztály közös őse, tehát ettől mindegyik örököl. Ez az osztály néhány alapvetően fontos metódust valósít meg, amelyeket később részletesen is megvizsgálunk (lásd 3.11. alfejezet).

2.2. A Helló, világ! program

A Java nyelv olyannyira objektumorientált, hogy nem is támogatja hagyományos, procedurális programok létrehozását, utasításokat ugyanis csak metódusokban használhatunk. A program belépési pontja ezért egy osztály statikus metódusa, amely konvenció szerint az alábbi szignatúrával rendelkezik:

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

A következőkben megnézzük Java nyelven a szokásos *Helló, világ!* programot. Ez nem csinál mást, csupán kiírja a képernyőre a Helló, világ! a szöveget. A program kódja így fest:

```
public class HelloVilag {
    public static void main(String[] args) {
        System.out.println("Helló, világ!");
    }
}
```

A kódot a `HelloVilag.java` fájlba kell mentenünk, a Java ugyanis megköveteli, hogy fájlonként egy publikusan elérhető osztályt definiáljunk, és a fájl neve egyezzen ennek nevével. Az első sor jelzi, hogy egy publikusan elérhető osztály definíciója következik. Az osztály definícióját kapcsos zárójelben kell megadni. A metódusok definícióját szintén kapcsos zárójelben kell írni. A `main()` metódus most csak egyetlen utasítást tartalmaz, egy másik metódus meghívását. A `System` osztály `out` osztályváltozója `PrintStream` típusú objektumra hivatkozik. A `PrintStream` objektum adatfolyamokat reprezentál, és a `println()` metódusával írhatunk a folyamba. A `System` osztálytól olyan példányt kapunk, amely az aktuális kimeneti adatfolyamhoz van rendelve, tehát a `println()` metódusnak átadott szöveg a kimenetre fog íródni. Megfigyelhetjük a programban, hogy a tagváltozóra és a metódusra való hivatkozást is a pont operátorral végezzük, az osztályhoz vagy objektumpéldányokhoz tartozó tagváltozók és metódusok esetén egyaránt. Szintén látható, hogy az utasításokat pontosvesszővel zárjuk. Megjegyezzük azt is, hogy a program tördelésének nincs jelentősége. A kulcsszavakat természetesen legalább egy szóközzel kell elválasztani, hogy azok egymástól megkülönböztethetők legyenek, de tetszőlegesen beszúrhatunk további szóközöket, sortöréseket és tabulátorokat, hogy a programkódot olvashatóbbá tegyük. A forráskód formázására konvencionális ajánlások is léteznek.

A programot lefordíthatjuk az Eclipse fejlesztőkörnyezettel (lásd B függelék) vagy parancssorból a következőképpen. Mindkét esetben feltételezzük, hogy a JDK 7-es verziója már telepítve van (lásd A függelék).

```
javac HelloVilag.java
```

Ekkor egy `HelloVilag.class` nevű fájlnak kell létrejönnie az aktuális könyvtárban. A program szintén futtatható az Eclipse programban vagy parancssorból:

```
java -cp . HelloVilag
```

Itt a `-cp .` opció jelzi a Java futtatókörnyezet számára, hogy az osztályhoz tartozó `.class` fájl az aktuális könyvtárban van. A parancs kiadása után a képernyőn a *Helló, világ!* szöveget kell látnunk.

2.3. A megjegyzések

A megjegyzések a programban elhelyezett szövegek, amelyek nem befolyásolják a működését, de segítik a forráskód későbbi megértését. A megjegyzések ismertetése azért került a fejezet elejére, mert a későbbi példákban ezek fogják mutatni, hogy egy adott programrészlet mit eredményez.

A jó programozási gyakorlat megkívánja, hogy a forráskódban használjunk megjegyzéseket a nehezen érthető részek előtt. Ez azonban nem pótolja azt, hogy a forráskódot is olvashatóan, az általánosan elfogadott programozási gyakorlatnak megfelelően írjuk. Az is fontos, hogy magától értetődő részeket ne lássunk el megjegyzésekkel, mert az csak nehezíti a megértést.

A Java nyelv kétféle szintaxist kínál a megjegyzések írásához. Az egysoros megjegyzéseket két perjel után írjuk. A megjegyzés az első perjelnél kezdődik, és a sor végén fejeződik be. Elhelyezhető kódsor elején vagy végén is:

```
// Sor elején kezdődő egysoros megjegyzés

a = b + c; // Ez kódsor végén van, de inkább ne is magyarázzuk.
```

Lehetőségünk van többsoros megjegyzések elhelyezésére is, ezt a `/*` és a `*/` jelek közt tehetjük meg. Néha egysoros megjegyzéshez is használjuk, ha nagy hangsúlyt akarunk annak adni:

```
/*
   Az alábbi kód beolvassa a kapcsolódási beállításokat a fájlokból,
   majd kapcsolódik a szerverhez, és letölti a friss adatokat,
   amelyekkel dolgozni fogunk.
*/

/*
   Ez egy egysoros, de fontos megjegyzés.
*/
```

A többsoros megjegyzések egy speciális változatát a `/**` és a `*/` jelek közt adjuk meg. Ezekből a Javadoc technológiával fejlesztői dokumentáció állítható elő. Ezt a 16.1. alfejezet ismerteti részletesen.

2.4. Az azonosítók

Mostantól rátérünk a Java programozási nyelv alapelemeinek átfogó és részletes ismertetésére. Ezt a nyelvi azonosítókkal kezdjük. Azonosítón az általunk deklarált változók és definiált osztályok, metódusok, konstansok és enumerációk nevét értjük, amellyel később hivatkozhatunk rájuk. Az azonosítókra érvényes megkötések a következőképpen foglalhatók össze:

- Az első karakter betű, dollárjel ($) vagy aláhúzásjel lehet. Használhatók ékezetes betűk is.
- A második karaktertől kezdődően ugyanezeket a karaktereket használhatjuk, illetve használhatunk számokat is.
- Az azonosító hossza nincs korlátozva.
- Az azonosító nem lehet foglalt szó, és nem kerülhet ki a `null`, `true` és `false` literálok közül sem.

Az azonosítók érzékenyek a kis- és nagybetűkre. A foglalt szavak a következők:

```
abstract    continue    for         new         switch
assert      default     if          package     synchronized
boolean     do          goto        private     this
break       double      implements  protected   throw
byte        else        import      public      throws
case        enum        instanceof  return      transient
catch       extends     int         short       try
char        final       interface   static      void
class       finally     long        strictfp    volatile
const       float       native      super       while
```

Annak ellenére, hogy az azonosítókra kevés megkötés van, szinte kivétel nélkül az alábbi konvenciókat használjuk:

- Az azonosítókban nem használunk ékezetes betűket, csak az angol ábécé betűit.
- Az osztályok, az interfészek és az enumerációk neveit nagybetűvel kezdjük, és a névben minden következő új szót nagybetűvel kezdünk, például: `AbstractMessageHandler`.
- A változók és a metódusok nevét kisbetűvel írjuk, de az új szavakat nagybetűvel kezdjük, például: `isSynchronized()` vagy `messageHandler`.
- A konstansok (`final` módosítóval deklarált változók) neveiben csak nagybetűket használunk, például: `PI`.

2.5. A csomagok

A Java nyelvben ún. csomagokat használhatunk arra, hogy az osztályainkat logikai csoportokba osszuk. Ez megkönnyíti a kód későbbi megértését, illetve annak modulonként való terjesztését. Például egy adatbázis-kezelő rendszernek külön csomagokba kerülhetnének a fájlelréssel, a felhasználói interakcióval, illetve a hálózati adatforgalommal kapcsolatos részei. A csomag a névütközés elkerülését is szolgálja. Azonos csomagban nem létezhetnek egyező nevű osztályok, interfészek és enumerációk, de ha más csomagban vannak, akkor lehet ugyanaz a nevük. Ebben az esetben a csomagnév segítségével adható meg, melyik osztályra, interfészre vagy enumerációra hivat-

kozunk. A csomagok további tárgyalása során csak osztályokat említünk, de a leírtak az interfészekre és az enumerációkra is érvényesek.

A csomagot, amelybe egy osztály tartozik, az osztályt definiáló forrásfájlban adjuk meg a package kulcsszó után. Ha nem adunk meg csomagnevet, akkor az osztály az alapértelmezett (névtelen) csomaghoz tartozik. Ha megadunk csomagot, akkor azonban annak a legelső utasításnak kell lennie, csak megjegyzések előzhetik meg.

A csomagnév azonosítójára is a már tárgyalt megkötések vonatkoznak. A csomagneveket hierarchiába is rendezhetjük, ekkor a név részeit pontokkal választjuk el egymástól, például server.messaging. Fontos, hogy csak a nevek hierarchikusak, a csomagok maguk nem. Tehát a server.messaging csomagban található osztályok nem részei a server csomagnak. Ez a láthatóság (lásd 2.6.8. alfejezet) témakörnél lesz majd fontos. Itt találhatunk néhány példát csomagnév-deklarációkra, de természetesen ezek nem szerepelhetnek ugyanabban a fájlban:

```
package server;
package server.messaging;
package client.net;
package client.net.tcp;
package client.gui;
```

A fájlok csomagokba való szervezését a könyvtárstruktúrának is tükröznie kell, amelyben a forrásfájlokat tároljuk. A server csomag fájljai tehát a server könyvtárban, a server.messaging csomag fájljai pedig a server\messaging könyvtárban találhatók.

A programokban az osztályokra hivatkozhatunk a teljes vagy a rövid nevükkel. A teljes név a csomag nevéből, egy elválasztó pontból és az osztály nevéből áll. A rövid név csak az osztály nevét tartalmazza. Ahhoz, hogy más csomagban található osztályokra a rövid nevükkel hivatkozzunk, a csomagot a használat előtt importálni kell, kivéve a java.lang csomag osztályait, ezek automatikusan importálódnak. Importálásra az import kulcsszó szolgál. Az importálandó csomagokat az opcionális package kulcsszó után, de a fájlban definiált osztály előtt kell felsorolni. Ha az importált csomagok között van névütközés, akkor az adott osztályokra mindig a teljes nevükkel kell hivatkozni.

Az import kulcsszót kétféleképpen használhatjuk. Hivatkozhatunk konkrét osztályra a teljes nevével, vagy importálhatunk egész csomagot is, ha a teljes névben az osztály helyett csillagot adunk meg. A következő példa importálja a client.net csomag összes osztályát. A client.net.tcp csomag osztályai nem importálódnak, mivel a csomagoknak csak a névtere hierarchikus, köztük nincs tartalmazási kapcsolat. Egy másik import utasítással viszont a client.net.tcp.TCPConnection osztályt is megadtuk:

```
import client.net.*;
import client.net.tcp.TCPConnection;
```

A kódban az osztályra a csomaggal megkülönböztetett nevével is hivatkozunk, ekkor viszont nem kell importálni. Ez terjengőssé teheti a kódot, ezért a gyakorlatban csak akkor használjuk, ha az importált csomagok tartalmaznak egyező nevű osztályokat.

Ebben az esetben csak így tudjuk egyértelműen megkülönböztetni őket, azonos nevű osztályok importálása ugyanis fordítási hibát eredményez.

Az import speciális típusát képezi a statikus import. Ezzel osztályváltozókat és metódusokat tudunk importálni, és azokra ezután egyszerűen a nevükkel hivatkozni, nem szükséges az osztálynév kiírása. Ilyen importhoz az `import static` kulcsszót használjuk. A statikus importnak két típusa van: importálhatunk egyetlen osztályváltozót vagy metódust, illetve egyszerre is importálhatjuk egy osztály összes osztályváltozóját és metódusát. Előbbi esetben az osztály után ponttal megadjuk a konkrét osztályváltozót vagy metódust, utóbbi esetben csillagot írunk helyette. Ha több azonos nevű osztályváltozót vagy metódust importálunk, akkor fordítási hibát kapunk. Az alábbi példa importálja a `Math` osztály összes osztályszintű konstansát és metódusát, ez gyakori matematikai műveletek elvégzését támogatja. Itt a csillagos jelölést használjuk. Szintén importáljuk a `System.out` osztályváltozót, amely egy `PrintStream` típusú objektum, és a szabványos kimenetre való írást teszi lehetővé:

```
import static java.lang.Math.*;
import static java.lang.System.out;
```

2.6. A változók és a literálok

A Java nyelvben változót bárhol deklarálhatunk, nem szükséges az osztályok, a metódusok vagy az utasításblokkok elején megtennünk. Ez lehetővé teszi, hogy a változókat közvetlen az első használat előtt deklaráljuk. Mivel a Java statikusan és erősen típusos nyelv, ezért a deklarációban meg kell adnunk a változó típusát, hogy a fordító a kifejezésekben szereplő operandusok típuskompatibilitását megfelelően ellenőrizni tudja. A deklaráció a típusból és a névből, valamint egy opcionális kezdőérték-adásból áll, ebben egyenlőségjel után adjuk meg a kezdeti értéket. A kezdeti érték lehet literál vagy egy kifejezés értéke is. A metódusokban deklarált változókat olvasás előtt kezdő értékkel kell incializálni, különben a program nem fordul le. Osztály- vagy példányváltozó esetén az inicializáció nem kötelező. Ha nem adunk meg kezdő értéket, akkor a változó a típusától függően `0`, `null` vagy `false` értékkel inicializálódik. Néhány példa változódeklarációra:

```
int limit = 5;
short i;
double pi = 3.1415;
double a = 45.0 / 16.0;
Button button = new JButton("OK");
```

A változóknak utólag is adhatunk értéket:

```
i = 15;
```

A Javában a típusokat alapvetően két nagy csoportba oszthatjuk: egyszerű típusokra és objektumreferenciákra. Az alábbiakban részletesen megvizsgáljuk őket.

2.6.1. Az egyszerű típusok

Az egyszerű típusok közé tartoznak a különféle egész- és lebegőpontos típusok, valamint a logikai típus. Ezek közös jellemzője, hogy nem objektumként reprezentálja őket a nyelv, és metódushíváskor érték szerint adódnak át.

Az egésztípusok a nekik megfelelő intervallumon képesek egész értékek tárolására. Tárolási hosszuk minden architektúrán egyértelműen elő van írva. A Java nyelv nem definiál külön előjeles és előjel nélküli módosítókat, egy típus kivételével mindegyik típus előjeles. Az előjeles számábrázolás a memóriában kettes komplemens alakban történik. Vigyázni kell a túl-, illetve alulcsordulásra, mert sok más nyelvhez hasonlóan a Java sem nyújt ezek ellen védelmet. A 2.1. táblázat összefoglalja az egésztípusokat.

2.1. táblázat: A Java nyelv egésztípusai

Név	Bithossz	Minimumérték	Maximumérték
byte	8	-128	127
short	16	-32 768	32 767
int	32	-2 147 483 648	2 147 483 647
long	64	-9 223 372 036 854 775 808	9 223 372 036 854 775 807
char	16	0	65 535

A char típus kivételnek tekinthető, mert valójában karakterek tárolására alkalmazzuk. Mivel a Java a Unicode szabvány UTF-16 kódolása szerint tárolja a karaktereket, ezért azok 16 bites értékekkel írhatók le. A karakterliterálokat aposztrófok közé írjuk. A billentyűzetről közvetlen nem begépelhető karaktereket a hexadecimális Unicode-kódjuk segítségével tudjuk megadni a \u karaktersorozat után. A gyakran használt karakterekhez könnyebben megjegyezhető escape-szekvenciák is használhatók, ezeket a 2.2. táblázat foglalja össze.

2.2. táblázat: Escape-szekvenciák a Java nyelvben

Escape-szekvencia	Karakter
\b	backspace
\t	vízszintes tabulátor
\n	soremelés
\f	lapdobás
\r	kocsivissza
\"	idézőjel
\'	aposztróf
\\	backslash

A karakterekkel ellentétben a karakterláncok a Java nyelvben nem tartoznak az egyszerű típusok közé, ezért őket később tárgyaljuk. Az alábbi programrészlet mutatja a karakterek használatát:

```
char fkjel = '!';
System.out.println("Szép napot kívánok" + fkjel + '\u0021');
```

A literálként bevitt egész számok alapértelmezésben `int` típusúak. A mögéjük írt `l` vagy `L` segítségével tehetjük őket `long` típusúvá.

Az egészek megadhatók decimális, hexadecimális, oktális és bináris formában is. Alapértelmezésben decimálisak. A hexadecimális megadás a `0x` vagy `0X` előtaggal kezdődik, és az A–F számjegyek is írhatók kis- vagy nagybetűkkel. Az oktális egész egy extra `0`-val kezdődik, a bináris megadás előtagja pedig `0b` vagy `0B`.

A Java SE 7-es verziójától kezdődően a számokban szerepelhetnek aláhúzásjelek (_), ezek azonban nem változtatják meg a számok jelentését, csupán azok tagolását segítik. Segítségükkel a megszokott hármas tagolás szerint írhatjuk le a nagy értékeket, vagy a bitmező jellegű adatok egyes komponenseit különíthetjük el. A következő rövid programrészlet szemlélteti az egésztípusú változók és literálok használatát, valamint a korábbi példaprogramban látott módszerhez hasonlóan kiírja azokat a szabványos kimenetre. A kimenetre írás részletei később válnak majd teljesen érthetővé. Alkalmaztunk többféle megadási módot, és némelyik számot aláhúzásjelekkel is tagoltuk. A kiírás során azonban ezek is decimálisan jelennek meg, hiszen a program számára a változó vagy a literál csupán az értéket tárolja, a többféle megadási mód csak a programozó munkáját könnyíti meg. Ha a változókat és a literálokat más formában akarjuk kiírni, akkor a kiírási formátumot is meg kell adnunk. Ezt a 2.6.3. alfejezet és a 14.1.1. alfejezet ismerteti. A példában a + operátor a szöveghez fűzi hozzá a szám szöveges reprezentációját. Az operátort a fejezet későbbi részében tárgyaljuk.

```
byte b = 12;
System.out.println("b = " + b);

short s = 345;
System.out.println("s = " + s);

int i = 5_666_777;
System.out.println("i = " + i);
System.out.println("Egy decimálisan megadott integer literál: " + ↺
1_234_567);
System.out.println("Egy hexadecimálisan megadott integer literál: " ↺
+ 0xFF_AA);
System.out.println("Egy oktálisan megadott integer literál: " + ↺
0664);
System.out.println("Egy binárisan megadott integer literál: " + ↺
0b1111_0000);

long l = 999_999_999_000_000l;
System.out.println("l = " + l);
System.out.println("Egy long literál: " + 123_456_789_000L);
System.out.println("Egy hexadecimálisan megadott long literál: " + ↺
0xFF_FF_FF_FFL);
```

A lebegőpontos számokhoz a `float` és a `double` típusokat használhatjuk. Mindkét típus az *IEEE-754* szabvány szerinti lebegőpontos aritmetikát követi. Az előbbi 32 bites, egyszeres pontosságú típus, az utóbbi 64 bites, dupla pontosságú. A típusok a normál lebegőpontos értékeken kívül tárolhatnak pozitív és negatív nullát, pozitív és negatív végtelent, valamint egy speciális, ún. `NaN` értéket. Ez érvénytelen műveletek végeredményeként áll elő, mint például a nullával való osztás. A 2.3. táblázat összefoglalja a lebegőpontos típusokat.

2.3. táblázat: A Java nyelv lebegőpontos típusai

Név	Bithossz	IEEE-743 pontosság	Speciális értékek
float	32	egyszeres	+0, −0, +∞, −∞, NaN
double	64	dupla	

A lebegőpontos literálokban mindig tizedespontot használunk, és a tizedestört-részt követheti egy opcionális exponens, amelyet az `e` vagy `E` karakter jelöl. Utána a kitevő következik, ez lehet pozitív, negatív vagy nulla is. Itt is használhatjuk az aláhúzásjellel történő tagolást, az értékek viszont csak decimálisan adhatók meg. A literálok alapértelmezésben `double` típusúak, `float` típusút `f` vagy `F` utótaggal adhatunk meg. A pozitív és negatív nulla is bevihető literálokkal, utóbbinál azonban figyelni kell arra, hogy `-0.0f` vagy `-0.0` alakban adjuk meg. A `-0` ugyanis `int` típusú literál. Ebben a típusban nem lehet negatív nullát ábrázolni, ezért értéke nulla ellentettje, azaz egyszerűen nulla lesz. Ez automatikusan konvertálódik `float` vagy `double` típusra, ha az adott kontextusban erre van szükség, a kapott érték azonban nem az elvárt lesz. Erre máshol is figyelni kell, például az osztást tartalmazó kifejezéseknél, mert az egészeken elvégzett osztás maradékos osztásként megy végbe.

A pozitív és a negatív végtelen, valamint a `NaN` értékek bevitelére a `Float` és a `Double` osztályokban definiált `POSITIVE_INFINITY` , `NEGATIVE_INFINITY` és `NaN` konstansokat használhatjuk. Ezekről az osztályokról bővebben a 2.6.3. alfejezetben szólunk. A következő programrészlet mutatja a lebegőpontos változók és a literálok használatát.

```
float f = 3.14f;
System.out.println("A pi értéke: " + f);
System.out.println("Az Euler-szám: " + 2.72F);

double d = 6.022e23;
System.out.println("Az Avogadro-szám: " + d);
System.out.println("A Boltzmann-állandó J/K mértékegységgel: " + ↵
1.380_650_424E-23);
System.out.println("Megadhatunk -0-t is:" + -0.0);
System.out.println("Vagy plusz végtelent: " + Double.↵
POSITIVE_INFINITY);
System.out.println("Vagy akár NaN-t: " + Double.NaN);
```

A logikai típus neve `boolean`, és ahogyan neve is tükrözi, a Boole-logika igaz és hamis értékeit tudja tárolni. Ezeket a `true` és a `false` literálokkal reprezentáljuk. A következő programlészlet szemlélteti használatukat.

```
boolean igaz = true;
boolean hamis = false;

System.out.println("Ez igaz lesz: " + igaz);
System.out.println("Ez pedig hamis: " + hamis);
```

2.6.2. A referenciatípusok

A típusok másik csoportját a referenciatípusok képviselik. A referenciatípusú változók objektumra hivatkoznak, illetve felvehetnek egy speciális, semmire sem hivatkozó `null` értéket. A változó deklarációjában interfészt vagy osztályt adunk meg típusnak, ez lesz a referencia *statikus típusa*. A változó olyan objektumokra hivatkozhat, amelyek osztálya megfelel a statikus típusnak. Ez konkrétan azt jelenti, hogy az osztály vagy valamelyik őse implementálja a statikus típusként megadott interfészt, vagy pedig leszármazott osztálya a statikus típusként megadott osztálynak. Az objektum tényleges típusát *dinamikus típusnak* nevezzük. A 3. fejezet ismerteti bővebben az osztályok hierarchiáját.

A referenciatípusok, ahogyan nevük is tükrözi, metódushíváskor cím szerint adódnak át. A Java nyelvben a karakterláncok objektumok, a `String` osztály példányai. Az osztály a karakterláncok tárolásán kívül néhány metódust is kínál, amelyekkel hasznos, karakterláncokhoz kapcsolódó funkcionalitásokat érhetünk el. A `String` objektum is példányosítható konstruktorhívással a már említett módon, de ebben az esetben a Java kényelmesebb jelölést is kínál. Az idézőjelekbe zárt karaktersorozatok karakterlánc-literálokat jelölnek, ezek mögött a háttérben egy `String` objektum áll. A karakterlánc-literálokat használhatjuk bárhol, ahol karakterláncokra van szükség, akár metódust is hívhatunk rajtuk. A karakterlánc-literálokban is alkalmazhatók a karaktereknél látott jelölések a közvetlenül be nem gépelhető karakterek bevitelére. A következő példa a karakterláncokkal mutatja be a referenciatípusú változók használatát.

```
String elso = "alma";
String masodik = new String("korte");
String harmadik = null;

System.out.println("elso: " + elso);
System.out.println("masodik: " + masodik);
System.out.println("harmadik: " + harmadik);
```

2.6.3. A csomagolóosztályok

A Java a primitív típusokhoz ún. csomagolóosztályokat is nyújt. Ezek gyakran használt funkcionalitást kínálnak a metódusaikon keresztül, illetve maguk is alkalmasak a reprezentált primitív típusnak megfelelő érték tárolására. A primitív típusok helyett tehát akár ezeket is használhatjuk. A 2.4. táblázat felsorolja ezeket az osztályokat:

2.4. táblázat: A primitív típusok csomagolóosztályai

Primitív típus	Csomagolóosztály
boolean	Boolean
byte	Byte
char	Character
double	Double
float	Float
int	Integer
long	Long
short	Short

A fenti csomagolóosztályok mindegyike rendelkezik olyan konstruktorral, amely a primitív típussal megadott értéket várja, illetve a `Character` osztály kivételével olyannal is, amelynek az érték karakterlánc-reprezentációja adható meg. Ha ennek olyan karakterláncot adunk meg, amely érvénytelen értéket reprezentál, akkor `NumberFormatException` kivétel váltódik ki (lásd 3.12. alfejezet). Az így példányosított csomagolóobjektum értéke az *xxx*`Value()` metódussal kapható meg a primitív típusban, ahol *xxx* a primitív típus neve. A `Character` és a `Boolean` osztályoktól csak a hozzájuk tartozó primitív típusnak megfelelő értéket kaphatjuk meg, a számokat reprezentáló osztályoktól viszont az összes többi számtípus szerinti értéket is. Ekkor a metódus az értéket az adott típusra konvertálja, viszont ilyenkor a pontosság csökkenhet. Például egy `Double` objektum által reprezentált érték nem biztos, hogy ábrázolható a `short` típussal, de még a `float` típusra alakításkor is veszíthet pontosságából. Az alábbi példákon láthatjuk a konstruktorhívást és az *xxx*`Value()` metódust:

```
Integer i1 = new Integer(5);
Integer i2 = new Integer("6");
Character c = new Character('c');

int i4 = i1.intValue(); // 5
double d1 = i1.doubleValue(); // 5
```

A fentiek alapján tudunk konvertálni primitív típusok és csomagolóosztályaik közt, erre azonban ritkán van szükség. A Java 5.0-ás verziójától kezdve a fordító ugyanis ezeket a konverziókat automatikusan elvégzi, és ha szükséges, akkor a megadott értéket *becsomagolja (boxing)* egy objektumba, vagy a csomagolóobjektumra hivatkozó referenciából *kicsomagolja (unboxing)* a primitív értéket. Mivel a csomagolóobjektumok használata költségesebb, mint a primitív típusú változóké, a virtuális gép a primitív típusok csomagolóobjektumaiból egy tárat tart fent, és automatikusan újrafelhasználja őket. Ha a becsomagolt literál vagy primitív változó `boolean` vagy `byte` típusú, ha `char` típusú és értéke `\u0000` és `\u007f` közé esik, illetve ha `int` vagy `short` típusú és értéke `-128` és `127` közé esik, akkor a becsomagolás elvégzése minden esetben ugyanazt a példányt adja vissza. A szabvány megengedi az ettől eltérő típusú vagy a megadott tar-

tományokon kívül eső értékek csomagolóobjektumainak a gyorstárazását is. Az alábbi példa mutatja be a becsomagolást és a kicsomagolást. Megfigyelhetjük, hogy a csomagolóobjektumnak primitív változó is értékül adható, és primitív változót is inicializálhatunk csomagolóobjektummal. Az azonos literálokat a fordító ugyanabba az objektumba csomagolja, de ha a csomagolóosztály konstruktorát hívjuk, akkor másik példány jön létre.

```
Integer i5 = 1; // boxing
int i6 = i5;  // unboxing
Integer i7 = 1;
Integer i8 = new Integer(1);
System.out.println(i5 == i7); // true
System.out.println(i5 == i8); // false
```

A csomagolóosztályokat főként az általuk nyújtott kiegészítő funkcionalitás miatt használjuk. A továbbiakban a `Character` és a `Boolean` típusoktól eltekintünk, és csak a számok csomagolóosztályait tárgyaljuk. Ezek sokféle konverziós műveletet támogatnak, ezeket statikus metódusokként teszik elérhetővé. A `parseXxx()`, ahol *Xxx* a primitív típus neve, karakterláncból képes beolvasni egy decimálisan leírt értéket, és primitív típussal adja vissza. A `valueOf()` vagy primitív típust vagy karakterláncot vár, és a reprezentált értéket egy csomagolóobjektumban tárolja el. A karakterláncot ez is decimálisan ábrázolva várja. Ha a megadott karakterlánc számként nem értelmezhető, akkor mindegyik metódus `NumberFormatException` kivételt vált ki. Valójában az automatikus be- és kicsomagolás miatt mindkét metódus eredményét értékül adhatjuk primitív típusú változónak és csomagolóobjektumnak is, de a be- és kicsomagolásnak költsége van, ezért ajánlatos a megfelelő metódust alkalmazni. Alább látható néhány példa:

```
double d2 = Double.parseDouble("15.5e2");
Double d3 = Double.valueOf(d2);
Double d4 = Double.valueOf("3.1415");
```

Egésztípusoknál a `parseXxx()` és a `valueOf()` is rendelkezik olyan változattal, amellyel a második paraméterben megadhatjuk, hogy a karakterlánc a számot milyen számrendszerben ábrázolva tartalmazza. Szintén egésztípusoknál használható a `decode()`, ez csak egy karakterlánc paramétert vár, de felismeri a Java literálok `0x`, `0X` hexadecimális és `0` oktális prefixumait is. Ez a metódus csomagolóobjektumban adja vissza az eredményt.

Hasznos lehet még a `toHexString()` metódus, amellyel a primitív típusokat karakterláncban kapjuk meg hexadecimálisan ábrázolva. Egész számoknál használható a `toBinaryString()` és a `toOctalString()` is, ezek rendre binárisan, illetve oktálisan adják vissza a számot:

```
long l1 = Long.parseLong("10", 2); // dec. 2
Long l2 = Long.valueOf("ff", 16); // dec. 255
Long l3 = Long.decode("0xff");  // dec. 255
System.out.println("hex: " + Long.toHexString(l3)); // ff
System.out.println("oct: " + Long.toOctalString(l3)); // 377
System.out.println("bin: " + Long.toBinaryString(l3)); // 11111111
```

A primitív típusok nem vehetnek fel tetszőlegesen nagy vagy kis értéket. A csomagolóosztályok a `MAX_VALUE` és a `MIN_VALUE` konstansokban tárolják el a felső és alsó korlátokat. A `SIZE` konstansból az adott típus bithossza olvasható ki. Lebegőpontos típusoknál rendelkezésünkre áll a `MAX_EXPONENT` és a `MIN_EXPONENT` konstans is, ezek az exponens rész korlátait tárolják.

Ugyan nem csomagolóosztályok, de jó szolgálatot tehetnek a `java.math` csomagban található `BigInteger` és a `BigDecimal` osztályok. Ezek tetszőleges pontosságú egész, illetve lebegőpontos számok ábrázolására szolgálnak. A szokásos aritmetikai és logikai műveleteket a metódusaik segítségével támogatják. A primitív típusok nemcsak korlátozott pontosságúak, de túl is csordulhatnak, és ez nehezen észrevehető hibákat eredményezhet. Ha ez gondot okoz, akkor megfontolhatjuk ezen osztályok használatát is. Ehhez a Java 7 Javadoc-referenciája adhat segítséget, itt bővebben nem tárgyaljuk használatukat.

2.6.4. A tömbök

A tömbök adott típusú változókból tárolnak többet, azokat egy logikai egységként kezelve. Felfoghatjuk a tömböt úgy, mint több rekeszből álló polcot, amelynek minden rekesze azonos méretű. A tömb mérete, vagyis a rekeszek száma azonban rögzített méretű, a létrehozás után már nem változtatható meg. Készíthetünk kétdimenziós tömböt is, ez azt jelenti, hogy minden egyes rekesz néhány továbbira van felosztva. Ez táblázatos ábrázolással is szemléltethető. A dimenziók számát tetszőlegesen növelhetjük, de később a jelentés már nem lesz ilyen szemléletes, és a gyakorlatban sem szokás kettőnél több dimenziójú tömböt használni. Az egydimenziós tömböt *vektornak*, a kétdimenziósat *mátrixnak* is nevezzük.

Egydimenziós tömböt úgy deklarálhatunk, hogy vagy a típus, vagy a változónév után üres szögleteszárójel-párt írunk. Javasolt ezt a típusnév után írni, mivel a tömbjelleg a típus részének tekinthető. Többdimenziós tömbnél a dimenziószámnak megfelelő zárójelpárt írunk. A tömböt ezután létre kell hozni, ezt a `new` operátorral tehetjük meg, ezt a típusnév és az utána szögletes zárójelben megadott elemszám követi. Ez a képzeletbeli rekeszek számát jelenti.

Ez a lépés létrehozza a tömböt, és kezdetben annak típustól függően minden eleme `0`, `false` vagy `null`. Az értékeket most már elérhetjük és módosíthatjuk. A tömb elemeinek számozása `0`-tól indul, a változónév után szögletes zárójelbe írt index megadásával hivatkozhatunk rájuk. Az utolsó használható index tehát a megadott elemszámnál eggyel kisebb. Ha ennél nagyobb indexet használunk, `ArrayIndexOutOfBoundsException` kivétel váltódik ki. A tömbök Java nyelven az objektumok speciális fajtáját képezik, tehát rendelkeznek néhány tagváltozóval és metódussal. Például a `length` tagváltozó tárolja a tömb méretét. Nézzünk erre egy példát:

```
String[] strVector;

strVector = new String[3];
strVector[0] = "Valami";
System.out.println("0: " + strVector[0]); // "Valami"
System.out.println("1:" + strVector[1]); // null
System.out.println("length: " + strVector.length); // 3
```

A kétdimenziós tömb valójában olyan tömb, amelynek elemei is tömbök. Ez azt jelenti, hogy először a „külső" tömböt hozzuk létre, majd végigmenve az elemein, mindegyiket inicializáljuk egydimenziós tömbként:

```
int[][] intMatrix = new int[2][];
intMatrix[0] = new int[2];
intMatrix[1] = new int[2];
intMatrix[0][0] = 5;
```

A tömböt a létrehozáskor azonnal is inicializálhatjuk. Ekkor nem kell megadni a méretét, mert az a felsorolt elemek számából következik. Az elemeket kapcsos zárójelben, vesszővel elválasztva soroljuk fel:

```
int[] uj = new int[] { 1, 2 };
System.out.println(uj.length); // 2
```

2.6.5. A változó hosszú paraméterlisták

Gyakran szükség van rá, hogy egy metódusnak értékek olyan sorozatát adjuk át, amelynek elemszáma előre nem ismert. Például ha a programból több címzettnek szeretnénk emailt küldeni, akkor készíthetünk egy metódust, amely az email címek alapján mindenkinek elküldi az üzenetet. Kézenfekvő és működőképes megoldás, ha a címeket a metódusnak tömbként adjuk át, ehhez azonban a címeket akkor is tömbbe kell szerveznünk, ha nem így állnak rendelkezésre. A változó hosszú paraméterlista használata lehetővé teszi, hogy a változó számú értékeket felsorolva is megadhassuk. Az értékeknek azonos típusúnak kell lenniük. A típus lehet primitív- vagy referenciatípus, akár tömb is. A változó hosszú lista mellett állandó paramétereket is megadhatunk a metódus szignatúrájában, de a változó hosszú listának a paraméterlista végén kell szerepelnie. Megadása abban különbözik a többi paramétertől, hogy a típus után három pont (...) szerepel. A változó hosszú lista tömbként járható be a metódusban. Az alábbi metódus felhasználóknak küld emailt. A szöveget és a tárgyat állandó paraméterben, a címzetteket pedig változó hosszú paraméterlistában veszi át.

```
public void sendMessage(String text, String subject, String... ↺
addresses) {
    for (String email : addresses) {
        ...
    }
}
```

A fenti metódus kétféleképpen hívható. A harmadik paraméterben megadhatunk tömböt, vagy az értékeket egyenként, a harmadik, negyedik, ... paraméterben is átadhatjuk. A következő példa szemlélteti a két hívási módot.

```
String[] rcpt = new String[] { "user1@example.com",
    "user2@example.com" };
sendMessage("Helló!", "Teszt 1", rcpt);

sendMessage("Helló!", "Teszt 2", "user1@exmaple.com",
    "user2@example.com");
```

A típus után írt három pont csak metódusok paraméterlistájában használható, máshol fordítási hibát eredményez.

2.6.6. Az enumerációk

Az enumeráció olyan típus, amelynek példányai a programozó által felsorolt értékeket vehetik fel. A gyakorlatban ez igen hasznos, például rendszerek állapotainak vagy egy választási helyzetben a lehetséges alternatíváknak a reprezentálására használható. Az enumeráció definiálásához az enum kulcsszó után meg kell adnunk a nevét, majd kapcsos zárójelben soroljuk fel a típus által megengedett értékeket vesszővel elválasztva. Konvenció szerint ezeket az értékeket csupa nagybetűvel írjuk.

Enumerációval változót úgy deklarálunk, hogy típusnak az enumeráció nevét adjuk meg. A felvethető értékek literáljai az enumeráció nevéből, majd a ponttal elválasztott értékből állnak. A következő programrészlet szemlélteti az enumerációkról elmondottakat:

```
public class Main {

    enum Napszak {
        REGGEL, DEL, DELUTAN, ESTE;
    }

    public static void main(String[] args) {
        // jelenlegi dátum és idő lekérése
        Date d = Calendar.getInstance().getTime();
        int ora = d.getHours(); // óra
        Napszak n;

        // napszak meghatározása
        if (ora < 12)
            n = Napszak.REGGEL;
        else if (ora < 13)
            n = Napszak.DEL;
        else if (ora < 19)
            n = Napszak.DELUTAN;
        else
```

```
            n = Napszak.ESTE;

        System.out.println("Napszak: " + n);
    }
}
```

Az enumerációk valójában speciális osztályok, és ennél összetettebb funkcionalitással is rendelkeznek. Ezt a 3.13. alfejezet tárgyalja.

2.6.7. A void kulcsszó

A Java nyelvben nincs típus nélküli változó, mint a C és C++ nyelvekben. Olyan metódusok azonban előfordulnak, amelyeknek nincs visszatérési értékük. Ennek jelzésére a visszatérési érték típusa helyén a `void` kulcsszót szerepeltetjük.

2.6.8. Az életciklus és a láthatóság

A konzervatívabb programozási nyelvekkel ellentétben a Javában nem szükséges, és nem is lehet a referenciák által hivatkozott objektumpéldányokat felszabadítani. A Java virtuális gép *szemétgyűjtő (garbage collector, GC)* komponenssel rendelkezik, és ez figyeli, hogy mely objektumpéldányokra létezik referencia. A már nem hivatkozott példányokat automatikusan felszabadítja. Ezeket az objektumokat ugyanis rájuk mutató referencia hiányában már lehetetlen elérni a programból, tehát biztosan nincs rájuk szükség. A Java nyelv ezzel a mechanizmussal igyekszik elkerülni a más programozási nyelvek esetén sokszor tapasztalt *memóriaszivárgást (memory leak)*, illetve a memóriafoglaló és -felszabadító metódusok hibás használatából eredő programhibákat. Az objektumok példányosítása úgy történik, hogy a `new` operátor segítségével meghívjuk az osztály konstruktorát. Ezután a referencián keresztül tudunk hivatkozni az objektumra, és azzal műveleteket végezhetünk. Ha már egyetlen referencia sem hivatkozik az objektumra, akkor az alkalmassá válik a szemétgyűjtésre, és a szemétgyűjtő bármikor eltávolíthatja. Azt azonban nem tudjuk, hogy ez mikor fog megtörténni, vagy egyáltalán megtörténik-e. A szemétgyűjtő a háttérben fut, és programozóként csak korlátozott beleszólásunk van a működésébe.

Változókat több helyen is deklarálhatunk. Ez azt is befolyásolja, hogy a változó meddig fog létezni, azaz meddig terjed az *életciklusa (lifecycle)*, illetve a program mely részein *látható (scope)*. Az első lehetséges típusba az objektumok példányváltozói tartoznak. Ezeket az osztálydefinícióban adjuk meg, életciklusuk az objektum példányosításától annak szemétgyűjtéséig tart. Az osztályban deklarálhatunk osztályváltozókat is, ezek az osztály betöltődésekor (első hivatkozáskor) jönnek létre, és egészen addig élnek, amíg az osztályt használjuk. Mindkét típus láthatóságáról a 3. fejezet fejezetben lesz szó.

Metódusokban is deklarálhatunk változókat, ezek életciklusa és láthatósága csak a deklaráció helyétől a metódus végéig tart. A nevük megegyezhet a metódus osztályában deklarált osztály- vagy példányváltozókéval. Ilyenkor a lokális változó *elfedi (shadowing)* őket, és a név automatikusan a lokális változót fogja jelenteni. Ha mégis az osztályváltozóra kell hivatkozni, akkor ki kell írni az osztály nevét, majd a pont operátorral hivatkozhatunk az osztályváltozóra. Példányváltozó esetén a `this`

kulcsszót használhatjuk, ez az aktuális objektumpéldány referenciáját jelenti. A referencia segítségével tagváltozóként már hozzáférhetünk a változóhoz.

A metódusok rendelkezhetnek paraméterváltozókkal is. Ezek azok a változók, amelyekben a metódus a paramétereket kapja meg a meghívásakor. A paraméterváltozók szintén elfedhetik a osztály- és példányváltozókat, de nevük nem egyezhet meg a lokális változókéval. A paraméterváltozók életciklusa és láthatósága a metódus kezdetétől a végéig terjed.

A kapcsos zárójelben megadott utasításblokkokban szintén deklarálhatunk lokális változót. Ezek is elfedhetik az osztály osztály- és példányváltozóit. Életciklusuk és láthatóságuk az utasításblokkra korlátozódik.

2.6.9. A konstansok

A Java nyelv valójában nem ismeri a konstansok fogalmát. A változók azonban a `final` módosítóval csak olvashatóvá tehetők. Az ilyen változót a kezdőértékadás után nem lehet megváltoztatni, ezért a Java-zsargon konstansoknak nevezi őket, még ha a nyelv nem is különbözteti meg őket élesen a többi változótól. A könyv is ezt a gyakorlatot követi. Ha a konstansokat osztályban és nem metódusban definiáljuk, akkor általában `public static` módosítókkal is megjelöljük őket (lásd 3. fejezet).

2.7. A kifejezések és az operátorok

Ebben a fejezetben áttekintjük, hogyan tudunk kifejezéseket létrehozni literálokból és már deklarált változókból.

2.7.1. Az aritmetikai operátorok

Az aritmetikai operátorok operandusai számok, és a belőlük alkotott aritmetikai kifejezések értéke is szám. A Java nyelvben is megtalálható a négy alapművelet operátora, az osztás (`/`) azonban maradékos osztásként működik, ha mindkét operandus egésztípusú. Ez azt jelenti, hogy az eredmény törtrésze eldobódik, kerekítés azonban nem történik. Ha hagyományos osztást szeretnénk alkalmazni egésztípusú változókon dolgozunk, akkor az egyiket lebegőpontossá kell konvertálni, például így:

```
(double)a / b
```

Ha literáljaink vannak, akkor az egyiket írjuk lebegőpontos alakban:

```
5.0 / 3
```

A `%` maradékképzésre szolgál. Megtalálhatjuk a C és C++ nyelvekből ismerős növelő és csökkentő operátorokat is. Ezek a többivel ellentétben egyoperandusú operátorok, és eggyel növelik vagy csökkentik a tagváltozóban tárolt értékeket. Bár használhatók lebegőpontos változókkal is, főleg egész típusú számlálóknál gyakori a használatuk. Létezik prefix és posztfix alakjuk is, előbbi a kifejezést már a növelés vagy csökkentés után értékeli ki, utóbbi csak a kiértékelés után növel vagy csökkent.

Fontos tudni, hogy az egészeken végzett aritmetikai műveletek értéke mindig `int` vagy `long` típusú. Ez két dolgot jelent. Egyrészt az osztás sem vezet ki az egész számok halmazából, ugyanis két egész operandus esetén az maradékos osztást jelent. Másrészt, két `byte` vagy `short` operandus esetén az eredmény `int` típusú lesz, még ha az eredmény elférne is az eredeti típusban. Ilyenkor az eredményt megfelelő körültekintés után konvertálhatjuk (lásd 2.7.9. alfejezet). A 2.5. táblázat összefoglalja a Java nyelv aritmetikai operátorait.

2.5. táblázat: A Java aritmetikai operátorai

Operátor	Típus	Jelentés	Példa
+	infix	összeadás	a + b
-	infix	kivonás	a - b
*	infix	szorzás	a * b
/	infix	maradékos osztás	a / b (a és b egész)
/	infix	osztás	a / b (a és b lebegőpontos)
%	infix	maradékképzes	a % b
++	prefix	kiértékelés előtti inkrementálás	++a
++	posztfix	kiértékelés utáni inkrementálás	a++
--	prefix	kiértékelés előtti dekrementálás	--a
--	posztfix	kiértékelés utáni dekrementálás	a--

Az alábbi programrészlet példákkal szolgál az aritmetikai kifejezések használatára:

```
int a = 5;
int b = 3;
float c = 3.0f;

System.out.println("a : " + a);
System.out.println("b : " + b);
System.out.println("c : " + c);
System.out.println("a + b : " + (a + b));
System.out.println("a - b : " + (a - b));
System.out.println("a * b : " + (a * b));
System.out.println("a / b : " + (a / b));
System.out.println("(double)a / b : " + ((double)a / b));
System.out.println("a / c : " + (a / c));
System.out.println("a % b : " + (a % b));
System.out.println("a++ : " + (a++));
System.out.println("++b : " + (++b));
```

2.7.2. Az előjeloperátorok

A Java szintén rendelkezik a matematikai pozitív és negatív előjeleknek megfelelő + és - prefix operátorokkal. Az értelmezésük teljesen megfelel a matematikai konvenciónak. Előbbit gyakorlatilag nem használjuk, mivel a literálok előjel nélkül megadva is pozitívak, kifejezésen alkalmazva pedig nincs hatása. Használata akkor lehet indokolt, ha egy literál pozitív előjelét hangsúlyozni akarjuk, például `+5.0`. A - operátort használjuk negatív számok literálként való megadásakor, például `-5.0`. Kifejezések előtt használva azok eredményét az ellentettjére változtatja.

2.7.3. Az összehasonlító operátorok

Az összehasonlító operátorok `boolean` típusú értéket adnak vissza. Mindig kétoperandusúak és infixek, a megadott két operandus között fejeznek ki valamilyen relációt, és attól függően adnak igaz vagy hamis eredményt, hogy a reláció teljesül-e. Egy részük csak számokon használható, ilyenek a kisebb és nagyobb relációk, valamint az egyenlőséget is megengedő változatuk: <, >, <=, >=.

Az egyenlőség vizsgálatára az == szolgál. Az operátor alkalmazható különböző típusú számok között, egyébként az összehasonlított értékek típusának egyeznie kell. A == operátor ellenpárja a `!=`, ez akkor ad igaz értéket, ha az operandusok nem egyenlők. Ezek az összehasonlítások objektumok esetén referenciális egyenlőségre vonatkoznak, tehát két referenciatípusú operandus akkor egyenlő, ha ugyanarra az objektumpéldányra hivatkozik. Ha létrehozunk egy másik objektumpéldányt ugyanabból a típusból, és összes tagváltozóját ugyanarra az értékre állítjuk, attól még az egyenlőség nem fog teljesülni, hiába hordozzák ugyanazt a jelentést. Az ilyen egyezés vizsgálatára más módszert kell alkalmaznunk (lásd 3.10. alfejezet). A `String` objektumok megvalósítása érdekes példát mutat erre. Mint láttuk, használhatunk literálokat a programban, és ezeket a fordító objektumpéldánnyal helyettesít. Akárcsak a csomagolóobjektumokat, a Java nyelv a `String` objektumokat újrafelhasználja, mindegyik ismétlődő literálhoz egy objektumpéldány készül. Ha tehát a literál egyenlőségét vizsgáljuk önmagához képest, akkor igazat kapunk. Más eredményre jutunk azonban, ha a `String` osztály konstruktorát hívjuk meg ugyanazzal a szöveggel, és a literált ezzel a példánnyal hasonlítjuk össze. A két karakterlánc hiába tartalmazza ugyanazt a szöveget, mégsem ugyanarra az objektumpéldányra hivatkozik. Karakterláncok esetén gyakorlatilag sosem a referenciális egyenlőségre van szükségünk, ezért jól jegyezzük meg, hogy karakterláncokat ne az == operátorral hasonlítsunk össze!

A számok esetén is találunk néhány furcsaságot. Az még nem is meglepő, hogy a `+0.0` és `-0.0` értékek egyenlőnek számítanak, viszont a pozitív és negatív végtelen nem. Az sokkal inkább figyelemre méltó, hogy a `NaN` érték önmagával sem egyenlő. Ez definíció szerint egy hibás érték, azaz azt jelöli, hogy nem lehetett értelmes eredményt meghatározni, ezért tehát az ilyen eredmény valamivel való egyenlőségéről beszélni értelmetlen. Ha meg akarjuk vizsgálni, hogy egy érték `NaN`-e, akkor használhatjuk a `Float`, illetve a `Double` osztályok statikus `isNaN()` metódusát. Az alábbi programrészlet bemutatja az összehasonlító operátorok használatát:

```
System.out.println("3 > 5 : " + (3 > 5));
System.out.println("3 < 5 : " + (3 < 5));
System.out.println("3 >= 5 : " + (3 >= 5));
System.out.println("3 <= 5 : " + (3 <= 5));
System.out.println("3 == 5 : " + (3 == 5));
System.out.println("3 != 5 : " + (3 != 5));
System.out.println("3 == 3.0f : " + (3 == 3.0f));
System.out.println("3 == 3.0 : " + (3 == 3.0));

System.out.println("+0.0 != -0.0 : " + (+0.0 != -0.0));
System.out.println("+0.0 == -0.0 : " + (+0.0 == -0.0));
System.out.println("Double.POSITIVE_INFINITY == Double.↺
NEGATIVE_INFINITY : " + (Double.POSITIVE_INFINITY == Double.↺
NEGATIVE_INFINITY));
System.out.println("Double.NaN == Double.NaN : " + (Double.NaN == ↺
Double.NaN));
System.out.println("Double.isNaN(Double.NaN) : " + (Double.isNaN↺
(Double.NaN)));

String a = "Hello";
String b = "Hello";
System.out.println("a == b : " + (a == b));
System.out.println("a == new String(\"Hello\") : "
  + (a == new String("Hello")));
System.out.println("a.equals(new String(\"Hello\") :" + (a.equals↺
(new String("Hello"))));

File elso = new File("fajlnev");
File masodik = new File("fajlnev");
System.out.println("elso == masodik : " + (elso == masodik));
```

2.7.4. A bitenkénti operátorok

A bitenkénti operátorok az egészeken alkalmazhatók, és bináris értékük bitjeit módosítják. Az *és, vagy* és *kizáró vagy* operátorokkal két számon bitenként végezhetjük el ezeket a műveleteket. Az operátorokat rendre az &, a | és a ^ karakterek jelölik. Az egyoperandusú ~ operátor pedig egy szám bitenkénti negáltját adja vissza:

```
int i1 = 0b11110000;
int i2 = 0b11001100;

int es = i1 & i2;
int vagy = i1 | i2;
int kvagy = i1 ^ i2;
int neg = ~i1;

System.out.println("i1:    " + Integer.toBinaryString(i1));
```

```
System.out.println("i2:    " + Integer.toBinaryString(i2));

// 11000000
System.out.println("es:    " + Integer.toBinaryString(es));
// 11111100
System.out.println("vagy:  " + Integer.toBinaryString(vagy));
// 111100
System.out.println("kvagy: " + Integer.toBinaryString(kvagy));
// 11111111111111111111111100001111
System.out.println("neg:   " + Integer.toBinaryString(neg));
```

A bitenkénti operátorok másik csoportjába tartoznak a léptető operátorok. A << az első operandus bitjeit a második operandusként megadott helyiértékkel balra tolja. A jobb oldalon belépő bitek mindig nullák. Ez azzal egyenértékű, mint ha az első operandust megszoroztuk volna kettőnek a második operandusra emelt hatványával. Természetesen az eredmény túcsordulhat, ekkor a magas helyiértékű bitek elvesznek. A >> ugyanígy működik, de az értékeket jobbra tolja, ez kettő hatványaival való osztásnak felel meg. Az operátor figyelembe veszi az előjelbitet, tehát valóban osztást végez. Más szóval, a léptetés során a bal oldalon belépő bitek függenek az első operandus előjelétől. A >>> operátor nem foglalkozik az előjelbittel, hanem mindig nullákat léptet be a bal oldalon:

```
int i3 = -55;
int i4 = i3 << 1;
int i5 = i3 >> 1;
int i6 = i3 >>> 1;

// 11111111111111111111111111001001
System.out.println("i3 :  " + Integer.toBinaryString(i3));
// 11111111111111111111111110010010
System.out.println("<< :  " + Integer.toBinaryString(i4));
// 11111111111111111111111111100100
System.out.println(">> :  " + Integer.toBinaryString(i5));
// 11111111111111111111111111100100
System.out.println(">>>:  " + Integer.toBinaryString(i6));
```

Felvetődhet a kérdés, hogy a balra léptető operátorból miért létezik csak egyféle. A válasz, hogy a kettes komplemens számábrázolásból adódóan a balra tolás egészen addig előjelhelyesen működik, amíg a léptetett szám túl nem csordul, ezért a kettő hatványaival való szorzás egybeesik a mechanikus léptetéssel.

2.7.5. Az értékadó operátorok

Példákból már láttuk, hogy egy változónak értéket az = operátorral adhatunk. Az értékadásnak azonban vannak egyéb műveletekkel kombinált formái is, és ezek rövidebbé, egyszerűbbé teszik a kódot. Például az `a += b` kifejezés eredményeképpen a értéke b-vel nagyobb lesz. A kifejezés teljesen egyenértékű az `a = a + b` kifejezéssel. Ilyen kombinált értékadás használható az összes aritmetikai és bitenkénti operátorral. A 2.6. táblázat felsorolja ezeket a rövidítéseket.

2.6. táblázat: A Java értékadó operátorai

Kifejezés	Rövidített forma
`a = a + b`	`a += b`
`a = a - b`	`a -= b`
`a = a * b`	`a *= b`
`a = a / b`	`a /= b`
`a = a % b`	`a %= b`
`a = a & b`	`a &= b`
`a = a ^ b`	`a ^= b`
`a = a | b`	`a |= b`
`a = a + b`	`a += b`
`a = a << b`	`a <<= b`
`a = a >> b`	`a >>= b`
`a = a >>> b`	`a >>>= b`

2.7.6. A logikai operátorok

A logikai operátorok logikai operandusokon vannak értelmezve, és az általuk alkotott kifejezés értéke is logikai. Idetartoznak az *és,* a *vagy,* illetve a *kizáró vagy* operátorok, amelyeket rendre az &, a | és a ^ karakterekkel jelölünk. Ezeket a karaktereket használtuk a bitenkénti műveleteknél is, de az operandusok itt más típusúak. Az operandusok típusa határozza meg tehát , hogy pontosan mit is jelentenek:

```
boolean igaz = true;
boolean hamis = false;

System.out.println("igaz & hamis: " + (igaz & hamis));
System.out.println("igaz | hamis: " + (igaz | hamis));
System.out.println("igaz ^ hamis: " + (igaz ^ hamis));
System.out.println("igaz & igaz: " + (igaz & igaz));
System.out.println("hamis ^ hamis: " + (hamis ^ hamis));
```

Az *és* esetén ha az első operandus hamis értékű, a második operandus kiértékelése nélkül is megállapítható, hogy a kifejezés értéke nem lehet igaz. Néhány programozási nyelvnél ilyenkor a második operandus ki sem értékelődik, ezért ha az mellékhatásokkal rendelkező kifejezés, akkor a mellékhatásai sem érvényesülnek. Ezt a jelenséget *rövidzár-kiértékelésnek (short circuit evaluation)* nevezzük. Hasonló a helyzet a *vagy* operátorral: ha az első operandus igaz, akkor a kifejezés értéke mindenképp igaz lesz. A fenti operátorok *nem* a rövidzár-kiértékelés szerint működnek, tehát a második operandus esetleges mellékhatásai mindig érvényesülni fognak. Ennek ellenére a mellékhatással rendelkező kifejezések használata nem javasolt logikai kifejezésben, mivel nehezen átlátható hibákhoz vezethet.

A Java nyelv rendelkezik az *és*, illetve a *vagy* operátorok olyan változatával is, amelyek rövidzár-kiértékelést alkalmaznak. Ezeket az && és a || jelöléssel érhetjük el:

```
int i = 4;
boolean b = true || (i++ >= 5);
System.out.println("b: " + b); // true
System.out.println("i: " + i); // 4

b = true | (i++ >= 5);
System.out.println("b: " + b); // true
System.out.println("i: " + i); // 5
```

Az egyetlen egyoperandusú logikai operátor a negáció. Ez mindössze ellentettjére fordítja az operandusának az értékét. Ennek jele a !:

```
System.out.println(!true); // false
System.out.println(!false); // true
```

2.7.7. A feltételes operátor

A feltételes operátor a Java egyetlen háromoperandusú operátora, és *a* ? *b* : *c* alakú. Az *a* mindenképpen logikai kifejezés, *b* és *c* lehet tetszőleges típus, de azonosak, vagy egymásnak megfeleltethetők. Ez azt jelenti, hogy elegendő, ha típusbővítéssel (lásd 2.7.9. alfejezet) kapunk azonos típust, valamint a `null` literál bármilyen referenciával kompatibilis. Ha *a* igaz, akkor a kifejezés értéke *b* lesz, különben *c*. A feltételes operátor nagyon hasznos, mert használatával elkerülhetők a rövid `if` utasítások (lásd 2.8.2. alfejezet). Ez tömörebbé és ezért jobban olvashatóvá teszi a kódot. Ha azonban a feltételek bonyolultak, akkor inkább ne erőltessük a feltételes operátort, mert éppen ellenkező hatást érhetünk el vele. Tipikusan ilyen az egymásba ágyazott feltételes operátorok esete, ez ugyanis igen áttekinthetetlen lehet. Az alábbi példaprogram szemlélteti a feltételes operátor használatát:

```
// aktuális dátum és idő
Date d = Calendar.getInstance().getTime();
System.out.println("Jó " + (d.getHours() > 11 ? "napot"
    : "reggelt") + " kívánok!");
```

2.7.8. Az objektumokkal kapcsolatos operátorok

Több különféle operátor létezik a Java nyelvben, amelyeket objektumreferenciákon használunk. Ezeket ebben az alfejezetben tekintjük át.

Rögtön az objektumok példányosításánál találkozhatunk ilyennel, ez pedig a new operátor. A new operátorral konstruktort hívhatunk, amelynek a neve egyezik az osztály nevével, és esetleg paramétereket is kaphat. Ezeket kerek zárójelben adjuk meg. Ha nincsenek paraméterek, az üres zárójelpárt akkor is ki kell tenni. Ahogyan korábban láttuk, a tömbök létrehozása is ezzel az operátorral történik. Az alábbi kód létrehoz egy File objektumot, ez egy (nem feltétlenül létező) fájlt reprezentál:

```
File f = new File("teszt.txt");
```

Miután létrehoztunk egy objektumpéldányt, általában műveleteket végzünk rajta. Kiolvashatjuk, illetve módosíthatjuk a tagváltozóit, valamint meghívhatjuk a metódusait. Mind a tagváltozók, mind a metódusok elérése a pont operátorral történik. Metódushíváskor a metódus neve után meg kell adni a paraméterlistát, ha pedig nincs paraméter, akkor az üres zárójelpárt. A metódushívás a zárójelek miatt tehát mindig egyértelműen megkülönböztethető a tagváltozó elérésétől. Tagváltozókat egyébként ritkán érünk el közvetlenül, hanem az objektumorientált programozás irányelveinek megfelelően getter és setter metódusokat alkalmazunk. A következő kódon megfigyelhetjük a metódushívásokat:

```
// ha nem létezik fájl, akkor létrehozunk egy üreset
if (!f.exists())
    f.createNewFile();
// beállítjuk, hogy írható legyen
f.setWritable(true);
```

A statikus, más néven osztályszintű metódusok és tagváltozók elérése is a . operátorral történik, de ebben az esetben az eléréshez általában az osztálynevet használjuk. Használhatunk tetszőleges objektumpéldányt is, de az osztályváltozó valójában az osztályhoz tartozik, ezért logikusabb ez a hivatkozás. Erre a fordító figyelmeztet is:

```
// osztályváltozó
System.out.println("elérésiút-elválasztó karakter: " + File.↺
pathSeparator);
// példányon keresztül is elérhető, de figyelmeztetést kapunk
System.out.println("Példányon keresztül is elérhető: " + f.↺
pathSeparator);
```

A this kulcsszó az aktuálisan futó metódus objektumpéldányára ad vissza referenciát. Ez használható például akkor, ha egy metódushívásban az objektumot szeretnénk paraméterben átadni. Szintén a this kulcsszó használata szükséges, ha a futó metódus vagy konstruktor lokális vagy paraméterváltozójának ugyanaza a neve, mint egy tagváltozó. Akkor ugyanis az előbbi elfedi a tagváltozót, és arra csak az objektumreferencia segítségével tudunk hivatkozni:

```
public class Main {
    String str = "Példányváltozó";

    public void teszt() {
        String str = "Lokális változó";
        // lokális változó
        System.out.println("this nélkül: " + str);
        // példányváltozó
        System.out.println("thisszel: " + this.str);
    }
```

Végül, az `instanceof` operátor arra használható, hogy megvizsgálja, hogy egy referencia kompatibilis-e egy adott osztállyal vagy interfésszel. A kompatibilitás azt jelenti, hogy a referencia típusa az osztályhierarchiában az adott típusból származik, azaz közvetlenül vagy közvetetten leszármazottja annak. A `null` érték minden típussal kompatibilis. Ha a megadott osztály nem ős- vagy leszármazott osztálya a változó deklarált típusának, akkor az operátor használata fordítási hibát eredményez. A változó ugyanis nem is tudná ilyen objektum referenciáját tárolni. Alább láthatunk példákat az operátor használatára:

```
// mindhárom true
System.out.println("str instanceof Object: " + (str instanceof ↺
Object));
System.out.println("str instanceof CharSequence: " + (str ↺
instanceof CharSequence));
System.out.println("str instanceof String: " + (str instanceof ↺
String));
// le sem fordul; File nem leszármazott osztálya a Stringnek
// System.out.println("str instanceof File: " + (str instanceof ↺
File));
// szintén true
System.out.println("null instanceof File: " + (null instanceof ↺
File));
```

2.7.9. A típuskonverziós operátor

A *típuskonverziós operátor (type cast)* segítségével egy kifejezés tartalmát más típusként érhetjük el. A kívánt új típust a kifejezés előtt kapcsos zárójelben adjuk meg.

Először a primitív típusok konvertálását vizsgáljuk. A típust bővíthetjük vagy szűkíthetjük. Előbbi azt jelenti, hogy az értéket bővebb típusra kovertáljuk, azaz olyanra, amelynek az értékkészlete bővebb az eredeti típusénál. Ezért az érték mindig probléma nélkül ábrázolható, és a konverziót a fordító automatikusan el is végzi, ha szükség van rá. Például, ha `long` vagy `float` érték helyett `int` típusút adunk meg, akkor a programunk bármiféle hiba vagy figyelmeztetés nélkül is lefordul, és működni fog. Típusszűkítésen az ellenkező irányú konverziót értjük, tehát ekkor nem bizonyos, hogy az eredmény pontosan ábrázolható az új típussal. Mégis elképzelhető olyan eset, amelynek során lehet értelme az ilyen típusú konverziónak. Emlékezzünk arra, hogy

az egészeken végzett aritmetikai műveletek eredménye mindig legalább `int` típusú. A következő program ezért konvertálás nélkül nem is fordulna le:

```
short a = 4;
short b = 5;
short c = (short) (a + b);
```

Referenciák konvertálásakor az osztályhierarchiát és a referencia típusának ebben elfoglalt helyét kell megvizsgálni. Mivel a hierarchia fastruktúrában ábrázolható, megkülönböztetünk *felfelé konvertálást (upcast)*, amikor általánosabb típusra konvertálunk, és *lefelé konvertálást (downcast)*, amikor konkrétabb típusra történik a konverzió. Ez tulajdonképpen megfelel a primitív típusoknál látott bővítésnek és szűkítésnek. Nem meglepő tehát, hogy a felfelé konvertálás automatikusan történik, mivel a jobban specializált típus példánya mindig példánya lesz az általánosabb típusnak is („minden bogár rovar"). Fordítva természetesen ez nem igaz („nem minden rovar bogár"), de néha tudjuk, hogy a referencia olyan objektumra hivatkozik, amely specializáltabb a referenica típusánál. Ekkor lehet értelme a lefelé konvertálásnak. A konvertálás előtt az `instanceof` operátorral ellenőrizhetjük, hogy a referencia által hivatkozott objektum ténylegesen kompatibilis-e a kívánt új típussal. Ha nem kompatibilis, és mégis konvertálni próbáljuk, akkor futásidőben `ClassCastException` kivétel váltódik ki. Természetesen olyan típusra semmiképpen sem konvertálhatunk, amely a fa másik ágán van, azaz a referencia típusának se nem leszármazott típusa, se nem őse. Az ilyen próbálkozás még csak le sem fordul.

A lefelé konvertálást alkalmazó megoldások használata sérti a polimorfizmus elvét, ezért ha túl sokszor van rá szükség, akkor érdemes megvizsgálni, hogy a forráskód átszervezhető-e úgy, hogy jobban kövesse az objektumorientált programozás irányelveit. A következő példa bemutatja a konvertálás használatát:

```
// automatikus felfele konvertálás
CharSequence r1 = new String("bla");

String r2;
// nem fordulna le, mert lefele kell konvertálni
// r2 = r1;

// így jó, de ha nem ismernénk r1 pontos típusát, akkor
// ellenőrizni is kellene:
if (r1 instanceof String)
    r2 = (String) r1;

// nem fordulna le, mert már fordításkor is bizonyos,
// hogy r1-ben nem lehetett File-példány
// File f = (String) r1;
```

2.7.10. A karakterlánc-műveletek

Szigorúan véve a karakterláncokon csak egyféle operátor, az összefűzés (+) van értelmezve. Az egyik operandus lehet primitív típus vagy referencia is. Ekkor a primitív típus értéke vagy referencia esetén az objektum `toString()` metódusa által visszaadott szöveges reprezentáció fűződik hozzá a karakterlánc-operandushoz. Az utóbbi metódust az `Object` osztály definiálja, ezért minden objektum esetén működik, még ha a visszaadott szöveges reprezentáció nem is mindig a legmegfelelőbb.

A `String` osztály metódusaival is végezhetünk néhány további karakterlánc-műveletet. A `String` objektumok által reprezentált szöveg azonban később már nem változtatható meg. Ezért például a `toLowerCase()` metódus, amely csupa kisbetűssé alakítja a karakterláncot, az átalakított karakterláncot egy új objektumban adja vissza. Alább felsoroljuk a `String` osztály legfontosabb metódusait.

`char charAt(int index)`
Visszaadja az adott sorszámú karaktert.

`boolean contains(CharSequence s)`
Megvizsgálja, hogy a karakterlánc tartalmazza-e a megadott karaktersorozatot.

`boolean endsWith(String suffix)`
Megvizsgálja, hogy a karakterlánc a megadott karakterláncra végződik-e.

`boolean equals(Object anObject)`
Megvizsgálja, hogy az átadott objektum ugyanazt a karaktersorozatot tartalmazó karakterlánc-e.

`boolean equalsIgnoreCase(String anotherString)`
Ellenőrzi, hogy a karakterláncok a kis- és nagybetűktől eltekintve egyeznek-e. A metódus a magyar nyelv ékezetes karaktereivel is helyesen működik.

`boolean isEmpty()`
Igazat ad vissza, ha a karakterlánc üres.

`int length()`
Visszaadja a karakterlánc karakterekben mért hosszát.

`String replace(char oldChar, char newChar)`
Új karakterláncot ad vissza, amelyben az első karakter összes előfordulását kicseréli a második karakterrel.

`String replace(CharSequence target, CharSequence replacement)`
Új karakterláncot ad vissza, amelyben az első karaktersorozat összes előfordulását kicseréli a második karaktersorozattal.

`boolean startsWith(String prefix)`
Megvizsgálja, hogy a karakterlánc a megadott karakterlánccal kezdődik-e.

`String substring(int beginIndex)`
Visszaadja a megadott indextől kezdődő részkarakterláncot.

`String substring(int beginIndex, int endIndex)`
Visszaadja a megadott indexek közé eső részkarakterláncot. Az utolsó index már nem tartozik bele az eredménybe.

`String toLowerCase()`
Visszaadja a kisbetűssé konvertált karakterláncot.

`String toLowerCase(Locale locale)`
Visszaadja az aktuális lokalizáció szerint kisbetűssé konvertált karakterláncot.

`String toUpperCase()`
Visszaadja a nagybetűssé konvertált karakterláncot.

`String toUpperCase(Locale locale)`
Visszaadja az aktuális lokalizáció szerint nagybetűssé konvertált karakterláncot.

`String trim()`
Visszaadja az adott karakterláncot kezdő és záró szóközök nélkül.

Az értékadó operátoroknál megismert += operátorral a bal oldalán álló `String` típusú referenciához fűzhetünk hozzá. A fenti metódusokhoz hasonlóan erre az operátorra is igaz, hogy nem a korábbi karakterlánc módosul, hanem új példány jön létre, és ez kerül a bal oldali referenciába. Alább láthatunk néhány példát a metódusok használatára:

```
String s1 = "JAVA 7 SE";
String s2 = "Java 7 SE";

// nullától kezdődik
System.out.println("s1.charAt(5): " + s1.charAt(5));
// kisbetű/nagybetű különbség
System.out.println("s1.contains(\"Java\"): " + s1.contains("Java"));
// false
System.out.println("s1.equals(s2): " + s1.equals(s2));
// true
System.out.println("s1.equalsIgnoreCase(s2): " + s1.equalsIgnoreCase
(s2));
// JAVA 6 SE
System.out.println("s1.replace(\'7\', \'6\'): " + s1.replace('7',
 '6'));
// java 7 se
System.out.println("s1.toLowerCase(): " + s1.toLowerCase());
```

2.7.11. Az asszociativitás és a precedencia

Asszociativitáson azt értjük, hogy egy bizonyos típusú kifejezés balról jobbra, vagy jobbról balra értékelődik-e ki. Ha a kétoperandusú kifejezés operandusai maguk is kifejezések, akkor a kétféle kiértékelés más eredményhez vezethet. A Java nyelven a legtöbb operátor kiértékelése balról jobbra történik. Ez alól kivételt képeznek az értékadó operátorok (=, +=, ...), a feltételes operátor, a növelő és csökkentő operátorok, az előjel operátorok, a bitenkénti negálás, a konvertálás és a `new` operátor.

Az operátorok másik fontos jellemzője a precedencia, vagyis a végrehajtási sorrend. Ha több különböző operátort használunk egyetlen kifejezésben, akkor tudnunk kell, hogy az egyes műveletek milyen sorrendben hajtódnak végre. Így tudunk megbizonyosodni arról, hogy a kifejezés ténylegesen azt jelenti-e, amit meg akartunk fogalmazni. Ha az alapértelmezett kiértékelési sorrendtől el akarunk térni, akkor kerek zárójelek használatával csoportosítanunk kell az egy egységként kiértékelendő részkifejezéseket. A matematikából ismert például, hogy a szorzást előbb kell elvégezni, mint az összeadást. Természetesen ez a Java nyelven is így van, az `a + b * c` ezért előbb `b` és `c` szorzatát számolja ki, és utána határozza meg az összeget. Ha ehelyett `a` és `b` összegét szeretnénk szorozni `c`-vel, akkor az `(a + b) * c` kifejezést kell használnunk.

A 2.7. táblázat csökkenő sorrendben sorolja fel az operátorok precedenciáját, tehát a feljebb lévő operátorok értékelődnek ki előbb. Azonos szinten lévő operátorok között a balról jobbra irány határozza meg a sorrendet.

2.7. táblázat: A Java-operátorok precedenciája csökkenő sorrendben

Leírás	Operátorok
Posztfix operátorok	`++, --`
Prefix operátorok	`++, --, +, -, ~, !`
Szorzás, osztás, maradékképzés	`*, /, %`
Összeadás, kivonás	`+, -`
Bitenkénti léptetés	`<<, >>, >>>`
Összehasonlító operátorok és típustesztelés	`<, >, <=, >=, instanceof`
Egyenlőségvizsgálat	`==, !=`
Bitenkénti és	`&`
Bitenkénti kizáró vagy	`^`
Bitenkénti vagy	`|`
Logikai és	`&&, &`
Logikai vagy	`||, |`
Feltételes operátor	`? :`
Értékadás	`=, +=, -=, *=, /=, %=, &=, ^=, |=, <<=, >>=, >>>=`

2.8. A vezérlési szerkezetek

2.8.1. A return

A `return` utasítás a metódust befejezi, és a végrehajtást visszaadja a hívó metódusnak. Ha a metódusnak visszatérési értéke is van, a `return` utasítás után olyan kifejezést kell megadni, amely ennek megfelelő típusú. Ha nincs visszatérési érték, akkor az utasítás önmagában áll:

```
return;
```

Az alábbi példa a és b számok maximumával tér vissza:

```
return a > b ? a : b;
```

2.8.2. Az if

Az `if` utasítás arra szolgál, hogy a program futása során egyes részek csak bizonyos feltételek teljesülése esetén hajtódjanak végre. A feltételeket logikai kifejezéssel adhatjuk meg. Az `if` után a feltételt mindig kerek zárójelbe kell tenni. Ezt követi a feltételesen végrehajtandó utasítás. Ebből több is megadható, de ilyenkor kapcsos zárójelbe tesszük őket. Egyes programozási irányelvek szerint akkor is ajánlott a kapcsos zárójel használata, ha csak egyetlen utasítást adunk meg, mert átláthatóbbá teszi a programkódot.

A fentiek szerint kiadott `if` utasítást követheti még tetszőleges számú `else if`-ág. Ha az `if` után megadott feltétel nem teljesül, akkor az `else if` ágak feltétele sorban kiértékelődik, és az első teljesülő feltételhez tartozó utasítás vagy utasítások fognak végrehajtódni. A feltétel és az utasítások megadása megegyezik a fentiekkel, azaz kerek, illetve kapcsos zárójelet használunk. Az `else if` ágak végén megadható egyetlen `else`-ág is, amelyhez nem tartozik feltétel, csak utasítás vagy utasítások. Itt olyan utasításhalmazt adhatunk meg, amely akkor hajtódik végre, ha egyetlen korábbi feltétel sem teljesült. Az `if` használatát korábbi példánk részletén nézhetjük meg:

```
// napszak meghatározása
if (ora < 12)
    n = Napszak.REGGEL;
else if (ora < 13)
    n = Napszak.DEL;
else if (ora < 19)
    n = Napszak.DELUTAN;
else
    n = Napszak.ESTE;
```

Ha if utasítsokat ágyazunk egymásba, akkor fontos figyelembe venni, hogy a kapcsos zárójelpár hiányában a fordító az else if és else ágakat mindig a hozzájuk legközelebb álló if utasításhoz rendeli hozzá. Ha nem ezt szeretnénk elérni, akkor egyetlen utasítás esetén is ki kell tenni a kapcsos zárójelet, hogy a fordítónak jelezzük, hova tartoznak a végrehajtási ágak. Tegyük fel, hogy a DEL értéket csak 12.30-ig akarjuk kiosztani. Ha ezt így fejeznénk ki, akkor nem a kívánt eredményt kapnánk:

```
if (ora < 12)
    n = Napszak.REGGEL;
else if (ora < 13)
    if (perc > 30)
        n = Napszak.DELUTAN;
    else if (perc < 30)
        n = Napszak.DEL;
else if (ora < 19)
    n = Napszak.DELUTAN;
else
    n = Napszak.ESTE;
```

A tördelés a programozó szándékait tükrözi, de a fordító ezt nem veszi figyelembe, és az utolsó két blokk, az else if és az else is a belső if utasításhoz fog tartozni. Annak első két feltétele viszont lefedi az összes lehetséges esetet, ezért a két utolsó ág sosem fog végrahajtódni. A megoldást a kapcsos zárójel használata jelenti:

```
if (ora < 12)
    n = Napszak.REGGEL;
else if (ora < 13) {
    if (perc > 30)
        n = Napszak.DELUTAN;
    else if (perc < 30)
        n = Napszak.DEL;
} else if (ora < 19)
    n = Napszak.DELUTAN;
else
    n = Napszak.ESTE;
```

2.8.3. A switch

Előfordulnak olyan esetek, hogy primitív típusú értéket visszaadó kifejezés konkrét értékei esetén különböző lépéseket kell végrehajtanunk. Ezt megoldhatjuk olyan if utasítással, amely sok else if ággal rendelkezik, és a feltételeiben egyenlőségvizsgálat van. A switch utasítás azonban kimondottan erre a problémára ad megoldást, és jobban olvasható programkódot eredményez. Az utasítást kerek zárójelben megadott kifejezés követi, majd kapcsos zárójelben a case kulcsszóval adjuk meg a kívánt értéket. Ezt kettőspont, majd a végrehajtandó utasításblokk követi. Ezeket az utasításokat itt nem tesszük kapcsos zárójelbe, de a következő case kulcsszó kezdete

nem is állítja meg a végrehajtást. A switch utasításból ilyenkor a break utasítással lehet kiugrani. Ennek hatására a végrehajtás a switch utáni első utasítással folytatódik.

Néha praktikus, ha a végrehajtás folytatódik a következő értékhez megadott kóddal, ilyen helyzet azonban ritkán áll fenn. Ezért érdemes ezeket az átcsúszó eseteket figyelemfelkeltő megjegyzéssel ellátni, például /* FALLTHROUGH */. Ez egyértelműen jelöli, hogy ez a kívánt működés, és nem csupán lefelejtettük a break utasítást. A default kulcsszó és az utána következő kettőspont alkalmazható olyan esetek kezelésére, amelyekhez nem adtunk meg külön belépési pontot a case kulcsszóval. A korábbi példa kiegészítése szemlélteti a switch utasítás használatát:

```
switch (n) {
case REGGEL:
    System.out.println("Jó reggelt kívánok!");
    break;
case DEL:
    /* FALLTHROUGH */
case DELUTAN:
    System.out.println("Jó napot kívánok!");
    break;
case ESTE:
    System.out.println("Jó estét kívánok!");
    break;
default:
    break;
}
```

Az utasítás a primitív értékeken kívül enumerációkkal és a Java SE 7-es verziója óta karakterláncokkal is használható. Karakterláncok esetén az összehasonlítás úgy történik, mint ha az equals() metódust hívtuk volna meg. Ha a kis- és nagybetűket nem akarjuk megkülönböztetni, akkor eljárhatunk úgy, hogy a switch utasításnak megadott karakterláncot először kisbetűsre alakítjuk, majd a case kulcsszó karakterlánc-literáljaiban is csupa kisbetűs írásmódot alkalmazunk.

2.8.4. A while és a do

A while utasítás ún. ciklusutasítás, a megadott kódot többször egymás után futtatja le. Az utasítás után kerek zárójelben megadunk egy logikai kifejezést, amelynek igaz értéke esetén fog lefutni az ezután megadott utasítás. Kapcsos zárójelet használva több utasítást is megadhatunk. Miután ezek lefutottak, a feltétel újra kiértékelődik, és ha igaz értéket eredményez, akkor az utasítások újra lefutnak. Ez addig ismétlődik, amíg a feltétel hamis nem lesz. Ha ez sosem következik be, akkor végtelen ciklusról beszélünk. Végtelen ciklus lehet programozói hiba eredménye, de néha szándékosan is létrehozunk ilyet. Többszálú programoknál elképzelhető például, hogy egy szál folyamatosan ugyanazt az ismétlődő feladatot látja el.

A do utasítás hasonlóan működik, de a feltétel kiértékelése a ciklusba foglalt utasítások után következik. Először tehát ezeket adjuk meg, majd a while kulcsszó, a ke-

rek zárójelbe tett feltétel és az utasítást záró pontosvessző következik. Ez a működés azt is eredményezi, hogy a ciklusba foglalt utasítások legalább egyszer lefutnak.

A ciklusból annak tetszőleges belső pontján ki is léphetünk a `break` utasítással. Találkozhatunk olyan esettel is, hogy sem a `while`, sem a `do` nem felel meg teljesen az elvárásainkank, mert a kilépési feltételeket nem a ciklus elején vagy végén, hanem belső ponton kell vizsgálni. Ekkor a `while` utasításnak feltételként megadhatjuk a `true` konstansot, majd a ciklus belsejében a megfelelő feltételek fennállása a `break` utasítással léphetünk ki a ciklusból. A következő programrészlet mutat példát a `while` ciklusra. A metódus a `while` ciklus használatával eldönti egy számról, hogy prím-e:

```
public static boolean primteszt(long p) {
    boolean prim = true;
    long oszto = 2;

    while (oszto < p) {
        if (p % oszto == 0) {
            prim = false;
            break;
        }
        p++;
    }
    return prim;
}
```

2.8.5. A for

A `for` is ciklusutasítás, ennek hagyományos formájában három kifejezést kell megadni. Az első az inicializáló utasítás, ez a legelső iteráció előtt hajtódik végre. A második a feltétel, ennek a teljesülése esetén a ciklus lefut. A harmadik pedig az egyes iterációk végén végrehajtódó utasítás. Ezeket kerek zárójelben, pontosvesszővel elválasztva adjuk meg. Ezt követi a ciklusban végrehajtandó utasítás, vagy kapcsos zárójelben megadott utasítások.

A `for` ciklus jól alkalmazható számláló jellegű működés megvalósítására. Inicializáláskor egy változó kezdeti értékét adjuk meg, a feltételben megvizsgáljuk, hogy ez valamilyen alsó vagy felső korláton belül van-e, majd az iteráció végén a harmadik utasítással megváltoztatjuk az értékét. A ciklus belsejében a változó értékével dolgozhatunk, így a kívánt utasításokat minden értékre elvégezhetjük. Nem muszáj megadnunk sem az inicializáló, sem az iterációk végén végrehajtandó utasítást. Ha egyiket sem adjuk meg, akkor a ciklus azzal lesz egyenértékű, mint ha a `while` utasítást használtuk volna. A feltétel is lehet üres, ez olyan, mint ha konstans igaz értéket adtunk volna meg. Szintén használható a `break` utasítás, hogy a ciklust tetszőleges ponton megszakítsuk. Használható még a `continue` utasítás is, ez a futást a következő iterációval folytatja. A következő példában láthatjuk a prímteszt `for` ciklussal megvalósított változatát:

```
public static boolean primteszt(long p) {
    boolean prim = true;

    for (long oszto = 2; oszto < p; oszto++) {
        if (p % oszto == 0) {
            prim = false;
            break;
        }
        p++;
    }
    return prim;
}
```

A for utasításnak létezik egy másik szintaxisa is, ennek segítségével tömbök vagy a később ismertetett kollekciók (lásd 5. fejezet) elemein lépkedhetünk végig. Ekkor a zárójelben változódeklaráció áll, utána kettőspontot írunk, majd megadjuk a bejárni kívánt adatstruktúrát. A deklarált változó minden egyes iterációban az adatstruktúra más elemét fogja tartalmazni, és hatóköre csak erre a ciklusra terjed ki. A bejárás sorrendje a bejárt tömbtől vagy kollekciótól függ. Az alábbi program a for ciklussal járja be az argumentunként kapott karakterlánctömböt, és kiírja a parancssori paramétereket a kimenetre:

```
public static void main(String args[]) {
    for (String s : args) {
        System.out.println(s);
    }
}
```

2.8.6. A címkék

A Java nyelvben nincs ugró utasítás, mert ez rossz programozási gyakorlatot alakíthat ki. Egymásba ágyazott ciklusoknál a break és continue utasítások a belső ciklusra vonatkoznak. Hasznos lenne azonban, ha a belső ciklusból is ki tudnánk ugrani a külsőn kívülre, hogy a végrehajtás a külső ciklus utáni utasításokkal folytatódjon. Erre ad megoldást a címkézett ciklusok alkalmazása. Ilyet úgy hozunk létre, hogy először a címkenevet adjuk meg, majd kettőspont után kezdődik a ciklusutasítás:

```
kulso: for (int i = 0; i < 5; i++) { ... }
```

A tetszőlegesen mélyen beágyazott ciklusból a következő utasítással tudunk kilépni:

```
break kulso;
```

Új iterációt is kezdhetünk a külső ciklusban:

```
continue kulso;
```

2.9. Az annotációk

Az annotációk olyan módosítók, amelyeket bármilyen deklarációs utasításhoz hozzárendelhetünk (csomag, osztály, interfész, tagváltozó, paraméterváltozó stb.). Az annotációt a neve azonosítja[1], és rendelkezhet paraméterekkel is. Az annotációkat @ jellel, majd a nevükkel adjuk meg az annotálni kívánt deklaráció előtt. Az annotáció esetleges paraméterei zárójelben vannak felsorolva név-érték párokként. Ha az annotáció csak egy paraméterrel rendelkezik, akkor a név-érték pár helyett írhatjuk csupán a nevet is.

Az annotáció kiegészítő információt rendel a deklarált elemhez, amelyet később fel lehet dolgozni. Ezáltal olyan információ is elhelyezhető a kódban, amely annak nem szerves része, de erősen kapcsolódik hozzá. Például az adatbázis-kezelést támogató keretrendszerek számára is lehet így információt nyújtani az adatbázistáblákra való leképezés módjáról (lásd 8.2. alfejezet). A fordító is használ néhány annotációt. Az `@Override` metódusokon használható, és azt jelzi, hogy az annotált metódus ősosztály metódusát definiálja felül vagy interfész metódusát implementálja. Ha elgépelnénk a metódus szignatúráját, akkor új metódust hoznánk létre, az újradefiniálni kívánt metódus pedig észrevétlenül öröklődne. Ezen segít az `@Override` annotáció, ez ugyanis fordítási hibát eredményez, ha az így megjelölt metódus nem szerepelt az ősosztályokban és ősinterfészekben. A `@SuppressWarnings` jelzi a fordítónak, hogy ne adjon ki *figyelmeztetést (warning)* bizonyos potenciálisan veszélyes műveletek, értékadások esetén. A figyelmeztetések természetesen hasznosak, de ha megbizonyosodtunk arról, hogy az adott helyzet veszélytelen, akkor jobb lehet ezeket lecsendesíteni. Így a figyelmeztetések tengerében nem fognak elveszni az újabb figyelmeztetések, amelyek esetleg veszélyes hibára vonatkoznak. Az annotációnak van egy `value` paramétere, ennek egy karakterlánctömböt adhatunk meg. Ebben soroljuk fel azonosítóikkal a kihagyni kívánt figyelmeztetéstípusokat. A leggyakoribb értékeket a 2.8. táblázat foglalja össze.

2.8. táblázat: Az @SuppressWarnings annotáció value paraméterének megadható értékek

Azonosító	Leírás
all	az összes figyelmeztetés
boxing	primitív típusok automatikus be- és kicsomagolása
cast	konvertálással kapcsolatos műveletek
deprecation	elavultnak jelölt metódus hívása
fallthrough	túlcsúszó case-ág
hiding	lokális változók, amelyek elfednek egy másikat
unused	nem használt kód

[1] Interfésznévre vonatkozó konvenció szerinti név, mivel az annotációt interfész valósítja meg.

A következő példa mutatja az annotációk használatát:

```
@SuppressWarnings(value = "unused")
int nemhasznalt;

// így is írható
@SuppressWarnings("unused")
Double d = 4.0;
```

HARMADIK FEJEZET
Az objektumorientált eszköztár

Az objektumorientált eszközök és elvek ugyan alapjaiban egységesek, a különböző programozási nyelvek mégis eltérő módon használják azokat. A fejezet bemutatja a Java nyelv objektumorientált eszköztárát. Tárgyaljuk az alapvető elveket, így az egyszeres öröklés és az interfészek használatát, valamint a Javára jellemző speciális objektumorientált eszközöket is. Idetartoznak például a belső osztályok és az enumerációk.

3.1. A tagváltozók és a metódusok

A Java-osztályok definíciója tagváltozók, metódusok és konstruktorok definícióját, valamint inicializációs blokkokat tartalmazhat. A tagváltozók az osztályhoz vagy objektumpéldányhoz kapcsolódó változók. Az előbbieket osztályváltozónak, utóbbiakat példányváltozónak nevezzük. A metódusok műveleteket végeznek, ennek során kiolvashatják és megváltoztathatják a tagváltozókban tárolt értékeket. Szintén lehetnek statikusak vagy példányhoz tartozók. A tagváltozókat és a metódusokat együtt az osztály tagjainak nevezzük. A konstruktorok olyan speciális metódusok, amelyek az osztály objektumpéldányait inicializálják, ezért nevük is megegyezik az osztály nevével. Objektumpéldány létrehozása mindig konstruktorhívással történik, még ha nem közvetlenül hívjuk is meg. Minden osztálynak rendelkeznie kell tehát konstruktorral, hogy példányokat lehessen létrehozni. A konstruktorok is rendelkezhetnek paraméterekkel, amelyek befolyásolják a létrejövő objektumpéldány jellemzőit. Gyakran előfordul, hogy nincs szükségünk inicializációs műveletekre, ekkor paraméterek nélküli, üres konstruktort definiálhatunk. Mivel ez az eset nagyon gyakori, a fordító automatikusan beszúr egy ilyen konstruktort `public` láthatósággal (lásd 3.2. alfejezet), ha a programozó nem definiál egy konstruktort sem. Ezt a konstruktort ezért *alapértelmezett konstruktornak (default constructor)* nevezzük.

Más objektumorientált nyelvekben definiálhatunk destruktort is, ez szintén speciális metódus, és az objektumpéldány életciklusának végén hajtódik végre. Ez az objektumpéldány által lefoglalt erőforrások felszabadítására használható. A Java nyelv osztályai nem kezelik külön fogalomként a destruktort, de az `Object` osztálytól örökölt üres `finalize()` metódust újradefiniálhatjuk, és használhatjuk erre a célra. Ezt a metódust a szemétgyűjtő hívja meg, mielőtt a memóriából eltávolítaná az objektumpéldányt. A szemétgyűjtés azonban nem befolyásolható a programozó által, ezért nincs garancia arra, hogy a `finalize()` metódus valaha is lefut. Az alábbi példa számlálót valósít meg. Ebben láthatjuk egy osztály vázát a benne szereplő tagváltozó-, metódus- és konstruktordefiníciókkal:

```
class Counter {
    // tagváltozó: a számláló értéke
    protected int value;

    // konstruktor: inicializálja a számlálót
    public Counter() {
        value = 0;
    }

    // metódus: növeli a számlálót
    public void increment() {
        value++;
    }

    // metódus: kiolvassa az értéket
    public int getValue() {
        return value;
    }
}
```

Az osztályokat specializálhatjuk, és a specializáció során funkcionalitásukat megváltoztathatjuk vagy kiegészíthetjük. A specializált osztályt leszármazott osztálynak nevezzük, az általánosabb változatot pedig ősosztálynak. A közvetlen ősosztályt szülőosztálynak is hívjuk. A leszármazott osztályok örökölhetnek példányváltozókat és metódusokat, de a konstruktorok sosem öröklődnek. Öröklésкor az örökölt metódus vagy példányváltozó a leszármazott osztályban is használható lesz, mint ha azonos módon definiáltuk volna. Ha a leszármazott osztályban a metódusnak másként kell viselkednie, akkor az új definíció megadásával felül kell bírálni. Néha a metódust nem akarjuk teljesen újradefiniálni, csak néhány új lépéssel kiegészítenénk ki. Ebben az esetben is meg kell ismételnünk a definíciót, de a `super`-referencia segítségével meghívhatjuk az ősosztály eredeti metódusát. Ezután elvégezhetjük a kívánt kiegészítő lépéseket. Az alábbi példában az előbbi osztályból hozunk létre leszármazott osztályt, amely kettesével számol:

```
class Counter2 extends Counter {

    @Override
    public void increment() {
        value += 2;
    }
}
```

A metódusok újradefiniálása megváltoztatja az osztály viselkedését. Ha a programozó nem ismeri az osztály belső működését, akkor az újradefiniálás helytelen viselkedéshez vezethet. Előfordulhat, hogy ilyenkor inkább megtiltanánk az osztály specializálását. Ebben az esetben az osztálydefiníció `class` kulcsszava előtt a `final` módosí-

tót kell használni. A metódusok újradefiniálása egyenként is tiltható, ha szignatúrájukban használjuk a `final` módosítót. Ez lehetővé teszi, hogy megengedjük az osztály specializációját, és csak a kritikus metódusok újradefiniálását tiltsuk. Az osztálykönyvtár `String` osztálya is `final` módosítóval van megjelölve, mivel a karakterláncokat a Java nyelv speciálisan kezeli. Nem kívánatos, hogy a belső működési mechanizmusait módosítsuk, ugyanis ez könnyen helytelen működéshez vezetne.

3.2. A láthatóság

A változók láthatóságát általánosan a 2.6.8. alfejezetben tárgyaltuk. Az objektumok tagváltozói, metódusai és konstruktorai esetében a láthatóság kérdése összetettebb. A láthatóságot a programozó adja meg láthatósági módosítók használatával. A láthatósági módosítók nemcsak a láthatóságot, hanem az öröklést is befolyásolják. A 3.1. táblázat összefoglalja a láthatósági módosítókat, és láthatósággal, illetve öröklléssel kapcsolatos hatásaikat.

3.1. táblázat: A Java nyelv hozzáférési módosítói

Módosító	Láthatóság	Öröklés
`public`	Mindenhonnan látható.	Öröklődik.
`protected`	Csomag osztályaiból és leszármazott osztályokból látható.	Öröklődik.
(nincs módosító)	Csomag osztályaiból látható.	Nem öröklődik.
`private`	Csak a definiáló osztályból látható.	Nem öröklődik.

A `private` és a `protected` módosítók némi magyarázatra szorulnak. Az előbbi azt jelenti, hogy csak a definiáló osztály érheti el a kulcsszóval megjelölt tagokat. Az osztály példánya azonban más példány privát tagjait is eléri. Ha a kódban referenciánk van másik példányra, akkor annak privát tagjait közvetlenül is elérhetjük, nem szükséges a getterek és setterek használata. Ez tipikusan az `equals()` (lásd 3.11. alfejezet) metódus újradefiniálásánál fordul elő.

A `protected` módosítóval rendelkező metódusokat a csomag osztályai látják, valamint a leszármazott osztályok öröklik. Ha a leszármazott osztály másik csomagban van, akkor is örökli a metódust, de a csomag többi osztálya nem fogja látni. A metódust újra kell definiálni, hogy a leszármazott osztály csomagjának többi osztálya is meg tudja hívni. A láthatósági módosító ugyan nem változtatható meg az újradefiniáláskor, de a definíció a leszármazott osztály csomagjába kerül. A csomag osztályai így már elérik a metódust. Előfordulhat, hogy egy metódust csak a láthatóság miatt definiálunk újra, a kódját nem kívánjuk megváltoztatni. Ekkor használható a `super` kulcsszó. Ilyen esetre példa a `clone()` (lásd 3.11. alfejezet) metódus használata.

A láthatósági módosítók konstruktorokon is alkalmazhatók. Egyedül a `protected` módosító nem használható, a konstruktorok nem öröklődő természete miatt ugyanis ez ekvivalens lenne a kulcsszó nélküli esettel. Ha az osztály rendelkezik publikus konstruktorral, akkor bárhol példányosítható. Előfordulhat azonban, hogy az osztály példányosítását jobban kézben szeretnénk tartani, és nem kívánjuk megengedni, hogy

tetszőleges módon példányosítható legyen. Ilyen eset például, ha az osztály példányosítása előtt be kell gyűjteni néhány paramétert, vagy a példányosítást más módon kell előkészíteni. Ekkor az Abstract Factory vagy a Factory Method objektumorientált tervezési mintákat alkalmazzuk [4]. A könyvben a mintákban szereplő osztályra és metódusra a *factoryosztály* és a *factorymetódus* elnevezésekkel hivatkozunk. A minták kikényszerítéséhez használhatunk csomagszintű (módosító nélküli) vagy `private` konstruktort. Előbbi csak az osztály csomagjaiba tartozó osztályokból hívható, utóbbi pedig kizárólag magából az osztályból. Az alábbi példában a Singleton tervezési mintát valósítjuk meg Java nyelven. A Singleton minta lényege, hogy csupán egy példány létezhet az osztályból, és ezt metódushívással kaphatjuk meg. Ehhez privát konstruktort kell használni, különben kívülről is meghívható lenne, és ezért nem lehetne kikényszeríteni, hogy csupán egy példány létezzen. Az egyetlen példányt osztályváltozóban tároljuk el, és statikus metódussal kapható meg az osztálytól. Az első híváskor a metódusnak létre is kell hoznia a példányt.

```
public class Singleton {
    private static Singleton instance = null;

    public static Singleton getInstance() {
        if (instance == null)
            instance = new Singleton();
        return instance;
    }

    private Singleton() {}

    public void hello() {
        System.out.println("Hello");
    }

    public static void main(String[] args) {
        Singleton i = Singleton.getInstance();
        i.hello();
    }
}
```

3.3. Az osztálydefiníciók láthatósága

Az osztályok és az interfészek definíciói ugyan nem változók, de esetükben is beszélhetünk láthatóságról a tekintetben, hogy a kódban hol hivatkozhatunk rájuk. A továbbiakban az osztály fogalmat használjuk, de az interfészek láthatóságára is ugyanazok a szabályok érvényesek. Osztálydefiníciók esetén csak publikus és csomagszintű láthatóságot adhatunk meg. Az előbbi a `public` kulcsszóval, utóbbi kulcsszó megadása nélkül történik. A publikus osztályok bármilyen kódból elérhetők, és érvényes rájuk az a megkötés, hogy fájlonként csupán egyet definiálhatunk. A fájl nevét az osztály nevéből és a `.java` kiterjesztésből képezzük. A csomagszintű osztály csak a csomagjába

tartozó kódból érhető el, valamint nem vonatkozik rá a fájlokkal kapcsolatos szabály, ugyanis többet is definiálhatunk belőlük egy fájlban. Publikus osztályt definiáló fájlban is létrehozhatunk csomagszintű osztályokat. Ennek ellenére a fájlonként egy osztály konvenció természetesen alkalmazható, elősegítve a forráskód áttekinthetőségét.

3.4. A konstruktorok

A konstruktorok az osztály példányait hozzák létre, és rendelkezhetnek paraméterekkel is. Ezek tipikusan kezdeti adatok átadására szolgálnak, ezekkel a konstruktor a létrejövő objektum példányváltozóit inicializálja. Több konstruktor is megadható, ha azok eltérő paraméterlistával rendelkeznek. Ekkor az objektumot példányosító program írója választhat, hogy melyik konstruktort hívja.

A konstruktorok nem öröklődnek, de a végrehajtásuk mindig végiggyűrűzik az ősökön, egészen az `Object` osztályig. A konstruktor legelső utasítása mindig olyan utasítás, amelyben vagy az ősosztály konstruktorát, vagy egyéb saját konstruktort hív meg. Ha a programozó nem ad meg mást, akkor a konstruktor első utasításának a fordító a `super()` utasítást szúrja be, amely a szülőosztály alapértelmezett konstruktorát hívja. Lehetséges paraméteres konstruktort is hívni. Ekkor a `super()` utasítást ki kell írnunk, és a zárójelben meg kell adnunk a szülőosztály konstruktorának átadandó paramétereket. Ennek az utasításnak mindig a konstruktor legelején kell szerepelnie, más utasítást nem adhatunk ki előtte.

Használhatjuk a `this()` operátort is. Ha az osztálynak van más konstruktora, akkor ezzel az utasítással hívhatjuk meg az előbb látott `super()` utasításhoz hasonlóan. Az utasításnak szintén az első helyen kell szerepelnie. Az osztály konstruktorai közül valamelyik mindenképpen az ősosztály konstruktorának kell, hogy átadja a végrehajtást, ha ugyanis az osztály konstruktorai mindig egymást hívnák meg, az végtelen rekurzióhoz vezetne, és a verem előbb-utóbb megtelne. Ez az eset nem vezet fordítási hibához, de ha ilyen osztályt kísérlünk meg példányosítani, akkor `StackOverflowError` hibát kapunk.

3.5. Az objektumok inicializálása

Nem a konstruktorok használata az egyetlen mód az osztályok példányváltozóinak inicializálására. Ahogy a lokális változóknál is, a példányváltozók deklarációjában is szerepelhet kezdőérték-adás. A harmadik lehetőség az inicializációs blokk használata. Ez kapcsos zárójelbe írt kódot jelent, amely az osztály példányosításakor hajtódik végre. Mind a kezdőérték-adás, mind az inicializációs blokk feltétel nélkül, bármely konstruktor hívása esetén végrehajtódik. Ezekkel tehát az általánosan érvényes inicializációs lépéseket érdemes megadni, míg az egy-egy eltérő használati esetre vonatkozó inicializációs lépéseket különböző konstruktorokban adhatjuk meg. Az osztály példányosításakor így az adott esetnek megfelelő konstruktor hívható meg.

Ha mindhárom módon megadunk inicializációs lépéseket, akkor azok a következő sorrendben hajtódnak végre:

1. Ha vannak olyan változódeklarációk, amelyekben kezdőérték-adás is szerepel, először ezek hajtódnak végre.
2. Ezután az inicializációs blokkok futtatódnak, megadásuk sorrendjében.
3. Végül a meghívott konstruktor utasításai jutnak érvényre.

Az alábbi programrészlet szemlélteti az inicializációs lehetőségeket.

```
public class Main {
    // Első
    private int a = 2;

    { System.out.println("Második, a értéke ekkor már " + a); }

    public Main() {
        System.out.println("Negyedik.");
    }

    { System.out.println("Harmadik."); }

    public static void main(String[] args) {
        Main m = new Main();
    }
}
```

3.6. Az absztrakt osztályok és az interfészek

Láttuk, hogy a Java nyelv alapegységei az osztályok. Ezek tagváltozókat, metódusokat és konstruktorokat definiálnak, és az előbbi kettőt a leszármazott osztályok örökölhetik. Lehetőség van interfészek használatára is. Ezek adott szignatúrájú metódusokat írnak elő, de nem adnak hozzájuk implementációt. Az interfészt megvalósító leszármazott osztálynak defniálnia kell a metódusokat. Interfészen általánosan egy komponens külvilág számára elérhetővé tett kommunikációs felületét értjük. Innen ered a Java-interfész elnevezése is, ezek ugyanis az általános értelemben vett interfész kialakításában segítenek. Ha fontos kiemelni, hogy az interfészről mint a Java nyelv egységéről beszélünk, nem pedig az általános programozási fogalomról, akkor a továbbiakban arra Java-interfészként hivatkozunk. A Java-interfészek tehát programozói hozzáférési felületet definiálnak, ezért csak publikus metódusokat írhatak elő. A `public` kulcsszó megadása nem kötelező, az interfészek metódusai mindig publikusak lesznek. Más láthatósági módosító nem is használható, mert az fordítási hibát eredményez. Az interfészben csak a metódusok szignatúráját írjuk elő, azokat nem implementáljuk, ezért a paraméterlista után a metódusdefiníció helyett pontosvesszőt teszünk. Az interfészben `public static final` módosítójú konstansok is definiálhatók, de ezek használata rossz programozási gyakorlatnak számít, ugyanis a konstansok az implementációs részletekhez tartoznak, nem pedig az osztály programozói interfészéhez. Az alábbi példa interfészt definiál a korábban látott számlálóhoz:

```
public interface Counter {
    void increment();
    int getValue();
}
```

Az objektomorientált programozási gyakorlat szerint ajánlott interfésztípusokat alkalmazni, ahol csak lehet, így később a konkrét típus könnyen lecserélhető. Tekintsünk például egy statisztikákat előállító programot. A kimenetet előállíthatjuk sok formában, például szöveges, HTML-, illetve Excel-formátumban. Ha definiálunk egy interfészt, amelyen keresztül a statisztikák átadhatók a kimenetet létrehozó objektumnak, akkor a különböző kimeneti formátumok a kódban egységesen kezelhetők, és később a konkrét implementáció könnyen lecserélhető.

A Java nyelv nem támogatja az osztályok közötti többszörös öröklést, de az osztályok tetszőleges számú interfészt implementálhatnak. Jó gyakorlat ezért az osztályhierarchiákat interfésszel kezdeni, és ahhoz egy alapértelmezett implementációs osztályt is készíteni, így a későbbi kiegészítések alapozhatók az osztályra, de az interfészre is. Ha egy osztályt valamiért több osztályhierarchiának is a részévé akarunk tenni, akkor csak az egyik hierarchia esetén használhatjuk az öröklést, a másik esetben interfészt kell implementálnunk. Az interfészek egymást is bővíthetik, újabb publikus metódusokat adva az ősinterfészhez.

Megtehetjük azt is, hogy bizonyos metódusokat implementálunk egy osztályban, de néhány másik metódus esetén nem adunk definíciót, csak előírjuk azok implementálását. Az ilyen osztályt absztrakt osztálynak nevezzük, és csak típusként, valamint más osztályok szülőjeként alkalmazható, de nem példányosítható. A nem implementált metódusokat az `abstract` kulcsszóval kell ellátni, valamint az osztályt is az `abstract class` kulcsszavakkal definiáljuk. Az interfésszel ellentétben az absztrakt osztályok absztrakt metódusai lehetnek csomagszintű vagy `protected` láthatóságúak is, a `private` módosító azonban nem használható. Ahhoz, hogy létrehozzunk olyan példányt, amely megfelel ennek az absztraktosztály-típusnak, az absztrakt osztályból leszármazott osztályt kell létrehoznunk, és abban az összes absztrakt metódust meg kell valósítanunk. Létrehozható olyan leszármazott osztály is, amely csak néhány metódust valósít meg, de ez még rendelkezni fog absztrakt metódusokkal, ezért magának az osztálynak is absztraktnak kell lennie. A fennmaradó absztrakt metódusokat újabb leszármazott osztálynak vagy osztályoknak kell implementálniuk. Az absztrakt osztályok jól használhatók akkor, ha az osztályhierarchia funkcionalitásának egy része jól általánosítható, de néhány rész erősen függ a konkrét leszármazott osztálytól, és ki akarjuk kényszeríteni, hogy a leszármazott osztályok ezt maguk implementálják. Természetesen sem absztrakt osztály absztrakt metódusa, sem interfész metódusa nem lehet `final` módosítóval megjelölve, ezeket ugyanis definiálni kell a leszármazott osztályban vagy az interfész implementációs osztályában, de a kulcsszó éppen ezt akadályozná meg. Az alábbi példában egy absztrakt osztályt láthatunk, amely dátumok formázására szolgál. A dátum beállítására szolgáló metódus öröklődik, és megköveteljük a formázást végrehajtó metódus implementálását a leszármazott osztályokban. Az osztály leszármazottjai különböző módokon formázhatják a dátumot, például a magyar, angol stb. konvenció szerint. A leszármazott osztályoknak csak ezt kell megvalósítaniuk. Az osztálykönyvtár `DateFormat` osztálya (lásd 14.1. alfejezet) gazdagabb dátumformázási funkcionalitást kínál, de ugyanezen az elven működik.

```
public abstract class DateFormatter {
    protected Date date;

    public void setDate(Date date) {
        this.date = date;
    }

    public abstract String formatDate();
}
```

Az osztályok egymás specializálására az `extends` kulcsszót használják. Interfész újabb interfésszel való kiterjesztése is az `extends` kulcsszóval lehetséges, ugyanakkor osztály az `implements` kulcsszóval implementálhat interfészeket. A kulcsszavakat az osztály definíciójában, a neve után adjuk meg, például:

```
public class HtmlFormatter extends PlainTextFormatter implements ↵
Formatter {
    ...
}
```

Interfészekből többet is felsorolhatunk vesszővel elválasztva. Változótípusként egyaránt használhatunk osztályt, absztrakt osztályt és interfészt. Leszármazott típuson egy specializáltabb típust értünk, amely lehet osztály leszármazott osztálya, interfész leszármazott interfésze vagy interfész implementációs osztálya is. Mivel ezek a típusok az őstípus teljes funkcionalitását megvalósítják, bárhol használhatók, ahol az őstípus példányára van szükség.

3.7. Az újradefiniálás és a túlterhelés

Az osztályok tehát örökölhetnek metódusokat az őseiktől. Ez azzal egyenértékű, mint ha azok definícióját egy az egyben átmásoltuk volna a leszármazott osztályba. A leszármazott osztály az örökölt metódust újradefiniálhatja, ha az eredeti működése nem megfelelő számára. Az új definícióban a metódus szignatúrája meg kell, hogy feleljen az eredetinek. Ez azt jelenti, hogy a paraméterlista megegyezik, a visszatérési érték és a kiváltott kivételek pedig vagy megegyeznek, vagy az eredeti típusok leszármazott típusai. Természetesen interfészek implementálása esetén is érvényes ez a korlátozás. A megkötés logikus, ugyanis a leszármazott osztály a szülőjének specializált változata, vagyis helyette is használhatónak kell lennie. Leszármazott típus visszaadása, illetve leszármazott kivétel kiváltása még nem sérti ezt a követelményt, de nagyobb eltérések esetén a kicserélhetőség már nem teljesülne.

A metódusok *újradefiniálását (override)* fontos megkülönböztetni a *túlterheléstől (overloading).* Utóbbi azt jelenti, hogy különféle paraméterlistájú metódusokat vagy konstruktorokat definiálunk ugyanazzal a névvel. Híváskor a paraméterlista alapján eldönthető, hogy pontosan melyik metódust kell végrehajtani. Leszármazott osztályban is definiálhatunk örökölt metódus nevével különböző paraméterlistájú metódust,

de ebben az esetben is túlterhelésről beszélünk, nem pedig újradefiniálásról. Ebből következik, hogy ha véletlenül nem ugyanúgy adjuk meg a paraméterlistát, vagy elírjuk a metódus nevét, akkor valójában nem fogjuk újradefiniálni. Ez nem várt és nehezen azonosítható hibákhoz vezethet. A fejlesztőeszközök természetesen segítenek ebben az automatikus kódkiegészítés funkcióval, de a biztonság kedvéért használjuk a már említett `@Override` annotációt is. Ez fordítási hibát eredményez, ha a vele megjelölt metódusdefiníció nem definiál újra örökölt metódust, vagy nem interfész által előírt metódust valósít meg.

3.8. Az osztályszintű változók és metódusok

A statikus tagváltozók és metódusok nem objektumpéldányhoz, hanem osztályhoz tartoznak. A rájuk történő hivatkozásnál általában az osztálynevet és nem példányra hivatkozó referenciát használunk. Használhatunk példányváltozót is, de ezek a tagváltozók és metódusok az osztályhoz tartoznak, ezért ésszerűbb az előző jelölés.

Osztályváltozót, illetve statikus metódust a `static` kulcsszó használatával deklarálunk. Az osztályváltozók incializálása történhet a kezdőérték megadásával a deklarációban, vagy használhatunk statikus inicializációs blokkot is. A statikus inicializációs blokk a példányok inicializálásánál említett blokkhoz hasonló, de a `static` kulcsszó előzi meg. Konstruktorokban ugyan elérhetjük az osztályváltozókat és metódusokat, de ezek csak példányosításkor futnak le. Mivel az osztályváltozók és statikus metódusok létező példány nélkül is elérhetők, ezért a konstruktorok használata nem megfelelő módszer az osztályváltozók inicializálásához.

Az osztályváltozók inicializációja az osztály betöltődésekor történik, ez az első statikus elemre történő hivatkozáskor vagy az első objektum példányosításakor megy végbe. Ebben az esetben is először a deklarációban elhelyezett kezdőérték-adás fut le, majd a statikus inicializációs blokkok a megadásuk sorrendjében. Lényeges különbség a példánymetódusokhoz képest, hogy a statikus metódusok sosem öröklődnek. A túlterhelés viszont alkalmazható, tehát azonos névvel deklarálhatunk eltérő paraméterlistájú metódusokat.

3.9. A belső osztályok

Az objektumorientált paradigma szerint a programok funkcionalitását specializált objektumokba zárjuk egységbe. Egy objektum egy adott, jól meghatározott feladatra készül. Előfordul azonban, hogy egy osztály írása során azt vesszük észre, hogy a funkcionalitás bizonyos részének külön osztályba kellene kerülnie, noha nagyon kötődik a jelenleg fejlesztett osztályhoz. Ilyenek például a grafikus felhasználói felület fejlesztése során használt eseménykezelők. Ezek adott interfészt implementálnak, amely előírja az esemény bekövetkeztekor meghívandó metódus megvalósítását. Ilyen esetben használhatunk belső osztályt az eseménykezelő megvalósítására. A belső osztályoknak több típusuk van, és a statikus belső osztály kivétel jellemző rájuk, hogy a tartalmazó osztály összes tagját elérik, még a `private` módosítóval deklaráltakat is. Ez az alfejezet a belső osztályokat tekinti át.

3.9.1. A hagyományos belső osztályok

A hagyományos belső osztályokat a tartalmazó osztály definícióján belül hozzuk létre. Használhatók rajtuk a láthatósági módosítók, valamint a `final` és `abstract` kulcsszavak is. A belső osztályt a tartalmazó osztályon belül a megszokott módon példányosítjuk. A belső osztály kódjában hivatkozhatunk a külső osztály bármely tagjára, mint ha azok a belső osztályban is definiálva lennének. A `this` kulcsszó ilyenkor a belső osztályra vonatkozik, a külső osztályra `KulsoOsztaly.this` formában hivatkozhatunk. A belső osztály statikus metódusai természetesen a tartalmazó osztálynak is csak a statikus tagjait érhetik el.

Az alábbi példa a hagyományos belső osztályt mutatja be mezőkből álló játékteret reprezentáló osztállyal. A mezőkre jellemző a koordinátájuk. A mezők a játéktér részei, de önmaguk is önálló egységnek tekinthetők, ezért praktikus őket belső osztályként megvalósítani. A játéktér osztály konstruktorának megadható, hogy hány sorból és oszlopból áll a játéktér, így könnyen létrehozhatjuk a mezőket.

```
public class Maze {
    class Field {
        int x;
        int y;

        Field(int x, int y) {
            this.x = x;
            this.y = y;
        }
    }

    private Field[][] fields;

    public Maze(int dimX, int dimY) {
        fields = new Field[dimX][];
        for (int x = 0; x < dimX; x++) {
            fields[x] = new Field[dimY];
            for (int y = 0; y < dimY; y++) {
                Field f = new Field(x, y);
                fields[x][y] = f;
            }
        }
    }

    public static void main(String[] args) {
        Maze m = new Maze(5,5);

        // Nem fordul le; a belső osztály a tartalmazó
        // osztály példánya nélkül nem használható
        // Maze.Field f = new Maze.Field(5, 0);
```

```
        // Ha létezik tartalmazó példány, akkor már működik
        Maze.Field f2 = m.new Field(5, 0);
    }
}
```

Ha a láthatósági módosító megengedi, akkor a belső osztály kívülről is elérhető. A tartalmazó osztállyal minősített osztálynévvel hivatkozunk rá. Példányosításához szükséges a tartalmazó osztály egy példánya, ahogy a fenti példa is mutatja. Alább láthatunk két példát a példányosításra:

```
Kulso kulso = new Kulso();
Kulso.Belso b1 = kulso.new Belso();
Kulso.Belso b2 = new Kulso().new Belso();
```

3.9.2. A lokális belső osztályok

Metódusokon belül is definiálhatunk osztályt. Ennek a láthatósága természetesen a metódusra korlátozódik, így csak az `abstract` és `final` módosítók használhatók. Azt várnánk, hogy a belső osztály a tartalmazó osztály tagjain kívül elérje a metódus lokális változóit is, de ez általánosságban nem igaz. A lokális belső osztályok ugyanis a veremben jönnek létre, és életciklusuk átívelhet a metódus határán is. A metódus lefutása után a lokális változók azonban már nem léteznek. A `final` módosítóval deklarált lokális konstansok azonban a lokális belső osztályból is elérhetők. Statikus metódusokban is létrehozhatók lokális belső osztályok, de természetesen ezek csak a tartalmazó osztály statikus tagjait érhetik el. A következő példa szemlélteti a lokális belső osztályok használatát:

```
public class MethodLocal {

    public void method() {
        int a = 5;
        final int b = 6;

        class MethodLocalC {
            MethodLocalC() {
                // Ez nem működik
                // System.out.println(a);

                // De ez igen
                System.out.println(b);
            }
        }

        MethodLocalC localC = new MethodLocalC();
    }
}
```

3.9.3. A névtelen belső osztályok

Névtelen belső osztályokat akkor használunk, ha osztályból szeretnénk leszármazott osztályt létrehozni vagy interfészt megvalósítani, de a keletkező osztályt csak egy helyen használjuk. Ilyenkor az osztályt nem szükséges külön névvel definiálni, egyszerűen csak létrehozzuk a használat helyén. Használhatók bárhol, ahol objektumpéldányt kell megadni. Tipikusan értékadó kifejezésben vagy paraméterátadásban használjuk őket. A névtelen osztályok létrehozásának szintaxisa kicsit furcsa, ugyanis a new kulcsszó után írjuk a konstruktorhívást az ősosztály vagy ősinterfész típusával, majd kapcsos zárójelben következik az újradefiniálandó vagy implementálandó metódusok definíciója. Ez az egyetlen eset, hogy a new kulcsszót interfésznév követhet, ezek ugyanis természetükből adódóan nem példányosíthatók. Névtelen osztály csak egyetlen ősosztályt vagy interfészt specializálhat. Ha ez nem elegendő, akkor hagyományos definíciót kell alkalmaznunk. Az alábbi példa egy periodikusan lefutó időzített taszkot hoz létre, amely másodpercenként kiírja felváltva a „tic" és „tac" szavakat. Az időzítőt az osztálykönyvtár `Timer` osztályával (lásd 11.4. alfejezet) lehet használni. Ennek `schedule()` metódusa a `TimerTask` absztrakt osztály leszármazott osztályát várja, ez megvalósitja a `run()` metódust. A példa névtelen belső osztályt használ erre a célra. A példából is látható, hogy a névtelen belső osztályok használata áttekinthetetlenné teszi a kódot, ezért használatuk kerülendő.

```
public class TicTac {
    public static void main(String[] args) {
        Timer timer = new Timer();
        timer.schedule(new TimerTask() {
            boolean even = false;

            @Override
            public void run() {
                System.out.println(!even ? "tic" : "tac");
                even = !even;
            }
        }, 0, 1000);
    }
}
```

3.9.4. A statikus belső osztályok

A statikus belső osztályok valójában csak a névtér szempontjából belső osztályok, nem rendelkeznek ugyanis azzal a privilégiummal, hogy elérjék a tartalmazó osztály tagjait. Igazából nem is statikusak, a statikus jelző csupán azt jelenti, hogy ezek az osztályok a tartalmazó osztály példánya nélkül is példányosíthatók. Az alábbi programrészlet a játéktér és mezők példáját mutatja be statikus belső osztállyal. Ha a játékmezők életciklusát külön akarjuk kezelni, akkor praktikus lehet a belső osztály statikussá tétele, így ugyanis a tartalmazó osztálytól függetlenül példányosítható.

```
public class Maze2 {
    static class Field {
        int x;
        int y;
        ...
    }
    ...
    public static void main(String[] args) {
        // A belső osztály tartalmazó példány nélkül is használható
        Maze2.Field f = new Maze2.Field(5, 0);
    }
}
```

3.10. Az egyenlőség értelmezése

A `String` osztály kapcsán már felmerült, hogy a referenciális egyenlőség, és a szemantikai egyenlőség nem ugyanaz a fogalom. Ha az == operátorral hasonlítunk össze két objektumot, akkor az összehasonlítás a referenciákra vonatkozik. Tehát két különböző példány esetén mindig hamis eredményt kapunk, még akkor is, ha mindkét példány összes példányváltozójának értéke megegyezik. A szemantikai egyenlőség vizsgálatára a `String` osztály esetén az `equals()` metódust használtuk. Ez akkor ad vissza igaz értéket, ha a két példány ugyanazt a karakterláncot reprezentálja. Valójában az `equals()` metódust az `Object` ősosztály definiálja, és ezt a leszármazott osztályok újradefiniálhatják, hogy az a szemantikai egyenlőség vizsgálatára használható legyen. Természetesen a szemantikai egyenlőség, mint neve is utal rá, az osztály által reprezentált adatok értelmezésétől függ, ezért az egyenlőség pontos kritériumait a programozó választja meg. Karakterláncok esetén a fenti értelmezés volt ésszerű. A `File` osztály esetén azonban az a logikus, hogy két példányt akkor tekintsünk ekvivalensnek, ha ugyanazt az elérési utat reprezentálják. Az osztálykönyvtár is így valósítja meg a `File` osztály `equals()` metódusát. Saját osztályok esetén az értelmezés a programozóra van bízva. Gyakran az összes példányváltozót össze kell hasonlítani. Más esetekben, mint például adatbázis-entitások objektumokra történő leképezésénél, egyéni azonosítókat használunk, és csak ezeket hasonlítjuk össze. Fontos, hogy a `null` értékű paraméter kezeléséről se feledkezzünk el. Ebben az esetben `false` értéket kell visszaadni. Mivel az `equals()` metódust az `Object` osztály definiálja, ezért `Object` típusú paramétert vár. Tehát az újradefiniáláskor először az `instanceof` operátorral ellenőrizni kell a kapott paraméter típusát. Ha a vizsgált objektum típusa nem kompatibilis a metódus osztályával, akkor értelemszerűen nem is lehet ekvivalens a példányaival. Ha a típusvizsgálat sikeres volt, akkor típuskonvertálás után össze kell hasonlítani a megfelelő példányváltozók értékét.

Ha újradefiniáljuk az `equals()` metódust, akkor fontos újradefiniálni a `hashCode()` metódust is. Ezt szintén az `Object` osztály definiálja, és nincs paramétere. Minden példányhoz egy hashértéket ad vissza, ennek segítségével az objektum hashelést használó struktúrákban tárolható el. A metódusnak tehát a hashelés szabályai szerint kell működnie, azaz az `equals()` szerint ekvivalens példányokhoz azonos értéket kell visszaadnia. Különböző példányokhoz visszaadható ugyanaz az érték, de minél

kevesebb egyezés fordul elő, annál hatékonyabban fog működni a hashelést alkalmazó gyorsítótár. A hashfüggvények hatékony implementációja a matematika összetett területe, ezért túlmutat a könyv keretein. A helyes működéshez azonban fontos, hogy az ekvivalens példányok azonos hashértékkel rendelkezzenek. Ez könnyen elérhető például úgy, hogy a megkülönböztető példányváltozókat vagy azok közül a leggyakrabban eltérőket vesszük figyelembe, majd összeadjuk azok hash kódját, illetve primitív értékek esetén magukat az értékeket, végül az eredményt megszorozzuk 31-gyel. A `boolean` típus két értéke itt az `1` és `0` számokkal helyettesíthető. Az alábbi osztály egy webbankrendszer felhasználói fiókjait reprezentálja. Egy fiók a felhasználói azonosítóval és a bankszámlával azonosítható egyértelműen, egy bankszámlához ugyanis a tulajdonos által engedélyezett személyek is hozzáférhetnek, illetve egy személy több bankszámlához is rendelkezhet hozzáféréssel. Ehhez implementáljuk az `equals()` és a `hashCode()` metódusokat úgy, hogy erre a két példányváltozóra hagyatkozzanak:

```
public class WebAccount {
    private String login;
    private long accountNo;
    ...

    @Override
    public int hashCode() {
        int result = 0;
        result += login.hashCode();
        result += accountNo;
        result *= 31;
        return result;
    }

    @Override
    public boolean equals(Object obj) {
        if (this == obj)
            return true;
        if (obj == null)
            return false;
        if (!(obj instanceof WebAccount))
            return false;
        WebAccount other = (WebAccount) obj;
        if (accountNo != other.accountNo)
            return false;
        if (login == null) {
            if (other.login != null)
                return false;
        } else if (!login.equals(other.login))
            return false;
        return true;
    }
}
```

3.11. Az Object osztály metódusai

A következő lista összefoglalja az `Object` osztály legfontosabb metódusait és rövid leírásukat. Az osztály rendelkezik még néhány további metódussal is. Ezek többszálú programok esetén használhatók, ezért a 11. fejezet ismerteti őket.

`protected Object clone() throws CloneNotSupportedExcption`
A metódus objektumok másolatát készíti el.

`boolean equals(Object obj)`
Szemantikai egyenlőség vizsgálatára alkalmazható, ha újradefiniáljuk. Az alapértelmezett implementáció referenciális egyenlőség alapján hasonlít össze, akárcsak az == operátor.

`protected void finalize()`
A szemétgyűjtő hívja a már nem hivatkozott objektumpéldányokon, mielőtt megszüntetné azokat.

`int hashCode()`
Objektumpéldányhoz állít elő hashkódot. Ha az `equals()` metódust újradefiniáljuk, akkor annak megfelelően ezt is újra kell definiálni.

`String toString()`
Az objektumpéldány szöveges reprezentációját adja vissza.

Objektumokról másolatot a `clone()` metódussal készíthetünk. A metódust az `Object` osztály definiálja `protected` láthatósággal, tehát minden osztály örökli. A metódust a `protected` láthatóság miatt azonban a csomag osztályai csak akkor tudják hívni, ha azt újradefiniáljuk. Az örökölt kód hívásához használhatjuk a `super.clone()` utasítást. Annak ellenére, hogy minden osztály örökli a `clone()` metódust, az újradefiniáláson kívül a klónozható osztályokat meg is kell jelölni a `Cloneable` üres interfész implementálásával. Ha ezt nem tesszük meg, akkor a `clone()` metódus hívásakor `CloneNotSupportedException` kivételt kapunk (lásd 3.12. alfejezet). Ez jelzett kivétel, tehát akkor is kezelni kell, ha az osztály klónozható. Az interfész implementálása esetén az örökölt `clone()` metódus az objektumból új példányt hoz létre, és abba az összes példányváltozó értékét átmásolja. Referencia típusú példányváltozók esetén a referencia másolódik, a hivatkozott objektumról nem készül másolat. Az ilyen másolatot ezért *felületes másolatnak (shallow copy)* nevezzük. Ha a referenciákat is lemásoló *mély másolatot (deep copy)* kívánunk készíteni, akkor a `clone()` metódusban a hivatkozott objektumról is másolatot kell készíteni. A következő példa bemutatja a klónozás használatát. Macskákról tároljuk a nevüket és a tulajdonosaikat. A tulajdonosoknak szintén van nevük. A macskák tulajdonosáról mély másolat készül. A másolás után láthatjuk, hogy a másolat referenciálisan tényleg különbözik, de a nevek referenciálisan is egyeznek, ugyanis azokról csak felületes másolat készül. A tulajdonosok neve megegyezik, de a tulajdonosok referenciálisan eltérnek. Ebből láthatjuk, hogy mély másolat készült.

```
class Owner implements Cloneable {
    String name;

    public Owner(String name) {
        this.name = name;
    }

    @Override
    protected Object clone() throws CloneNotSupportedException {
        return super.clone();
    }
}

public class Cat implements Cloneable {
    String name;
    Owner owner;

    public Cat(String name, Owner owner) {
        this.name = name;
        this.owner = owner;
    }

    @Override
    protected Object clone() throws CloneNotSupportedException {
        Cat ret = (Cat) super.clone();
        ret.owner = (Owner) this.owner.clone();
        return ret;
    }

    public static void main(String[] args) throws ↺
CloneNotSupportedException {
        Owner o1 = new Owner("Gábor");
        Cat c1 = new Cat("Fuyur", o1);
        Cat c2 = (Cat) c1.clone();

        System.out.println(c1 == c2);
        System.out.println(c1.name.equals(c2.name));
        System.out.println(c1.owner.name.equals(c2.owner.name));
        System.out.println(c1.owner == c2.owner);
    }
}
```

3.12. A kivételkezelés

Egyes programozási nyelveken a hibákat úgy kezelik, hogy a könyvtári metódusok valamilyen kitüntetett, egyébként érvénytelen értékkel térnek vissza. Ha a könyvtári metódus eredménye bármilyen érték lehet, akkor ez nem valósítható meg. Ilyenkor szokás azt a megoldást választani, hogy a metódus maga egy státuszkóddal tér vissza, amely jelzi, hogy az sikeresen lefutott-e, a tényleges eredményt pedig egy paraméteren keresztül kapjuk meg.

A fenti módszerek hátránya, hogy az eredmény és a hibajelzés keveredik, ráadásul metódus meghívása után mindig vizsgálni kell, hogy hiba nélkül végrehajtódott-e. A Java objektumorientált kivételkezelésével elkerülhetők ezek a nem kívánt hatások. Amikor egy hívott metódusban hiba történik, akkor kivételt vált ki, a futása megszakad, és a hívó metódus kezd el futni. Itt a programozó kétféleképpen dönthet: vagy kezeli a kivételt, és valamilyen módon reagál rá (például egy hibaüzenet kiírásával), vagy továbbadja a hibát. Utóbbi esetben ez a metódus is abbahagyja a futást, és a kivétel ennek hívójához kerül, amely szintén ezekkel a választásokkal élhet. A kivétel így végiggyűrűzhet egészen a `main()` metódusig, amellyel a program elkezdte futását. Ekkor természetesen a futás is megszakad, és a virtuális géptől kapunk hibaüzenetet.

Ebben az alfejezetben a kivételkezelést tekintjük át.

3.12.1. A kivételtípusok

A kivételek valójában objektumok, tehát egy osztály példányai. Ez lehetővé teszi, hogy a hibákat típusuk alapján megkülönböztessük, és a kivétel a hiba típusát tükrözze. Egyrészt a kivétel konkrét típusa is hordoz információt, másrészt az objektum tagváltozói is tárolhatnak adatokat.

Az osztályokkal való kapcsolat miatt a kivételosztályok is osztályhierarchiába szerveződnek, tehát a kivételeknek nemcsak típusuk van, hanem tetszőleges mélységig megkülönböztethetünk leszármazott típusokat is. Az összes kiváltható kivétel a `Throwable` osztályból származik, ennek két fontos leszármazott osztálya van: az `Error` és az `Exception`. Az előbbi példányait a szó szoros értelmében nem is kivételeknek, hanem hibáknak nevezzük. Ezek olyan súlyos problémák, amelyek után a program már nem is tud folytatódni. Ha például ha elfogy a memória, akkor `OutOfMemoryError` hiba váltódik ki. Ezeket nem is szokás kezelni, és a továbbadásukat sem kell deklarálni.

Az `Exception` osztályba tartoznak a valódi kivételek. Ezek olyan problémákat reprezentálnak, amelyekre már érdemes reagálni. Az `IOException` jelzi például az I/O műveletek során fellépő hibákat. Ha valamely metódus ilyen típusú hibát válthat ki, vesszővel elválasztva fel kell sorolni ezeket a paraméterlista végén a `throws` kulcsszót követően. Ezért ezeket a kivételeket *jelzett kivételeknek (checked exception)* nevezzük. Metódus kétféleképpen válthat ki kivételt: vagy közvetlenül, valamilyen hiba esetén, vagy nem kezeli egy hívott metódus kivételeit, és akkor azok továbbadódnak. Mind a kétféle kivételt fel kell sorolnunk a `throws` kulcsszó után. Ez a deklaráció jelzi a metódust hívók számára, hogy ilyen kivételek váltódhatnak ki a metódushívás során. Az alábbi példa mutatja ezt a deklarációt:

```
public void parseFile(File f) throws FileNotFoundException, ↺
ObjectStreamException {
    ...
}
```

Az `Exception` osztálynak van egy `RuntimeException` leszármazott osztálya is. Ennek példányai olyan problémákat jelölnek, amelyek programozási hibából erednek. Például `IndexOutOfBoundsException` váltódik ki, ha egy tömb nem létező elemére hivatkozunk. Ezeket a kivételeket tehát a hibákhoz hasonlóan nem szükséges kezelni, sem továbbadásukat deklarálni. Az ilyen kivételeket *jelzetlen kivételeknek (unchecked exception)* nevezzük.

Természetesen saját kivételt is létrehozhatunk. Ezt úgy tehetjük meg, hogy valamely kivételosztályból leszármazott osztályt hozunk létre. A választott ős lehet a három nagy típus egyike vagy akár egy specifikusabb kivételosztály is. Például ha a készülő alkalmazás rendelkezik valamilyen konfigurációs fájllal, akkor az `IOException` osztályból származtathatunk új kivételt, amely a rossz formátumú konfigurációs fájlt jelzi. A kivételeket az osztályoknál érvényes konvenció szerint nevezzük el, és nevük az *Exception* szóra végződik.

3.12.2. Kivétel kiváltása

Kivételt tehát egy metódusban kétféleképpen válthatunk ki. Az egyik lehetőség, hogy egyszerűen nem kezeljük a hívott metódus által kiváltott kivételt. A másik lehetőség az, hogy a kiváltandó kivételből létrehozunk egy példányt, majd a `throw` utasítással kiváltjuk. Az utasítás utáni kódrészletek ekkor már nem futnak le. A kiváltás előtt a kivételt igényeink szerint inicializálhatjuk. Az osztálykönyvtárban a legtöbb kivételnek van olyan konstruktora, amelynek karakterláncban adhatjuk meg a hiba szöveges leírását. Ilyen konstruktort a saját kivételosztályokban is érdemes definiálni. A kivételt kezelő programrészlet így a `Throwable` osztálytól örökölt `getMessage()` segítségével ki tudja olvasni, és a hibaüzenet részeként megjeleníthető. Ezen kívül a saját kivételosztályokban a konkrét hibát jellemző egyéb járulékos információt is tárolhatunk. Az alábbi programrészlet példa kivétel kiváltására:

```
public static double divide(double dividend, double divisor) throws ↺
InvalidDivisorException {
    if (divisor == 0.0)
        throw new InvalidDivisorException("Cannot divide by zero.");
    return dividend / divisor;
}
```

3.12.3. A kivételek kezelése

Ahol a kivételekre reagálni szeretnénk, ott kezelni kell őket, és végre kell hajtani a válasznak szánt műveleteket. A kezelés a `try` utasítással történik. A `try` utasítás hagyományosan a következőképpen néz ki:

```
try {
    // ide kerülnek a kivételt kiváltó metódushívások
} catch (FirstException e) {
    // Itt kezeljük le az FirstException típusú kivételeket
} catch (SecondException e) {
    // Itt pedig a SecondException típusúakat
}

// Szerepelhet itt még tetszőleges számú catch-blokk

finally {
   // Ide kerül a try-blokk végén mindig lefuttatandó kód
}
```

Látható, hogy különféle kivételtípusok esetén választhatunk különféle hibakezelést. A `finally`-blokkban megadott utasítások a `try`-blokk után mindig lefutnak, akár váltódott ki kivétel, akár nem. Ez praktikusan használható az erőforrások felszabadítására. A `finally`-blokk el is hagyható, ha nincs rá szükség. A `catch`-blokkok is elhagyhatók, ekkor viszont a `finally`-blokknak mindenképpen szerepelnie kell. Ebben az esetben a kivételt nem kezeljük, hanem továbbadjuk a hívó metódusnak, előtte azonban végrehajtódik a `finally`-blokk.

A Java SE 7-es verziójától kezdve arra is lehetőség van, hogy egy `catch`-blokkban több különböző kivételt kezeljünk. Ez akkor hasznos, ha ugyanúgy akarjuk őket kezelni, mert így nem szükséges a hibakezelő kódot megismételnünk. A kivételeket | jellel elválasztva soroljuk fel:

```
catch (FirstException|SecondException e) { ... }
```

Szintén a 7-es verzió újítása az erőforrások kényelmesebb kezelése. Ez a mechanizmus automatikusan felszabadítja az `AutoCloseable` interfészt megvalósító erőforrásokat, tehát a `finally`-blokk elhagyhatóvá válik. Az osztálykönyvtár erőforrásosztályai ilyenek. Ezt a mechanizmust úgy használhatjuk, hogy a `try` kulcsszó és az utána következő kapcsos nyitójel közt kerek zárójelben deklaráljuk és inicializáljuk az erőforrásként kezelt objektumokat. Ez a mechanizmus az *elnyomott kivételek (suppressed exceptions)* problémájára is megoldást ad. Ha a `try`-blokkban kivétel váltódik ki, majd a `finally` lefutása újabb kivételt eredményez, akkor a hívó metódus az utóbbit kapja csak meg. A második kivétel egyszerűen elnyomja az elsőt. A `try` utasítás új szintaxisával ettől a második kivételtől megkaphatjuk az elnyomott kivételt is a `getSuppressed()` metódus használatával. Ezért az elnyomott kivételre is tudunk reagálni.

A következő példaprogram bemutatja a `try` utasítást. A korábbi `divide()` metódust hívjuk meg, és kezeljük az általa kiváltott kivételt. Kivétel esetén hibajelzést adunk, illetve megjelenítjük a kivételben található üzenetet is. Az eredményt fájlba írjuk, ehhez a `try`-blokk erőforrás-kezelését használjuk. Itt szintén kiváltódhat kivétel, ha a fájl nem írható:

```
public static void main(String[] args) {
    if (args.length < 2) {
        System.out.println("Meg kell adni paraméternek két számot");
        System.exit(-1);
    }

    double dividend = Double.parseDouble(args[0]);
    double divisor = Double.parseDouble(args[1]);

    try (FileWriter writer = new FileWriter("result");) {
        double result = divide(dividend, divisor);
        writer.append(Double.toString(result));
    } catch (InvalidDivisorException e) {
        System.out.println("Hiba az osztás során: " +
            e.getMessage());
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

Az `AutoCloseable` interfész egyetlen `close()` metódust deklarál. Az interfészt implementálva saját osztályainkat is használhatjuk erőforrásként a `try`-blokkban.

3.12.4. Kivételkezelési tanácsok

Mivel a kivételek hierarchikus típusokba tartoznak, ezért az `Exception` ősosztály kezelésével egyetlen `catch`-blokkban is kezelhetnénk az összes hibát. A lusta programozót ez könnyen kísértésbe ejtheti, de ez a megoldás több szempont miatt sem lenne szerencsés. Először is, a típusos kivételek lényege pontosan az, hogy a kivételtípus a hiba típusára utaljon. Ez a megoldás elfedi ezt a megkülönböztetést. Másodszor, ha a `try`-blokkban hívott kód módosul, és esetleg új, eddig nem használt kivételeket is kiváltunk, akkor ezt valószínűleg észre sem vesszük, mert ez is kezelődik. Ha típusonkénti `catch`-blokkokat adtunk volna meg, akkor az új kivételt nem kezelné egyik sem, így a program nem fordulna le. Ez jelezné a programozónak, hogy új kivételt is kell kezelni. A második problémára megoldást jelent a kivételek | jellel való felsorolása, de az első probléma még mindig fennáll. Ezt úgy orvosolhatjuk, hogy a naplóba vagy a képernyőre írt hibaüzenetbe a kivétel típusát is belefoglaljuk, így később azonosítható lesz a konkrét hiba.

A `throws` deklarációban is meg kell fontolni, hogy mennyire specifikus kivételt adunk meg. Itt sem jó az általános `Exception` használata, mert nem érvényesül a kivételek típusossága, és a kivételt kezelő metódus nem tudja a hibatípusokat megkülönböztetni. A másik probléma szintén fennáll, azaz ha a kódot módosítjuk, és új kivétel is kiváltódhat a metóduson belül, akkor a hívók erről nem értesülnek. Soroljuk fel ezért mindig egyenként az összes lehetséges kivételtípust.

A jelzetlen kivételek kezelhetők ugyan, de azok általában programozói hibára utalnak. Ha ilyen kivétel váltódik ki, akkor a program már valószínűleg hibás állapotban van, ahonnan a futása nem tud megbízhatóan folytatódni. Az ilyen hibákat a kódban

kell javítani, a jelzetlen kivételek kezelése csak elfedné ezeket a hibákat. Ne kezeljük ezért ezeket a kivételeket, hanem hagyjuk, hogy a hiba megjelenjen, és a felhasználó vagy a tesztelő a fejlesztőknek jelezhesse.

3.13. Az enumeráció, mint osztály

Az enumeráció a Java nyelvben valójában többet jelent a 2.6.6. alfejezetben leírtaknál. Az enumeráció tulajdonképpen osztály, amelynek minden lehetséges értékét az osztály egy-egy példánya reprezentálja. Az értékek nevével elérhetjük az értékeket reprezentáló objektumpéldányokat. Ha az enumerációt hagyományos osztállyal reprezentálnánk, akkor a nevek olyan osztályváltozóknak felelnének meg, amelyek egy-egy lehetséges példányra hivatkoznak. Az enumeráció más osztályokhoz hasonlóan definiálhat metódusokat, és van néhány közös metódus, amelyet minden enumeráció örököl a típusparaméterrel rendelkező (lásd 5.2. alfejezet) `Enum` osztálytól. Az enumerációk metódusait a következő lista sorolja fel. Az adott enumeráció konkrét típusát `E` jelöli.

`public static E valueOf(String arg)`
: A karakterláncként megadott értékhez tartozó enumerációpéldányt adja vissza.

`public static E[] values()`
: Visszaadja az összes enumerációpéldányt.

`public final int compareTo(E o)`
: Az aktuális példányt hasonlítja a megadotthoz a sorszáma szerint. Aszerint ad vissza rendre negatív, nulla vagy pozitív értéket, hogy az aktuális példány pozíciója kisebb, egyenlő vagy nagyobb a megadottnál.

`public final boolean equals(Object other)`
: Enumeráció értékének összehasonlítására használható. Mivel minden értékhez egyetlen példány létezik, ezért használható az == operátor is, és az enumerációk a `switch` utasítással is működnek.

`public final String name()`
: Az enumerációpéldány nevét adja vissza.

`public final int ordinal()`
: Az adott enumerációpéldány sorszámát adja vissza az értékek felsorolásában elfoglalt helye szerint.

`public String toString()`
: Az adott enumerációpéldány nevét adja vissza, akárcsak a `name()`, de ez a metódus újradefiniálható, így kiírhat emberközelibb reprezentációt is.

Saját metódusok is definiálhatók, akár statikusak is. Szintén szükség lehet a példányok bizonyos jellemzőinek eltárolására. Ezeket példányváltozókban tárolhatjuk el. A példányváltozóknak konstruktorokkal adunk értéket. A szokott módon definiálunk egy konstruktort, amely paraméterben átveszi az incializálni kívánt értéket, majd az enumeráció értékei után zárójelben megadjuk, hogy az adott értékhez milyen

paraméterekkel hívódjon a konstruktor. A következő példa szemlélteti az enumerációk lehetőségeit. Az enumeráció hosszmértékegységeket reprezentál, mindegyik értékhez eltárolja a mértékegység rövidítését és a méterhez viszonyított váltószámot. Az enumeráció egy statikus metódusa a rövidítés alapján képes visszaadni a megfelelő enumerációpéldányt. Ezt úgy teszi, hogy az enumerációknál rendelkezésre álló `values()` metódus segítségével bejárja az összes értéket, amíg meg nem találja a keresett enumerációpéldányt. A `toString()` metódus úgy lett újradefiniálva, hogy a rövidítést írja ki, a `toMeters()` pedig az enumerációpéldánynak megfelelő mértékegységben értelmezett hosszt adja vissza méterben:

```
public enum LengthUnit {
    METER(1.0, "m"), YARD(0.9144, "yd"), FOOT(0.3048, "ft"), INCH↩
(0.0254, "in");

    private String abbrev;
    private double meters;

    private LengthUnit(double meters, String abbrev) {
        this.meters = meters;
        this.abbrev = abbrev;
    }

    public static LengthUnit parseAbbrev(String abbrev) {
        for (LengthUnit u : LengthUnit.values()) {
            if (u.abbrev.equals(abbrev))
                return u;
        }
        return null;
    }

    public double toMeters(double x) {
        return x * meters;
    }

    @Override
    public String toString() {
        return abbrev;
    }
}
```

Az alábbi példában normál osztálydefiníciót adtunk meg, amely a fenti enumerációhoz hasonlóan működik. Ez a példa csupán annak szemléltetésére szolgál, hogyan viszonyul az enumeráció az osztályokhoz. Látható, hogy az enumeráció funkcionalitása gyakorlatilag egy átlagos osztállyal is megvalósítható, de az enumeráció definíciójának szintaxisa egyszerűbb, és jobban tükrözi az enumeráció mögötti jelentést. Az osztály konstruktora `private`, így csak az osztály kódjából tudjuk meghívni. Egy statikus inicializációs blokkban hozzuk létre a lehetséges értékeknek megfelelő példányokat, és

ezeket osztályváltozókban tároljuk el. A példányok a megfelelő értékekkel vannak inicializálva, akárcsak az enumerációban. A `values()` metódust itt nekünk kellett definiálni, mivel nem enumerációt használtunk:

```
public class LengthUnitClass {
    public static LengthUnitClass METER;
    public static LengthUnitClass YARD;
    public static LengthUnitClass FOOT;
    public static LengthUnitClass INCH;

    public static LengthUnitClass[] values() {
        return new LengthUnitClass[] { METER, YARD, FOOT, INCH };
    }

    static {
        METER = new LengthUnitClass(1.0, "m");
        YARD = new LengthUnitClass(0.9144, "yd");
        FOOT = new LengthUnitClass(0.3048, "ft");
        INCH = new LengthUnitClass(0.0254, "in");
    }

    private String abbrev;
    private double meters;

    private LengthUnitClass(double meters, String abbrev) {
        this.abbrev = abbrev;
        this.meters = meters;
    }

    public static LengthUnitClass parseAbbrev(String abbrev) {
        for (LengthUnitClass u : LengthUnitClass.values()) {
            if (u.abbrev.equals(abbrev))
                return u;
            }
            return null;
    }

    public double toMeters(double x) {
        return x * meters;
    }

    @Override
    public String toString() {
        return abbrev;
    }
}
```

Az enumerációk példányai tehát szintén specializációt fejeznek ki. Felvetődhet a kérdés, hogy különböző entitások megkülönböztetésére mikor milyen programozási megoldást használjunk. Leszármazott osztály használata akkor célszerű, ha a leszármazott osztályokat ténylegesen külön típusnak kell tekinteni, és azok viselkedése algoritmikusan is különbözik, nemcsak paraméterekben tér el. Ha ugyanis a viselkedés paraméterezhető, akkor egyrészt nehezen beszélhetünk különböző típusról, másrészt a paramétereket példányváltozókban tárolhatjuk. A kívánt viselkedés tehát ekkor objektumpéldányokkal is elérhető, nem szükséges típushierarchiát létrehozni. Az enumeráció is ehhez hasonló funkcionalitást nyújt, de minden paraméterkombináció csak egyszer létezik, valamint az enumeráció definíciójának módosítása nélkül nem tudunk új enumerációpéldányokat létrehozni. Enumerációt tehát akkor érdemes használni, ha kevés felvehető állapotkombináció van, és ezek előre ismertek, illetve csak egyetlen példányra van szükség belőlük. Ha egy osztályból sok statikus példányt hozunk létre, akkor érdemes elgondolkodni azon, hogy az enumeráció használata megfelelőbb-e.

3.14. A JavaBeans-konvenciók

A *JavaBeans szabvány* Java-objektumok újrafelhasználható komponensként történő felhasználását szabványosítja. Definiál néhány programozási konvenciót, amelyeket a JavaBeans-osztályoknak követniük kell, illetve osztálykönyvtárat biztosít a konvencióknak megfelelő osztályok kezeléséhez. Az osztálykönyvtár olyan funkcionalitást kínál a JavaBeans-osztályokhoz, mint például tulajdonságok kezelése, tulajdonságszerkesztők támogatása, változásokkal kapcsolatos eseménykezelés. A Swing keretrendszer (lásd 10. fejezet) jelentősen épít a JavaBeans-szolgáltatásokra. Most csak a programozási konvenciókat ismertetjük. Ezek közül az elnevezési szabályokat érdemes akkor is követni, ha nem vesszük igénybe a JavaBeans-szolgáltatásokat. A következő lista foglalja össze a JavaBeans-konvenciókat:

- Az osztály legyen sorosítható (lásd 6. fejezet).
- Az osztálynak legyen alapértelmezett konstruktora.
- A tulajdonságokat reprezentáló példányváltozók beállításához létezzen setter metódus, és ennek neve a `set` szóból és a tulajdonság nevéből álljon, például:

  ```
  void setSizeLimit(long sizeLimit)
  ```

  A kiolvasáshoz biztosítsunk getter metódust, ennek neve a `get` vagy `boolean` típus esetén az `is` szóval kezdődjön, például:

  ```
  long getSizeLimit()
  ```

  Ez a konvenció lehetővé teszi, hogy a tulajdonságokat a különböző keretrendszerek fel tudják deríteni, valamint azok könnyen szerkeszthetők legyenek.

NEGYEDIK FEJEZET
A Java SE osztálykönyvtára

Az előző fejezetek bemutatták a Java nyelv elemeit és az objektumorientált eszközök használatát. Mindez erős eszköztárat ad a programozó kezébe, de még nem elég a valós, életszerű alkalmazások kifejlesztéséhez. Ehhez az osztálykönyvtárat is igénybe kell vennünk, ez teszi ugyanis lehetővé olyan funkcionalitások elérését, mint például a karakterláncok feldolgozása, a bonyolult matematikai műveletek elérése, a dátumok kezelése, a fájlok írása és olvasása, valamint a könyvtár- és fájlműveletek végrehajtása. A fejezet részletesen bemutatja az osztálykönyvtár használatát.

4.1. A karakterláncok kezelése

Ez az alfejezet a karakterláncok kezeléséhez használható technikákat ismertet. Tárgyaljuk a karakterlánc-műveletek hatékony elvégzését, a reguláris kifejezések használatát és a tokenekre bontást is.

4.1.1. A gyakran változó karakterláncok

Ugyan a + operátorral lehetőség van karakterláncok összefűzésére, illetve a `String` osztály metódusaival is sok műveletet végezhetünk, a `String` osztály példányai alapvetően nem változtathatók meg. Ha megfigyeljük a `String` osztály metódusait, láthatjuk, hogy nem az eredeti karakterláncot módosítják, hanem újat adnak vissza. Ugyanez történik az összefűzés során is. Ez elfogadható egy-egy művelet esetén, de ha sok műveletet kell végezni egy karakterláncon, akkor számos `String`-példány jön létre. Ez jelentős költséggel jár, ezért ebben az esetben hatékonyabb megoldást jelent a `StringBuilder` osztály használata. Az osztály szintén karakterláncot reprezentál, habár nem leszármazottja neki. A `String` osztály ugyanis `final` módosítóval van megjelölve, ezért nem hozhatók létre belőle leszármazott osztályok. A `StringBuilder` metódusai a `String` osztályéhoz hasonló funkcionalitást nyújtanak, de nem hoznak létre új példányt, hanem a meglévő állapotát módosítják.

Az `append()` metódusnak megadhatunk primitív típust, karakterláncot, karaktertömböt, másik `StringBuilder`-példányt vagy akármilyen objektumot, és azt vagy a karakterlánc reprezentációját hozzáfűzi az aktuális `StringBuilder` által tárolt karakterlánchoz. Az `insert()` beszúrást végez el, első paramétere a kezdőpozíció, ahova a beszúrandó karakterek kerülnek, második paramétere pedig szintén bármilyen típus lehet. A `delete()` metódus a karakterlánc egy részének törlésére szolgál, és két `int` paramétert vár. Az első a törlés kezdőpozíciója, a második pedig az utolsó törlendő pozíciónál eggyel nagyobb szám. A `deleteCharAt()` csupán egy karaktert töröl ki, és ennek az indexét várja paraméterben. A `replace()` cserét hajt végre. Három paramétert vár: az első kettő a cserélendő pozíciót jelöli ki, ahogyan a `delete()` metódus is, a

harmadik pedig a helyette beszúrandó `String`. A `reverse()` megfordítja a karakterlánc karaktereinek sorrendjét.

Az osztálykönyvtár és az egyéb keretrendszerek metódusai legtöbbször `String` paramétereket várnak, és mivel a `StringBuilder` nem leszármazottja ennek, közvetlenül nem használható helyette. Amikor a kívánt műveleteket elvégeztük a karakterláncon, a `StringBuilder` objektumot tehát `String` típusúra kell konvertálni, hogy más metódusnak átadhassuk. Ezt a `toString()` metódussal tehetjük meg. Ha csak a karakterlánc egy részére van szükség, akkor használható a `substring()` metódus is. Ennek két változata van. Egyiknek csak a kezdőpozíciót kell megadni, és attól kezdve a karakterlánc végéig tartó részt választja ki. A másik változattal a karakterlánc végét is kijelölhetjük a végső pozíciónál eggyel nagyobb számmal. Az alábbi példa szemlélteti a `StringBuilder` használatát:

```
StringBuilder sb = new StringBuilder("Helló,");
sb.replace(4, 5, "ó");
sb.append(" világ!");
sb.deleteCharAt(11);
// eredmény: Helló, világ!
System.out.println(sb);
```

A `StringBuffer` osztály a `StringBuilder` funkcionalitását nyújtja, de szinkronizáltak a metódusai, ezért többszálú környezetben használható (lásd 11.7. alfejezet). Ha csak egy szálból szükséges a karakterláncot módosítani, akkor nem javasolt a használata, a szinkronizáció ugyanis költséggel jár, ezért csökkenti a hatékonyságot.

4.1.2. A reguláris kifejezések használata

A reguláris kifejezések szövegminták, amelyekre szöveg vagy annak része illeszkedhet. A reguláris kifejezésekkel karakterláncok elvárt tulajdonságait adhatjuk meg, ezért ezeket használhatjuk például részletes kereséshez vagy a felhasználó által megadott bemenet validálásához. Például HTTP-erőforrásokra hivatkozó URL-ek formátuma hozzávetőlegesen megadható a következő reguláris kifejezéssel: *`http://[a-zA-Z./]+`*. Ez a reguláris kifejezés azt jelenti, hogy az URL a *`http://`* karakterlánccal kezdődik, majd a szögletes zárójelben megadott karakterek ismétlődnek legalább egyszer. Erre a kifejezésre érvénytelen URL-ek is illeszkednek, például *`http://a//`*, de számos alapvető hibát kiszűr, például a ponton és a perjelen kívül nem engedi meg írásjelek előfordulását az URL-ben. Az olvasóra bízzuk, hogy ennél pontosabb reguláris kifejezést készítsen. A reguláris kifejezések megvalósítása programozási nyelvenként eltér, a 4.1. táblázat összefoglalja a Java nyelv reguláris kifejezéseiben leggyakrabban használt elemeket.

4.1. táblázat: A reguláris kifejezésekben használt elemek

Jelölés	Jelentés
x	az *x* karakter
\\	a \ karakter
\t	a tabulátor karakter
\n	a soremelés karakter
\r	a kocsivissza karakter
[*abc*]	az *abc* karakterek bármelyike
[^*abc*]	az *abc* karakterektől eltérő karakter
[*a-z*]	az *a-z* intervallumba eső bármely karakter
.	bármely karakter
\d	számjegy
\D	nem számjegy
\s	whitespace karakter
\S	nem whitespace karakter
\w	számjegy, betű vagy aláhúzásjel
\W	nem számjegy, betű, aláhúzásjel
^	a sor eleje
$	a sor vége
\b	szóhatár
\B	nem szóhatár
\A	a bemenet kezdete
\G	az előző illeszkedés vége
\z	a bemenet vége
X?	*X* nullaszor vagy egyszer
*X**	*X* nullaszor vagy többször
X+	*X* egyszer vagy többször
X{*n*}	*X* pontosan *n* alkalommal
X{*n*,}	*X* legalább *n* alkalommal
X{*n*,*m*}	*X* legalább *n*, legfeljebb *m* alkalommal
XY	*X*, utána *Y*
X|*Y*	*X* vagy *Y*
(*X*)	*X* mint részkifejezés
n	*n*-edik részkifejezés

A `String` osztály rendelkezik négy metódussal, amelyek reguláris kifejezéseket használnak, ezeket alább ismertetjük.

`boolean matches(String regex)`
Igazat ad vissza, ha a karakterlánc illeszkedik a megadott reguláris kifejezésre.

`String replaceAll(String regex, String replacement)`
Új karakterláncot ad vissza, amelyben a megadott reguláris kifejezés összes illeszkedését kicseréli a megadott karakterlánccal.

`String replaceFirst(String regex, String replacement)`
Új karakterláncot ad vissza, amelyben a megadott reguláris kifejezés első illeszkedését kicseréli a megadott karakterlánccal.

`String[] split(String regex)`
A reguláris kifejezés illeszkedéseinél darabokra vágja a karakterláncot, és visszaad egy tömböt a darabokból.

A `java.util.regex` csomag osztályai bővebb funkcionalitást kínálnak a reguláris kifejezések használatához. Először a mintával `Pattern` objektumot kell létrehoznunk. Ennek segítségével készíthető egy `Matcher` objektum, ezzel bonyolult illesztések és cserék végezhetők. Ezeket a fejezetben nem részletezzük. Az alábbi kódrészlet bemutatja, hogyan használhatók a reguláris kifejezések e-mail címek validálására, illetve a @ jel mentén tokenizálásra. A validáláshoz használt reguláris kifejezés természetesen csak a leggyakoribb eseteket fedi le, és nem vizsgálja például a felső szintű domének helyességét.

```
String email1 = "foo@bar.hu";
String email2 = "rossz@email@cim.com";
final String re = "([a-zA-Z0-9_]*)@([a-zA-Z0-9_]*)\\.[a-zA-Z]*";

System.out.println(email1.matches(re));
System.out.println(email2.matches(re));
System.out.println(Arrays.toString(email1.split("@")));
```

4.1.3. A tokenizálás

A `String` osztály `split()` metódusával is lehetséges karakterláncok tokenekre bontása, de a metódus reguláris kifejezést vár, és ez körülményessé teszi több határolókarakter megadását. Ezen kívül a reguláris kifejezések feldolgozása lassabb, mint az egyszerű karakterek mentén történő felbontás. A `StringTokenizer` osztály segítségével egyszerűbben és hatékonyabban végezhető el a tokenizálás. A konstruktornak meg kell adni a tokenizálandó karakterláncot. Alapértelmezésben a felbontás a whitespace karakterek mentén történik. Az opcionális második paraméterben karakterláncként megadhatók a határolókarakterek. A karakterlánc minden karaktere határolóként fog működni. Ezután a `StringTokenizer` objektum `countTokens()` metódusa visszaadja a tokenek számát. A soron következő token a `nextToken()` metódussal kapható meg, a `hasMoreTokens()` pedig azt adja meg, hogy van-e még fennmaradó token. Az alábbi példa szemlélteti vesszővel felsorolt szavak tokenizálását.

```
String str = "Merkúr,Vénusz,Mars,Föld";
StringTokenizer tokenizer = new StringTokenizer(str, ",");
while (tokenizer.hasMoreTokens())
    System.out.println(tokenizer.nextToken());
```

4.2. A System osztály

A `System` osztály a futtatókörnyezettel kapcsolatos általános információk lekérdezésére és végrehajtására használható. Nem példányosítható, minden metódusa osztályszintű. Egyik kiemelt feladata a rendszerbeállítások kezelése. Ezek olyan alapvető információkat tárolnak a rendszerről, mint például a Java futtatókörnyezet verziója vagy az aktuális munkakönyvtár. A legfontosabb rendszerbeállításokat a 4.2. táblázat ismerteti.

4.2. táblázat: A rendszerbeállítások és jelentésük

Rendszerbeállítás	Jelentése
java.version	a JRE verziója
java.vendor	a JRE készítője
java.home	a Java telepítési könyvtára
java.library.path	az elérési utak, ahol a Java alapértelmezésben osztálykönyvtárakat keres
java.io.tmpdir	az alapértelmezett ideiglenes könyvtár
os.name	az operációs rendszer neve
os.arch	az operációs rendszer architektúrája
file.separator	az elérési utak komponenseinek elválasztókaraktere (Windowson \)
path.separator	több elérési út elválasztókaraktere a classpath megadásánál (Windowson ;)
line.separator	a platform sorhatároló karaktere (Windowson kocsivissza és soremelés)
user.name	a programot futtató felhasználó felhasználóneve
user.home	a felhasználó home könyvtára
user.dir	az aktuális munkakönyvtár

Ezektől eltérő alkalmazásspecifikus rendszerbeállítások is megadhatók a **java** parancs **-D** kapcsolójával. A rendszerbeállítások értéke a `getProperty()` metódussal kérdezhető le. Ennek a rendszerbeállítás nevét adjuk meg karakterláncként, és az értéket is ilyen formában kapjuk meg. Az opcionális második paraméterben alapértelmezett értéket lehet megadni. Ha a rendszerbeállítás nincs beállítva, akkor a metódus ezt adja vissza. A rendszerbeállítások a `Properties` objektumban egyszerre is lekérdezhetők a `System` osztálytól (lásd 6.1. alfejezet).

A System osztály tagváltozóival érhető el a program szabványos ki- és bemenete, illetve szabványos hibafolyama is. Utóbbi a szabványos kimenethez hasonló, de nem az általános kimenet, hanem a hibaüzenetek kiírására szolgál. A három folyamot a 4.3. táblázat írja le.

4.3. táblázat: A szabványos folyamok

Tagváltozó	Típus	Funkció
err	PrintStream	szabványos hibafolyam
in	InputStream	szabványos bemeneti folyam
out	PrintStream	szabványos kimeneti folyam

A System osztály többi fontos metódusát az alábbi lista foglalja össze.

`void exit(int status)`
A megadott státuszkóddal befejezi a program működését. Konvenció szerint a nulla státuszkód sikeres futást jelent, a negatív értékek pedig hibát.

`void gc()`
Javaslatot tesz a virtuális gépnek a szemétgyűjtés elvégzésére. A virtuális gép figyelmen kívül hagyhatja a kérést, ezért nem biztos, hogy a szemétgyűjtés ténylegesen megtörténik.

`Map<String,String> getenv()`
Szótárban adja vissza a környezeti változókat és értéküket.

`String getenv(String name)`
Visszaadja a megadott környezeti változó értékét, vagy ha az nincs beállítva, `null` értéket.

`void setErr(PrintStream err)`
A szabványos hibafolyamot a megadott folyamba irányítja át.

`void setIn(InputStream in)`
A szabványos bemeneti folyamot a megadott folyamba irányítja át.

`void setOut(PrintStream out)`
A szabványos kimeneti folyamot a megadott folyamba irányítja át.

4.3. A matematikai műveletek

A Math osztály statikus metódusokat kínál matematikai műveletek elvégzéséhez. Konstansokként teszi elérhetővé ez e-nek, a természetes logaritmus alapjának, illetve a π-nek az értékét. Az alábbi lista a konstansokat és a legfontosabb metódusokat foglalja össze. Sok metódus többféle paramétert is elfogad, ezeket nem soroljuk fel, de helyüket három pont jelzi.

`static double E`
A természetes logaritmus alapját tárolja.

`static double PI`
A π értékét tárolja.

`static long abs(long a)`
`...`
A megadott szám abszolút értékét adja vissza.

`static double cbrt(double a)`
A szám köbgyökét számítja ki.

`static double ceil(double a)`
Az adott számot felfelé kerekíti egészre.

`static double cos(double a)`
A szám koszinuszát adja vissza.

`static double exp(double a)`
Az e szám *a-adik* hatványát határozza meg.

`static double floor(double a)`
Az adott szám egész részét adja vissza.

`static double log(double a)`
A megadott érték természetes logaritmusát számítja ki.

`static double log10(double a)`
A megadott érték tízes alapú logaritmusát számítja ki.

`static double max(double a, double b)`
`...`
A két szám maximumát adja vissza.

`static double min(double a, double b)`
`...`
A két szám minimumát adja vissza.

`static double rint(double a)`
A számot a legközelebbi egészre kerekíti, az öt tizedre végződő számokat mindig a páros szomszédra.

`static long round(double a)`
`static int round(float a)`
A megadott értéket a matematikai kerekítés szabályai szerint egészre kerekíti, azaz a számokat öt tizedtől kerekíti felfelé.

`static double sin(double a)`
A megadott érték szinuszát számítja ki.

`static double sqrt(double a)`
A szám négyzetgyökét számítja ki.

`static double tan(double a)`
A megadott érték tangensét adja vissza.

Ezeken a műveleteken kívül gyakran szükséges véletlenszámot előállítani. Ehhez a `java.util` csomag `Random` osztálya használható. Az osztály valójában álvéletlenszám-generátor. Azaz *seed értékkel* inicializáljuk, és azonos seed érték, valamint azonos metódushívások esetén mindig ugyanazokat az értékeket adja vissza ugyanabban a sorrendben. Az alapértelmezett konstruktora törekszik arra, hogy mindig más seed értékkel inicializálja az objektumot. Van `long` paramétert váró konstruktora is, ennek közvetlenül megadhatjuk a seed értékét. A nehezebb megjósolhatóság érdekében a seed megállapításához alapul vehetjük például a milliszekundumban mért aktuális időt. A seed később is megváltoztatható a `setSeed()` metódussal. Miután az osztály példánya létrejött, különböző típusú véletlenszámokat állíthatunk vele elő. Az ehhez használható metódusokat a következő lista foglalja össze:

`boolean nextBoolean()`
Véletlen `boolean` értéket állít elő.

`void nextBytes(byte[] bytes)`
Véletlen bájtokat állít elő, és beteszi azokat a megadott tömbbe.

`double nextDouble()`
Véletlen `double` értéket állít elő a [0,0, 1,0) intervallumon.

`float nextFloat()`
Véletlen `float` értéket állít elő a [0,0, 1,0) intervallumon.

`double nextGaussian()`
Véletlen `double` értéket állít elő a Gauss-eloszlás szerint. Az érték várható értéke `0`, szórása `1`, értelmezési tartománya a $-\infty$ és $+\infty$ közti valós számok halmaza.

`int nextInt()`
Véletlen egész értéket állít elő az `int` típus teljes értékkészletén egyenletes eloszlással.

`long nextLong()`
Véletlen egész értéket állít elő a `long` típus teljes értékkészletén egyenletes eloszlással.

A `Random` osztály a hétköznapi alkalmazásokhoz megfelelő, de nem nyújt kriptográfiai értelemben biztonságos véletlenszám-generálást. Ha ilyenre van szükségünk, használjuk a `java.security.SecureRandom` osztályt. Ez leszármazottja a `Random` osztálynak, ezért helyette bárhol használható.

4.4. A dátum és az idő kezelése

A dátum és az idő kezelésére az osztálykönyvtár `Date` és `Calendar` osztályait használhatjuk. Az előbbi legtöbb metódusa elavult, ugyanis nem támogatja az internacionalizációt (lásd 14. fejezet), és a dátumokkal kapcsolatos számítások elvégzését sem túl célszerűen valósítja meg. Az osztály megismerése mégis hasznos, mert hidat alkot a `Calendar` és a dátumok nyelvfüggő kiírásához használt `DateFormat` osztályok között. Ezen kívül találkozhatunk az osztállyal régebbi kódokban, valamint egyszerűen használható a dátum és az idő gyors kiírására az alábbi módon.

```
Date date = new Date();
System.out.println(date);
```

A `Date` valójában az időt `long` értékként, az 1970. január 1-je 00:00:00 óta eltelt milliszekundumokkal reprezentálja. Erre a dátumra később *referenciaidőként (epoch time)* hivatkozunk.

A `Calendar` osztály jobban támogatja dátumokkal kapcsolatos számítások elvégzését. Ez absztrakt osztály, ezért közvetlen nem példányosítható, hanem általában a `getInstance()` statikus factorymetódus valamelyik változatát használjuk. A paraméter nélküli változat az alapértelmezett lokalizációnak megfelelő példányt adja vissza. Valójában a Java 7 csak egyféle implementációt kínál, ez pedig a `GregorianCalendar`, azaz a világon legelterjedtebb Gergely-naptár megvalósítása. Természetesen egyes kultúrák saját naptárrendszerrel rendelkeznek, de kivétel nélkül ismerik és használják a Gergely-naptárt is. A Gergely-naptár hónapjaira és napjaira a különböző nyelvek saját megnevezést használnak. A lokalizációtól függően a visszaadott példány eltérhet ezekben a részletekben, de alapvetően a Gergely-naptárt használja. Az osztálykönyvtár tehát nem kínál teljes internacionalizációt a naptárrendszerhez, de a `Calendar` osztályra építve létrehozhatunk saját implementációkat.

A `Calendar` metódusai három nagy csoportba sorolhatók. A metódusok az osztályban definiált konstansokra épülnek. A konstansok segítségével jelöljük ki, hogy a dátum vagy az idő melyik komponensét akarjuk lekérdezni vagy beállítani. Az alábbi lista ismerteti a legfontosabb konstansokat:

`YEAR, MONTH, WEEK_OF_YEAR, WEEK_OF_MONTH, DAY_OF_YEAR, DAY_OF_MONTH, DAY_OF_WEEK`

: A dátumok lekérdezésénél vagy beállításánál határozzák meg rendre a dátum következő komponenseit: az év, a hónap, a hét sorszáma az évben, a hét sorszáma a hónapban, a nap sorszáma az évben, a nap sorszáma a hónapban, a nap sorszáma a héten. A hét napjainak számozása vasárnap kezdődik a `0` értékkel, de a hónapok és a napok sorszámát is konstansok tárolják (lásd alább). A konstansok sorszámozásától függetlenül a naptárrendszerben a hét számítása vasárnaptól különböző nappal is kezdődhet.

`AM_PM, HOUR, HOUR_OF_DAY, MINUTE, SECOND, MILLISECOND`

: Az idő lekérdezésénél vagy beállításánál határozzák meg rendre az idő következő komponenseit: napszak (délelőtt vagy délután), 12 órás óra, 24 órás óra, perc, másodperc, ezredmásodperc.

`SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY`
A hét napjait reprezentáló konstansok. A számozás nullától kezdődik, ez a `SUNDAY` konstansnak felel meg.

`JANUARY, FEBRUARY, MARCH, APRIL, MAY, JUNE, JULY, AUGUST, SEPTEMBER, OCTOBER, NOVEMBER, DECEMBER, UNDECIMBER`
A hónap napjait reprezentáló konstansok. A számozás a `JANUARY` konstanssal, nullától kezdődik. Az `UNDECIMBER` konstans az egyes naptárrendszerekben előforduló tizenharmadik hónapot jelöli. Ez a Gergely-naptártól eltérő naptárrendszerek megvalósításánál használható.

`AM, PM`
Délelőttöt és délutánt reprezentáló konstansok.

`SHORT, MEDIUM, LONG`
A dátum és az idő részeinek szöveges kiíratásakor használható stílusok.

Az első csoport metódusai az aktuálisan használt naptárrendszerről adnak általános információt, azt mondják például meg, hogy melyik a hét első napja. A metódusokat az alábbi lista ismerteti:

```
int getActualMaximum(int field)
int getActualMinimum(int field)
```
Az adott komponens aktuális idő szerinti maximumát és minimumát adják vissza. Például a zsidó naptárban az évtől függően tizenkét vagy tizenhárom hónap van.

```
String getDisplayName(int field, int style, Locale locale)
```
A tárolt dátum adott komponensének megjeleníthető nevét adja vissza a megadott stílus és lokalizáció szerint.

```
Map<String, Integer> getDisplayNames(int field, int style,
Locale locale)
```
Az adott komponens lehetséges értékeinek megjeleníthető neveit adja vissza a megadott stílus és lokalizáció szerint.

```
int getFirstDayOfWeek()
```
A hét első napjához tartozó konstansot adja vissza.

```
int getGreatestMinimum(int field)
int getLeastMaximum(int field)
```
Az adott komponens legnagyobb minimumát, illetve legkisebb maximumát adja vissza, azaz a `getActualMinimum()` által visszaadható legnagyobb értékét, illetve a `getActualMaximum` által visszaadható legkisebbet.

```
int getMaximum(int field)
int getMinimum(int field)
```
Visszaadja a naptárrendszerben az adott komponens által felvehető legnagyobb és legkisebb értékeket.

A metódusok második csoportja a jelenlegi dátummal és idővel kapcsolatos információ lekérdezésére szolgál pl. hányadika van? egy adott dátum a hét mely napjára esik? Ezeket a metódusokat az alábbi lista foglalja össze:

`int get(int field)`
A paraméterben megadott dátum- vagy időkomponens értékét adja vissza.

`Date getTime()`
A tárolt dátumot és időt `Date` objektumban adja vissza. Ez például akkor alkalmazandó, ha a dátumot a `DateFormat` osztállyal formázni kell (lásd 14.1.2. alfejezet).

`long getTimeInMillis()`
A referenciaidő óta eltelt milliszekundumok számával adja meg a reprezentált időt.

`TimeZone getTimeZone()`
A reprezentált idő időzónáját adja meg `TimeZone` objektumként.

A harmadik csoportba azok a metódusok tartoznak, amelyek dátumokkal és idővel kapcsolatos számítások végézésre szolgálnak. Ezeket alább tekintjük át.

`void add(int field, int amount)`
Az adott komponenst növeli a megadott értékkel. Negatív érték esetén ez csökkentést jelent. A változtatás során frissülnek a magasabb időegységek is, például ha a dátum öt nappal történő növelése során megváltozik a hónap, akkor annak értéke is frissül.

`boolean after(Object when)`
Akkor ad vissza igaz értéket, ha ez a példány a paraméterben megadott `Calendar`-példánynál későbbi időpontot reprezentál. A paraméter típusa `Object`, ezért más típusú objektumok is megadhatók, de ebben az esetben a visszatérési érték mindig hamis lesz.

`boolean before(Object when)`
Mint az előző metódus, de akkor ad vissza igazat, ha ez az objektum korábbi időpontot reprezentál.

`Object clone()`
Másolatot készít az objektumról.

`int compareTo(Calendar anotherCalendar)`
A megadott másik `Calendar`-példányhoz hasonlítja ezt a példányt, és pozitív értéket ad vissza, ha ez a példány reprezentálja a későbbi időt, nullát, ha a reprezentált idő egyezik, és negatív értéket, ha a paraméterben kapott példány a későbbi.

`void roll(int field, int amount)`
Az adott komponenst növeli a megadott értékkel. Negatív érték esetén ez csökkentést jelent. A változtatás során *nem* frissülnek a magasabb időegységek, de megváltozhatnak más komponensek értékei, ha egyébként érvénytelen dátum jönne létre. Például ha március 31-i dátumot növelünk egy hónappal, akkor április 30-at kapunk, mert április csak 30 napos.

```
void set(int field, int value)
void set(int year, int month, int date)
void set(int year, int month, int date, int hourOfDay, int minute)
void set(int year, int month, int date, int hourOfDay, int minute,
int second)
```

A metódus első változata az adott komponenst állítja be a megadott értékre. A többi változat több időkomponens együttes beállítását teszi lehetővé.

`void setFirstDayOfWeek(int value)`

Beállítja, melyik napot tekintsük a hét első napjának.

`void setTime(Date date)`

A megadott `Date`-példányban tárolt időt állítja be.

`void setTimeInMillis(long millis)`

A referenciaidő óta eltelt milliszekundumok megadásával teszi lehetővé az idő beállítását.

`void setTimeZone(TimeZone value)`

Az időzónát állítja be.

Az alábbi rövid programrészlet példa a fenti ismertetett metódusok használatára.

```
// Gyors módszer a dátum kiírására
System.out.println("Dátum kiírása a Date osztállyal: " + new Date());

// Elérhető lokalizációk listázása
Locale[] locales = Calendar.getAvailableLocales();
System.out.println("Elérhető lokalizációk a Calendar osztályhoz: "
    + Arrays.toString(locales));

// Spanyol naptár szerint írunk ki pár adatot
Locale locale = new Locale("es");
Calendar calendar = Calendar.getInstance(locale);
int firstDay = calendar.getFirstDayOfWeek();
calendar.set(Calendar.DAY_OF_WEEK, firstDay);
System.out.println("A hét első napja: "
    + calendar.getDisplayName(Calendar.DAY_OF_WEEK, Calendar.LONG,
    locale));
System.out.println("A legrövidebb hónap napjainak száma: "
    + calendar.getLeastMaximum(Calendar.DAY_OF_MONTH));
System.out.println("A leghosszabb hónap napjainak száma: "
    + calendar.getMaximum(Calendar.DAY_OF_MONTH));

// Idő kiírása magyar lokalizáció szerint
locale = new Locale("hu");
calendar = Calendar.getInstance(locale);
System.out.print(calendar.get(Calendar.YEAR) + " ");
System.out.print(calendar.getDisplayName(Calendar.MONTH,
```

```
    Calendar.SHORT, locale) + " ");
System.out.print(calendar.get(Calendar.DAY_OF_MONTH) + ", ");
System.out.println(calendar.getDisplayName(Calendar.DAY_OF_WEEK,
    Calendar.SHORT, locale));

// Egyszerű számítás
Calendar calendar2 = Calendar.getInstance(locale);
calendar2.set(2018, Calendar.APRIL, 22);
long diff = calendar2.getTimeInMillis() - calendar.getTimeInMillis();
long napok = diff / 1000 / 60 / 60 / 24;
System.out.println("2018. március 22-ig hátralévő napok: " + napok);
// 2018. március 22-t kell kapni, ha ezt hozzáadjuk
calendar.add(Calendar.DATE, (int) napok);
System.out.println(calendar.getTime());
```

4.5. A java.io API

A Java nyelv osztálykönyvtára két API-t is nyújt adatok olvasására és írására. Az első a `java.io`, a második a `java.nio` csomag része. A két csomag osztályaira röviden az *IO API* és *NIO API* kifejezésekkel is hivatkozunk. A NIO API későbbi fejlesztés, a *new IO (új be- és kimenet)* rövidítése. Az új API széleskörűbb funkcionalitást nyújt, legfontosabb újítása az aszinkron olvasás. Ennek használata elősegíti a hatékony és skálázható többszálú programok kifejlesztését. Ilyen szempontból a NIO API fejlettebb, ugyanakkor a programokban a be- és kimenetet pufferekkel kezeljük, és ez megnehezíti a programozást. Ráadásul az aszinkron be- és kimenetre többszálú programok esetén sincs mindig szükség. Az IO API ezért a legtöbbször még mindig megállja a helyét, a programozása ráadásul könnyebb, sok meglévő keretrendszer használja, illetve a NIO API előnyei csak összetett IO-kezelés esetén használhatók ki. A NIO API ugyanakkor könnyebbé teszi a fájl- és könyvtárműveletek kezelését, valamint a karakterkészletek közti konverzióra is lehetőséget ad. Ezek az IO API-val együtt is jól használhatók. Ezeket a tényezőket szem előtt tartva, a könyv mindkét API-t ismerteti, de a be- és kimenet tekintetében az IO API, a fájlkezelés és a karakterkészletek témákban pedig a NIO API kap nagyobb hangsúlyt. Az API-k tárgyalását az IO API-val kezdjük.

4.5.1. A be- és a kimenet kezelése

Az IO API négy alapvető osztályt nyújt a be- és a kimenet kezeléséhez. Választhatunk bájtokkal vagy karakterekkel dolgozó osztályok között. Másképpen fogalmazva ez azt jelenti, hogy „nyers adattal" vagy szöveggel dolgozunk-e. Ezeket az adatáramlás iránya szerint kétfelé oszthatjuk: amelyek bemenetet olvasnak, és amelyek kimenetet írnak. Az osztályok nem támogatják egyszerre az írást és az olvasást, valamint az adatok forrását adatfolyamként kezelik, ezért a pozíció visszaállítása csak korlátozottan van támogatva. A 4.4. táblázat összefoglalja a négy alapvető IO-osztályt.

4.4. táblázat: Az IO API osztályai

	Bájtalapú	Karakteralapú
Olvasás	InputStream	Reader
Írás	OutputStream	Writer

Az `InputStream` tehát bájtokból álló adatfolyam olvasására szolgál. Az osztálytól lekérdezhetjük, hány bájt áll rendelkezésre, illetve olvashatunk bájtonként vagy bájttömbbe. Lehetséges adott számú bájt átugrása is, illetve bizonyos leszármazott osztályok támogathatják az aktuális pozíció megjelölését, így ide olvasás után visszaugorhatunk. Végül a használat után az adatfolyamot le kell zárni. A következő lista összefoglalja az osztály metódusait:

`int available()`
Visszaadja a kiolvasható bájtok becsült számát.

`void close()`
Lezárja az adatfolyamot.

`void mark(int readlimit)`
Megjelöli az adott pozíciót, hogy később vissza lehessen oda ugorni. A paraméter adja meg, hány bájt kiolvasásáig marad érvényes a megjelölt pozíció.

`boolean markSupported()`
Visszaadja, hogy a folyam támogatja-e pozíció megjelölését a `mark()` metódussal.

`int read()`
Egyetlen bájtot olvas a folyamból, majd visszaadja azt. Ha elérte a folyam végét, akkor `-1` értéket ad vissza.

`int read(byte[] b)`
Feltölti a tömböt a folyamból olvasott adatokkal, majd visszaadja a beolvasott bájtok számát, illetve `-1`-et, ha nem tudott adatot beolvasni.

`int read(byte[] b, int off, int len)`
Mint a fenti metódusok, de a második paraméterben megadott eltolástól tölti fel a tömböt, és legfeljebb a harmadik paraméterben megadott számú bájtot olvas.

`void reset()`
Visszaállítja a pozíciót a `mark()` metódussal megjelölt helyre. Ha a `mark()` metódus nem lett hívva, vagy nincs támogatva, akkor `IOException` kivétel váltódhat ki.

`long skip(long n)`
Megkísérel kihagyni megadott számú bájtot a folyamban. Visszaadja a ténylegesen átugrott bájtok számát. Ha negatív számot ad vissza, akkor nem ugrott át egy bájtot sem.

Az `OutputStream` bájtokból álló adatfolyam írását teszi lehetővé. Funkcionalitása bájt vagy bájttömb írására, pufferelt adatok azonnali lemezre írására, valamint az adatfolyam lezárására korlátozódik. A metódusokat az alábbi lista foglalja össze:

`void close()`
Lezárja az adatfolyamot.

`void flush()`
Kiírja az összes pufferelt adatot.

`void write(int b)`
A folyamba ír egy bájtot.

`void write(byte[] b)`
Kiírja a megadott tömbben tárolt bájtokat a folyamba.

`void write(byte[] b, int off, int len)`
A tömbből a megadott eltolástól ír ki megadott számú bájtot.

A `Reader` osztály karakteres adatok olvasását támogatja. Funkcionalitása igen hasonlít az `InputStream` osztályéhoz, de ez az osztály karakterekkel és karaktertömbökkel dolgozik. Metódusait az alábbi lista foglalja össze:

`void close()`
Lezárja az adatfolyamot.

`void mark(int readlimit)`
Megjelöli az adott pozíciót, hogy később vissza lehessen oda ugorni. A paraméter adja meg, hány karakter kiolvasásáig marad érvényes a megjelölt pozíció.

`boolean markSupported()`
Visszaadja, hogy a folyam támogatja-e pozíció megjelölését a `mark()` metódussal.

`int read()`
Egyetlen karaktert olvas a folyamból, majd visszaadja azt. Ha elérte a folyam végét, akkor `-1` értéket ad vissza.

`int read(char[] b)`
Feltölti a tömböt a folyamból olvasott adatokkal, majd visszaadja a beolvasott karakterek számát, illetve `-1`-et, ha nem tudott adatot beolvasni.

`int read(char[] b, int off, int len)`
Mint a fenti metódusok, de a második paraméterben megadott eltolástól tölti fel a tömböt, és legfeljebb a harmadik paraméterben megadott számú karaktert olvas.

`boolean ready()`
Visszaadja, hogy a folyam készen áll-e olvasásra.

`void reset()`
Visszaállítja a pozíciót a `mark()` metódussal megjelölt helyre. Ha a `mark()` metódus nem lett hívva, vagy nem támogatott, akkor `IOException` kivétel váltódhat ki.

`long skip(long n)`
Megkísérel kihagyni megadott számú karaktert a folyamban. Visszaadja a ténylegesen átugrott kaarakterek számát. Ha negatív számot ad vissza, akkor nem ugrott át egy karaktert sem.

A `Writer` osztály az `OutputStream`hez hasonló, de karakterekkel és karaktertömbökkel dolgozik, akárcsak a `Reader`. Metódusait az alábbi listában tekinthetjük át:

`void close()`
Lezárja az adatfolyamot.

`void flush()`
Kiírja az összes pufferelt adatot.

`void write(int c)`
A folyamba ír egy karaktert.

`void write(char[] cbuf)`
Kiírja a megadott tömbben tárolt karaktereket a folyamba.

`void write(char[] cbuf, int off, int len)`
A tömbből a megadott eltolástól ír ki megadott számú karaktert.

`void write(String str)`
Kiírja a megadott karakterláncot a folyamba.

`void write(String str, int off, int len)`
A karakterlánc eltolással és hosszal meghatározott részét írja ki.

A fenti négy osztály mind absztrakt, mindig azok specifikusabb leszármazott osztályait példányosítjuk. Például fájlból történő olvasáshoz a `FileInputStream` vagy `FileReader` osztályok szolgálnak, attól függően, hogy bájtokkal vagy karakterekkel kívánunk dolgozni. A négy alaposztály leszármazottjai támogatják fájlok, bájttömbök, Java-primitívek, illetve karakterláncok olvasását és írását. Ezeket a 4.5. táblázat foglalja össze.

A listából kiemelendők a `PrintStream` és `PrintWriter` osztályok. Ezek a `print()` és a `println()` metódusokkal bármilyen típusú adatot ki tudnak írni, ezért jól használhatók az adatok kiírásának magas szintű kezelésére. Az utóbbi sortörést is kiír az adat után. Objektumok esetén a metódusok a `toString()` metódus által visszaadott karakterláncot írják ki. A `format()` és a `printf()` metódus a C-ből ismert `printf()` függvénynél megszokott módon formázott kiírást is el tud végezni. Java nyelven a karakterláncok könnyű összefűzhetősége miatt azonban erre ritkán van szükség, ezért a formázott kiírást nem tárgyaljuk.

4.5. táblázat: A java.io csomag főbb osztályai

	Bájtalapú		Karakteralapú	
	Olvasás	**Írás**	**Olvasás**	**Írás**
Alaposztály	InputStream	OutputStream	Reader	Writer
Tömbök	ByteArrayInputStream	ByteArrayOutputStream	CharArrayReader	CharArrayWriter
Fájlok	FileInputStream	FileOutputStream	FileReader	FileWriter
Csővezeték	PipedInputStream	PipedOutputStream	PipedReader	PipedWriter
Karakterláncok			StringReader	StringWriter
Adat	DataInputStream	DataOutputStream		
Objektum	ObjectInputStream	ObjectOutputStream		
Formázott adat		PrintStream		PrintWriter

4.6. táblázat: A java.io csomag csomagolóosztályai

	Bájtalapú		Karakteralapú	
	Olvasás	**Írás**	**Olvasás**	**Írás**
Pufferelés	BufferedInputStream	BufferedOutputStream	BufferedReader	BufferedWriter
Bájtból karakter			InputStreamReader	OutputStreamWriter

Az API használata során nagy szerepet kap a csomagolóobjektumok (Decorator minta [4]) használata is. Ez azt jelenti, hogy egy objektum funkcionalitását úgy bővítjük ki, hogy másik, bővebb funkcionalitást nyújtó objektumba csomagoljuk. A csomagolóobjektum példányosításkor megkapja az eredeti objektum referenciáját, és az eredetinél bővebb funkcionalitást valósít meg, de az alapvető műveleteket a csomagolt objektumnak delegálja. Például a `BufferedInputStream` és `BufferedOutputStream` objektumok pufferelt olvasást és írást valósítanak meg `InputStream` és `OutputStream`-példányok fölött. Pufferelő csomagolóosztály a `Reader` és `Writer` osztályokhoz is létezik. A csomagolóosztályok arra is lehetőséget adnak, hogy bájtalapú adatfolyamokat karakteresen érjünk el. A csomagolóosztályokat 4.6. táblázat foglalja össze.

Az alábbi példaprogram szemlélteti a be- és a kimenet kezelését. A program egy reguláris kifejezést, egy karakterláncot és egy fájlnevet vár, majd a fájlban a reguláris kifejezés illeszkedéseit soronként kicseréli a karakterláncra. Ha fájlnak a - jelet adjuk meg, akkor a szabványos bemenetről olvas. A soronkénti olvasáshoz a `BufferedReader` osztályt használja. Az eredményt először a `StringWriter` osztály segítségével karakterláncba írja, majd a szabványos kimeneten is megjeleníti.

```
public static void main(String[] args) {
    if (args.length != 3) {
        System.out.println("Használat: SearchAndReplace regex ↺
új_szöveg fájl");
        System.exit(-1);
    }

    BufferedReader br = null;
    try {
        br = (args[2].equals("-")) ? new BufferedReader↺
(new InputStreamReader(System.in)) : new BufferedReader(new ↺
InputStreamReader(new FileInputStream(args[2])));
    } catch (FileNotFoundException e) {
        e.printStackTrace();
    }

    String s;
    try (StringWriter writer = new StringWriter();) {
        while ((s = br.readLine()) != null) {
            writer.append(s.replaceAll(args[0], args[1]));
            writer.append('\n');
        }
        br.close();
        System.out.println(writer.toString());
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

Létezik még egy fontos, de a fenti kategóriákba nem illő osztály is. Ez a `RandomAccessFile`. Az osztály segítségével fájlokat olvashatunk és írhatunk véletlen hozzáféréssel, azaz tetszőleges pozíciótól. Az osztály `readXxx()` és `writeXxx()` metódusaival, ahol *Xxx* a típus neve, számos formában képes adatot olvasni és írni.

4.5.2. A fájlműveletek

Fájlműveletek a `File` osztály segítségével végezhetők. Valójában az osztály neve nem tükrözi pontosan a rendeltetését, az osztály ugyanis ténylegesen elérési utat reprezentál. A reprezentált elérési út ugyanis könyvtárra is hivatkozhat, de nem is szükséges léteznie. `File` objektumot legegyszerűbben az egyparaméteres konstruktorral hozhatunk létre, ennek az elérési utat kell megadni karakterláncként vagy `URI` objektumként. A megadott elérési út relatív is lehet, ekkor a rendszer az aktuális munkakönyvtárhoz képest értelmezi. Léteznek kétparaméteres konstruktorok is. Ezek egy elérési útból és egy ahhoz képest értelmezett relatív elérési útból hoznak létre új elérési utat. Az első paraméter `File` objektummal vagy karakterlánccal adja meg, hogy honnan értelmezzük a relatív elérési utat, a második paraméter pedig maga a relatív elérési út karakterláncként megadva.

Ha létrehoztunk egy `File` objektumot, akkor az `exists()` metódussal tudjuk ellenőrizni, hogy az elérési úton ténylegesen létezik-e könyvtár vagy fájl. Ha nem létezik, akkor az elérési úton fájlt vagy könyvtárat kell létrehoznunk, hogy használni tudjuk. Fájl létrehozására a `createNewFile()` szolgál, ez üres fájlt hoz létre. A metódus akkor ad vissza `true` értéket, ha a fájl nem létezett, de sikerült létrehozni. Könyvtár létrehozására az `mkdir()` metódus használható. Ez akkor működik, ha az elérési útnak az utolsót kivéve az összes komponense létezik. Ha közbülső könyvtárakat is létre kell hozni, akkor az `mkdirs()` metódus alkalmazandó. Mindkét metódus akkor ad vissza `true` értéket, ha a könyvtárat vagy könyvtárakat létrehozta.

Ha az elérési út létezik, akkor az `isFile()` és az `isDirectory()` metódusokkal állapíthatjuk meg, hogy fájlra vagy könyvtárra hivatkozik-e. A `File` osztály számos metódussal rendelkezik, némelyikük csak fájlon, mások csak könyvtáron használhatók. Vannak olyan metódusok is, amelyek fájlt vagy könyvtárt reprezentáló elérési úton egyaránt működnek, valamint az osztálynak van néhány statikus metódusa is, amely az aktuálisan használt operációs rendszerrel kapcsolatos adatok lekérdezésére szolgál, mint például az elérési utakban használt elválasztókarakterek. A NIO API `Path` osztálya kiküszöböli a `File` osztály sok korlátozását, ezért új programokban javasolt annak a használata. A `File` osztály alapvető ismerete mégis fontos a régebbi programok és keretrendszerek használata miatt. Az osztály `toPath()` metódusa, valamint a `Path` osztály `toFile()` metódusa valósítja meg a két osztály közötti átjárást.

4.6. A java.nio API

Ebben az alfejezetben röviden áttekintjük a NIO API lehetőségeit.

4.6.1. A be- és a kimenet kezelése

A Java NIO API alapja a `Buffer` osztály és annak leszármazottjai. Ezek az osztályok a `java.nio` csomag részei, és adatpuffereket reprezentálnak, amelyekbe a beolvasás, illetve amelyekből a kiírás történik. A pufferek tulajdonképpen adott típusú adatok véges sorozatát foglalják magukban, és ennek a sorozatnak a manipulálásához nyújtanak gazdag funkcionalitást. A pufferek tehát kibővített funkcionalitású tömbökhöz hasonlíthatók. Minden egész- és lebegőpontos típushoz létezik leszármazott osztály, amelynek neve a típus nevéből és a `Buffer` szóból tevődik össze, például `ByteBuffer` vagy `IntBuffer`. Mivel a fájlokat leggyakrabban bájt- vagy karaktersorozatként kezeljük, ezért a `ByteBuffer` és `CharBuffer` osztályokat használjuk a leggyakrabban. Létezik még egy speciális puffertípus, a `MappedByteBuffer`. Ennek segítségével fájlt érhetünk el közvetlenül bájttömbként, és a tömbön végzett változások visszaíródnak a fájlba. Ez a mechanizmus a POSIX operációs rendszerek `mmap()` függvényével egyezik meg, és sokkal hatékonyabb fájlműveleteket tesz lehetővé, mint a hagyományos írás és olvasás. Ráadásul a tömbre leképezett fájl módosítását az operációs rendszer végzi el, így a Java-program esetleges összeomlásakor is végbemegy. A puffereket nem lehet közvetlenül példányosítani, de minden konkrét osztály rendelkezik statikus `allocate()` factorymetódussal, amelynek a létrehozandó puffer kapacitását kell megadni. A pufferekнek három fontos jellemzőjük van:

- a kapacitás: a puffer mérete, azaz a benne tárolható adat mennyisége;
- a pozíció: az aktuális pozíció, amelytől kezdve a következő írási vagy olvasási művelet végbemegy;
- a limit: a jelenleg kiírható vagy olvasható adat mennyisége.

A pufferekbe olvasás, illetve azok kiírása csatornákon keresztül történik, ezeket a `Channel` interfészt megvalósító, `java.nio.channels` csomagban található osztályok reprezentálják. A `FileChannel` osztály használható fájlok kezelésére. Más csatornatípusokat hálózati kommunikációhoz (lásd 9.1. alfejezet) és egyéb speciális célokra használunk. A csatorna metódusait használjuk a puffer kiírására vagy adattal történő feltöltésére. Miután a pufferbe adatokat olvastunk, a puffer `flip()` metódusát használhatjuk arra, hogy a beolvasott adat máshova kiírható legyen. A metódus ugyanis visszaállítja a pozíciót a pufffer elejére, és beállítja a limitet a jelenleg tartalmazott adat szerint, tehát pontosan a pufferben levő adat fog kiíródni. Újbóli feltöltés előtt a `rewind()` metódus hívandó, ez visszaállítja a pozíciót a puffer elejére.

4.6.2. A karakterkódolások

A Java nyelv a fájlok feldolgozásánál használt alapértelmezett karakterkódolást a `file.encoding` rendszerbeállításban tárolja, ha pedig ez nincs megadva, akkor *UTF-8* karakterkódolást használ. A `char` típus karakterei és a karakterláncok is *UTF-16* kódolással vannak tárolva. Mindkét karakterkódolás a Unicode szabvány része, ezért

egymásba könnyen konvertálhatók. Manapság jó választás az UTF-8 használata, de megeshet, hogy régebbi rendszerekkel való együttműködés miatt más karakterkódolással kell adatokat olvasnunk és írnunk. A `java.nio.charset` csomag a karakterkódolások közötti konverzióhoz nyújt segítséget. Mivel a csomag a NIO API része, szintén a `Buffer` osztályokra épül.

A csomag legfőbb osztálya a `Charset`, ennek példányai konkrét karakterkódolásokat reprezentálnak. Az osztálynak nincs publikus konstruktora, példányokat háromféleképpen érhetünk el. Az egyik módszer a `forName()` statikus metódus használata, ennek karakterláncként adjuk meg a használandó karakterkódolás nevét. A Java szabvány megköti, hogy a következő karakterkódolásokat minden implementáció támogassa: `US-ASCII`, `ISO-8859-1`, `UTF-8`, `UTF-16BE`, `UTF-16LE` és `UTF-16`. A támogatott karakterkódolásokat a statikus `availableCharset()` metódussal kérdezhetjük le. Ez `SortedMap<String, Charset>` típusú visszatérési értékkel rendelkezik, tehát a visszaadott szótárból a megfelelő példányt is kinyerhetjük. A harmadik módszer a szintén statikus `defaultCharset()` használata, ez az alapértelmezett karakterkódoláshoz tartozó példányt adja meg.

Miután megszereztünk egy példányt, lekérhetünk róla pár alapvető információt a következő metódusok segítségével:

`Set<String> aliases()`
: A karakterkódolás alternatív neveit kérdezi le.

`String displayName()`
`String displayName(Locale locale)`
: Visszaadja a karakterkódolás nevét a megadott vagy az alapértelmezett lokalizáció szerint.

`boolean isRegistered()`
: Visszaadja, hogy a karakterkódolás regisztrálva van-e az IANA (Internet Assigned Numbers Authority) listájában.

`String name()`
: A karakterkódolás kanonikus nevét adja meg.

A `Charset` osztállyal konverziót is végezhetünk a Java belső reprezentációja (UTF-16) és az aktuális karakterkódolás között. Ha a Java reprezentációjára konvertálunk az idegen kódolásból, akkor dekódolásról, fordított esetben kódolásról beszélünk. Mivel a dekódolás eredménye Java-karakterek sorozata, ezért az eredmény `CharBuffer` objektumban tárolható. Az idegen karakterkódolásban reprezentált szöveg a Java nyelv számára nem rendelkezik jelentéssel, ezért az általánosabb `ByteBuffer` osztállyal tároljuk. Az osztály a következő metódusokat biztosítja a kódolás és a dekódolás elvégzéséhez:

`boolean canEncode()`
: Megállapítja, hogy használhatjuk-e a karakterkészletet kódoláshoz.

`CharBuffer decode(ByteBuffer bb)`
: Dekódolja a bájtpufferben megadott szöveget.

`ByteBuffer encode(CharBuffer cb)`
`ByteBuffer encode(String str)`
Kódolja a karakterpufferben vagy karakterláncban megadott szöveget.

A `Charset` osztály `newEncoder()` és `newDecoder()` metódusaival elérhetjük a `CharsetEncoder` és `CharsetDecoder` osztályok megfelelő példányát, ezek bővebb funkcionalitást nyújtanak a kódoláshoz és dekódoláshoz. Például kódolhatjuk vagy dekódolhatjuk a bemeneti puffernek csak egy részét, lekérdezhetjük, hogy egy bemeneti vagy kimeneti bájt átlagosan hány karaktert jelképez, vagy kezelhetjük az érvénytelen és a nem leképezhető bemeneteket. Ezeket az osztályokat a könyv nem tárgyalja.

Az alábbi példa olyan egyszerű programot valósít meg, amellyel fájlt konvertálhatunk át adott karakterkódolásból új karakterkódolásba, és a végeredményt új fájlba írjuk. A program szintén szemlélteti a NIO API-val történő olvasást és írást a `FileChannel` osztály segítségével, ugyanis a `Charset` API pufferekkel dolgozik, amelyeket a `FileChannel` osztály segítségével lehet kényelmesen feltölteni és kiírni.

```
public static void main(String args[]) {
    if (args.length < 4) {
        System.out.println("Használat: java Iconv bemeneti_fájl "
                + "kimeneti_fájl bemeneti_kódolás kimeneti_kódolás");
        System.exit(-1);
    }
    Charset charsetIn = null;
    Charset charsetOut = null;
    try {
        charsetIn = Charset.forName(args[2]);
        charsetOut = Charset.forName(args[3]);
    } catch (UnsupportedCharsetException e1) {
        System.out.println("Érvénytelen kódolás: "
            + e1.getMessage());
        System.exit(-1);
    }

    ByteBuffer bufIn = ByteBuffer.allocate(5);
    try (FileChannel channelIn = FileChannel.open(Paths.get↵
(args[0]), StandardOpenOption.READ);
        FileChannel channelOut = FileChannel.open(Paths.get↵
(args[1]), StandardOpenOption.WRITE, StandardOpenOption.CREATE);) {
        while (channelIn.read(bufIn) > 0) {
            bufIn.flip();
            CharBuffer tmpBuf = charsetIn.decode(bufIn);
            ByteBuffer bufOut = charsetOut.encode(tmpBuf);
            channelOut.write(bufOut);
            bufIn.rewind();
        }
```

```
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

4.6.3. A fájlműveletek

A NIO API fájlokkal és könyvtárakkal kapcsolatos funkcionalitása a Java 7-es verziójában került csak be az osztálykönyvtárba, ezért NIO 2 API-ként is szokás hivatkozni rá. Ezt a funkcionalitást a `java.nio.file` és a `java.nio.file.attribute` csomagok tartalmazzák. Az első csomag legfontosabb eleme a `Path` interfész, amely elérési utat reprezentál, akárcsak a `File` osztály. A csomagok nem tartalmaznak a `Path` interfészt megvalósító publikus osztályt, így a `Paths` osztály statikus `get()` metódusát kell használnunk ahhoz, hogy `Path`-példányt kapjunk. A metódusnak két változata van, egyik `URI`-t vár, a másik tetszőlegesen sok karakterláncot, amelyekből összeállítja az elérési utat.

A `File` osztállyal szemben a `Path` interfész metódusai csak az elérési utak kezelésével kapcsolatos műveletekre alkalmasak, mint például a relatív elérési utak feloldására, a gyökérkönyvtár lekérésére, az utolsó komponens megállapítására stb. Ezeket az alábbi lista foglalja össze:

`boolean endsWith(Path other)`
`boolean endsWith(String other)`
Megvizsgálja, hogy az elérési út a megadott másik elérési úttal végződik-e.

`Path getFileName()`
Visszaadja az elérési út utolsó komponensét, ez fájl vagy könyvtár lehet.

`FileSystem getFileSystem()`
Visszaadja az elérési úthoz tartozó fájlrendszert.

`Path getName(int index)`
Visszaadja az elérési út adott sorszámú komponensét. A számozás a gyökértől kezdődik, a legelső komponens indexe `0`, az utolsóé a `getNameCount()` metódus által visszaadott értéknél eggyel kisebb.

`int getNameCount()`
Megadja az elérési út komponenseinek a számát.

`Path getParent()`
Visszaadja a szülőt, illetve `null`t, ha az elérési útnak nincs szülője.

`Path getRoot()`
Visszaadja a gyökeret, illetve `null`t, ha az elérési útnak nincs gyökere.

`boolean isAbsolute()`
Megvizsgálja, hogy az elérési út abszolút elérési út-e.

`Iterator<Path> iterator()`
Iterátort ad vissza, amellyel az elérési út komponensei járhatók be.

`Path normalize()`
Normalizált elérési utat ad vissza, azaz megszünteti a felesleges elemeket.

`Path relativize(Path other)`
A megadott elérési útnak az ehhez az elérési úthoz képest vett relatív formáját adja vissza.

`Path resolve(Path other)`
`Path resolve(String other)`
Ha a megadott elérési út relatív, akkor ehhez az elérési úthoz viszonyítva abszolút elérési úttá oldja fel.

`Path resolveSibling(Path other)`
`Path resolveSibling(String other)`
Ha a megadott elérési út relatív, akkor ennek az elérési útnak a szülőjéhez viszonyítva abszolút elérési úttá oldja fel.

`boolean startsWith(Path other)`
`boolean startsWith(String other)`
Megvizsgálja, hogy az elérési út a megadott másik elérési úttal kezdődik-e.

`Path toAbsolutePath()`
Az elérési utat abszolút formában adja vissza.

`File toFile()`
Az elérési út `File` objektummal történő reprezentációját adja vissza.

`URI toUri()`
Az elérési út URI-reprezentációját adja vissza.

A tényleges fájl- és könyvtárműveleteket a `Files` osztály statikus metódusain keresztül érhetjük el. Az osztálynak igen sok metódusa van. Ezek között számos olyan is akad, amely a műveletet meghatározó opciókat enumerációk változó hosszúságú paraméterlistájában veszi át. Ezért az API ismertetését itt mellőzzük, az a Javadoc-referenciában megtekinthető. Az alábbi példaprogram primitív parancssoros shellt valósít meg, amely csak másolni, áthelyezni, törölni tud, illetve képes megjeleníteni az aktuális könyvtárat, listázni annak tartalmát és más könyvtárra váltani. A hibásan kiadott parancsok kezelését most az egyszerűség kedvéért mellőzzük.

```
public static void main(String[] args) {
    BufferedReader br = new BufferedReader(new InputStreamReader
(System.in));
    String currentDir = System.getProperty("user.dir");
    Path cwd = Paths.get(currentDir);

    while (true) {
        String line = "";
        try {
            line = br.readLine();
        } catch (IOException e) {
```

```
            e.printStackTrace();
        }

        String[] params = line.split(" ");
        switch (params[0]) {
        case "cp":
            try {
                Files.copy(Paths.get(params[1]),
                    Paths.get(params[2]));
            } catch (IOException e) {
                e.printStackTrace();
            }
            break;
        case "mv":
            try {
                Files.move(Paths.get(params[1]),
                    Paths.get(params[2]));
            } catch (IOException e) {
                e.printStackTrace();
            }
            break;
        case "rm":
            try {
                Files.delete(Paths.get(params[1]));
            } catch (IOException e) {
                e.printStackTrace();
            }
            break;
        case "cd":
            Path newDir = cwd.resolve(params[1]);
            if (newDir != null)
                cwd = newDir.normalize();
            else
                System.out.println("Nem létező könyvtár.");
            break;
        case "ls":
            try (DirectoryStream<Path> directoryStream = Files.↺
newDirectoryStream(cwd)) {
                for (Path path : directoryStream)
                    System.out.println(path);
            } catch (IOException ex) {
                ex.printStackTrace();
            }
            break;
        case "pwd":
            System.out.println(cwd.toAbsolutePath());
            break;
```

```
            case "exit":
                return;
            }
        }
}
```

A NIO2 API `FileSystem` osztálya az elérhető fájlrendszerekről képes néhány alapvető adatot lekérdezni. Az osztályt a `FileSystems` metódus statikus metódusaival lehet példányosítani. A fájlrendszerek kezelését a könyv nem tárgyalja.

ÖTÖDIK FEJEZET
A generikus programozás

Generikus programozáson olyan osztályok készítését értjük, amelyek bármilyen típusú objektumon képesek elvégezni általános feladatokat. Tipikusan idetartoznak a generikus kollekciók, mint például a halmazok és a listák, amelyek ezeket az adatstruktúrákat valósítják meg, és tetszőleges típusú objektummal használhatók. A fejezet bemutatja a generikus programozást, majd részletesen ismerteti a Java Collections keretrendszerét. Ez széles körű és hatékony implementációt nyújt generikus kollekciókhoz.

5.1. Az Object használata

Ha generikus megoldásokat kell kifejlesztenünk, amelyek bármilyen típusú objektumon működnek, akkor kézenfekvő megoldásnak tűnhet az `Object` típusú referenciákkal való munka, ez az osztály ugyanis az összes többi osztály őse. A Java 5-ös verziója előtt az osztálykönyvtár által megvalósított generikus kollekciók így működtek. A módszer hátránya, hogy nem kényszeríti ki a típusbiztosságot, mert nem korlátozhatjuk, hogy a struktúrában pontosan milyen típusú elemeket tárolunk. Ha explicit ellenőrzést építünk be az osztályba, például az `instanceof` operátorral, akkor a megoldás elveszti általánosan újrafelhasználható jellegét. Ezért a Java 5-ös verziója bevezette a típusparamétereket, ezek ezt a problémát hivatottak megoldani.

5.2. A típusparaméterek használata

A típusparaméterek segítségével a kezelt osztály típusa is paraméterezhetővé válik. Generikus programrészek írásakor a típusra paraméterváltozóval hivatkozunk, ezt egyetlen nagybetűvel szokás jelölni. A generikus komponens használatakor típusparaméterben megadjuk az alkalmazott típust is, ezért fordítási időben ellenőrizhető lesz a típusbiztosság. Kompatibilitási okok miatt futási időben nem tárolódik el a típusokra vonatkozó információ, a kód valójában `Object` típusú referenciákat használó osztályra fordul le. A típusparaméter kétféleképpen használható: teljes osztályokra adunk meg paramétert, vagy csak metódusokra alkalmazzuk őket. Az alábbiakban áttekintjük mindkét lehetőséget.

5.2.1. A típusparaméteres osztályok

Típusparaméteres osztályok esetén a típusparamétert az osztály neve után < ... > jelek között adjuk meg. Több paramétert is megadhatunk, vesszővel elválasztva. Ezután a típusparaméter úgy használható, mint ha valós típus lenne: használhatjuk metódusok visszatérési értékeként, paraméterek típusaként, vagy változót is deklarálhatunk vele. Konstruktort azonban nem hívhatunk. Ez a korlátozás logikus, a konstruktorok

ugyanis nem öröklődnek, ezért általános esetben nem tudjuk, hogy egy osztálynak milyen szignatúrájú konstruktorai vannak. Az alábbi példa generikus tárat valósít meg, ebben újrafelhasználható erőforrásokat tárolunk. Ha erőforrásra van szükségünk, akkor a tártól kaphatunk szabad példányt. A példány használata után a tárnak jelezzük, hogy az erőforrásra nincs már szükségünk. A tár tehát gondoskodik arról, hogy egy példányt egyszerre csak egy igénylőnek adjon ki, és a visszaadott példányok újra fel legyenek használva.

```
public class RentalStore<T> {
    private T[] store;
    // ez tárolja, melyik elem szabad
    private boolean[] free;

    // Meg kell adni egy tömböt a konstruktorban, mivel
    // generikus módon nem tudunk példányosítani
    public RentalStore(T[] store) {
        this.store = store;
        free = new boolean[store.length];
        Arrays.fill(free, true);
    }

    public T reserve() {
        for (int i = 0; i < free.length; i++)
            if (free[i]) {
                free[i] = false;
                return store[i];
            }
        return null;
    }

    public void release(T e) {
        for (int i = 0; i < store.length; i++)
            if (store[i].equals(e)) {
                free[i] = true;
                return;
            }
    }
}
```

Típusparaméteres osztály példányosításakor az osztály neve után meg kell adni a típusparamétert is a < ... > jelek között. Ha ezt nem tesszük meg, attól a program még lefordul, de a fordító nem végez típusellenőrzést, mint ha egyszerűen csak `Object` típusú referenciákat használtunk volna. Ezért a Java 5-ös verziója előtti programok is kompatibilisek maradtak az azóta generikussá tett osztálykönyvtárakkal. Az alábbi példaprogram szemlélteti a fenti generikus tár használatát.

```
public static void main(String args[]) {
    String[] strings = new String[] { "alma", "körte" };
    RentalStore<String> store = new RentalStore<>(strings);

    // ilyenkor nincs típusellenőrzés
    RentalStore store2 = store;

    // lefoglaljuk sorban a karakterláncokat,
    // a harmadik már null lesz
    String a = store.reserve();
    String b = store.reserve();
    String c = store.reserve();
    System.out.println(a + " " + b + " " + c);

    // elengedjük a másodikat, újbóli foglaláskor ezt kapjuk vissza
    store.release(b);
    System.out.println(store.reserve());
}
```

Típusparaméteres osztály specializálásakor a leszármazott osztály maga is deklarálhat típusparamétert, és örökölheti az ősosztály metódusait generikusan, vagy rögzítheti is a típusparaméterek értékét, ha az extends kulcsszó után a megfelelő helyettesítéssel adja meg az ősosztályt. Interfész is lehet generikus, erre természetesen ugyanez vonatkozik.

5.2.2. A típusparaméteres metódusok

Metódusokat külön is elláthatunk típusparaméterrel. Ebben az esetben is < ... > jelek között adjuk meg a típusparamétert, de most a metódus módosítói és a visszatérési érték között tesszük. A típusparaméter ezután ugyanúgy használható, mint az előző esetben. A következő generikus metódus visszaadja az objektumokból álló tömb legtöbbször előforduló elemét. Ha több olyan elem van, amelynek az előfordulási száma maximális, akkor a metódus ezek közül a legalacsonyabb indexszel rendelkezőt választja.

```
public static <T> T maxOccurrence(T[] arr) {
    Map<T, Integer> m = new HashMap<>();
    for (T t : arr) {
        Integer count = m.containsKey(t) ? m.get(t) : 0;
        m.put(t, ++count);
    }
    T max = null;
    int maxCount = 0;
    for (Entry<T, Integer> e : m.entrySet())
        if (e.getValue() > maxCount) {
```

```
            maxCount = e.getValue();
            max = e.getKey();
        }
    return max;
}
```

5.2.3. A típusparaméterek és a polimorfizmus

A polimorfizmus a típusparaméterek esetén másképp működik, mint az objektumreferenciák esetén. Azt várnánk, hogy a következő kód megfelelően működik:

```
class Animal {}

class Dog extends Animal {
    public void bar() {}
}

class GermanShepherd extends Dog {}

class Cat extends Animal {
    public void meow() {}
}

public class Main {

    public static void fillListWithDogs(List<Animal> list) {
        for (int i = 0; i < 5; i++)
            list.add(new Dog());
    }

    public static void main(String[] args) {
        List<Cat> catList = new ArrayList<>();
        fillListWithDogs(catList);
    }
}
```

Ez azonban nem fordul le. A metódus által várt lista típusa `List<Animal>`, a főprogramban létrehozott referencia típusa pedig `List<Cat>` . Ugyan a `Cat` egyben `Animal` is, ezért logikusan várjuk, hogy a kód működjön, a macskákat tartalmazó lista azonban mégsem leszármazott típusa az állatokat tartalmazó listának. A fenti példaprogramban az is megfigyelhető, miért van ez így. A hívott metódus ugyanis kutyákat tesz a listába. A `Dog` is az `Animal` osztályból származik, ezért az állatokat tartalmazó listába kutyákat is felvehetünk. Ha azonban a nyelv megengedné, hogy a metódusnak macskákat váró listát adjunk át az állatokat tartalmazó lista referenciáján keresztül, akkor a macskák listájába kutya is kerülhetne. Ez nyilván nem lenne szerencsés, mivel a típusok nem kompatibilisek, és a típusparaméterek csak fordítási időben nyújtanak védelmet, ezért futásidőben sem lenne ellenőrizhető a típusbiztosság. A típusparamétereknek tehát

pontosan meg kell egyezniük, nem adható meg sem leszármazott, sem szülő. Ez a működés azért zavaró, mert a generikus osztályok előtt már létező tömbök esetén lehetséges, hogy tömbreferenciának olyan tömbpéldányt adjunk át, amelynek típusa leszármazott osztályok tömbje. Tehát a fenti programrészlet tömböket használva lefordítható, annak ellenére is, hogy a macskák tömbjében kutyát próbálunk tárolni.

```
public static void fillArrayWithDogs(Animal[] arr) {
    for (int i = 0; i < arr.length; i++)
        arr[i] = new Dog();
}

public static void main(String[] args) {
    Cat[] catArray = new Cat[5];
    fillArrayWithDogs(catArray);
}
```

A fordító a fenti kód esetén még csak nem is figyelmeztet minket, hogy a fenti kód veszélyes. A tömbök azonban a generikus osztályokkal szemben futásidőben is típusosak, ezért a hibás beszúrási kísérlet futtatásakor `ArrayStoreException` kivételt kapunk. A generikus osztályok kezelését azért nem lehetett a tömbökkel összhangban megvalósítani, mert a generikus osztályok esetén futásidőben már nem rendelkezünk információval a konkrét típusokról, ezért nem detektálható a típusütközés.

A fenti szabályok jelentősen korlátozzák a generikus osztályok használatát, a megszorítások ugyanis nem mindig indokoltak. Például ha a listát nem módosítjuk, csupán annak elemeit olvassuk, akkor biztonságos lenne leszármazott osztályok listáját megadni. Szerencsére erre is van lehetőség. Ha például a típus `Dog` vagy annak leszármazottja kell, hogy legyen, akkor a típusparaméter helyén a `? extends Dog` jelölést adjuk meg. Ennek jelentése bármely példány, amely kompatibilis a `Dog` osztállyal. Interfész esetén ugyanezt a jelölést használjuk, típusparaméterben nem használható az `implements` kulcsszó. Vesszővel elválasztva több őstípus is megadható. A következő példakód mutatja a jelölés használatát.

```
public static void barkAll(List<? extends Dog> list) {
    for (Dog d : list)
        d.bark();
}

public static void main(String[] args) {
    List<GermanShepherd> gsList = new ArrayList<>();
    gsList.add(new GermanShepherd());
    barkAll(gsList);
}
```

Hangsúlyozzuk, hogy ez a megoldás csak akkor biztonságos, ha nem tárolunk el semmit a generikus osztályban, csak kiolvasunk belőle, vagy metódusokat hívunk a benne tárolt példányokon. Olyan eset is előfordul azonban, hogy csak tárolunk valamit a

listában, de az elemeket nem olvassuk. Ilyenkor biztonságosan tárolhatók leszármazott osztályok is. A `? super Dog` jelölés a típusparaméterben azt fejezi ki, hogy minden olyan típus megfelelő, amelynek a `Dog` a leszármazottja. Ugyanis ha a lista általánosabb elemek referenciáját tárolja, akkor a `Dog` is tárolható lesz. Az első kódrészletben tehát ez a legpontosabb működő típusmegadás.

```
public static void fillListWithDogs(List<? super Dog> list) {
    for (int i = 0; i < 5; i++)
        list.add(new Dog());
}
```

Létezik a `<?>` jelölés is, amely bármely típust elfogad. Ez nem összekeverendő az `<Object>` jelöléssel, mert az kizárólag `Object`-példányokra vonatkozik, leszármazott típusokat sem enged meg. Az előbbivel a `<? extends Object>` jelölés ekvivalens.

5.3. A Collections API

A Java SE osztálykönyvtárának része a Collections API, ez a generikus osztályok segítségével általánosan használható tárolóstruktúrákat valósít meg. A generikus kollekciók a `java.util` csomagban vannak, és két fő interfésztípusra épülnek. Az első a `Collection`, ez a példányok különböző típusú gyűjteményeit reprezentálja. Ezt három további interfész terjeszti ki. A `Set` halmazt ír le, tehát egy elem csak egyszer szerepelhet benne. A `List` listát valósít meg, azaz az elemeket sorrendezetten tárolja. A `Queue` várakozási sorokat reprezentál, tehát az elemei rendezettek, és belőle sorrendben kivehetők.

A másik fontos interfész a a `Map`. Ez kulcs-érték párok kezelését teszi lehetővé, ezért két típusparaméterrel rendelkezik, ugyanis a kulcs és az érték is tetszőleges típusú lehet. Ebben az alfejezetben interfészenként áttekintjük a különböző struktúrákat és gyakran használt implementációikat.

5.3.1. A sorba rendezhető objektumok

Kollekciók esetén kétféle rendezettségről beszélhetünk. Sajnos a magyar nyelvben mindkét fogalomra rendezettségként hivatkozunk, nem létezik megfelelő terminológia a két fogalom megkülönböztetésére. Az első értelemben rendezett *(ordered)* kollekción azt értjük, hogy az elemek valamilyen kötött, megjósolható sorrendben nyerhetők ki a kollekcióból, például a beszúrás sorrendjében. A második értelemben rendezett *(sorted)* kollekcióelemeket a programozó által implementált rendezőalgoritmus rendezi, és ebben a sorrendben tárolja el. Alább a rendezési algoritmus megadásának módját vizsgáljuk meg.

A Java nyelv kétféle rendezési mechanizmust kínál. Az egyik a *természetes sorrend (natural order)* támogatása a `Comparable<T>` interfészen keresztül. Az interfész egy `int compareTo(T o)` szignatúrájú metódust ír elő, ez az implementáló osztály két példányának összehasonlítására használható. Attól függően, hogy az aktuális objektum kisebb, egyenlő vagy nagyobb a paraméterben megadott példánynál, a metódus rendre negatív, nulla vagy pozitív értéket ad vissza. Ezzel a módszerrel osztályonként

csupán egyféle rendezést adhatunk meg, ezért célszerű úgy megadni, hogy a legtermészetesebb sorrendet adja az osztály példányai között. Erre utal a természetes sorrend elnevezés is. Habár nem kötelező, a rendezésnek ajánlott konzisztensnek lennie az `equals()` metódus működésével, azaz ha két példány az `equals()` szerint azonos, akkor rendezésük is legyen azonos. Az alábbi kódrészlet ad példát a természetes rendezés megvalósítására gépjárművek rendszáma szerint. A `String` osztály maga is megvalósítja a `Comparable` interfészt, ezért a feladat könnyen megfogalmazható karakterláncok összehasonlításával. Természetesen bonyolultabb logika is megadható.

```
public class Vehicle implements Comparable<Vehicle> {
    private String ownerName;
    private String licensePlate;
    ...

    @Override
    public int compareTo(Vehicle v) {
        return licensePlate.compareTo(v.licensePlate);
    }
}
```

Előfordulhat azonban, hogy egy osztály többféle lehetséges rendezési módjával is dolgoznunk kell. Gépjárműveket reprezentáló objektumokat például rendezhetünk a rendszám, a tulajdonos neve, a hengerűrtartalom stb. szerint. A második rendezési mechanizmus tetszőleges számú eltérő rendezési algoritmus megvalósítását teszi lehetővé a komparátorok segítségével. A komparátorok a `Comparator<T>` interfészt valósítják meg, ahol `T` típusparaméter az összehasonlítandó osztályokat jelöli. Az interfész `int compare(T o1, T o2)` szignatúrájú metódust követel meg, amely negatív, nulla vagy pozitív értékkel tér vissza, ha az első paramétere rendre kisebb, egyenlő vagy nagyobb, mint a második. A Java dokumentációja felhívja rá a figyelmet, hogy a komparátorok `equals()` metódusát nem kötelező újradefiniálni, de ha megtesszük, akkor annak pontosan akkor kell igaz értéket visszaadnia, ha a két komparátor azonos rendezést eredményez. A megfelelően újradefiniált `equals()` metódus jelentősen meggyorsíthat néhány osztálykönyvtári műveletet. Az alábbi példában a fenti osztályhoz láthatunk egy komparátort, ez a tulajdonos neve szerint rendez.

```
class OwnerComparator implements Comparator<Vehicle> {

    @Override
    public int compare(Vehicle o1, Vehicle o2) {
        return o1.getOwnerName().compareTo(o2.getOwnerName());
    }
}
```

5.3.2. A kollekciók bejárása

A kollekciók bejárhatók a `for` ciklus segítségével (lásd 2.8.5. alfejezet) vagy iterátorokkal is. Habár a `for` ciklus igen tömör és jól olvasható szintaxissal rendelkezik, néha szükség van az iterátorok használatára az általuk nyújtott kiegészítő funkcionalitás miatt. Az iterátor olyan objektum, amely adatstruktúrák bejárását teszi lehetővé, és közben a bejárás aktuális állapotát tárolja. A `Collection` interfészben deklarált `iterator()` metódus iterátort ad vissza az adott kollekció elemeinek bejárásához. A visszaadott iterátor az `Iterator<E>` interfész implementációja, a típusparamétere a bejárás során visszaadott elemek típusát adja meg. Kezdetben az iterátor az első elem előtti pozícióra hivatkozik. A `hasNext()` metódus megvizsgálja, hogy van-e még bejárandó elem, a `next()` hívása pedig vissza is adja ennek értékét. Némelyik struktúra iterátora a `remove()` metódust is támogatja, ez a legutóbb kiolvasott elemet törli a struktúrából. Ha ez nincs támogatva, akkor ennek hívása `UnsupportedOperationException` kivételt eredményez. A bejárás közben a kollekciót csak a `remove()` metódussal lehet módosítani. Ha a kollekciót a bejárás közben más módon változtatjuk meg, akkor az iterátor metódusai `ConcurrentModificationException` kivételt eredményeznek. A következő lista összefoglalja az iterátorok metódusait:

`boolean hasNext()`
: Megvizsgálja, hogy van-e még bejárandó elem a struktúrában.

`E next()`
: Visszatér a következő bejárandó elemmel, ha nincs következő elem, akkor `NoSuchElementException` kivételt kapunk.

`void remove()`
: Ha a törlés lehetséges, akkor kitörli a struktúrából a legutoljára visszaadott elemet, különben `UnsupportedOperationException` kivételt kapunk.

5.3.3. A Collection interfész

A `Collection` interfész a kollekciók legalapvetőbb metódusait írja elő. Az összes `Set`, `List` és `Queue` implementáció rendelkezik ezekkel. Metódusait a következő lista ismerteti:

`boolean add(E e)`
: Ha lehetséges, akkor beszúrja a paraméterben megadott elemet a kollekcióba, és `true` értékkel tér vissza, ha a kollekció a beszúrás eredményeként megváltozott. Ha nincs támogatva, akkor `UnsupportedOperationException` kivételt kapunk.

`boolean addAll(Collection<? extends E> c)`
: Mint az `add()` metódus, de a megadott kollekció összes elemét felveszi.

`void clear()`
: Ha lehetséges, akkor kiüríti a kollekciót, különben `UnsupportedOperationException` kivételt kapunk.

`boolean contains(E e)`
: Visszaadja, hogy a megadott elem benne van-e a kollekcióban.

`boolean containsAll(Collection<?> c)`
Visszaadja, hogy a megadott kollekció elemei benne vannak-e a kollekcióban. Csak akkor ad vissza `true` értéket, ha minden elem része annak.

`void set(E e)`
Ha támogatott a művelet, akkor az utoljára kiolvasott elemet felülírja a megadottal, különben `UnsupportedOperationException` kivétel váltódik ki.

`boolean isEmpty()`
Ha a kollekció üres, akkor `true` értéket ad vissza, különben `false`.

`Iterator<E> iterator()`
Visszaad egy `Iterator`-példányt, amellyel a kollekció elemei bejárhatók.

`boolean remove(Object o)`
Ha támogatott a művelet, akkor törli a paraméterben megadott elemet a kollekcióból, és `true` értékkel tér vissza, ha a kollekció a törlés eredményeként megváltozott. Ha nincs támogatva, akkor `UnsupportedOperationException` kivételt kapunk.

`boolean removeAll(Collection<?> c)`
Mint a `remove()`, de a paraméterben átadott kollekció összes elemét törli.

`boolean retainAll(Collection<?> c)`
Ha támogatott a művelet, akkor csak azokat az elemeket tartja meg, amelyek a megadott kollekciónak is elemei, és `true` értékkel tér vissza, ha a művelet eredményeként a kollekció megváltozott. Ha nincs támogatva, akkor `UnsupportedOperationException` kivételt kapunk.

`int size()`
A kollekció elemeinek számát adja vissza.

`Object[] toArray()`
A kollekció elemeit `Object[]` tömbben adja vissza.

`<T> T[] toArray(T[] a)`
A kollekció elemeit tömbben adja vissza. A tömb típusa a paraméterben megadott tömbével fog megegyezni. Az átadott tömböt a metódus nem módosítja, csupán a típus megállapításához használja. Akár nulla méretű tömb is megadható, például `new String[0]`.

A Java-szabvány nem mond semmit a `Collection` interfészt megvalósító osztályok `equals()` metódusáról, de kiköti, hogy mindig szimmetrikusnak kell lennie, azaz `a.equals(b)` pontosan akkor teljesül, ha `b.equals(a)` is igaz. Ezen kívül a `List` és a `Set` interfészek specifikációja szerint lista csak listával, halmaz pedig csak halmazzal lehet egyenlő. Saját kollekcióosztályok megvalósításánál ügyelni kell arra, hogy mindhárom kritérium teljesüljön.

5.3.4. A Set interfész

A `Set` interfészt megvalósító objektumok a matematikai halmazokat modellezik, azaz minden elem csak egyszer szerepelhet bennük. Az elemek egyenlőségét az osztály az `equals()` metódus alapján dönti el, ezért azt mindig megfelelően újra kell definiálni azokban az osztályokban, amelyeket halmazban szándékozunk tárolni. Ez a `hashCode()` metódusra is igaz. A halmazok működése nincs definiálva abban az esetben, ha az objektum állapota olyan módon változik meg, hogy az az `equals()` és `hashCode()` metódusok eredményét befolyásolja. Halmaz nem tartalmazhatja önmagát. Ha mégis megkíséreljük a halmazban önmagát tárolni, annak eredménye `StackOverflowError` lehet. Általánosan a halmazok tartalmazhatnak egy `null` elemet, de a konkrét implementáció további megszorításokkal élhet a tartalmazott elemeket érintően.

A `Set` interfész nem ír elő újabb metódusokat a `Collections` interfész metódusain kívül, csupán jelzi, hogy az implementáló osztályok a halmazok szemantikáját valósítják meg. Az interfész leggyakrabban használt implementációs osztályai a `HashSet` és a `LinkedHashSet`. A fő különbség, hogy az előbbi bejárása nem megjósolható sorrendben történik, míg a második láncolt listában tárolja el az elemeket, ezért azokat a beszúrás sorrendjében járja be. A láncolt lista fenntartása miatt azonban néha csekély mértékben lassabb lehet a `HashSet` osztálynál.

Létezik egy `SortedSet` leszármazott interfész is, ez rendezett halmazok ősosztályaként szolgál. Az implementáló osztályok a rendezést vagy a természetes sorrend, vagy a megadott komparátor szerint végzik el. Ugyan interfész nem tud konstruktort előírni, a szabvány javaslata az, hogy az osztály implementációi négy konstruktorral rendelkezzenek:

1. Alapértelmezett konstruktor, amely üres, természetes sorrend szerint rendező halmazt hoz létre.

2. `Comparator`-példányt váró konstruktor, amely üres, a komparátor szerint rendező halmazt hoz létre.

3. `Collection`-példányt váró konstruktor, amely a kollekció elemeit rendezi természetes sorrend szerint a halmazban.

4. `SortedSet`-példányt váró konstruktor, amely azonos módon rendező másolatot készít a rendezett halmazról.

Az interfész a következő metódusokat írja elő:

`Comparator<? super E> comparator()`
Visszaadja a használt komparátort, illetve `nullt`, ha a halmaz a természetes sorrend szerint van rendezve.

`E first()`
Visszaadja a halmaz első elemét.

`SortedSet<E> headSet(E toElement)`
Visszaadja azt a részhalmazt, amely a megadottnál szigorúan kisebb elemekből áll. A részhalmaz és a teljes halmaz kapcsolatban marad, az egyikben történt változás a másikban is tükröződik. A részhalmazba azonban csak a felső korlátnál kisebb elemek szúrhatók be, különben `IllegalArgumentException` kivételt kapunk.

`E last()`
Visszaadja a halmaz utolsó elemét.

`SortedSet<E> subSet(E fromElement, E toElement)`
Visszaadja azt a részhalmazt, amely az első elemnél nagyobb vagy egyenlő, de a második elemnél szigorúan kisebb elemekből áll. A részhalmaz és a teljes halmaz kapcsolatban marad, az egyikben történt változás a másikban is tükröződik. A részhalmazba azonban csak a megadott korlátok közé eső elemek szúrhatók be, különben `IllegalArgumentException` kivételt kapunk.

`SortedSet<E> tailSet(E fromElement)`
Visszaadja azt a részhalmazt, amely a megadottnál szigorúan nagyobb elemekből áll. A részhalmaz és a teljes halmaz kapcsolatban marad, az egyikben történt változás a másikban is tükröződik. A részhalmazba azonban csak az alsó korlátnál nagyobb elemek szúrhatók be, különben `IllegalArgumentException` kivételt kapunk.

A `NavigableSet` interfész még szélesebb körű navigációs lehetőségeket biztosít a rendezett halmaz elemein, ennek metódusait azonban nem részletezzük. A `TreeSet` implementálja a `SortedSet` és a `NavigableSet` interfészt is. Az alábbi példa szöveget olvas a szabványos bemenetről, szavakra bontja, majd egy rendezett halmaz segítségével betűrendben kiírja az összes előforduló szót.

```
public static void distinct() throws IOException {
    Set<String> set = new TreeSet<>();
    String s = br.readLine();
    while (s != null) {
        s = s.toLowerCase();
        StringTokenizer tok = new StringTokenizer(s,
            " .,:\n-?!+\"\'()");
        while (tok.hasMoreTokens())
            set.add(tok.nextToken());
        s = br.readLine();
    }
    System.out.println("A bemeneten kapott szavak:");
    for (String t : set)
        System.out.println(t);
}
```

5.3.5. A List interfész

A `List` interfész elemek rendezett listáját valósítja meg. Másolatok és `null` elemek általában meg vannak engedve, de az egyes implementációs osztályok tehetnek további megkötéseket. A listába elemek tetszőleges helyre beszúrhatók, majd bejárással vagy a sorszámuk alapján is elérhetők. A lista speciális iterátorral (`ListIterator`) rendelkezik, ez kétirányú navigációt, valamint beszúrást és cserét is lehetővé tesz. A lista úgy is felfogható, mint egy dinamikusan változó méretű tömb, amelybe könnyen be lehet szúrni elemeket. Ügyelni kell azonban arra, hogy több listaimplementáció valamilyen láncoltlista-mechanizmust használ, ezért az egyes elemek eléréséhez sokszor be kell járni az azt megelőző elemeket. Ha több különböző elemet kell egymás után elérni, akkor célszerűbb lehet tehát egyetlen bejárást alkalmazni az egyes elemek indexelt elérése helyett. A `List` interfész a `Collection` metódusain kívül az alábbiakat írja elő:

`void add(int index, E element)`
: Ha támogatott a művelet, akkor beszúrja a paraméterben megadott elemet az adott indexű helyre, különben `UnsupportedOperationException` kivételt kapunk.

`boolean addAll(int index, Collection<? extends E> c)`
: Ha támogatott a művelet, akkor beszúrja a paraméterben megadott kollekció összes elemét az adott indexű helynél kezdve, különben `UnsupportedOperationException` kivételt kapunk. Akkor tér vissza `true` értékkel, ha a beszúrás során a lista megváltozott.

`E get(int index)`
: Visszaadja az adott indexű elemet. Ha a megadott index nem létezik, akkor `IndexOutOfBoundsException` kivétel váltódik ki.

`int indexOf(Object o)`
: Visszaadja a megadott objektum első előfordulásának indexét, illetve `-1`-et, ha az nem eleme a listának.

`int lastIndexOf(Object o)`
: Visszaadja a megadott objektum utolsó előfordulásának indexét, illetve `-1`-et, ha az nem eleme a listának.

`ListIterator<E> listIterator()`
: Visszaadja a lista elemeinek bejárásához használható listaiterátort (lásd 5.3.2. alfejezet).

`ListIterator<E> listIterator(int index)`
: Visszaadja a lista elemeinek bejárásához használható listaiterátort (lásd 5.3.2. alfejezet), a bejárás a megadott indexű pozíciónál kezdődik.

`E remove(int index)`
: Törli a listából az adott indexű elemet, és visszaadja. A lista ezt követő elemeinek indexe eggyel csökken.

`E set(int index, E element)`
: Lecseréli az adott indexű elemet a paraméterben átadottal, és visszaadja az előző értéket.

`List<E> subList(int fromIndex, int toIndex)`
A listának azt a részét adja vissza, amelyre az index nagyobb vagy egyenlő az első paraméternél, de szigorúan kisebb a másodiknál. A visszaadott lista összekapcsolódik az eredetivel, azaz az egyiken elvégzett módosítások a másikon is tükröződnek. Ha az eredeti listában törlést vagy beszúrást hajtunk végre, akkor azonban a részlista további működése nincs definiálva.

A szabvány előírja továbbá, hogy lista az `equals()` metódus szerint csak listákkal lehet egyenlő. Két lista pontosan akkor egyenlő, ha méretük egyenlő, és ugyanazokat az elemeket tartalmazzák ugyanabban a sorrendben. Az interfész leggyakrabban használt megvalósításai az `ArrayList` és a `LinkedList` osztályok. Előbbi tömbbel valósítja meg a listát, és valamivel jobb teljesítményt nyújt, mint az utóbbi. Az utóbbi kétszeresen láncolt listát alkalmaz. Mindkét osztály megvalósítja az összes opcionális metódust, és megengedi a `null` elemeket. A `Vector` osztály ekvivalensnek tekinthető az `ArrayList` osztállyal, de annak metódusai szinkronizáltak (lásd 11.5. alfejezet), így többszálú alkalmazásokban is használható. Ebből adódóan ugyanakkor teljesítménye némileg elmarad az `ArrayList` osztálytól. Az alábbi példa az `ArrayList` osztályban tárolja el a bemeneten kapott szavakat, majd azokat fordított sorrendben írja ki őket. Ehhez listaiterátort használunk.

```
public static void reverseWithList() throws IOException {
    List<String> list = new ArrayList<>();
    String s = br.readLine();
    while (s != null) {
        s = s.toLowerCase();
        StringTokenizer tok = new StringTokenizer(s,
            " .,:\n-?!+\"\'()");
        while (tok.hasMoreTokens())
            list.add(tok.nextToken());
        s = br.readLine();
    }
    System.out.println("A bemeneten kapott szavak fordított ↵
sorrendben:");
    // listaiterátorral járjuk be visszafele
    ListIterator<String> it = list.listIterator(list.size());
    while (it.hasPrevious())
        System.out.println(it.previous());
}
```

A `List` interfész `listIterator()` metódusa `ListIterator<E>` típusú iterátort ad vissza. Ez az általános iterátor funkcióin kívül képes visszafelé is lépkedni, valamint a listába elemet beszúrni. Ennek a kiegészítő metódusait az alábbi lista ismerteti.

`void add(E e)`
Ha támogatott a művelet, akkor az aktuális pozícióra beszúrja a megadott elemet, különben `UnsupportedOperationException` kivételt kapunk.

`boolean hasPrevious()`
Megvizsgálja, hogy az aktuális pozícióhoz képest létezik-e előző elem.

`int nextIndex()`
A következő elem indexét adja vissza, vagy a tömb méretét, ha már az utolsó elemet is kiolvastuk.

`E previous()`
Az előző elemet adja vissza, és eggyel visszaállítja az aktuális pozíciót. Ha nincs előző elem, akkor `NoSuchElementException` kivétel váltódik ki.

`int previousIndex()`
Az előző elem indexét adja vissza, illetve `-1`-et, ha az aktuális pozíció a lista eleje.

`void set(E e)`
Ha támogatott a művelet, akkor az utoljára kiolvasott elemet felülírja a megadottal, különben `UnsupportedOperationException` kivételt kapunk.

5.3.6. A Queue interfész

A `Queue` interfész várakozási sort reprezentál. A sorok megvalósítása lehet FIFO (first in, first out, azaz először be, először ki), LIFO (last in, first out, azaz először be, utoljára ki) vagy prioritásos. Első esetben a legkorábban tárolt elemet kapjuk meg először a lekérdezésnél, míg a LIFO sor a legkésőbb beszúrt elemet adja vissza először. A prioritásos sorok az elemeket komparátor vagy természetes sorrend szerint rendezik, és az elemek kinyerése is eszerint történik. A `Queue` interfész kiterjeszti a `Collection` interfészt, de rendelkezik néhány további, a várakozási sorokra jellemző metódusokkal is. Három jellemző művelet végezhető el a sorokon: felvehetünk új elemeket a sorba, lekérdezhetjük a sor első elemét anélkül, hogy azt kivennénk, illetve ki is vehetjük azt a sorból. A `Queue` interfész mindhárom műveletre kétféle metódust kínál: az egyik kivételt eredményez, ha a műveletet nem lehet elvégezni, a másik pedig visszatérési értékkel jelzi a művelet sikertelenségét. Az alábbi lista ezt a hat metódust foglalja össze:

`boolean add(E e)`
Beszúrja az elemet a sorba, ha van elegendő hely. Korlátozott méretű soroknál lehetséges, hogy az elem számára már nincs hely, ezért nem szúrható be. Ha sikeres volt a művelet, akkor `true` értékkel tér vissza, különben `IllegalStateException` kivételt kapunk.

`boolean offer(E e)`
Beszúrja az elemet a sorba, ha van elegendő hely. Visszaadja, hogy sikeres volt-e a beszúrás.

`E element()`
Visszaadja a sor első elemét, de nem veszi ki a sorból. Ha üres a sor, akkor `NoSuchElementException` kivételt kapunk.

`E peek()`
Visszaadja a sor első elemét, de nem veszi ki a sorból. Ha üres a sor, akkor `nullt` ad vissza.

`E remove()`
Visszaadja a sor első elemét, és ki is veszi a sorból. Ha üres a sor, akkor `NoSuchElementException` kivételt kapunk.

`E poll()`
Visszaadja a sor első elemét, és ki is veszi a sorból. Ha üres a sor, akkor `null`t ad vissza.

A várakozási sorokon az `equals()` metódusnak nincs természetes értelmezése, ezért nem definiálják újra, hanem a referenciák egyenlőségét vizsgálják. A már ismertetett `LinkedList` osztály a `Queue` interfészt is implementálja, és ez a legegyszerűbb, legáltalánosabb implementáció, amely FIFO elven működik. A `PriorityQueue` természetes rendezés, vagy komparátor alapján rendezi a beszúrt elemeket, és sorrendben adja vissza őket.

A várakozási soroknak specializáltabb formái is léteznek. A `Deque` interfészt megvalósító osztályok olyan várakozási sorok, amelyeknek mindkét végéről kiszedhetők az elemek. Ezzel a mechanizmussal már LIFO elven működő sorok is készíthetők. A `LinkedList` a `Deque` interfészt is implementálja. Az `ArrayDeque` egy másik, tömböket használó implementáció. Az interfész metódusait nem részletezzük, de a következő példa ennek használatával fordítja meg a bemeneten kapott szavak sorrendjét.

```
public static void reverseWithDeque() throws IOException {
    Deque<String> q = new ArrayDeque<>();
    String s = br.readLine();
    while (s != null) {
        s = s.toLowerCase();
        StringTokenizer tok = new StringTokenizer(s,
            " .,:\n-?!+\"\'()");
        while (tok.hasMoreTokens())
            q.offerFirst(tok.nextToken());
        s = br.readLine();
    }
    System.out.println("A bemeneten kapott szavak fordított ↵
sorrendben:");
    while (!q.isEmpty())
        System.out.println(q.pollFirst());
}
```

Szintén a `Queue` interfészre épülnek a `BlockingQueue` és `BlockingDeque` interfészek. Ezek olyan metódusokat írnak elő, amelyek elaltatják a hívó szálat, ha a sorban nincs hely elem beszúrására, vagy nincs kivehető elem. Ezek tehát jól használhatók termelő-fogyasztó jellegű problémáknál többszálú környezetben (lásd 11.7. alfejezet).

5.3.7. A Map interfész

A Map interfész kulcs-érték párok tárolására szolgál, tehát két típusparaméterrel is rendelkezik. Ezt az adatstruktúrát szótárnak is nevezik, és a könyv is ezt a terminológiát használja. Minden kulcshoz csak egy érték tartozhat. Az interfész nem tiltja a null kulcs vagy értékek használatát, de egyes implementációk tehetnek további megkötéseket a használható értékekre. Tulajdonképpen a szótárak három kollekciót foglalnak magukban: a kulcsok halmazát, az értékek kollekcióját és a kulcs-érték hozzárendelések halmazát. A kulcsokat a kollekció a hashCode() és equals() metódusok alapján azonosítja, ezért fontos ezen metódusok megfelelő újradefiniálása a kulcsként használt osztályban. Ugyan az interfészek nem írhatnak elő konstruktorokat, a szabvány azt javasolja, hogy az interfészt megvalósító osztályoknak legyen alapértelmezett és egyetlen Map paramétert fogadó konstruktora. Előbbi üres szótárt, utóbbi pedig a megadott szótár másolatát hozza létre. A Map interfész metódusait a következő lista foglalja össze:

`void clear()`
: Ha lehetséges, törli a szótár tartalmát. Ha nem támogatott a művelet, UnsupportedOperationException kivételt kapunk.

`boolean containsKey(Object key)`
: Visszaadja, hogy a szótár tartalmaz-e értéket a megadott kulcshoz.

`boolean containsValue(Object value)`
: Visszaadja, hogy van-e olyan kulcs, amelyhez a megadott érték tartozik.

`Set<Map.Entry<K,V>> entrySet()`
: Halmazt ad vissza, amelynek elemei Map.Entry típusú objektumot, valamint minden objektum egy kulcs-érték párt tartalmaz. A Map.Entry osztály getKey() és getValue() metódusa használható a kulcs és az érték eléréséhez, illetve ha támogatott a művelet, akkor az érték meg is változtatható a setKey() metódussal.

`V get(Object key)`
: Visszaadja a kulcshoz tartozó értéket, illetve nullt, ha a megadott kulcshoz nem tartozik érték.

`boolean isEmpty()`
: Visszaadja, hogy üres-e a szótár.

`Set<K> keySet()`
: A kulcsok halmazát adja vissza.

`V put(K key, V value)`
: A megadott kulcshoz rendeli a megadott értéket, ha támogatva van a művelet, különben UnsupportedOperationException kivételt kapunk. Visszaadja a kulcshoz tartozó korábbi értéket, vagy nullt, ha a kulcshoz nem tartozott érték.

`void putAll(Map<? extends K,? extends V> m)`
: Ha támogatva van a művelet, átmásolja a megadott szótárban lévő bejegyzéseket, különben UnsupportedOperationException kivételt kapunk.

`V remove(Object key)`
Ha támogatva van a művelet, törli a megadott kulcshoz tartozó bejegyzést, különben `UnsupportedOperationException` kivételt kapunk.

`int size()`
Visszaadja a szótár méretét.

`Collection<V> value()`
Kollekcióként adja vissza a szótárban tárolt értékeket.

Az interfész leggyakrabban használt implementációja a `HashMap`. Ez vödrös hasheléssel tárolja el és keresi ki az elemeket. A bejárás sorrendjét a Java-szabvány nem határozza meg. A `LinkedHashMap` kétszeresen láncolt listát használ, ezért a bejárási sorrend a beszúrás sorrendjével egyezik meg. A legtöbb művelet konstans időben végezhető el mindkét implementációval. A bejárás azonban a `HashMap` esetén általában valamivel költségesebb. Az `IdentityHashMap` a `HashMap` olyan változata, amely nem az `equals()` metódus alapján hasonlít össze, hanem referenciális egyenlőséget vizsgál. Akárcsak a `Set` interfész esetében, a `Map` is rendelkezik `SortedMap` és `NavigableMap` variánsokkal. Az ezek által nyújtott kiegészítő funkcionalitás itt is hasonló, az implementáció neve pedig `TreeMap`. Az alábbi példaprogram a `Map` segítségével megszámolja, hogy a bemenetként kapott szövegben melyik szó hányszor fordul elő. A szavakat és az előfordulásukat `TreeMap`-példányban tároljuk, hogy az eredményt betűrendben írhassuk ki.

```
public static void countWords() throws IOException {
    Map<String, Integer> map = new TreeMap<>();
    String s = br.readLine();
    while (s != null) {
        s = s.toLowerCase();
        StringTokenizer tok = new StringTokenizer(s,
            " .,:\n-?!+\"\'()");
        while (tok.hasMoreTokens()) {
            String tmp = tok.nextToken();
            Integer count = map.containsKey(tmp) ? map.get(tmp) : 0;
            map.put(tmp, ++count);
        }
        s = br.readLine();
    }
    System.out.println("A bemeneten kapott szavak előfordulási ↵
száma:");
    for (Map.Entry<String, Integer> e : map.entrySet())
        System.out.println(e.getKey() + ": " + e.getValue());
}
```

5.3.8. A Collections osztály

A Collections osztály a statikus metódusai segítségével széles körű kiegészítő funkcionalitást kínál a kollekciók kezeléséhez. Az osztályról általában elmondható, hogy metódusai `null` paraméter esetén `NullPointerExceptiont` kivételt eredményeznek. Ha módosítással járó műveletet végzünk, de a megadott kollekció nem támogatja a `Collection` interfész ehhez szükséges opcionális műveleteit, akkor `UnsupportedOperationException` kivételt kapunk. Az osztály sok metódusa a Decorator tervezési mintát használja [4], azaz kiegészítő funkcionalitást valósít meg az átadott kollekción, de az alapvető funkcionalitást annak delegálja. Érdemes az osztály metódusait a Javadoc-referenciában átnézni, mert sok gyakori problémában segíthetnek, és használtuk rövidebbé teszi a kódot. Továbbá ezek a metódusok esetenként hatékonyabbak, mint egy naiv implementáció. Az osztály metódusai olyan funkcionalitást nyújtanak, mint üres és egyelemű kollekciók létrehozása, dinamikusan típusbiztos, szinkronizált és csak olvasható csomagolóobjektumok, rendezéssel kapcsolatos műveletek és listák manipulációja. Például az `unmodifiableList()` a megadott lista csak olvasható nézetét adja vissza.

5.3.9. Az Arrays osztály

Az `Arrays` a `Collections` osztályhoz hasonló funkcionalitást kínál tömbök számára. Mivel a tömbök primitív típusokat is tárolhatnak, az osztály legtöbb metódusa rendelkezik az objektumok, illetve az összes primitív típus kezeléséhez szükséges változattal. Az osztály metódusai magukban foglalják a mély egyenlőségvizsgálat és a hashkódszámítás, az adott elemmel való feltöltés, a másolás és az átméretezés, a rendezés, a keresés és a listanézet funkciót. Kiemelendő a `copyOf()` metódus használata, amelynek a második paraméterben hosszt is meg kell adni. Ha a másolat rövidebb a réginél, akkor csak a tömb elejéről készül másolat. Ha hosszabb, akkor a fennmaradó helyek a típustól függően `0`, `false` vagy `null` értékekkel lesznek feltöltve. A metódus ezért jól használható átméretezésre, de körültekintéssel kell használni, mert a tömbről valójában mindig másolat készül. Csökkentheti a másolat költségét például, ha a tömböt ritkábban, de több elemmel növeljük meg, és a ténylegesen kihasznált méretet változóban tároljuk.

HATODIK FEJEZET
Az állapot elmentése

Bár a grafikus felhasználói felület kifejlesztését eddig nem tárgyaltuk, a korábbi fejezetek alapján már bonyolult Java-alkalmazások is fejleszthetők. Összetett alkalmazásokban gyakran van szükség állapotmentésre. Tulajdonképpen ez az osztálykönyvtár fájlkezelési funkcióival tetszőleges módon is megoldható, a Java nyelv a feladathoz jobban illő megoldásokat is kínál. Az egyik lehetőség a Properties API használata. Ennek segítségével kulcs-érték párokat tudunk tárolni, ezért leginkább konfigurációs adatok tárolására alkalmas. A másik megoldás az objektumok sorosítása és visszatöltése, ez a programállapot mentésére használható. A fejezet bemutatja a két módszert. Az XML-formátum is alkalmas állapotmentésre, erről a 7. fejezetben szólunk.

6.1. A Properties API használata

A Properties API segítségével kulcs-érték párban tárolhatunk `String` típusú konfigurációs értékeket. A `Properties` osztály reprezentálja a konfigurációs beállítások meghatározott halmazát. Az osztály a `Hashtable` osztályból származik, ezért örököl olyan metódusokat is, amelyek `String`től eltérő típusú kulcsokat és értékeket is megengednek. Ezek beszúrása azonban hibás működéshez vezet, ezért ügyeljünk az elkerülésére. Az osztály azt is támogatja, hogy megadjunk alapértelmezett értékeket arra az esetre, ha a beolvasott konfiguráció nem tartalmaz egy-egy adott kulcshoz értékeket. Az alapértelmezett kulcs-érték párokat is egy `Properties`-példányban adjuk meg. Ez a példány is rendelkezhet alapértelmezéssel, és a példányok sora ilyen módon végtelen mélységig láncolható. Az osztály két formátumban támogatja a beállítások mentését és beolvasását: egyszerű szöveges formátumban és XML-szintaxis szerint. A beállításokkal való munkát három szakaszra oszthatjuk:

1. Az alkalmazás indulásakor a kimentett beállítások beolvasása és az alkalmazás inicializációja.

2. Az értékek felhasználása, esetleg megváltoztatása az alkalmazás futása során.

3. A futás befejeződése előtt az aktuális beállítások kimentése.

6.1.1. A beállítások betöltése

Először példányt szükséges létrehozni a `Properties` osztályból. Az osztály rendelkezik alapértelmezett, illetve `Properties`-példányt váró konstruktorral. Utóbbi szolgál az alapértelmezett értékek megadására. Ezután a `load()` metódust használhatjuk a szöveges formátumból való beolvasásra. Ennek az `InputStream` vagy a `Reader` példányát kell átadni. Az XML-formátumba mentett beállításokat a `loadFromXML()` metódussal olvashatjuk be, ennek azonban csak `InputStream` típusú paramétert adhatunk meg.

A fejezetben olyan példaprogramot írunk, amely a felhasználót köszönti, és kiírja az utolsó köszöntés óta eltelt órák számát. A program a Properties API segítségével fájlba menti a felhasználó nevét és a legutóbbi üdvözlés idejét. A mentést megvalósító metódust az alábbi kódrészlet mutatja be.

```
public static void loadSettings() {
    Properties props = new Properties();
    File file = new File(PROPSFILE);
    try {
        if (!file.exists())
            file.createNewFile();
        // beállítások betöltése
        props.load(new FileInputStream(file));
    } catch (FileNotFoundException e) {
        e.printStackTrace();
    } catch (IOException e) {
        e.printStackTrace();
    }
...
}
```

6.1.2. A konfigurációs értékek felhasználása

Adott kulcshoz tartozó értéket a `getProperty()` metódussal lehet lekérdezni. Ha a példány nem rendelkezik a kulcshoz tartozó értékkel, de meg lett adva az alapértelmezett értékek listája, akkor a metódus abból próbálja meg kiolvasni az értéket. Ha az alapértelmezett értékek listájában sem található a keresett érték, akkor a metódus `null` értéket ad vissza. A metódusnak van kétparaméteres változata is. Ennek a második paraméterben megadható egy további érték, amelyet ebben az esetben vissza kell adnia. A kulcsok halmaza is lekérdezhető. Erre a `stringPropertyNames()` szolgál, amely `Set<String>` típusú objektumot ad vissza. Itt láthatjuk a példaprogram azon részét, amely beolvassa a konfigurációt. A beolvasott adatok kulcsát konstansokban tároltuk, hogy elkerüljük az elgépelést.

```
// név beolvasása
name = props.getProperty(PROP_NAME);

// utolsó látogatás óta eltelt idő milliszekundumokban
String lastSeenStr = props.getProperty(PROP_LASTSEEN);
if (lastSeenStr != null)
    lastSeen = Long.valueOf(lastSeenStr);
```

Értékeket módosítani a `setProperty()` metódussal lehet. Ez két `String` paramétert vár: a kulcsot és a beállítandó értéket. A metódus a `Hashtable` osztály `put()` metódusát hívja, ezért `Object` típusú a visszatérési értéke, és a kulcshoz tartozó korábbi értéket adja vissza, vagy `null`t, ha nem volt korábbi érték. Az alábbi példa szemlélteti a beállítások felhasználását, azaz az üdvözlőszöveg megjelenítését:

```
String input = reader.readLine();
switch (input) {
case "1":
    StringBuffer greeting = new StringBuffer("Üdvözlöm");
    if (name != null) {
        greeting.append(", ");
        greeting.append(name);
    }
    greeting.append("!");
    System.out.println(greeting.toString());
    long now = new Date().getTime();
    if (lastSeen > 0) {
        long hours = (now - lastSeen) / 360000;
        System.out.println("Több mint " + hours + " órája nem ↺
láttam.");
    }
    lastSeen = now;
```

6.1.3. A beállítások mentése

Az alkalmazás bezárása előtt a megváltozott beállításokat el kell menteni, hogy a következő induláskor vissza lehessen állítani. Szöveges formátumba mentéshez a `store()` metódus használható, ennek első paramétere `OutputStream` vagy `Writer` típusú, második paramétere pedig karakterláncként megadott megjegyzés. XML-formában történő mentésre a `storeToXML()` metódus szolgál, ennek csak `OutputStream` objektumot használó változata van. Rendelkezik azonban háromparaméteres változattal, ennek utolsó paramétere a karakterláncként megadott karakterkódolás (lásd 4.6.2. alfejezet). Az alábbi példa bemutatja, hogyan lehet az alkalmazás bezárása előtt a konfigurációt elmenteni:

```
public static void saveSettings() {
    Properties props = new Properties();
    File file = new File(PROPSFILE);
    try {
        if (!file.exists())
            file.createNewFile();
        if (name != null)
            props.setProperty(PROP_NAME, name);
        if (lastSeen != 0)
            props.setProperty(PROP_LASTSEEN,
                Long.toString(lastSeen));
```

```
        props.store(new FileOutputStream(file), "Greeting program ↩
configuration");
    } catch (FileNotFoundException e) {
        e.printStackTrace();
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

6.2. Az objektumok sorosítása

Az objektumok *sorosítása (serialization)* lehetővé teszi, hogy az objektumpéldány teljes állapotát, azaz példányváltozóinak értékét elmentsük, majd egy későbbi időpontban visszaállítsuk. Mindezt a Java nyelv transzparens módon támogatja, a legtöbb esetben nincs szükség arra, hogy a részletekbe beleavatkozzunk. A következőkben megismerkedünk a sorosítás működésével.

6.2.1. A sorosítás működése

Az objektumok alapértelmezésben nem sorosíthatók, egyes osztályok ugyanis olyan információt reprezentálnak, amelyek a program következő futása során már értelmetlenek lennének (például a szálakat reprezentáló `Thread` osztály példányai), vagy biztonsági kockázatot jelentenének (például olyan osztályok, amelyek privilégiumokat vagy hozzáférési jelszavakat tárolnak). A sorosítható osztályokat ezért explicit módon meg kell különböztetni. Ez a `Serializable` interfész implementálásával tehető meg. Az interfész üres, ún. *marker interfész*, azaz nem ír elő egyetlen metódust sem, csak jelzi, hogy az őt megvalósító osztályok sorosíthatók.

A sorosítható osztályokat ezután az `ObjectOutputStream` és az `ObjectInputStream` osztályokkal tudjuk sorosítani, illetve betölteni. Az `ObjectOutputStream` konstruktora `OutputStream`-példányt (lásd 4.5.1. alfejezet) vár. Ez reprezentálja az adatfolyamot, amelybe az objektumot sorosítjuk. Ennek típusa lehet például `FileOutputStream`, ha az objektum állapotát fájlba akarjuk menteni, vagy `ByteArrayOutputStream` is, így a sorosított objektumot egyéb módon is feldolgozhatjuk, például hálózaton is elküldhetjük. Az `ObjectOutputStream`be az objektumot a `writeObject()` metódussal írhatjuk ki. A metódus automatikusan kiírja az objektum összes példányváltozóját az adatfolyamba. Ha az objektum példányváltozói között más objektumok is vannak, akkor ezeknek is sorosíthatóknak kell lenniük, és a sorosítás során ezek is kimentődnek, végtelen mélységig. Ha van nem sorosítható példányváltozó is, akkor a sorosítás során `NotSerializableException` váltódik ki. Ezeket, vagy más példányváltozókat, amelyek sorosíthatók ugyan, de elmentésük nem fontos, a `transient` kulcsszóval megjelölve tranziessé tehetjük. A tranziens változók nem sorosítódnak. A sorosításhoz készült példaprogram egyszerű határidőnaplót valósít meg. A bejegyzéseket rendezett halmazban tároljuk, majd mentéskor a halmazt fájlba írjuk ki. A kollekciók sorosíthatók, ha a bennük tárolt elemek mind megvalósítják a `Serializable` interfészt. Az alábbi kódrészlet végzi a határidőnapló fájlba írását.

```
System.out.println("Fájlnév:");
input = reader.readLine();
File outputFile = new File(input);
ObjectOutputStream oos = new ObjectOutputStream(new FileOutputStream
(outputFile));
oos.writeObject(events);
oos.close();
```

A betöltéshez az `ObjectInputStream` osztály használható, ennek konstruktora `InputStream`-példányt vár. Ebből olvassa be a sorosított objektumot a `readObject()` metódus hívásakor. Általános jellege miatt ez `Object` típusú referenciát ad vissza, tehát konvertálni kell. Az alábbi példában beolvassuk a kimentett határidőnaplót.

```
System.out.println("Fájlnév:");
input = reader.readLine();
File inputFile = new File(input);
ObjectInputStream ois = new ObjectInputStream(new FileInputStream
(inputFile));
events = (Set<Event>) ois.readObject();
ois.close();
```

Fontos speciális eset az, amikor az osztály, amelynek példányait sorosítani kívánjuk, nem sorosítható szülővel rendelkezik. A szülőtől örökölt példányváltozók nem mentődnek el, hanem a szülő konstruktora fut le, és azok az inicializálás utáni kezdeti értékeiket veszik fel. Mivel a közvetlen szülő konstruktora meghívódik, ezért tulajdonképpen az összes ős konstruktora le fog futni, és a tőlük örökölt példányváltozók eszerint inicializálódnak. A sorosítható osztály példányváltozóinak értékét az adatfolyamból olvassuk be, és ezekkel az értékekkel inicializáljuk őket, ezért ennek az osztálynak a konstruktora nem fut le.

6.2.2. A sorosítás testre szabása

A sorosítás testre szabására kétféle megoldás létezik. Az első továbbra is a szabványos sorosítási formátumot alkalmazza, és inkább kiegészítések hozzáadására, mintsem a folyamat teljes megváltoztatására alkalmas. A második módszer esetén egyáltalán nem támaszkodunk a szabványos sorosítási folyamatra, ezért az teljesen egyedi módon implementálható.

Általában elegendő az első megoldás használata. Két fő esetben van rá szükség, hogy a sorosítási folyamatot kiegészítsük. Az egyik eset, hogy a szülőosztály nem sorosítható, annak néhány példányváltozóját mégis el akarjuk menteni. A másik esetben a sorosítani kívánt objektumnak olyan példányváltozója van, amely nem sorosítható. Kézenfekvő megoldás lenne, hogy azt is sorosíthatóvá tegyük, vagy hozzunk létre belőle sorosítható leszármazott osztályt, és helyette ezt használjuk. Ezek a megoldások azonban nem mindig lehetségesek, elképzelhető ugyanis, hogy az osztály az osztálykönyvtár része, így nem tudjuk módosítani, vagy `final` kulcsszóval van megjelölve, ezért nem lehet belőle leszármazottat létrehozni. Lehetnek más, gyakorlati okai is annak, hogy ezek a megoldások nem alkalmazhatók. Ekkor csak az a megoldás használ-

ható, hogy a példányváltozót tranzienssé tesszük, és annak állapotát magunk mentjük el. A sorosítást úgy tudjuk kiegészíteni, hogy az alábbi két metódust valósítjuk meg az osztályban:

```
private void writeObject(ObjectOutputStream os) throws IOException
private void readObject(ObjectInputStream is) throws IOException
```

Ezek a metódusok `private` módosítóval vannak megjelölve, hogy ne öröklődjenek. Az osztálykönyvtár azonban el tudja érni őket, és sorosításkor, valamint az állapot betöltésekor ezek fognak meghívódni. A `writeObject()` metódusban a paraméterben kapott `ObjectOutputStream` metódusait használjuk arra, hogy az objektum példányváltozóit elmentsük. A `defaultWriteObject()` az alapértelmezett módon végzi el a sorosítást, mint ha azt nem is módosítottuk volna. Ezt rendszerint meghívjuk, előtte vagy utána pedig egyéb adatokat is kiírhatunk. Ehhez az `ObjectOutputStream` osztály `writeXxx()` metódusait használjuk, ahol *Xxx* a kiírni kívánt típus neve. A beolvasás hasonlóan történik. A `readObject()` metódusban az `ObjectInputStream` metódusait használjuk beolvasásra. A `defaultReadObject()` beolvassa az alapértelmezésben sorosított állapotot, a `readXxx()` metódusokkal pedig tetszőleges típusú adatot olvashatunk be, és azzal a példányváltozókat inicializálhatjuk. Fontos, hogy az adatokat ugyanabban a sorrendben olvassuk be, ahogyan azokat a `writeObject()` metódusban kimentettük. Az olvasáshoz és íráshoz használt metódusok kiválthatnak kivételt, amelyeket akár tovább is adhatunk, de érdemes őket itt kezelni.

A sorosítás testre szabásának másik módja az `Externalizable` interfész implementálása. Az interfész az alábbi két metódust írja elő:

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

Az interfész implementálásával teljesen kézben tarthatjuk a sorosítási folyamatot. Alapértelmezésben csak az objektum típusa íródik ki, a teljes állapot mentése és visszaállítása a programozó feladata. Ehhez az `ObjectOutput` és `ObjectInput` metódusait használhatjuk. Visszaállításkor az osztályból először létrejön egy példány az alapértelmezett konstruktor meghívásával, majd lefut a `readExternal()` metódus. Az `Externalizable` interfészt megvalósító osztályoknak tehát rendelkezniük kell alapértelmezett konstruktorral. A `Serializable` interfész használata esetén ez nem követelmény.

6.2.3. Megfontolások a sorosításhoz

A sorosítás az objektumok állapotának mentésére szolgál. Az osztályváltozók az osztályhoz és nem annak példányaihoz kapcsolódnak, tehát nem képezik részét az objektumok állapotainak. Ezért sorosítás során ezek nem mentődnek el.

A sorosítás során használt formátum alapértelmezésben nem tartalmaz semmiféle titkosítást, és a Java-szabvány egyértelműen specifikálja azt. A sorosítás használata ezért biztonsági kérdéseket is felvet. Kényes adatok sorosítását mellőzni kell, vagy megfelelően titkosított formátumban kell tárolni őket. A sorosítással ráadásul nem-

csak a bizalmasság, hanem az integritás is sérülhet. A sorosítás ugyanis alapértelmezésben nem használ védelmi mechanizmust az integritás ellenőrzésére. Az integritás védelmére a sorosítandó objektumot a `java.security.SignedObject` osztály példányába csomagolhatjuk. A bizalmasság betartására a `javax.crypto.SealedObject` nyújt hasonló megoldást. Ezeket könyvünk nem tárgyalja.

Előfordulhat, hogy osztályt származtatunk sorosítható osztályból, de nem akarjuk megengedni ennek sorosítását. Ebben az esetben implementáljuk a `writeObject()` és a `readObject()` metódust, és dobjunk bennük `NotSerializableException` kivételt.

Ahogy az alkalmazások fejlődnek, a sorosítható osztályok is módosulhatnak. Ez befolyásolhatja a sorosítást, a régi verzióval sorosított objektumok ugyanis nem mindig állíthatók vissza megfelelően. Szerencsére a Java az objektumok verziózására is kínál megoldást. Az osztály verziószámát egy privát, osztályszintű, `long` típusú konstansban adhatjuk meg, ezt `serialVersionUID` névvel kell ellátni. Ha a sorosított objektum verziószáma eltér a jelenlegitől, akkor visszaállításkor `InvalidClassException` kivétel váltódik ki. A verziószámot nem kell minden változtatás után frissíteni, csak akkor, ha a változtatás sérti a kompatibilitást. Ilyen változtatásnak számít a példányváltozók törlése, statikussá vagy tranzienssé tétele, a primitív tagváltozók típusának megváltoztatása, az osztály helyének megváltoztatása az osztályhierarchiában, az `enum` típusról normál osztályra váltás vagy fordítva, illetve a `Serializable` és az `Externalizable` interfészek közti váltás. Kompatibilis változtatás például új példányváltozók és metódusok hozzáadása, osztályváltozó példányváltozóvá tétele, tranziens változó perzisztenssé tétele vagy példányváltozó hozzáférési módosítójának megváltoztatása. Ha nem definiáljuk az osztályban a verziószámot, akkor azt a fordító generálja az osztály definíciója alapján. Tehát a generált verziószám gyakorlatilag minden változtatáskor frissül, és a kompatibilis változtatások is inkompatibilitáshoz vezetnek. Ha fontos a kompatibilitás, akkor kezdjük azzal az osztályok implementációját, hogy megadjuk a verziószámot, és csak inkompatibilis változtatások esetén növeljük azt.

HETEDIK FEJEZET

XML-feldolgozás Java nyelven

Napjaink szoftvereiben számos különböző feladatra használunk XML-adatokat. Az XML-formátumot sokszor használjuk konfigurációs fájlként, de lehet szöveges szakterületi nyelvek hordozószintaxisa is, tehát szakterületi programok is készíthetők vele. Használható adatok sorosítására is. A W3C-szabvány Web Services technológiája is az XML-formátumot használja távoli metódusok meghívására, illetve a sorosított adatok átvitelére. Az XML ugyanis gyártó- és technológiafüggetlen szabvány, ezért jól alkalmazható különböző technológiákkal megvalósított rendszerek közti integrációra. Szintén használható szöveges dokumentumok jelölésekkel való ellátására. Jelen könyv is XML-formátumban íródott a DocBook séma alapján. Az XML használatával tehát a legtöbb szoftverfejlesztő találkozik. A fejezet az XML feldolgozását mutatja be. A SAX és DOM szabványokat csak röviden tekintjük át, részletesen csak a JAXB technológiát tárgyaljuk. Az XML alapjaival nem foglalkozunk, azt [5] mutatja be részletesen.

7.1. Az XML-feldolgozás módszerei

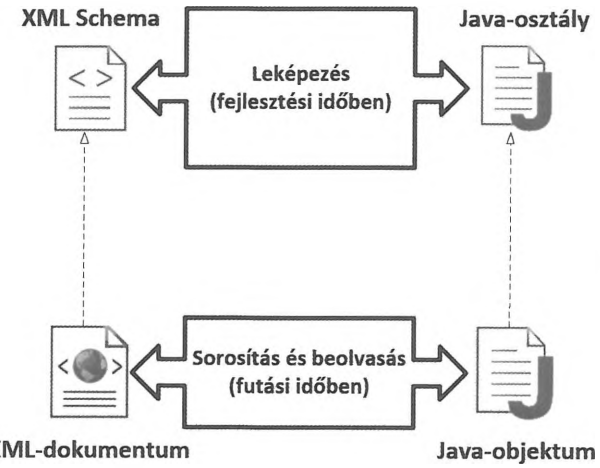
Az XML-dokumentumok feldolgozásának két legelterjedtebb módját a SAX és DOM szabványok írják le. Ezek általános szabványok, a feldolgozáshoz használt interfészt úgy definiálják, hogy az bármilyen programozási nyelven megvalósítható legyen. A SAX alapelve, hogy az XML-dokumentumot szekvenciálisan olvassa, és annak részeit eseményekre képezi le. Események tehát az elemek, attribútumok, megjegyzések, feldolgozási utasítások, amelyeket az elemző a beolvasás sorrendjében talál. A SAX-keretrendszer az események bekövetkeztekor a programozó által regisztrált eseménykezelő objektum metódusait hívja. Ezekben adható meg, hogyan kell a programnak az adott elem, attribútum stb. előfordulására reagálnia. A SAX technológia előnye, hogy a szekvenciális végigolvasás közben nem kell állapotot tárolni, tehát a dokumentumok gyorsan és kevés memóriafelhasználással feldolgozhatók. Hátránya, hogy nem lehet visszafelé navigálni, sem módosítani a dokumentumot.

A másik szabvány, a DOM, a memóriában fastruktúrát épít az XML elemeiből. Ezt DOM-fának nevezzük. A DOM-fa tulajdonképpen egy absztrakt szintaxisfa. Miután a fa felépült, a programból tetszőleges módon bejárható és módosítható. A módszer hátránya, hogy a DOM-fa felépítése több időbe telik, mint az eseményvezérelt feldolgozás, valamint nagy dokumentum esetén jelentős memóriaterületet foglal. Előnye a már említett tetszőleges bejárás és módosíthatóság. Ennek köszönhetően a dokumentumban található kereszthivatkozások is könnyen kezelhetők.

A DOM szabvány általánossága nehézkessé teszi a használatát, és a DOM API nem illeszkedik jól a Java nyelv programozási konvencióihoz. A JDOM keretrendszer célkitűzése, hogy olyan API-t nyújtson XML-dokumentumok manipulálásához, amely a DOM elvein alapul, de jobban illeszkedik a Java nyelvhez. A JDOM például kihasználja a Java kollekcióit.

7.2. A JAXB technológia

A Java nyelv a SAX és DOM szabványon kívül magasabb szintű megoldást is kínál XML-dokumentumok kezeléséhez. Ennek neve: Java Architecture for XML Binding (JAXB). A JAXB a Java 6-os verziójától része az osztálykönyvtárnak, a régebbi verziókhoz külön tölthető le. A JAXB szabvány az XML Schema típusú sémák és Java-osztályok közti kötést teszi lehetővé. Az XML-világban a séma tölti be az adattípusnak a szerepét, és ennek a példányai az XML-dokumentumok. Ez analóg a Java-osztályok és Java-objektumok közti kapcsolattal. A JAXB alapelve, hogy a típusok szintjén, sémák és osztályok között hoz létre leképezést. Ennek segítségével lehetővé válik, hogy az összekötött séma és osztály példányait egymás közt transzformáljuk. Tehát a kötött osztály példányobjektumait sorosíthatjuk XML-formátumba, illetve a kötött sémának megfelelő dokumentumokat Java-objektumokba olvashatjuk be. A kötés kétirányú, azaz ha a séma áll rendelkezésre, akkor abból a kötés megadása után el is készíthetjük a Java-osztályokat. A Java-osztályokat is elkészíthetjük elsőként, és azokból sémát hozhatunk létre. 7.1. ábra mutatja a kötésben részt vevő elemek kapcsolatát.



7.1. ábra: A JAXB-keretrendszer működése

A JAXB tehát elfedi az XML-dokumentumok kezelésének részleteit, és a dokumentumokban tárolt információt objektumokban adja vissza. Ez kevesebb programozással jár, és a SAX és DOM API-k ismeretét sem igényli. A magas szintű feldolgozás miatt azonban a teljesítmény elmaradhat a hagyományos SAX- vagy DOM-alapú feldolgozásétól.

7.2.1. Osztályból séma

Osztályok sémára történő leképezése az osztályon elhelyezett annotációkkal történik. A gyökérelemként használt osztályt az `@XmlRootElement` annotációval kell ellátni. Az annotációnak a `name` és `namespace` paraméterekben megadható az XML-elem neve és névtere, amelyre az osztály példányait le akarjuk képezni. A JAXB által kezelt osztályoknak rendelkezniük kell alapértelmezett konstruktorral vagy paraméter nélküli statikus factorymetódussal, hogy a keretrendszer az adatok

beolvasása előtt példányosítani tudja azokat. Az opcionális `@XmlType` annotáció `factoryClass` és `factoryMethod` paramétereiben adható meg a factoryosztály és -metódus. Az annotáció `propOrder` paraméterében pedig felsorolható a tagok leképezésének sorrendje. Mivel a példányváltozók és a JavaBeans-konvenciók szerint használt getter és setter metódusok is leképezhetők XML-elemekre, ezért a továbbiakban röviden a tagok leképezéséről beszélünk. Felhívjuk rá a figyelmet, hogy a tag szó a tárgyalásban tehát a tagváltozókra és a metódusokra vonatkozik, nem az XML-elemeket határoló tagekre.

Az `@XmlRootElement` annotáció elhelyezése után már használható a JAXB-keretrendszer alapértelmezett leképezése. További konfiguráció csak akkor szükséges, ha ez nem megfelelő. Alapértelmezés szerint a JAXB az összes publikus tagot XML-elemekre képezi le. Először azt kell meghatároznunk, hogy mely tagokat akarjuk leképezni, utána megadhatjuk azt is, hogyan történjen a leképezés. Az `@XmlAccessorType` annotációval módosíthatjuk, hogy a JAXB az osztály mely tagjait képezze le alapértelmezésben. Az annotáció elhelyezhető csomagon vagy osztályon, és a szülőosztályoktól is öröklődik. Legmagasabb prioritással az osztály saját annotációja rendelkezik, majd az örökölt annotáció. Ha a szülőosztályok sem rendelkeznek ezzel az annotációval, akkor a csomagra megadott alapértelmezés léphet érvényre. A 7.1. táblázat ismerteti az annotációnak megadható leképezési stratégiákat.

7.1. táblázat: XML-leképezési stratégiák

Érték	Jelentés
`XmlAccessType.FIELD`	A nem tranziens példányváltozók képeződnek le, kivéve az `@XmlTransient` annotációval megjelöltek.
`XmlAccessType.NONE`	Csak az egyenként megjelölt tagok képeződnek le.
`XmlAccessType.PROPERTY`	A JavaBeans szabvány (lásd 3.14. alfejezet) szerinti getter-setter párok képeződnek le, kivéve az `@XmlTransient` annotációval megjelöltek.
`XmlAccessType.PUBLIC_MEMBER`	Az összes publikus getter-setter pár és a publikus példányváltozók képeződnek le, kivéve az `@XmlTransient` annotációval megjelöltek.

A leképezési stratégia által kiválasztott elemekből tehát az `@XmlTransient` annotációval zárhatunk ki egyes elemeket. Hozzávételhez vagy finomhangoláshoz a kiegészítő annotációt kell elhelyezni a megfelelő tagon. Ha a leképezési stratégia nem választotta ki az adott tagot, akkor az annotáció elhelyezésével az is bekerül a leképezett tagok közé. Ha a leképezési stratégia kiválasztotta ugyan a tagot, de az alapértelmezett leképezéstől el kell térni, akkor a tagon elhelyezett annotációval felülbírálható. Az `@XmlElement` annotáció a tagot XML-elemre képezi le. Paraméterben megadható a már ismert `name` és `namespace`, valamint a `defaultValue`, `nillable` és `required` paraméterekkel rendre megadhatunk alapértelmezett értéket, illetve beállítjuk, hogy az elem lehet-e `null`, és kötelező-e. Az `@XmlID` és az `@XmlIDREF` annotáció mind elemre, mind attribútumra leképezett tagon elhelyezhető, és azt XML ID-ként vagy ID-hivatkozásként képezi le.

Meg kell még említeni a tagok típusaként használt osztályok leképezését. Az alapértelmezett leképezés ezekre is vonatkozik rekurzívan, nincs szükség további beállítások elvégzésére. A hivatkozott osztály tagjai természetesen annotációkkal finomhangolhatók. Magán az osztályon az `@XmlType` annotációval végezhetünk finomhangolást, mint a tagok leképezésének sorrendje, vagy így adhatjuk meg a factorymetódust. Ha tömböket vagy kollekciókat használunk egy osztályban, akkor azok a hivatkozott típus példányainak egymás utáni sorosítását eredményezik. A tömbön vagy kollekción alkalmazható az `@XmlElementWrapper` annotáció, ez azt eredményezi, hogy a sorosított tagokat körülveszi egy csomagolóelem.

Enumerációk értékeit a keretrendszer alapértelmezésben az értékek konstansaival reprezentálja. Az `@XmlEnum` annotációval megadható az enumeráción, hogy milyen típusra képeződjön le. Az egyes értékeken az `@XmlEnumValue` annotációval adható meg, hogy milyen érték reprezentálja őket az XML-dokumentumban.

A leírtakat egy egyszerű telefonkönyv-alkalmazással szemléltetjük. A telefonkönyvet a `PhoneBook` osztály reprezentálja, ez fog a gyökérelemre leképeződni. A kódban látható, hogy a `phonebook` elemnevet választottuk az alapértelmezett `phoneBook` helyett. Az osztály leképezését a példányváltozók alapján végezzük. Az osztály egyetlen példányváltozója a bejegyzések listája. Ez az `@XmlElement` annotáció alapján `person` elemek sorozata lesz, az `@XmlElementWrapper` annotáció pedig azt eredményezi, hogy egy `entries` elembe kerülnek. A `name` paraméteren keresztül megadható lett volna más név is.

```
@XmlRootElement(name = "phonebook")
@XmlAccessorType(XmlAccessType.FIELD)
public class PhoneBook {
    @XmlElement(name = "person")
    @XmlElementWrapper
    private List<Person> entries;
    ...
}
```

A hivatkozott `Person` osztály annotációk nélkül is működik, de itt megfigyelhetünk némi finomhangolást. A leképezés szintén a példányváltozók alapján történik: a bejegyzés típusát kötelező attribútumra képezzük le, a címet pedig tranzienssé tesszük. Ezen kívül az `@XmlType` segítségével a tagok sorrendjét is megadjuk.

```
@XmlType(propOrder= {"number", "firstName", "lastName"})
@XmlAccessorType(XmlAccessType.FIELD)
public class Person {
    @XmlAttribute(required = true)
    private EntryType type;
    private String firstName;
    private String lastName;
```

```
    @XmlTransient
    private String address;
    private String number;
    ...
}
```

A bejegyzés típusaként használt enumerációt is testre szabjuk, hogy az egész értékeket használjon az XML-dokumentumban:

```
@XmlEnum(Integer.class)
public enum EntryType {
    @XmlEnumValue("1") PERSONAL,
    @XmlEnumValue("2") PROFESSIONAL;
}
```

Az annotációkkal megjelölt osztályok már használhatók sorosításhoz és beolvasáshoz (lásd 7.2.3. alfejezet). A dokumentumokat a keretrendszer az annotált osztályokból kikövetkeztetett séma szerint kezeli. Ugyan sémát eddig nem hoztunk létre, az a színfalak mögött létezik. Séma létrehozásához a **schemagen** program használható, ennek a `.java` fájlokat kell megadni:

```
C:\__work\javabook_ws\F7>schemagen.exe src\phonebook\domain\*.java

warning: The apt tool and its associated API are planned to be
removed in the next major JDK release.  These features have been
superseded by javac and the standardized annotation processing API,
javax.annotation.processing and javax.lang.model.  Users are
recommended to migrate to the annotation processing features of
javac; see the javac man page for more information.
Note: Writing C:\__work\javabook_ws\F7\schema1.xsd
Note: Writing C:\__work\javabook_ws\F7\schema2.xsd
```

A parancs lefutása után az aktuális könyvtárban megtalálhatók a `schema1.xsd` és `schema2.xsd` fájlok.

7.2.2. Sémából osztály

A sémából történő alapértelmezett leképezés sokszor önmagában is kielégítő eredményt ad. Ha mégsem felelne meg az alapértelmezett leképezés, akkor kétféleképpen is módosíthatjuk. Mindkét módszerrel XML-formátumban adjuk meg a leképezési információt. Az egyik módszer közvetlenül az XML-sémában adja meg a leképezést a JAXB névterébe tartozó XML-elemek beágyazásával. Ezzel a módszerrel könnyebben megadható a leképezés, de a séma és a leképezési információ keveredik, és ez nehezen kezelhetővé teszi a sémát, ráadásul a leképezés is a szerves részévé válik. A másik módszer külön kötési fájlban adja meg a leképezést. Ennek előnye, hogy a leképezési jelölés elkülönül a sémától, és akár többféle leképezés is készíthető hozzá külön fájlokban. Hátránya, hogy a leképezési információt XPath-kifejezésekkel kell a séma ele-

meihez rendelni, és ez az XML-technológiában kevésbé jártas fejlesztőknek nehézséget jelenthet.

A fejezetben az alábbi sémát kötjük Java-osztályokhoz. A séma egy gépkocsi-adatbázist ír le, ebben tulajdonosok és gépkocsik vannak felsorolva. A gépkocsik tulajdonosaira az azonosítójukon keresztül hivatkoznak.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<xs:schema version="1.0" xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xs:complexType name="person">
    <xs:sequence>
      <xs:element name="firstName" type="xs:string"/>
      <xs:element name="lastName" type="xs:string"/>
    </xs:sequence>
    <xs:attribute name="id" type="xs:ID" use="required"/>
  </xs:complexType>

  <xs:complexType name="car">
    <xs:sequence>
      <xs:element name="licensePlate" type="xs:string"/>
    </xs:sequence>
    <xs:attribute name="owner" type="xs:IDREF" use="required"/>
  </xs:complexType>

  <xs:complexType name="carDatabase">
    <xs:sequence>
      <xs:element name="person" type="person" minOccurs="0" ↺
maxOccurs="unbounded"/>
      <xs:element name="car" type="car" minOccurs="0" ↺
maxOccurs="unbounded"/>
    </xs:sequence>
  </xs:complexType>
</xs:schema>
```

Az alapértelmezett leképezés jól használható objektummodellt készít a sémából. Az egyetlen szépséghibája, hogy az ID-hivatkozás `Object` típusú. Természetesen a sémából nem derül ki, hogy mindig személyekre hivatkozunk, de a most nyilvánvalóan ez a célunk, és az XML-elemek elnevezései is ezt tükrözik. Ezért készítünk egy kötési fájlt, amellyel megadjuk, hogy a hivatkozás mindig `Person` típusú legyen. A kötési fájl is XML-formátumú, és a gyökéreleme a `<jaxb:bindings>`. Ennek attribútumként meg kell adni a verziószámot, valamint a kapcsolódó séma elérési útját. A gyökérelembe egy újabb `<jaxb:bindings>` elem kerül, ez a `node` attribútumban megadott XPath-kifejezés segítségével választja ki, hogy a séma mely elemére hivatkozik. Jelen esetben ez az `owner` attribútumot leíró elem. Ehhez adjuk meg a `Person` osztályt mint leképezésben használt típust. A kötési fájl alább látható.

```
<jaxb:bindings version="2.1"
  xmlns:jaxb="http://java.sun.com/xml/ns/jaxb"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  schemaLocation="car.xsd">

  <jaxb:bindings node="//xs:attribute[@name='owner']">
    <jaxb:property>
      <jaxb:baseType name="Person"/>
    </jaxb:property>
  </jaxb:bindings>
</jaxb:bindings>
```

A kötési fájl segítségével már létrehozhatjuk a kívánt Java-osztályokat. Ezt az **xjc** segédprogrammal tesszük meg, ennek a **-b** parancssori opciójával tudjuk megadni a kötési fájlt, valamint a **-p** opciójával a kimeneti csomag nevét. Meg kell továbbá adnunk a sémát is, amelyből az osztályokat elkészítjük:

```
xjc.exe -b binding.xjb -p cardb.domain car.xsd
parsing a schema...
compiling a schema...
cardb\domain\Car.java
cardb\domain\CarDatabase.java
cardb\domain\ObjectFactory.java
cardb\domain\Person.java
```

Látható, hogy egy `ObjectFactory` nevű objektum is létrejött. Ez factorymetódusokat tartalmaz a létrehozott osztályokhoz. A leképezés további testre szabási beállításait a könyv nem részletezi, azok leírása elérhető angolul a JAXB dokumentációjában.[1]

7.2.3. A sorosítás és a beolvasás

A fenti kettő közül bármelyik irányt választhatjuk a fejlesztéshez. Miután rendelkezésre állnak a sémához kötött Java-osztályok, már tudunk a JAXB segítségével XML-formátumból adatokat olvasni, illetve abba sorosítani. A telefonkönyv példáját használjuk a sorosítás és a beolvasás szemléltetésére. A JAXB API fő elérési pontja a `JAXBContext` osztály. Ebből példányt a `newInstance()` factorymetódussal hozunk létre, ennek meg kell adni a kötött osztályt vagy osztályokat, amelyekkel dolgozunk. A sorosítás a `Marshaller` osztállyal történik, ebből példányt a `JAXBContext` objektum `createMarshaller()` metódusával kapunk. A sorosítást a `Marshaller` objektum `marshal()` metódusával végezhetjük el, ennek első paramétere a sorosítandó osztály, második paramétere pedig a cél, ahova a sorosított XML-adat kerül. Ez többféle típus lehet, például `File` vagy `OutputStream`. A beolvasás hasonlóan történik, de az `Unmarshaller` osztályt és az `unmarshal()` metódust használjuk. Az utóbbi `Object` típust ad vissza, ezért az eredményt konvertálni kell. A beolvasás forrása szintén számos típusból kerülhet ki. A JAXB osztályainak több metódusa is `JAXBException` kivételt vált-

[1] https://jaxb.java.net/2.2.7/docs/index.html

hat ki, ezeket megfelelően kezelni kell. Az alábbi példa létrehoz néhány adatot, és ezekből elkészíti az XML formátumú telefonkönyvet, majd kiírja egy fájlba, valamint a szabványos kimenetre is. Ezután beolvassa a fájlból a kiírt adatokat, és összehasonlítja az eredetiekkel.

```
// Adatok inicializálása
Person p = new Person("Gábor", "Kövesdán", "Magyar Tudósok krt. 2. ↺
QB224\nBudapest 1117", "+3614632713");
Person p2 = new Person("Doktorandusz", "Dániel", "Magyar Tudósok ↺
krt. 2. QB224\nBudapest 1117", "+3614632713");
PhoneBook phoneBook = new PhoneBook();
phoneBook.addEntry(p);
phoneBook.addEntry(p2);

try {

    // Kell egy JAXBContext, amellyel létrehozzuk a Marshallert
    JAXBContext ctx = JAXBContext.newInstance(PhoneBook.class);
    Marshaller marshaller = ctx.createMarshaller();

    // Formázzuk a kiírt adatokat
    marshaller.setProperty(Marshaller.JAXB_FORMATTED_OUTPUT, true);

    // Kiírjuk a PhoneBook osztályt fájlba és képernyőre
    File f = new File("tmp.xml");
    marshaller.marshal(phoneBook, f);
    marshaller.marshal(phoneBook, System.out);

    // Visszaolvassuk fájlból, és megnézzük ugyanazt kapjuk-e
    Unmarshaller unmarshaller = ctx.createUnmarshaller();
    PhoneBook phoneBook2 = (PhoneBook) unmarshaller.unmarshal(f);
    if (phoneBook.equals(phoneBook2))
        System.out.println("Helyesen beolvastuk a kiírt adatot.");

} catch (JAXBException e) {
    e.printStackTrace();
}
```

NYOLCADIK FEJEZET
Az adatbázisok kezelése

Az előző fejezetek tárgyalták az állapotmentést, valamint az XML-fájlok használatát. Néha egyik megoldás sem kielégítő, és helyettük adatbázisszerverhez kell fordulni. Ilyen eset például, ha ki akarjuk használni az aggregált adatok hatékony kezelését, amelyet a relációsadatbázis-szerver nyújt, vagy az alkalmazást adatszinten akarjuk más szoftverekkel integrálni. A fejezet a *Java Database Connectivity (JDBC)* rövid ismertetése után bemutatja a *Java Persistence API (JPA)* technológiát. Ez a JDBC-re épül, de magas szintű *objektumrelációs leképezést (Object-Relational Mapping, ORM)* tesz lehetővé.

8.1. A JDBC technológia

A *Java Database Connectivity (JDBC)* a Java nyelv alacsony szintű relációsadatbázis-elérést megvalósító technológiája. A JDBC kifejlesztésekor más adatbázisok gyakorlatilag még nem léteztek. A technológia két interfészt biztosít: egy *szolgáltatói interfészt (Service Provider Interface, SPI),* ezen keresztül a különböző adatbázisszerverek saját protokolljukon keresztül érhetők el. Ennek megvalósításához a használt adatbázisszerver működésének alapos ismerete szükséges. Az SPI-t az adatbázisszerverek fejlesztői használják, hogy elkészítsék, és az alkalmazásfejlesztők rendelkezésére bocsássák a saját adatbázisuk elérését szolgáló komponenst. Ennek a komponensnek a neve JDBC-driver, és általában JAR-fájlként (lásd 16.2. alfejezet) tölthető le az adatbázisszerver honlapjáról. Az interfész implementálásával biztosíthatók tehát a gyártófüggő alacsonyszintű funkciók, amelyekre alapozva magasabb szintű funkcionalitás építhető. Az SPI-re alapozva a JDBC egy gyártófüggetlen interfészt is nyújt, amelyet az alkalmazásfejlesztők használhatnak az adatbázisműveletek hordozható megvalósításához. Ez az *alkalmazásfejlesztői interfész (Application Programming Interface, API).*

A JDBC SQL-lekérdezések végrehajtására, és az eredmény lekérdezésére használható. Készültek magasabb szintű keretrendszerek is, ezek gyorsabb fejlesztést és transzparensebb adatbáziskezelést tesznek lehetővé. Ezért a könyv a JDBC közvetlen használatát nem tárgyalja, csak a rá épülő JPA szabványt.

8.2. A JPA technológia

A program fejlesztése során a programozó a jól megszokott objektummodellel tud hatékonyan dolgozni, de az elterjedt adatbázisszerverek még nagyrészt relációs adatmodellt használnak. A relációs adatmodell tábláit és az azok közötti joinműveleteket azonban bonyolult programból kezelni. Ez indokolja, hogy a két adatmodell között leképezéssel kapcsolatot teremtsünk. A leképezésnek a két modell közötti paradigmaüt-

közést fel kell oldania. Ezt a mechanizmust objektumrelációs leképezésnek nevezzük. A Java Persistence API (JPA) is ilyen leképezést valósít meg. Fontos kiemelni, hogy a technológia célja a fejlesztés megkönnyítése, de nem pótolja a relációsadatbázis-szerverek ismeretét.

A JPA alapelve, hogy a perzisztencia támogatása ne nehezítse meg az alkalmazásfejlesztést. A perzisztenssé tett objektumok ezért *hagyományos Java-objektumok (Plain Old Java Object, POJO)*, nem igénylik semmilyen interfész megvalósítását, és csak néhány egyszerű követelménynek kell megfelelniük. Ez a unittesztelést is megkönnyíti (lásd 15.2. alfejezet). A leképezést XML-leíróval vagy a perzisztens osztályon és a tagjain elhelyezett annotációkkal adjuk meg. Az utóbbi megoldás elterjedtebb és egyszerűbb, ezért a fejezet csak ezt ismerteti. A szabvány megad egy alapértelmezett leképezést, és csak az ettől eltérő módon leképezni kívánt elemeket kell testre szabni. Ez jelentősen megkönnyíti a fejlesztést.

Akárcsak a JAXB technológia (lásd 7.2. alfejezet), a JPA is kétirányú kötést hoz létre. A leképezést ugyan mindig a Java-programban adjuk meg, de választhatunk, hogy meglévő adatbázishoz készítünk objektummodellt, vagy az objektummodell alapján hozzuk létre az adatbázissémát. Az első esetben a leképezést úgy kell megadni, hogy az objektummodell pontosan a meglévő adatbázissémára képeződjön le. Léteznek olyan eszközök, amelyek ebben segítenek, és az adatbázissémából annotált Java-osztályokat hoznak létre. A második esetben az annotált osztályokból a JPA-keretrendszer készíti el az adatbázissémát, és az alkalmazás indításakor a táblákat a megfelelő beállítás esetén létre is hozza.

A fejezet során egyszerű gépkocsi-adatbázist készítünk, amely tárolja a gépkocsik és tulajdonosaik adatait, valamint lehetővé teszi manipulálásukat és lekérdezésüket.

8.2.1. Az entitások leképezése

A perzisztenssé tett objektumokat *entitásoknak (entity)* nevezzük. Az entitások jellemzője, hogy egyértelmű *azonosítóval (identifier, ID)* rendelkeznek. Ezek az adatbázisban kulcsra képeződnek le. Célszerű az `equals()` és a `hashCode()` metódust újradefiniálni, hogy az azonosító szerint értelmezzék a szemantikai egyenlőséget. Az entitások másik jellemzője, hogy összetartozó elemi adatokat fognak össze. Például egy karakterlánc önmagában nem entitás, de a gépkocsi, amelynek márkája, típusa és rendszáma van, már igen. Az entitások az alkalmazás követelményleírásában azonosíthatók be jól, és általában megegyeznek a szakterületi modell doménosztályaival.

Az entitásosztályokban annotációkat használunk a leképezés megadására. Az annotációk és a JPA-keretrendszer osztályai a `javax.persistence` csomagban vannak. Az entitásosztályt az `@Entity` annotációval kell megjelölni. Az így megjelölt osztály táblára képeződik le, és ennek neve alapértelmezésben az osztály nevével egyezik. A tábla neve és egyéb jellemzői az opcionális `@Table` annotáció paramétereivel állíthatók be. Az entitásosztályokra az alábbi korlátozások vonatkoznak:

- Az entitásoknak rendelkezniük kell publikus vagy `protected` láthatóságú paraméter nélküli konstruktorral.

- Sem az osztály, sem a perzisztens példányváltozók vagy metódusok nem lehetnek `final` módosítóval jelölve.

Ezután a tábla oszlopait kell megadni, ezek az osztály példányváltozóinak vagy JavaBeans-tulajdonságainak felelnek meg. Ezek típusa lehet bármilyen primitív típus vagy azok csomagolóosztálya, karakter- vagy bájttömb, `BigInteger`, `BigDecimal`, `String`, `Date`, `Calendar`, enumeráció, bármilyen sorosítható osztály vagy a JDBC időt jelölő típusai. A leképezéshez két megközelítést választhatunk. Használhatjuk a tagváltozókat közvetlenül vagy a JavaBeans-konvenciónak (lásd 3.14. alfejezet) megfelelő getter-setter párokat.

Először a kulcsmezőként használt azonosítót kell kijelölni. Ez lehet primitív egész típusú és karakter vagy ezek csomagolóosztálya, `String`, `BigInteger` vagy `Date` típusú. Lehetséges a lebegőpontos azonosító használata is, de a lebegőpontos típusok kerekítése miatt ez nem ajánlott. Az azonosító megadása kijelöli azt is, hogy melyik leképezési megközelítést alkalmazzuk. Ha az `@Id` annotációt tagváltozóra helyezzük, akkor a JPA a példányváltozókat veszi alapul a leképezéshez. Ha az annotációt a változó getterére tesszük, akkor a JavaBeans-tulajdonságok szerint történik leképezést. A többi példányváltozót vagy gettert már nem kell annotálni, mert azok a példányváltozó, vagy a JavaBeans-tulajdonság nevének megfelelő oszlopra képeződnek le a Java-típusuk szerinti adatbázisbeli típusra. A leképezett példányváltozók nem lehetnek publikusak, a settereknek és a gettereknek pedig `public` vagy `protected` láthatóságúaknak kell lenniük.

Az egyszerűség kedvéért a továbbiakban példányváltozókat említünk a leképezés során, de természetesen helyettük mindig használhatók a JavaBeans-tulajdonságok is. Alapértelmezés szerint a példányváltozók neve adja az oszlop nevét. A név és az oszlop néhány egyéb tulajdonsága (csak olvasható, nullázható stb.) a `@Column` annotációval szabható testre. Az annotációt a példányváltozón adjuk meg. A `@Transient` annotáció tranziensé teszi a kapcsolódó példányváltozót, így nem képeződik le oszlopra.

Az időt és a dátumot reprezentáló típusokon meg kell adni a `@Temporal` annotációt is. Ennek paramétere határozza meg, hogy a dátumot, az időt vagy mindkettőt tárolják-e. A lehetséges értékek `TemporalType.DATE`, `TemporalType.TIME` és `TemporalType.TIMESTAMP`. Az enumerációk leképezése alapértelmezésben működik. Az adatbázisban az enumeráció numerikus oszlopra képeződik le, és a lehetséges értékeket a sorszám reprezentálja. Az `@Enumerated(EnumType.STRING)` annotáció megadásával lehet szöveges oszlopra váltani, amelyben a felsorolt értékek konstansai tárolódnak. A `@Lob` annotáció nagy adatmennyiséget hordozó példányváltozók megjelölésére szolgál. Tipikusan `byte[]` vagy `char[]` tömbökön használjuk. Közvetlen hatása nincs, de jelzi a JDBC-drivernek, hogy az oszlopban sok adatot kell tárolni.

A kulcsmezőben sokszor generálunk egyedi értékeket. Ezt a `@GeneratedValue` annotációval adhatjuk meg. Az annotációnak a `strategy` paraméterben megadható a létrehozáshoz használt stratégia. Az alapértelmezés `GenerationType.AUTO`, azaz a JPA-keretrendszer tetszőleges mechanizmust választhat. A `GenerationType.IDENTITY` értékkel az adatbázis támogatását használjuk az egyedi értékű oszlop kitöltésére. A `GenerationType.SEQUENCE` adatbázis-szekvenciát használ az értékek előállítására. Ez a két módszer csak akkor használható, ha az adatbázis támogatja ezeket a mechanizmusokat. A `GenerationType.TABLE` esetén a JPA-keretrendszer hozza létre az azonosítókat, és táblában tartja számon a már felhasznált értékeket.

Az alábbi kódrészlet bemutatja a példaalkalmazás gépkocsi entitását. Az entitásban a példányváltozók alapján végezzük a leképezést, és felülbíráljuk a rendszám alapértelmezett oszlopnevét. A gépkocsi utolsó frissítésének dátumát és idejét is eltároljuk.

```
@Entity
public class Car {
    @Id
    @Column(name = "id")
    private String licensePlate;
    private String brand;
    private String model;
    @ManyToMany(cascade = { CascadeType.ALL })
    @JoinTable(name = "CarPerson")
    private Set<Person> owners = new HashSet<>();
    @Temporal(TemporalType.TIMESTAMP)
    private Calendar updated = null;
    ...
}
```

Használható példányváltozókból és JavaBeans-tulajdonságokból álló hibrid leképezés is. Ekkor az osztályon az `@Access` annotációt kell megadni, ennek paramétere `AccessType.FIELD` vagy `AccessType.PROPERTY` lehet. Ez jelöli ki az elsődlegesen használt mechanizmust. Tételezzük fel, hogy alapértelmezésben a példányváltozók alapján végezzük a leképezést. Ekkor a getter és a setter alapján leképezni kívánt példányváltozókat a `@Transient` annotációval kell megjelölni, a getterükön pedig az `@Access(AccessType.PROPERTY)` annotációt kell megadni. Fordított esetben természetesen a `@Transient` annotáció a getterre kerül, és a példányváltozón lesz az `@Access` annotáció az ellenkező elérési móddal.

Az alábbi kódrészlet bemutatja a példaalkalmazás személy entitását. Ennél a JavaBeans-tulajdonságokon alapuló leképezést vesszük alapul, de a gépkocsik halmazát nem kívánjuk közvetlen módosíthatóvá tenni, ezért ahhoz nem fog setter tartozni. Ezért a hibrid megközelítést alkalmazzuk, és a gépkocsik halmazát a példányváltozón keresztül képezzük le. Megfigyelhető az is, hogy az egyedi azonosítókat az adatbázis támogatásával hozzuk létre.

```
@Entity
@Access(AccessType.PROPERTY)
public class Person {
    private long id;
    private String firstName;
    private String lastName;
    private PersonSex sex;
    @Access(AccessType.FIELD)
    @ManyToMany(mappedBy = "owners")
    private Set<Car> cars = new HashSet<>();
    ...

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    public long getId() {
```

```
        return id;
    }

    @Enumerated(EnumType.STRING)
    public PersonSex getSex() {
        return sex;
    }

    @Transient
    public Set<Car> getCars() {
        return Collections.unmodifiableSet(cars);
    }
    ...
}
```

8.2.2. Az entitáskapcsolatok leképezése

Ha egy entitásnak van olyan példányváltozója, amely nem az előző alfejezetben felsorolt típusú, akkor az általában másik entitásosztállyal való kapcsolatot jelöl. Ritkább esetben lehet beágyazott osztály is, ezt később tárgyaljuk.

Entitáskapcsolatokat relációs adatmodellben kulcsokra való hivatkozással tudunk tárolni. Például egy gépkocsihoz eltároljuk a tulajdonosai azonosítóit. Amikor kapcsolódó objektumokkal dolgozunk, akkor a kapcsolatokat ilyen hivatkozásokra kell leképeznünk. A kapcsolatok irányuk szerint lehetnek egy- vagy kétirányúak, kardinalitásuk szerint pedig egy-egy, egy-több, több-egy vagy több-több kapcsolatok. Minden esetben az egyik entitásosztály lesz a kapcsolat tulajdonosa, a másik az inverz oldal. Az alábbiakban bemutatjuk az egyes esetek megvalósítását.

Egy-egy kapcsolat esetén az egyik entitásosztály táblájában soronként tároljuk a kapcsolódó entitás megfelelő példányának azonosítóját. Az az entitás lesz a kapcsolat tulajdonosa, amelynek a táblájában van a hivatkozás. Egyirányú kapcsolat esetén csak a tulajdonos oldalán adjuk meg a `@OneToOne` annotációt a kapcsolatot megvalósító példányváltozón. Kétirányú kapcsolatnál az inverz oldal példányváltozóján is meg kell ezt adni, és a `mappedBy` paraméterében jelezzük, melyik példányváltozó tartozik a kapcsolathoz a tulajdonos oldalán. Ennek a paraméternek a hiányában a JPA-keretrendszer egy másik kapcsolat tulajdonos oldalát hozná létre.

Kétirányú egy-több kapcsolat esetén a több oldal a tulajdonos, ugyanis erről az oldalról nézve minden példányhoz csak egy példány tartozhat, így a hivatkozások egy oszlopban megadhatók. Az egy oldalon ez nem lenne lehetséges. A több oldalon a kapcsolatot megvalósító példányváltozó valamilyen kollekciótípus. Ezen a példányváltozón a `@ManyToOne` annotációt kell megadni. Az egy oldalon a `@OneToMany` annotáció szükséges, és ennek a `mappedBy` paraméterben meg kell adni a tulajdonos oldal megfelelő példányváltozóját. Ha az egy-több kapcsolat egyirányú, akkor a több oldalon nem tárolhatók a hivatkozások, a kapcsolat ugyanis onnan nem érhető el. Ebben az esetben csak a `@OneToMany` annotációt kell megadni a `mappedBy` paraméter nélkül. A kapcsolat tárolásához a JPA jointáblát alkalmaz. Ez olyan tábla, amelynek soraiban páronként szerepelnek a kapcsolódó entitáspéldányok azonosítói. A jointábla neve és oszlopnevei a `@JoinTable` annotációval szabhatók testre.

Kétirányú több-egy kapcsolat a kétirányú egy-több kapcsolat inverz oldala, ezért ebben az esetben a fent elmondottak alkalmazandók. Egyirányú több-egy kapcsolat esetén csak a `@ManyToOne` annotációt adjuk meg a több oldalon.

Több-több kapcsolat esetén jointábla alkalmazása szükséges. Tetszőleges oldal lehet a kapcsolat tulajdonosa. Ezen az oldalon `mappedBy` paraméter nélkül használjuk a `@ManyToMany` annotációt, és itt helyezhető el a `@JoinTable` annotáció is. Ha kétirányú a kapcsolat, akkor az inverz oldalon is meg kell adni a `@ManyToMany` annotációt a `mappedBy` paraméterrel.

8.2.3. A beágyazott osztályok

Az objektum típusú példányváltozók nem mindig entitások. Ilyenkor azt vagy tranziessé, vagy beágyazhatóvá kell tenni. A beágyazás azt jelenti, hogy a beágyazott osztály példányváltozói úgy képeződnek le, mint ha a tartalmazó osztályban lennének deklarálva. A beágyazott osztályt az `@Embeddable` annotációval kell ellátni, példányváltozóin pedig a már megismert annotációk alkalmazhatók a leképezés testre szabásához. A tartalmazó osztályban a tartalmazott osztályra hivatkozó példányváltozót meg kell jelölni az `@Embedded` annotációval. Ha a beágyazott osztály leképezése nem felel meg az igényeknek, akkor a példányváltozón az `@AttributeOverrides` annotáció alkalmazható a felülbírálására.

8.2.4. Az öröklés leképezése

A relációs adatmodell nem ismeri az öröklés fogalmát, ezért az osztályhierarchiák leképezése nem magától értetődő. A legegyszerűbb esetben csak specifikusabb osztályok példányosodnak, az ősosztály nem. Ilyenkor az ősosztály megjelölhető `@MappedSuperclass` annotációval. Ekkor az osztálytól örökölt példányváltozók az entitásosztályok részeként perzisztensek lesznek, de az ősosztályhoz nem tartozik saját tábla. Az ősosztály önmagában nem entitás, közvetlen példányai nem tárolhatók el az adatbázisban, és olyan lekérdezések sem fogalmazhatók meg, amelyek az ősosztály típusával adnak vissza eredményt. Az így megjelölt ősosztály a JPA számára az absztrakt Java-osztályok fogalmával tekinthető analógnak. Nincs előírva, hogy az ősosztály absztrakt legyen, de ez ajánlatos. Az ősosztály ugyanis nem menthető el az adatbázisban, és lekérdezések eredményeként sem kaphatunk belőle közvetlen példányt, ezért nincs értelme a példányosításának.

Ha az ősosztály maga is entitás, akkor bonyolultabb a helyzet. Ez az osztály is közvetlen példányosodhat, ezért a relációs adatmodellnek támogatnia kell a perzisztenciáját. Attól függően, hogy az ősosztály táblája és a leszármazott osztályok táblái hogyan viszonyulnak egymáshoz, három megközelítést alkalmazhatunk. Ezt az ősosztályon az `@Inheritance` annotáció `strategy` paraméterével lehet megadni. Az alapértelmezett érték `InheritanceType.SINGLE_TABLE`. Ez azt eredményezi, hogy az hierarchiában szereplő összes osztály egyetlen táblára képeződik le. Ebben az összes osztály példányváltozóihoz tartozik oszlop. Ha egy példány nem rendelkezik valamely példányváltozóval, akkor annak sorában az oszlop értéke `NULL`. A táblában szükséges egy kiegészítő oszlop is, amely jelzi, hogy a sorban szereplő adatok mely osztály közvetlen példányától származnak. Ezt *diszkriminátor oszlopnak (discriminator column)* nevezzük, és az oszlop neve az ősosztályon a `@DiscriminatorColumn` annotációval állítható. Az egyes leszármazott osztályokat az oszlopban alapértelmezésben az osztály-

név azonosítja. Ez a leszármazott osztályokon elhelyezett `@DiscriminatorValue` annotációval megváltoztatható. Az egy táblára történő leképezés előnye a hatékonyság, ugyanis az osztályhierarchia bármely osztályának példánya lekérdezhető joinművelet nélkül. Hátránya, hogy a tábla az osztályhierarchia összes példányváltozóját tartalmazza, ezért hatalmasra nőhet, illetve a nem létező példányváltozók helye `NULL` értékekkel van kitöltve.

Az `InheritanceType.JOINED` mechanizmus minden leszármazotthoz új táblát hoz létre, de abban csak azokat a példányváltozókat tárolja, amelyeket az osztály nem örököl. Az örökölt példányváltozók az ősosztály táblájában tárolódnak. Ezért entitáspéldány lekérdezésekor az entitásosztály és az ősök tábláit joinművelettel egyesíteni kell, hogy a példányt ki lehessen nyerni. A beszúrás és a módosítás is több tábla bevonásával történik. Ez nagy osztályhierarchiák esetén igen költséges lehet. A mechanizmus azonban jól tükrözi a polimorfizmust, és erősen normalizált, redundanciamentes adatbázissémát eredményez. Új osztály létrehozása is könnyű, mert csak annak tábláját kell létrehozni.

Az `Inheritance.TABLE_PER_CLASS` megközelítés minden osztályhoz külön táblát hoz létre, de abban az összes példányváltozót tárolja, az örökölteket is beleértve. Az ennek eredményeként létrejövő séma nagyon hatékony, ha a hierarchia alján lévő osztályokkal dolgozunk, a beszúrásuk, lekérdezésük és módosításuk ugyanis csak egyetlen táblát érint. Sokkal rosszabb a helyzet polimorf lekérdezések esetén, mert a példányok bármely leszármazottból származhatnak. Ráadásul a táblákat egyesíteni sem lehet, ezért több különálló `SELECT` utasítást kell megfogalmazni. Ennek megvalósítása bonyolult, ezért a szabvány ezt az öröklési stratégiát opcionálisként írja elő a gyártók számára.

A fenti módszerek mindegyike rendelkezik előnyökkel és hátrányokkal. Nincs univerzálisan jó megoldás, az öröklési módszert a konkrét probléma ismeretében kell megválasztani. Ha nincsenek polimorf lekérdezések, akkor érdemes a `@MappedSuperclass` használatát megfontolni. Ha elhanyagolható a polimorf lekérdezések száma, és a használt JPA-implementáció támogatja a `TABLE_PER_CLASS` stratégiát, akkor jó döntés lehet ennek használata. A fennmaradó esetekben figyelembe kell venni, hogy a teljesítmény vagy a normalizált adatbázisséma fontosabb-e. További nehézség, hogy mindkét lehetséges módszer hátrányai skálázódnak az osztályhierarchiák növekedésével.

8.2.5. Műveletek az entitásokkal

A perzisztens osztályok halmazát és az adatbázis-konfigurációt magában foglaló konkrét beállításhalmazt *perzisztenciaegységnek (persistence unit)* nevezzük. A perzisztenciaegységet az `EntityManagerFactory` osztály reprezentálja, ez a `Persistence` osztály statikus factorymetódusával példányosítható. A példaalkalmazásban megfigyelhető a perzisztenciaegység elérése.

```
EntityManagerFactory factory = Persistence.↺
createEntityManagerFactory("CarPU");
```

A CarPU elnevezés a perzisztenciaegységet azonosítja, ennek az adatait a konfigurációs fájlban adjuk meg. A perzisztenciaegység segítségével példányosítjuk az EntityManager osztályt, ennek a metódusaival az entitásosztályokon végezhetünk műveleteket.

```
EntityManager em = factory.createEntityManager();
```

A használat befejezése után mindkét objektumot le kell zárni a close() metódus meghívásával, hogy a lefoglalt erőforrások felszabaduljanak.

Az entitásokon végzett műveletek adatbázisműveletek, ezért azoknak az olvasási műveleteken kívül tranzakcióban kell futniuk. Java SE-alkalmazás esetén a tranzakciókezelést mindig a JDBC natív tranzakciói biztosítják. A tranzakciós határokat a programozónak kell kijelölnie. Az EntityManager osztály getTransaction() metódusa EntityTransaction típusú objektumot ad vissza, a tranzakció ezzel kezelhető. Tranzakció a begin() metódussal kezdhető, majd a commit() és a rollback() metódussal kommittálható, illetve görgethető vissza. A setRollBackOnly() előírja, hogy a tranzakció mindenképpen visszagörgetéssel végződjön, a getRollbackOnly() metódus pedig visszaadja, hogy ez be lett-e már állítva. Az isActive() metódussal ellenőrizhető, hogy van-e aktív tranzakció. Ha van aktív tranzakció, akkor annak határain belül tetszőleges műveleteket végezhetünk az entitásokon.

Az entitásokon az EntityManager metódusaival végezhetünk műveleteket. Az osztály CRUD (Create, Read, Update and Delete, azaz létrehozás, olvasás, frissítés és törlés) műveleteket támogat, és ezzel a *Data Access Object (DAO)* tervezési minta (lásd [6]) megvalósítását segíti elő. A DAO minta technológiafüggetlen interfészt kínál az entitásokon végezhető műveletek elvégzéséhez, ezért elfedi a konkrét adatbázistól függő részleteket.

Az EntityManager metódusainak megismerése előtt fontos megérteni a JPA működését. Az entitásobjektumok nem állnak folyamatosan kapcsolatban az adatbázissal. Példányosítás után *lecsatolt (detached)* állapotban vannak, majd mentéskor *csatolt (managed)* állapotba kerülnek. A lekérdezés során kapott példányok is csatolt állapotban vannak. Az EntityManager példánya tehát az entitásoknak egy időpillanatban csak egy halmazát kezeli, ebbe a csatolt objektumok tartoznak. Ezt a halmazt *perzisztenciakontextusnak (persistence context)* nevezzük. Bizonyos események hatására az objektumok kikerülnek a perzisztenciakontextusból, és lecsatolt állapotba kerülnek. Ez azért előnyös, mert így az adatbázissal való kapcsolat megszűnik, és a kapcsolat fenntartására használt erőforrások felszabadulnak. Ez Java SE környezetben a következő esetekben történik meg:

- Az EntityManager lezárásakor, amely a close() metódus hívásával történik, az összes csatolt objektum kikerül a perzisztenciakontextusból.
- Az EntityManager objektum clear() metódusának hívása lecsatolja az összes csatolt objektumot.
- Az EntityManager objektum detach() metódusa csak a megadott entitáspéldányt csatolja le.

- Tranzakció visszagörgetésekor a tranzakcióban részt vevő összes entitáspéldány lecsatolódik.

- Entitáspéldány sorosított formája is kikerül a perzisztenciakontextusból.

Lecsatolt objektumok szabadon változtathatók, de a változtatások nem kerülnek be az adatbázisba egészen addig, amíg az `EntityManager` osztály `merge()` metódusával nem csatoljuk az objektumot. A metódus hatására a lecsatolt objektumban végzett módosítások bekerülnek az adatbázisba, és az objektum újra csatolttá válik. A `refresh()` metódus is csatolttá teszi a lecsatolt objektumot, de nem az adatbázis tartalmát frissíti az objektum alapján, hanem az objektum állapotát írja felül az adatbázisban lévő adatokkal. Fontos megjegyezni, hogy a csatolt objektum ugyan kapcsolatban van az adatbázissal, de a rajta végzett módosítások egészen addig nem véglegesednek, amíg az aktív tranzakció nem kommittál. Mivel egy visszagörgetett tranzakció lecsatolttá teszi a résztvevő entitáspéldányokat, célszerű az alkalmazás írásakor feltételezni, hogy azok lecsatolt állapotban vannak.

Az `EntityManager` osztály entitások manipulációjához használható metódusait az alábbi lista foglalja össze.

`void clear()`
Lecsatolja az összes csatolt objektumot.

`boolean contains(Object entity)`
Akkor ad vissza igaz értéket, ha a megadott entitás csatolva van.

`boolean detach(Object entity)`
Lecsatolja a megadott entitást.

`<T> T find(Class<T> entityClass, Object primaryKey)`
Az entitásosztály azon példányát kérdezi le az adatbázisból, amely a megadott elsődleges kulccsal rendelkezik. Ha nem létezik ilyen példány, akkor `null` értéket ad vissza.

`void flush()`
Szinkronizálja a perzisztenciakontextust az adatbázissal. Az adatok a tranzakció kommittálásáig még nem véglegesednek.

`boolean isOpen()`
Akkor ad vissza igaz értéket, ha az `EntityManager` még nincs lezárva.

`<T> T merge(T entity)`
Az adatbázisba írja a lecsatolt példány állapotát, és csatolja a példányt.

`void persist(Object entity)`
Perzisztálja az entitást, és csatolttá teszi.

`void refresh(Object entity)`
Az adatbázis adataival frissíti az entitáspéldány állapotát, és csatolttá teszi.

`void remove(Object entity)`
Törli az entitáspéldányt az adatbázisból.

Az EntityManager-példányt általában a fenti említett DAO tervezési mintával használjuk, és nem tesszük elérhetővé az alkalmazás többi osztálya számára. A DAO osztály ráadásul az elemi műveletekből nagyobb szemcsézettségű (coarse-grained) alkalmazásspecifikus műveleteket is létrehozhat. Az alkalmazás egyszerűbbé válik, ha ezeket használjuk. A példaalkalmazás DAO osztályának egy részlete alább látható.

```
public class CarRepository implements AutoCloseable {
    EntityManagerFactory factory = Persistence.↺
createEntityManagerFactory("CarPU");
    EntityManager em = factory.createEntityManager();
    EntityTransaction tx = em.getTransaction();

    public void save(Object o) {
        tx.begin();
        em.persist(o);
        tx.commit();
    }

    public void merge(Object o) {
        tx.begin();
        em.merge(o);
        tx.commit();
    }

    public void remove(Object o) {
        tx.begin();
        em.remove(o);
        tx.commit();
    }

    public void addOwnerForCar(Car c, Person p) {
        tx.begin();
        c.addOwner(p);
        em.merge(c);
        tx.commit();
    }

    public void removeOwnerForCar(Car c, Person p) {
        tx.begin();
        c.removeOwner(p);
        em.merge(c);
        tx.commit();
    }
    ...
```

```
    @Override
    public void close() {
        em.close();
        factory.close();
    }
}
```

Az osztály megvalósítja az `AutoCloseable` interfészt, hogy a `try` utasításban erőforrásként használható legyen (lásd 3.12. alfejezet).

Az `EntityManager` által nyújtott elemi műveletek néha nem elegendők összetett funkcionalitás megvalósításához. A JPA saját lekérdezőnyelvet definiál, ezt *Java Persistence Query Language-nek (JP-QL)* nevezzük. A lekérdezőnyelv az SQL nyelvhez igen hasonló. A JP-QL nyelvet a könyv részletesen nem tárgyalja, az SQL nyelv ismeretében ugyanis könnyen megtanulható. A nyelv segítségével megfogalmazhatunk lekérdező, módosító, felvivő és törlő utasításokat, akárcsak az SQL nyelvben. Ezekre a műveletekre együttesen a lekérdezés szóval hivatkozunk, még ha nem adnak is vissza adatot. Lekérdezéseket az `EntityManager` osztály `createQuery()` metódusával hozunk létre. A metódus `Query` típusú objektumot ad vissza. Ennek metódusaival hajtjuk végre a lekérdezést, és ha a lekérdezés adatokat ad vissza, akkor ezekkel érjük el. Az `executeUpdate()` metódus olyan lekérdezést hajt végre, amely nem ad vissza adatot. A `getSingleResult()` egyetlen objektumot visszaadó lekérdezések esetén használható. A `getResultList()` metódus akkor hívandó, ha több visszaadott adat van. A példaprogramban az összes autó listáját az alábbi lekérdezéssel kapjuk meg.

```
public List<Car> getCars() {
    Query q = em.createQuery("SELECT c FROM Car c");
    return q.getResultList();
}
```

A lekérdezés egyes részeit gyakran paraméterben kapjuk meg. A lekérdezést nem biztonságos karakterláncok összefűzésével előállítani, a módszer ugyanis nem elég átlátható, ezért könnyű hibázni. Másrészt, a módszer *SQL injection* sebezhetőséghez vezet, azaz a felhasználó jól megválasztott bemenettel rosszindulatú lekérdezéseket állíthat elő. Ezért a JP-QL beépített támogatással rendelkezik a paraméterek használatához. Az alábbi példakód adott személy gépkocsijait kérdezi le. A lekérdezésben a személyt számozott paraméterrel adjuk meg.

```
public List<Car> getCars(Person p) {
    Query q = em.createQuery("SELECT c FROM Car c WHERE ?1 MEMBER ↺
OF c.owners");
    q.setParameter(1, p);
    return q.getResultList();
}
```

A paraméterek el is nevezhetők. Az alábbi kódrészlet a fentivel ekvivalens módon működik, de elnevezett paramétert használ.

```
public List<Car> getCars(Person p) {
    Query q = em.createQuery("SELECT c FROM Car c WHERE ?1 MEMBER ↺
OF c.owners");
    q.setParameter(1, p);
    return q.getResultList();
}
```

8.2.6. Az entitáskapcsolatok kaszkádosítása

Az entitáskapcsolatok fontos jellemzője a kaszkádosítás. A kaszkádosítás azt jelenti, hogy entitás mentése, módosítása, törlése stb. esetén a művelet az entitáskapcsolatokon végiggyűrűzik, és a kapcsolódó entitásokon is végbemegy. A kaszkádosítás kapcsolatkonként állítható be minden egyes műveletre. A kapcsolatot jelző annotációk cascade paraméterében tömbként adjuk meg az egyes műveleteket reprezenzáló konstansokat. A példaalkalmazásban a gépkocsik tulajdonosaira a `CascadeType.ALL` konstanst adtuk meg, ez minden művelet esetén kaszkádosít. Ezért ha új gépkocsihoz új tulajdonost is felveszünk, akkor elég csak a gépkocsit menteni, vele együtt tulajdonosa is az adatbázisba mentődik. A 8.1. táblázat összefoglalja a megadható értékeket.

8.1. táblázat: A kaszkádosításhoz megadható értékek

Érték	Jelentés
`CascadeType.ALL`	minden művelet kaszkádosítása
`CascadeType.DETACH`	lecsatolás kaszkádosítása
`CascadeType.MERGE`	mentés kaszkádosítása
`CascadeType.PERSIST`	mentés kaszkádosítása
`CascadeType.REFRESH`	frissítés kaszkádosítása
`CascadeType.REMOVE`	törlés kaszkádosítása

8.2.7. Az entitások életciklusának kezelése

Az entitáspéldányok életciklusa során több esemény is bekövetkezik: eltároljuk, lekérdezzük, frissítjük, esetleg töröljük őket. Ezekre az eseményekre az entitás eseménykezelő metódusokkal reagálhat. Az eseménykezelő metódusok `void` típusúak, és nincs paraméterük. Több módon konfigurálhatjuk őket, de a fejezet csak a legegyszerűbb módszert ismerteti. A metódusokat az entitásosztályban definiáljuk, láthatóságuk tetszőleges lehet. A metódusokon ezután az eseménynek megfelelő annotációt kell elhelyezni. A 8.2. táblázat ismerteti a használható annotációkat és jelentésüket.

8.2. táblázat: Az entitások eseménykezelő metódusain használható annotációk

Annotáció	Jelentés
`@PrePersist`	mentés előtt hajtódik végre
`@PostPersist`	mentés után (kommittálás vagy flush alatt) megy végbe
`@PostLoad`	az adatbázisból való kiolvasás után fut le
`@PreUpdate`	módosítás előtt hajtódik végre
`@PostUpdate`	módosítás után (kommittálás vagy flush alatt) fut le
`@PreRemove`	törlés előtt hajtódik végre
`@PostRemove`	törlés után (kommittálás vagy flush alatt) hajtódik végre

Ha az ősosztály is entitás, és van eseménykezelője, akkor ezeket a leszármazott entitások is öröklik. A végrehajtás az ősosztály eseménykezelőivel kezdődik. Az eseménykezelők öröklése kikapcsolható a leszármazott osztályon elhelyezett `@ExcludeSuperclassListeners` annotációval. A példaprogramban a gépkocsik utolsó frissítését úgy tároljuk, hogy a dátumot a mentés, illetve a frissítés előtt beállítjuk. Ezt a következő kódrészlet mutatja.

```
@Entity
public class Car {
    ...
    private Calendar updated = null;
    ...

    @PreUpdate
    @PrePersist
    private void update() {
        updated = Calendar.getInstance();
    }
}
```

8.2.8. A konfiguráció

A program futtatásához a classpathban kell lennie a JPA-keretrendszer valamely implementációjának, illetve a használt relációs adatbázis JDBC-driverének. Ezen kívül szükséges egy `persistence.xml` nevű XML formátumú konfigurációs fájl készítése, és ezt a JAR-csomag (lásd 16.2. alfejezet) `META-INF` könyvtárában kell elhelyezni.

A perzisztenciaegység foglalja magában a JPA-t megvalósító osztály, illetve a JDBC-driver osztályának nevét, valamint az adatbázis eléréséhez szükséges részleteket. Itt adjuk meg a tranzakciókezelés módját is, valamint hogy az alkalmazás indításakor a keretrendszer létrehozza-e a táblákat.

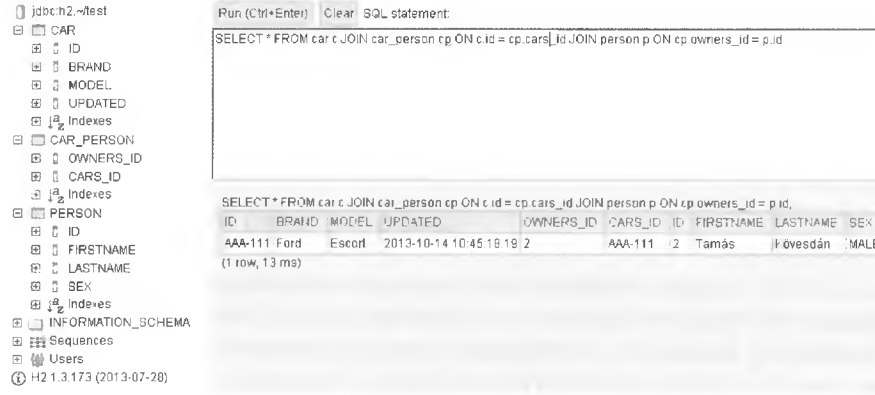
A példaprogramban a perzisztenciaegység neve `CarPU`, és mivel az alkalmazás Java SE környezetben fut, ezért a beállított tranzakciótípus `RESOURCE_LOCAL`. JPA-keretrendszerként az EclipseLink megvalósítást használjuk, ezért az osztálynevet ennek megfelelően adjuk meg. A perzisztenciaegység magában foglalja az összes annotált

entitásosztályt. A használt adabázis a H2 beágyazott adatbázis, ezért ennek JDBC-driverét is megadjuk, és az elérési adatok is a H2 megadási módját követik. Az alkalmazás induláskor eldobja és újra létrehozza a táblákat. Ez a fejlesztés alatt hasznos, természetesen éles környezetben nem alkalmazandó. Az alkalmazás XML-konfigurációja alább olvasható. Az Eclipse fejlesztőkörnyezet (lásd B függelék) segítségével a fájl űrlapok segítségével szerkeszthető, nincs szükség az egyes XML-elemek ismeretére. A példaalkalmazás a szükséges osztálykönyvtárakat a Maven keretrendszer segítségével kezeli. Az Eclipse fejlesztőkörnyezet használata esetén a függőségek automatikusan letöltődnek.

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.1"
 xmlns="http://xmlns.jcp.org/xml/ns/persistence" xmlns:xsi="http://
www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence http://
xmlns.jcp.org/xml/ns/persistence/persistence_2_1.xsd">
 <persistence-unit name="CarPU" transaction-type="RESOURCE_LOCAL">
  <provider>org.eclipse.persistence.jpa.PersistenceProvider</
provider>
  <exclude-unlisted-classes>false</exclude-unlisted-classes>
  <properties>
   <property name="eclipselink.ddl-generation" value="drop-and-↺
create-tables" />
   <property name="javax.persistence.jdbc.driver" value="org.h2.↺
Driver" />
   <property name="javax.persistence.jdbc.url" value="jdbc:h2:~/
test" />
   <property name="eclipselink.target-database"
    value="org.eclipse.persistence.platform.database.H2Platform" />
  </properties>
 </persistence-unit>
</persistence>
```

8.2.9. A példaalkalmazás futtatása

A példaalkalmazás a DAO osztály metódusait hívva mutatja be az adatbázis használatát. A `main()` metódusban példányosítjuk a DAO osztályt, majd adatokat viszünk fel. Ezután lekérdezéssel kiírjuk az adatbázis tartalmát, és módosításokat végzünk, majd újra kiírjuk a tartalmat. Ennek során megfigyelhető, hogy az adatok ténylegesen eltárolódtak. A program futása után a H2 adatbázis webes konzoljában is megtekinthető az alkalmazás által létrehozott adatbázis sémája és tartalma is. Ezt a 8.1. ábra szemlélteti.



8.1. ábra: Az adatbázis tartalma a program lefutása után

KILENCEDIK FEJEZET
A hálózati kommunikáció

Eddig hagyományos, önállóan működő alkalmazásokkal foglalkoztunk. A fejezetben megismerjük a hálózati kommunikáció megvalósítását Java nyelven. Először röviden említjük a TCP- és UDP-socketek kezelését, ezekkel tetszőleges protokollt implementálhatunk. A fejezet második része a *Remote Method Invocation (RMI)* szabványt mutatja be, ez a Java saját, távoli objektumelérést megvalósító protokollja. A protokoll segítségével transzparens módon kezelhetünk távoli objektumokat, tehát nem szükséges az alacsony szintű hálózati működéssel foglalkoznunk.

9.1. A socketek programozása

A socket hálózati adatátviteli végpontot jelent, amellyel adatokat küldhetünk és fogadhatunk. A legtöbb hálózati alkalmazás az *Internet Protocolt (IP)* használja, ezért a socketek az IP-címmel és porttal azonosíthatók. A socketek programozását támogató osztályok a `java.net` csomag részei. TCP-kliens-socketet a `Socket`, -szerver-socketet pedig a `ServerSocket` osztály példányosításával hozhatunk létre. A szerver-socket csak fogadja a kliensek kapcsolatát, majd minden elfogadott kapcsolathoz kliens-socketet hoz létre. A tényleges adatátvitel tehát mindig kliens-socketeken történik. Az adatátvitel történhet az IO és NIO API-kkal is (lásd 4. fejezet). A `Socket`-példánytól `InputStream`, `OutputStream` és `SocketChannel` objektumokat kaphatunk. Ezeken keresztül végezhetjük el az írást és olvasást.

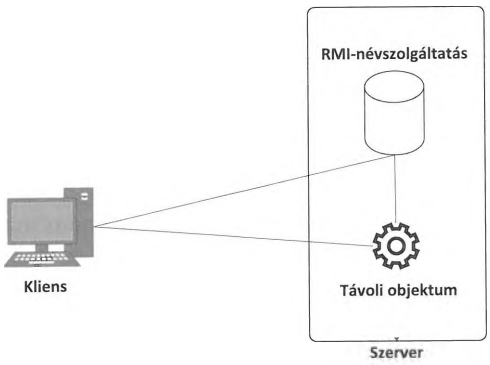
Az osztálykönyvtár UDP-socketek kezelését is támogatja a `DatagramSocket` osztállyal. Mivel az UDP kapcsolat nélküli protokoll, küldéshez és fogadáshoz az adatfolyamokkal dolgozó `OutputStream` és `InputStream` osztályok nem használhatók. Helyettük a `DatagramSocket` a `send()` és a `receive()` primitíveket nyújtja, ezek a `DatagramPacket` osztállyal reprezentált datagramcsomagokat képesek küldeni és fogadni. A NIO API-nak megfelelő `DatagramChannel` objektum azonban megkapható a `DatagramSocket` objektumtól.

A socketek programozásának előnye, hogy bármilyen protokollt használó hálózati alkalmazással kommunikálhatunk, még ha az nem is Java nyelven készült. Tetszőleges saját protokollt is készíthetünk. A teljes hálózati kommunikáció kézben tartható, ezért gyors, alacsony hálózati forgalmat eredményező protokollok készíthetők. A megoldás hátránya, hogy alacsony szinten programozzuk a hálózati kommunikációt, ezért a kifejlesztés bonyolult és időigényes. Ha Java-alkalmazások közt szeretnénk hálózati kommunikációt megvalósítani, akkor magasabb szintű eszköz is rendelkezésünkre áll, amely gyorsabb és hibamentesebb fejlesztést tesz lehetővé. Ezt mutatja be a következő alfejezet.

9.2. A távoli metódushívások

A Remote Method Invocation (RMI) protokoll távoli Java-objektumok transzparens elérését teszi lehetővé. Az inicializáció után a távoli objektumon ugyanúgy hívhatunk metódusokat, mint ha az a lokális Java virtuális gépben futna. A távoli objektumok elérése az RMI *névszolgáltatásán (registry)* keresztül történik. A szerveralkalmazás példányt hoz létre a *távoli objektumból (remote object)*, amelyet a kliens számára elérhetővé kell tenni, majd azt egyedi névvel regisztrálja a névszolgáltatásban. A kliens a név alapján tud referenciát szerezni az objektumra. Pontosabban szólva, a kliens nem magára a távoli objektumra szerez referenciát, hanem az ún. *csonkra (stub)*. A csonk proxyobjektumként [4] viselkedik, tehát elrejti a hálózati kommunikációt, és a metódushívásokat a távoli objektumhoz továbbítja.

A távoli objektumot a kliensnek nem kell ismernie, annak bájtkódját az RMI protokoll futásidőben képes letölteni a szerveralkalmazástól. Felmerülhet a kérdés, hogy a kliens honnan tudja, milyen metódusokat hívhat a távoli objektumon. A távoli objektumoknak egy *távoli interfészt (remote interface)* kell implementálniuk. A távoli interfésznek ki kell terjesztenie a `Remote` interfészt, valamint minden metódusának `RemoteException` jelzett kivételt kell deklarálnia a szignatúrában. A kliensben a távoli interfészt használjuk, ezen keresztül érjük el a távoli objektumot. A tényleges implementáció tetszőleges lehet, azt az RMI protokoll dinamikusan le tudja tölteni. A 9.1. ábra szemlélteti a protokoll működését.



9.1. ábra: Az RMI architektúrája

Távoli metódusok hívásakor paramétereket adhatunk át, illetve visszatérési értékként adatokat fogadhatunk. Ezeket az adatokat az RMI a hálózaton küldi és fogadja, tehát sorosítani kell őket (lásd 6.2. alfejezet). A távoli metódusok paraméterei és a visszatérési értéke tehát csak primitív típus vagy sorosítható objektum lehet.

Az RMI jól alkalmazható tehát saját alkalmazások közötti kommunikáció megvalósítására. Teljes távoliobjektum-protokollt ad a programozó kezébe, ez elfedi a bájtkód letöltését, a távoli metódushívást és a paraméterek átvitelének részleteit. Ha más mechanizmust használó programmal szeretnénk kommunikálni, akkor azonban közvetlen socketalapú megoldást kell kifejlesztenünk, vagy a saját protokollt megvalósító keretrendszert szükséges használni.

9.2.1. A szerveralkalmazás kifejlesztése

A szerveralkalmazás kifejlesztése során a következő lépéseket kell elvégeznünk:

1. A távoli interfész elkészítése. Ennek ki kell terjesztenie a `Remote` interfészt, valamint minden metódusának `RemoteException` kivételt kell deklarálnia.
2. A távoli interfész implementálása. Ennek az implementációs osztálynak a példánya fog távoli objektumként működni.
3. A távoli objektum példányosítása és a csonk létrehozása.
4. A csonk regisztrációja az RMI névszolgáltatásban, hogy a kliens a név alapján megtalálhassa.

A példában albérletkereső szolgáltatást valósítunk meg. A szolgáltatás kiadó lakások hirdetéseit fogadja és tárolja, valamint ár alapján képes keresni és ajánlatokat visszaadni. Kiadó lakás felvitelekor a hirdető kap egy azonosítót, ez alapján a hirdetés később törölhető. A lakások bejegyzései tartalmazzák a hirdető elérhetőségét, tehát a szolgáltatás a hirdetővel való kommunikációt nem támogatja, csupán tárolás és keresés a feladata. Először elkészítjük a `Flat` osztályt, ez a lakásokat reprezentálja. Ebben tároljuk el a lakás elhelyezkedését, alapterületét, szobáinak számát, havi bérleti díját, valamint egy szöveges megjegyzést, amelyben kiegészítő adatok adhatók meg. Az osztály megvalósítása a korábbi fejezetek ismeretében nem okoz kihívást, ezért itt azt nem ismertetjük. Megjegyezzük azonban, hogy a törléshez használt kulcs a szerveroldalon tárolandó, a keresést végrehajtó kliensek nem kaphatják meg. Ezért a kulcsot tranziens példányváltozóban tároljuk el. Ezután létre kell hoznunk a távoli interfészt, ez megvalósítja a beküldést, a keresést és a törlést:

```
public interface RentalAgencyService extends Remote {
    Set<Flat> search(double minPrice, double maxPrice) throws ↺
RemoteException;
    String add(Flat f) throws RemoteException;
    boolean remove(String key) throws RemoteException;
}
```

Az interfész implementációjában az osztályokat halmazban tároljuk, és a keresés során ezt a halmazt járjuk be a megfelelő hirdetések kikereséséhez. A távoli objektumot egyszerre több kliens is hívhatja, ezért a megfelelő szinkronizációról is gondoskodni kell. Ehhez a `synchronized` kulcsszót alkalmazzuk (lásd 11. fejezet). Az osztály implementációját az alábbi kódrészlet mutatja.

```
public class RentalAgencyServiceImpl implements RentalAgencyService {
    private Set<Flat> flats = new HashSet<>();
    private SecureRandom random = new SecureRandom();

    @Override
    public Set<Flat> search(double minPrice, double maxPrice) ↺
throws RemoteException {
```

```
        Set<Flat> resultSet = new HashSet<>();
        for (Flat f : flats) {
            if ((f.getRent() < maxPrice) && (f.getRent() > minPrice))
                resultSet.add(f);
        }
        return resultSet;
    }

    @Override
    public String add(Flat f) throws RemoteException {
        f.setKey(new BigInteger(130, random).toString(32));
        synchronized (flats) {
            flats.add(f);
        }
        return f.getKey();
    }

    @Override
    public boolean remove(String key) throws RemoteException {
        for (Flat f : flats) {
            if (f.getKey().equals(key)) {
                synchronized (flats) {
                    flats.remove(f);
                }
                return true;
            }
        }
        return false;
    }
}
```

Mielőtt a szolgáltatást elérhetővé tennénk, a tárban elhelyezünk néhány hirdetést, hogy a kliensalkalmazásból legyen mit tesztelnünk. Most már csak a példányosítás, a csonk létrehozása, valamint az RMI-névszolgáltatásban történő regisztráció van hátra. Ezt az alábbi rövid kódrészlettel tehetjük meg. A csonkot a `UnicastRemoteObject` osztály statikus `exportObject()` metódusával hozhatjuk létre. A visszaadott referencia `Remote` típusú, ezért konvertálni kell. A névszolgáltatást a `Registry` osztály reprezentálja, ebből példányt a `LocateRegistry` statikus factorymetódusaival hozhatunk létre. Lehetőség van futó névszolgáltatáshoz kapcsolódni, vagy programból indítani. A példa az utóbbi megoldást valósítja meg, a névszolgáltatást az `1099`-es szabványos porton indítva. A csonk felvétele a névszolgáltatásba a `Registry` osztály `bind()` vagy `rebind()` metódusaival lehetséges. Ha már van csonk rendelve a megadott névhez, akkor az utóbbi felülírja a régi bejegyzést, az előbbi viszont `AlreadyBoundException` kivételt vált ki.

```
public class Server {
    private static final String SERVICE_NAME = "RentalService";

    public static void main(String[] args) {
        RentalAgencyService service = new RentalAgencyServiceImpl();

        Flat f1 = new Flat("Budaörs", 54, 2, 55000,↺
 "info@kovacsistvan.hu");
        Flat f2 = new Flat("Dunakeszi", 67, 4, 82000,↺
 "alberlet@ingyenmail.hu");
        Flat f3 = new Flat("Angyalföld", 35, 1, 40000,↺
 "angyalfoldalbi@email.hu");

        try {
            service.add(f1);
            service.add(f2);
            service.add(f3);

            RentalAgencyService stub = (RentalAgencyService) ↺
UnicastRemoteObject.exportObject(service, 0);
            Registry registry = LocateRegistry.createRegistry(1099);
            registry.rebind(SERVICE_NAME, stub);
            System.out.println("A szolgáltatás elérhető.");
        } catch (RemoteException e) {
            e.printStackTrace();
        }
    }
}
```

9.2.2. A kliensalkalmazás kifejlesztése

A kliensalkalmazás elkészítéséhez szükséges a távoli interfész kódja, a távoli objektumot ugyanis ezen keresztül érhetjük majd el. A kliensben először is referenciát kell szereznünk a csonkra, majd ugyanúgy hívhatjuk a metódusait, mint ha lokális objektum lenne. A metódusok azonban `RemoteExceptiont` kivételt válthatnak ki, mivel a távoli interfész ezt írja elő. Ez a kivétel jelzi, hogy a távoli metódushívás során kommunikációs hiba lépett fel. Az alábbi példa bemutatja, hogyan használhatjuk a korábban elkészített szerveralkalmazást. A referenciaszerzéshez kapcsolódni kell a szerver névszolgáltatásához. Ehhez a már ismert `LocateRegistry` osztály `getRegistry()` statikus metódusát használjuk. A metódusnak meg kell adni a szerver címét vagy hosztnevét, és a portot is, ha a névszolgáltatás nem az alapértelmezett `1099`-es porton fut. A referenciaszerzés után feladunk egy hirdetést, majd keresést hajtunk végre. Végül töröljük a feladott hirdetést, és meg is bizonyosodunk arról, hogy törlődött.

```
public class Client {
    private static final String SERVICE_NAME = "RentalService";

    public static void main(String args[]) {
        try {
            Registry registry = LocateRegistry.getRegistry(args[0]);
            RentalAgencyService service = (RentalAgencyService) ↺
registry.lookup(SERVICE_NAME);
            Set<Flat> results = service.search(60_000, 90_000);
            System.out.println("60 000 és 90 000 Ft közötti bérleti ↺
díjjal rendelkező albérletek");
            for (Flat f : results)
                System.out.println(f.toString());
            Flat f = new Flat("Káposztásmegyer", 50, 2, 62_000,↺
 "sanyialberlet@email.hu");
            String key = service.add(f);
            results = service.search(60_000, 90_000);
            if (results.contains(f))
                System.out.println("Hirdetés sikeresen hozáadva.");
            service.remove(key);
            results = service.search(60_000, 90_000);
            if (!results.contains(f))
                System.out.println("Hirdetés sikeresen törölve.");
        } catch (RemoteException | NotBoundException e) {
            e.printStackTrace();
        }
    }
}
```

Az elosztott alkalmazások kifejlesztése összetett probléma. A fejezet nem ismerteti az RMI által nyújtott összes lehetőséget, sem az elosztott alkalmazások során felmerülő problémákat, mint például a skálázhatóságot és a holtpontok elkerülését. A témakörről [7] ad átfogó képet.

9.2.3. A példaalkalmazás futtatása

A szerver indításához a server.Server osztályt kell futtatni. Mivel a szerver maga hozza létre az RMI-névszolgáltatást, nem szükséges a szerver előtt elindítani. Ha a szervert úgy valósítottuk volna meg, hogy futó névszolgáltatáshoz csatlakozzon, akkor előbb az **rmiregistry** paranccsal el kellene indítani. A szerver indítása után a következő kimenetet kapjuk:

```
C:\__work\javabook_ws\F9\bin>java server.Server
A szolgáltatás elérhető.
```

Most már elindíthatjuk a klienst is. Meg kell adni parancssori paraméterben a szerver címét vagy hosztnevét. Jelen esetben ez **localhost:**

```
C:\__work\javabook_ws\F9\bin>java client.Client 127.0.0.1
60 000 és 90 000 Ft közötti bérleti díjjal rendelkező albérletek
Lakás 67m2 alapterülettel, 82000.0/hó
        4 szoba
        Régió: Dunakeszi
        Nincs leírás.
        alberlet@ingyenmail.hu
Hirdetés sikeresen hozzáadva.
Hirdetés sikeresen törölve.
```

A kimenetben láthatjuk, hogy a feltételeknek megfelelő hirdetést valóban kiírja a program, majd a hozzáadás és a törlés is sikeresen végrehajtódik.

TIZEDIK FEJEZET
A grafikus felhasználói felület

A korábbi fejezeteket elolvasva az olvasó mostanra már alapos ismeretekkel rendelkezik a Java nyelvű programozásról. A grafikus felhasználói felület kifejlesztéséről eddig azonban nem volt szó. Manapság a legtöbb alkalmazás grafikus felhasználói felülettel rendelkezik, ezek ugyanis növelik a felhasználók komfortérzetét, és a kevesebb számítógépes ismerettel rendelkezők számára különösen segítik az eligazodást. A fejezet a Swing keretrendszert mutatja be, ennek segítségével platformfüggetlen módon készíthetünk grafikus alkalmazásokat Java nyelven.

10.1. Az AWT és Swing keretrendszerek

A Java első grafikus keretrendszere az *Abstract Window Toolkit (AWT)*. Mivel a Java platform nagy hangsúlyt fektet a hordozhatóságra, az AWT keretrendszer programozói interfésze is független a konkrét platform grafikus alrendszerétől. A különböző operációs rendszerek grafikus alrendszerei azonban nagyon különböznek, és az AWT úgy biztosítja a hordozhatóságot a különböző platformok közt, hogy csak a mindenhol elérhető elemeket támogatja. Ennek következtében az AWT funkcionalitása igen korlátozott, és valós életbeli alkalmazások kifejlesztésekor igen sok megszorítást jelent. Ez szorgalmazta a *Swing keretrendszer* megjelenését.

A Swing keretrendszer a hordozhatóságot úgy valósítja meg, hogy nem az operációs rendszer grafikus alrendszerének komponenseit használja, hanem saját, rajzolt komponensekkel rendelkezik. Ezeket ezért *pehelysúlyú (lightweight) komponenseknek* nevezzük, nem kötődnek ugyanis hozzájuk operációsrendszer-szintű grafikus elemek. Az AWT komponenseire *nehézsúlyú (heavyweight) komponensekként* hivatkozunk. A Swing alapvetően az AWT-re épül, használja számos részét, például az eseménykezeléssel és a komponensek elrendezésével kapcsolatos részeit. A Swing ugyanakkor lecseréli az AWT összes komponensét. A Swing-komponensek osztályainak neve J betűvel kezdődik. Például az AWT-ben a gombokat a `Button` osztály segítségével használhatjuk, a Swing pedig a `JButton` osztályt biztosítja gombok létrehozására. Még ha a Java 7-es verziója támogatja is az AWT és a Swing komponenseinek a keverését, jobb ezt elkerülni. Használjuk mindig a J-vel kezdődő pehelysúlyú komponenseket.

10.2. Az MVC és a Model-Delegate

A grafikus felülettel rendelkező alkalmazások bonyolultak, ezért a *felelősségek szétválasztásának elve (separation of concerns)* szerint a grafikus elemeket különböző részekre bontják szét. Gyakran alkalmazzák a *Model-View-Controller (MVC)* architektúrát, ez a nevében szereplő három réteggel valósítja meg a grafikus felhasználói felületet. Az architektúra elemeire a könyvben a modell, a nézet és a kontroller kifeje-

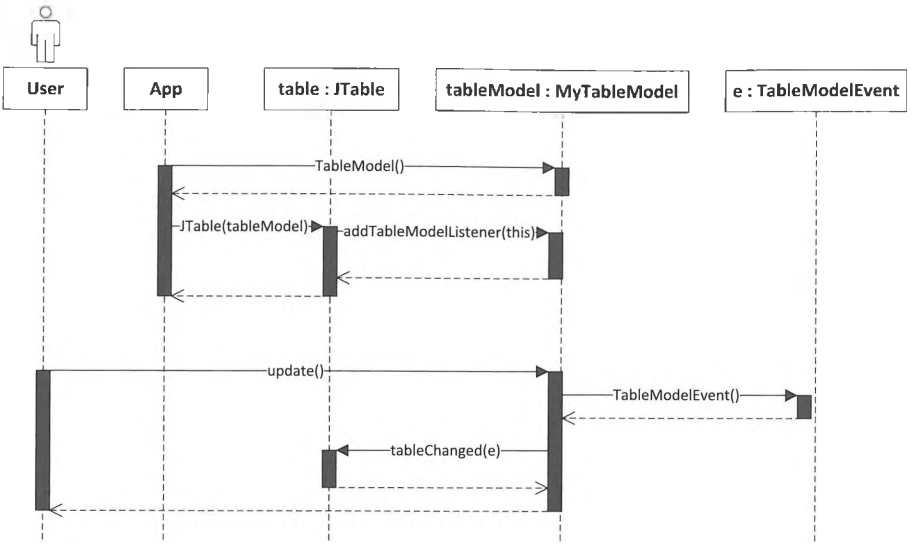
zésekkel hivatkozunk. A modell csak a grafikus elem által reprezentált, megjelenítéstől független információt tárolja. Például egy dokumentumszerkesztő komponens esetén ilyen a dokumentum szövege és formázása, de nem tartozik ide a szerkesztő kurzorának pozíciója. A nézet a megjelenésért felel, és feliratkozik a modell változásaira, ezekről események formájában kap értesítést (Observer tervezési minta, lásd [4]). Szintén a nézet tartja a kapcsolatot a felhasználóval, és a felhasználótól érkező bemenetet a kontrollernek adja át. A kontroller végzi a bemenet feldolgozását, és a bemenet alapján frissíti a modellt.

Az MVC architektúra egyszerűsítése a *Model-Delegate*. A Swing komponensei is ezt a mintát valósítják meg. A Model-Delegate mintában az MVC nézet és kontroller felelősségeit összevonjuk, és a delegate részben valósítjuk meg. Például a `JTable` komponens táblázatot valósít meg. A delegate maga a `JTable` osztály, de mindig kapcsolódik egy `TableModel` osztályhoz is, amely a modell szerepét tölti be. A modell metódusainak segítségével lehetséges a tábla celláinak módosítása. A módosítás történhet a tábla celláinak szerkesztésével vagy más módon is. A modell változása esetén esemény váltódik ki, a modell ezzel értesíti a regisztrált komponenseket a változásról. A delegate is regisztrálja magát az esemény figyelérésére, hogy a megjelenített adatokat frissíteni tudja.

A modellek által kiváltott események megfigyelőinek az eseményhez tartozó interfészt kell megvalósítaniuk. Ennek neve konvenció szerint a *Listener* szóval végződik. A `TableModel` osztály eseményeinek figyelésére például `TableModelListener` típusú objektumokat regisztrálhatunk. A modell az interfészek által előírt metódusokat hívja az események bekövetkeztekor. Ezek neve tükrözi az esemény jellegét, például a `TableModelListener` interfész egyetlen metódusának neve `tableChanged()`. A metódusok általában paraméterben megkapják egy eseményobjektumban a bekövetkezett esemény jellemzőit. Az eseményobjektum az esemény típusától függ, és konvenció szerint az *Event* szóval végződik. A `TableModelListener` interfész `tableChanged()` metódusa például `TableModelEvent` objektumban adja át az esemény jellemzőit. Az 10.1. ábra UML-szekvenciadiagramon mutatja be a példában használt események kezelését.

Az alkalmazás létrehozza a táblázatmodellt, majd a táblázat delegate-jét is, és átadja a modell referenciáját. A delegate megvalósítja a `TableModelListener` interfészt, ezért regisztrálhatja magát a modell változásaira. Később a felhasználó megváltoztatja a modellt. A modell a változás után létrehoz egy `TableModelEvent` objektumot, ez tartalmazza az esemény jellemzőit, majd meghívja a regisztrált megfigyelőket, és átadja nekik az eseményobjektumot. A delegate is regisztrálva van megfigyelőként, ezért értesítést kap a változásról, amelynek hatására frissíti a megjelenített táblázatot.

Eseménykezelést nemcsak a keretrendszer használ a delegate- és modellkomponensek között, hanem a programból is regisztrálhatók megfigyelők, amelyek az eseményekre reagálni tudnak. Eseményeket egyes delegate osztályok is kiváltanak, például az egérműveletek esetén minden komponens `MouseEvent` eseményt vált ki, ezt `MouseListener` megfigyelőkkel lehet kezelni.



10.1. ábra: A Swing eseménykezelése a JTable komponens használata során

10.3. A Helló, Swing! alkalmazás

A Swinggel történő ismerkedést a Helló, Világ! program Swing-megfelelőjével kezdjük. A program ablakában egyetlen címkét jelenít meg a „Helló, Swing!" szöveggel, valamint egy gombot kínál az ablak bezárására. A Swing ablakot megvalósító osztályának neve `JFrame`. Ezt az osztályt közvetlen is példányosíthatjuk, de a program ablakai lehetnek ennek leszármazottjai is. Ezzel a megközelítéssel minden egyes konkrét ablakot osztály zár egységbe, ezért a példában ezt követtük. Az osztályban a `JFrame` osztálytól örökölt metódusokkal tudunk dolgozni, ezekkel módosítjuk az ablak megjelenését, illetve helyezünk el komponenseket azon. A `setTitle()` metódus használható az ablak címének megadására. A `setMinimumSize()` állítja be az ablak minimális méretét, ez azért szükséges, hogy a komponensek elférjenek rajta. A `setDefaultCloseOperation()` metódusban konstansokkal adjuk meg, hogyan reagáljon az ablak a jobb felső sarkában található X jel megnyomására. Alapértelmezés szerint a gomb megnyomása csak elrejti az ablakot, de nem függeszti fel a program futását. A példában ezért ezt a működést átállítjuk. Alább látható a kapcsolódó kódrészlet.

```
// ablak címe
setTitle("Helló Swing!");
// minimális ablakméret
this.setMinimumSize(new Dimension(400, 300));
// jobb felső x-re bezárás
this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
// megjelenítés
this.setVisible(true);
```

Az ablakhoz komponenseket az `add()` metódussal adhatunk. A példában címkét hozunk létre a `JLabel` osztály példányosításával. Ennek konstruktorában megadható a címke szövege, ez később a `setText()` metódussal is változtatható. A címkében HTML-szövegrészlet is megadható, ha a szöveget a `<html>` karakterekkel kezdjük. Szintén megadhatunk grafikus ikont, sőt szöveget és ikont is egyszerre. A példában csak egyszerű szöveget adunk meg, de azt középre igazítjuk a `setHorizontalAlignment()` metódus hívásával. A metódusnak a `SwingConstants` osztály konstansaival adható meg az igazítás módja.

A másik komponens, amelyet az ablakhoz adunk, egy gomb. A gombot a `JButton` osztály valósítja meg, és a címkéhez hasonlóan képes HTML-szöveg és grafikus ikon akár egyidejű megjelenítésére. A gomb megnyomásakor a `JButton` osztály `ActionEvent` eseményt vált ki. Az esemény kezelésére `ActionListener` objektumok megadásával lehet feliratkozni. Ezeket a `JButton` osztály `addActionListener()` metódusával regisztrálhatjuk. A gombhoz tartozik egy karakterlánccal leírt akció. Alapértelmezésben ez megegyezik a gomb szövegével, de a `setActionCommand()` metódussal át is állítható. Eltérő akció megadása lokalizáció esetén szükséges, hogy az akció neve az aktuálisan beállított nyelvtől független maradjon. Az eseménykezelő metódusnak átadott `ActionEvent` objektum `getActionCommand()` metódusával érhető el az akció. Az akció vizsgálatával több gombhoz is használhatjuk ugyanazt az eseménykezelőt. A példában belső osztályként hozunk létre eseménykezelőt, és abban csupán bezárjuk a programot. Az alábbi kódrészlet mutatja a komponensek hozzáadását az ablakhoz.

```
// címke hozzáadása
JLabel label = new JLabel("Helló, Swing!");
label.setHorizontalAlignment(SwingConstants.CENTER);
this.add(label, BorderLayout.CENTER);

// gomb hozzáadása
JButton button = new JButton("Bezár");
// gomb eseménykezelője
button.addActionListener(new ExitActionListener();
this.add(button, BorderLayout.SOUTH);
```

A Swing keretrendszer szálkezeléssel kapcsolatos sajátosságai miatt fontos, hogy a felhasználói felületet `Runnable` interfészt megvalósító osztály `run()` metódusában hozzuk létre, majd az osztály példányát az `EventQueue` osztály statikus `invokeLater()` metódusával futassuk. Ennek bővebb magyarázata megtalálható a 11.8. alfejezetben. A példaprogram az alábbi `main()` metódussal indul.

```
public static void main(String[] args) {
    EventQueue.invokeLater(new HelloSwing());
}
```

A fejezet hátralevő részében lemezkatalógus-alkalmazást valósítunk meg, amely képes CD-, DVD- és Blu-Ray lemezeket katalogizálni. A katalógusok sorosítással (lásd 6.2. alfejezet) fájlba menthetők, és onnan beolvashatók. A katalógus bejegyzéseit táblázatos formában jelenítjük meg, és lehetőséget adunk új elem felvitelére, módosításra és törlésre.

10.4. A konténerek és az elrendezés

A Swing keretrendszer komponenseit két nagy csoportra oszthatjuk. Az első csoportba a konténerek tartoznak, ezek más komponenseket tartalmazhatnak. A második csoportot a normál komponensek képviselik, ezek a konténereken elhelyezhetők, de maguk nem funkcionálnak konténerként. A megkülönböztetés a két csoport között inkább logikai, mintsem szigorú. A három felső szintű konténer (`JFrame`, `JWindow` és `JDialog`) kivételével ugyanis az összes konténer és komponens a `JComponent` osztályból származik, utóbbi pedig az AWT `Container` osztályának leszármazottja. Ez definiálja a gyermek komponensek menedzseléséhez használható `add()`, `remove()` és `removeAll()` metódusokat. A Swing keretrendszerben tehát minden komponens implicit módon konténer is, legfeljebb figyelmen kívül hagyja a gyermek komponensek hozzáadására irányuló kísérletet.

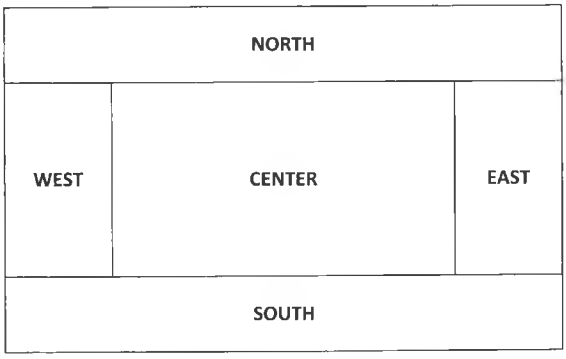
A konténerhez adott gyermek komponensek képpontokban mért méretét és helyét bonyolult egyenként megadni, ráadásul a konténer átméretezésekor ezeket újra kell számolni. A gyakorlatban ezért a felhasználói felület fejlesztésekor *elrendezésmenedzsereket (layout manager)* használunk a gyermekkomponensek kívánt elrendezésének kialakításához. Minden elrendezésmenedzser a `LayoutManager` interfészt valósítja meg, és jól meghatározott stratégia szerint rendezi el az elemeket a konténerben. Egyes elrendezésmenedzserek a megjelenített komponensekhez megszorításokat kapcsolhatnak. Ezek kiegészítő információt hordoznak arról, hogyan kell az adott elemet a konténerben elhelyezni. A 10.1. táblázat foglalja össze a leggyakrabban használt elrendezésmenedzsereket.

10.1. táblázat: Gyakran használt elrendezésmenedzserek

Elrendezésmenedzser	Leírás
`FlowLayout`	A komponenseket egymás mellé teszi, amíg azok egy sorban elférnek. Csak akkor kezd új sort, ha több komponens már nem fér el a sorban.
`BorderLayout`	Öt részre osztja a konténer területét, és a komponensek az öt rész egyikén helyezhetők el. A részt megszorítás segítségével választjuk ki.
`GridLayout`	A konténert táblázatosan osztja fel, és a következő komponens a soron következő cellába kerül.
`GridBagLayout`	Az előző általánosítása. A komponensek több cellát is kitölthetnek, és a cellaméreteknek nem kell egyenlőnek lenniük. Ezeket a kiegészítő információkat megszorítás segítségével adjuk meg.

A FlowLayout tehát sorban jeleníti meg a komponenseket, és csak akkor kezd új sort, ha a sorban már nem fér el több elem. A sorokat balról jobbra vagy jobbról balra is feltöltheti, ez a konténer tájolásától függ, ezt a setComponentOrientation() metódussal állíthatjuk be. Az értékek a ComponentOrientation osztály LEFT_TO_RIGHT , RIGHT_TO_LEFT és UNKNOWN konstansaival adhatók meg. Az utóbbi az alapértelmezés, és balról jobbra rendezi az elemeket. A FlowLayout elrendezésmenedzser jól használható például egymás mellett megjelenő gombok elrendezésére. Ez a JPanel konténer alapértelmezett elrendezésmenedzsere.

A BorderLayout a konténer által meghatározott téglalapot öt részre osztja. Az alsó és felső részek a téglalap két széléig terjedő sávokat határoznak meg. A középső sáv bal, középső és jobb részre van osztva. A felosztást a 10.2. ábra mutatja. A komponensek felvételekor a konténer add() metódusának a második paraméterben meg kell adni a kényszert, amely meghatározza, hogy a komponens a konténer melyik részére kerül. A kényszerek az osztály konstansaival adhatók meg. Egyes részekhez több konstans is tartozik. A példaprogramban a CENTER, NORTH, SOUTH, EAST és WEST konstansokat használjuk. Ezek az égtájakhoz való kapcsolódásuk miatt szemléletesek és könnyen megjegyezhetők. Ha nem adunk meg kényszert, akkor a komponens a CENTER részre fog kerülni. Minden egyes részen a legutóbbi hívással megadott komponens fog megjelenni, egy részre nem helyezhető el egyszerre több komponens. A konténerekhez adott komponensek azonban maguk is lehetnek konténerek, így ez a korlátozás kiküszöbölhető. Például a NORTH részen elhelyezhetünk menüsort, amely maga is tartalmaz további komponenseket, vagy a SOUTH részen JPanel segítségével több gombot foghatunk össze. Ezzel a mechanizmussal a BorderLayout igen jól használható az ablakok komponenseinek elrendezéséhez, ezért ez a felső szintű konténerek alapértelmezett elrendezésmenedzsere. A példaprogram főablaka is ezt használja. Felülre kerül a menüsor, középre pedig panelbe összefogott komponenseket helyezünk el.



10.2. ábra: A BorderLayout elrendezésmenedzser által használt felosztás

A GridLayout táblázatos formában rendezi el az elemeket. A táblázat sorainak és oszlopainak számát a konstruktorban lehet megadni. Az egyik érték nulla is lehet, ekkor ez a komponensek száma szerint változik. Például ha oszlopnak nullát, sornak egyet adunk meg, akkor a komponensek egy sorban és annyi oszlopban jelennek meg, ahány komponenst a konténerhez adunk. Az alapértelmezett konstruktor is egysoros elren-

dezést használó példányt készít. Az elrendezésmenedzser a táblázat celláit az első sortól kezdve sorban tölti fel a komponensekkel. A feltöltés iránya a konténer tájolásától függ. Az elrendezésmenedzser jól alkalmazható olyan komponenscsoportok elrendezéséhez, amelyek illeszkednek a táblázatos formához, például összetartozó címkék és szövegmezők esetén.

A `GridBagLayout` a táblázatos elrendezés általánosításával annál sokkal nagyobb rugalmasságot nyújt. A komponensek konténerhez adásakor kényszerrel kell megadnunk az elrendezési paramétereket. Az elrendezésmenedzsernek nem kell külön megadni az oszlopok és sorok számát, ezt a felvett komponensek kényszerei alapján határozza meg. Az elrendezésmenedzser konstruktora tehát nem kap paramétert, hanem a konténer `add()` metódusának hívásakor a második paraméterben a kényszereket a `GridBagConstraints` osztály példányával adjuk meg. Az osztályt létrehozhatjuk az alapértelmezett konstruktorral, ekkor az egyes megszorításokat a setterekkel be kell állítani. Az osztálynak van olyan konstruktora is, amely tizenegy paramétert vár. Ez a konstruktorhívás a paraméterek nagy száma miatt kevésbé áttekinthető, de használata jelentősen csökkenti a kódsorok számát, ugyanis nem szükséges további settereket hívni. A példaprogram ezt a megközelítést alkalmazza. Az alábbi lista áttekinti a kényszerben használható megszorításokat a konstruktornak történő megadás sorrendjében.

`int gridx`
: A kezdőcella oszlopának száma (nullától kezdve).

`int gridy`
: A kezdőcella sorának száma (nullától kezdve).

`int gridwidth`
: A vízszintesen lefedett cellák száma.

`int gridheight`
: A függőlegesen lefedett cellák száma.

`double weightx`
: A szélesség megállapításánál használt súly.

`double weighty`
: A magasság megállapításánál használt súly.

`int anchor`
: Ha a komponens kisebb, mint a tartalmazó terület, akkor ez a megszorítás határozza meg a komponens igazítását. Az igazítás az osztályban definiált konstansokkal adható meg. Az alapértelmezet érték `CENTER`, azaz középre igazítás.

`int fill`
: Ha a komponens által igényelt terület kisebb, mint a megjelenítési terület, akkor ez a megszorítás határozza meg, hogy az elrendezésmenedzser átméretezze-e a komponenst. Az átméretezés konstansokkal adható meg, az alapértelmezett érték `NONE`, azaz nincs átméretezés.

`Insets insets`
A külső kitöltést határozza meg, azaz a komponens négy oldala és a megjelenítési terület közti minimális távolságot. Alapértelmezett értéke `new Insets(0, 0, 0, 0)`.

`int ipadx`
A vízszintes belső kitöltést határozza meg, azaz hogy hány képpontot adjon hozzá az elrendezésmenedzser a komponens minimális szélességéhez. Alapértelmezett értéke `0`.

`int ipady`
A függőleges belső kitöltést határozza meg, azaz hogy hány képpontot adjon hozzá az elrendezésmenedzser a komponens minimális magasságához. Alapértelmezett értéke `0`.

A példaprogram főablakában a katalógusba felvitt lemezeket táblázat jeleníti meg. Az adatok manipulását a táblázat alatti gombokkal lehet elvégezni. A táblázatot és a gombokat panellel fogjuk össze, és a panel `GridBagLayout` elrendezésmenedzsert használ. Az alábbi kódrészlet szemlélteti az elemek elhelyezését a panelen.

```
panel.add(scrollPane, new GridBagConstraints(0, 0, 4, 1, 4, 80, ↺
GridBagConstraints.FIRST_LINE_START, GridBagConstraints.HORIZONTAL, ↺
new Insets(0, 0, 0, 0), 10, 10));
...
panel.add(newButton, new GridBagConstraints(1, 1, 1, 1, 1, 1, ↺
GridBagConstraints.FIRST_LINE_START, GridBagConstraints.HORIZONTAL, ↺
new Insets(0, 0, 0, 0), 10, 10));
panel.add(deleteButton, new GridBagConstraints(2, 1, 1, 1, 1, 1, ↺
GridBagConstraints.FIRST_LINE_START, GridBagConstraints.HORIZONTAL, ↺
new Insets(0, 0, 0, 0), 10, 10));
panel.add(detailsButton, new GridBagConstraints(3, 1, 1, 1, 1, 1, ↺
GridBagConstraints.FIRST_LINE_START, GridBagConstraints.HORIZONTAL, ↺
new Insets(0, 0, 0, 0), 10, 10));
```

10.5. A gyakran használt komponensek

Egyszerű adatok beviteléhez használjuk a `JTextField` komponenst, ez egysoros szöveget jelenít meg a komponens szélességében. A szélesség a karakterek számával adható meg a konstruktorban, egyébként az elrendezéstől függ. A szövegmezővel a komponens szélességénél hosszabb szöveget is be lehet vinni, ekkor a szövegmező csak a kurzor környezetét mutatja, ezért a teljes szöveg megtekintéséhez a kurzorral végig kell lépkedni rajta. Az osztály `getText()` és `setText()` metódusaival lehet a szövegmező szövegét elérni és módosítani. A szövegmező mellett balra általában címkét helyezünk el a már megismert `JLabel` komponenssel. A logikailag összetartozó szövegmezőket érdemes egységbe foglalni a `JPanel` komponens segítségével. A panel teljesen átlátszó, nem rendelkezik saját megjelenéssel, de saját elrendezésmenedzsere van. Ezért a panelen lévő elemeket a panel konténerétől eltérő módon is elhelyezhetjük. A panel sze-

géllyel is körülvehető. Szegély beállítására a `setBorder()` metódus szolgál. Szegélyeket a `BorderFactory` osztály statikus factorymetódusaival példányosíthatunk könnyen.

Numerikus adatok beviteléhez szövegmező helyett megfontolandó a spinner komponens használata, ezt a `JSpinner` osztállyal érhetünk el. A komponens által reprezentált érték valójában általános, `Object` típusként kapható meg, ugyanis bármilyen rendezett értékhalmazt támogat. Az értéket a `getValue()` és `setValue()` metódusokkal kérdezhetjük le és állíthatjuk be. A spinnerben kiválasztható értékeket a spinnerhez tartozó modell határozza meg, ennek típusa `SpinnerModel`. Számok kezeléséhez a `SpinnerNumberModel` konkrét osztályt használhatjuk. Ezt a `setModel()` metódussal állíthatjuk be. A `SpinnerNumberModel` osztály használata esetén az érték numerikus típusokra konvertálható. A komponensben a szerkesztőt is külön objektum valósítja meg. Ezt is le kell cserélni ahhoz, hogy a komponens csak numerikus értékek begépelését engedje meg. Ez a `setEditor()` metódussal tehető meg. A példaprogramban a lemezek felvitelét és szerkesztését megvalósító párbeszédablak szövegmezőket használ az adatok bekéréséhez. A lemez tartalmának hosszát percekben állíthatjuk be spinnerrel, amely csak számok bevitelét engedi. Az alábbi kódrészlet mutatja a komponensek elhelyezését.

```
public class EditDialog extends JDialog {
    private JTextField titleField = new JTextField();
    private JTextField authorField = new JTextField();
    private JSpinner minSpinner = new JSpinner();
    ...

    {
        setLayout(new GridBagLayout());

        JPanel dataPanel = new JPanel();
        dataPanel.setLayout(new GridLayout(0, 2));
        dataPanel.setBorder(BorderFactory.createTitledBorder↺
("Adatok:"));
        dataPanel.add(new JLabel("Cím:"));
        dataPanel.add(titleField);
        dataPanel.add(new JLabel("Szerző:"));
        dataPanel.add(authorField);
        dataPanel.add(new JLabel("Hossz (perc):"));
        minSpinner.setModel(new SpinnerNumberModel(0, 0, 1000, 1));
        minSpinner.setEditor(new JSpinner.NumberEditor(minSpinner));
        dataPanel.add(minSpinner);
        ...
        this.add(dataPanel, new GridBagConstraints(0, 0, 1, 1, ↺
8, 4, GridBagConstraints.FIRST_LINE_START, GridBagConstraints.↺
HORIZONTAL, new Insets(0, 0, 0, 0), 0, 0));
        ...
    }
    ...
}
```

A lemez típusa CD, DVD vagy BD lemez lehet. Ezt rádiógombokkal lehet kiválasztani. Rádiógombokat a `JRadioButton` osztállyal lehet létrehozni. A rádiógomb konstrukturában adható meg a gomb felirata, majd akciót kell beállítani a gombhoz a `setActionCommand()` metódussal. A rádiógombokat a megszokott módon kell hozzáadni valamely konténerhez, de azokat csoporttá is össze kell fogni, hogy közülük egyszerre csak egy legyen kiválasztható. Ezért a gombokat a `ButtonGroup` osztály egy példányához is hozzá kell adni. Az osztály gombokat csoportosít, és valamely rádiógomb megjelölése esetén leveszi a jelölést a csoport többi gombjáról. Az egy csoportba tartozó gombokat érdemes szegéllyel rendelkező panelbe foglalni, hogy az összetartozás a felhasználói felületen látható legyen. Az alábbi kódrészlet mutatja be a példaprogramban alkalmazott rádiógombokat.

```
public class EditDialog extends JDialog {
    ...
    private ButtonGroup buttonGroup = new ButtonGroup();
    private JRadioButton CDRadio = new JRadioButton("CD");
    private JRadioButton DVDRadio = new JRadioButton("DVD");
    private JRadioButton BDRadio = new JRadioButton("BD");

    {
        ...
        JPanel typePanel = new JPanel();
        typePanel.setLayout(new GridLayout(0, 1));
        typePanel.setBorder(BorderFactory.createTitledBorder↺
("Típus:"));
        CDRadio.setActionCommand("CD");
        buttonGroup.add(CDRadio);
        typePanel.add(CDRadio);
        DVDRadio.setActionCommand("DVD");
        buttonGroup.add(DVDRadio);
        typePanel.add(DVDRadio);
        BDRadio.setActionCommand("BD");
        buttonGroup.add(BDRadio);
        typePanel.add(BDRadio);

        JPanel descPanel = new JPanel();
        descPanel.setLayout(new CardLayout());
        descPanel.setBorder(BorderFactory.createTitledBorder↺
("Leírás:"));
        descPanel.add(descArea);
        ...
```

```
        this.add(typePanel, new GridBagConstraints(1, 0, 1, 1, ↺
2, 4, GridBagConstraints.FIRST_LINE_START, GridBagConstraints.↺
HORIZONTAL, new Insets(0, 0, 0, 0), 0, 0));
        ...
    }
    ...
}
```

A lemez leírása hosszú, többsoros szöveg is lehet, tehát a szövegmezők nem alkalmasak a bevitelére. A leírás beviteléhez ezért a `JTextArea` komponenst használjuk, ez többsoros szöveg bevitelét teszi lehetővé. A komponens konstruktorában megadjuk a sorainak a számát és a szélességét. A komponens szövegét lekérdezni, illetve beállítani a `getText()` és a `setText()` metódussal lehet. A példában a komponenst nem közvetlenül adjuk hozzá a konténerhez, hanem a `JScrollPane` osztályba csomagoljuk. Ez görgetősávot jelenít meg a komponensen, ha a szöveg nem fér el a rendelkezésre álló területen. A következő kódrészlet mutatja, hogyan használja a példaprogram a `JTextArea` komponenst.

```
public class EditDialog extends JDialog {
    ...
    private JTextArea descArea = new JTextArea(10, getWidth());
    ...
    {
        ...
        JPanel descPanel = new JPanel();
        descPanel.setLayout(new CardLayout());
        descPanel.setBorder(BorderFactory.createTitledBorder↺
("Leírás:"));
        descPanel.add(new JScrollPane(descArea));
        ...
        this.add(descPanel, new GridBagConstraints(0, 1, 2, 2, ↺
8, 8, GridBagConstraints.FIRST_LINE_START, GridBagConstraints.↺
HORIZONTAL, new Insets(0, 0, 0, 0), 0, 0));
        ...
    }
    ...
}
```

A példaprogram főablakában a katalógusba felvitt lemezeket látjuk táblázatos formában. A táblázatos megjelenést a `JTable` komponenssel valósítjuk meg. A komponens tetszőleges adatot képes táblázatos formában megjeleníteni, de meg kell adni a megjelenítés módját. Ezt a táblázatmodell elkészítésével tesszük meg. A táblázatmodell a `TableModel` interfészt implementálja. Az alapértelmezett `DefaultTableModel` implementáció `Vector` objektumokban tárolt `Vector` objektumokként tárolja el a cellák adatait. Ez elég általános, de a megjelenítendő adatot általában doménosztályok kollekciójaként tároljuk. A `DefaultTableModel` osztály ezért korlátozottan használha-

tó, és gyakran célravezetőbb saját táblázatmodell készítése. A példaprogram is ezt a gyakorlatot követi. Saját táblázatmodell készítéséhez jól használható az `AbstractTableModel` absztrakt osztály, ugyanis ez implementál néhány metódust, amely kiváltja a modell változását jelző eseményeket. Ezzel a modell jelezni tudja a delegate számára, hogy a táblázatot frissíteni kell. A táblázatmodellben olyan metódusokat kell implementálnunk, amelyek visszaadják, hány oszlop és sor van a táblázatban, mi az egyes oszlopok címe, milyen típusú adatot jelenítünk meg az oszlopban, illetve mi az adott cellában található érték. A példaprogram táblázatmodellje az oszlopneveket és oszlopszámot fixen tartalmazza, a sorok adatait pedig listából veszi. A táblázatmodell ezen kívül biztosít metódusokat a belső lista frissítéséhez. A táblázatmodell kódja alább látható.

```
public class CatalogTableModel extends AbstractTableModel {
    private static final long serialVersionUID = 1L;
    private List<CatalogEntry> list;
    private String[] colNames = { "Cím", "Előadó", "Típus", "Hossz ↺
(perc)", "Leírás" };

    public CatalogTableModel() {
        this.list = new ArrayList<>();
    }

    public CatalogTableModel(List<CatalogEntry> list) {
        this.list = list;
    }

    @Override
    public int getColumnCount() {
        return 5;
    }

    @Override
    public int getRowCount() {
        return list.size();
    }

    @Override
    public Object getValueAt(int entryNo, int propNo) {
        CatalogEntry e = list.get(entryNo);
        switch (propNo) {
        case 0:
            return e.getTitle();
        case 1:
            return e.getAuthor();
        case 2:
            return e.getType().toString();
        case 3:
```

```
            return Integer.toString(e.getLenghtInMins());
        case 4:
            return e.getDesc();
        }
        return null;
    }

    @Override
    public String getColumnName(int column) {
        return colNames[column];
    }

    @Override
    public Class<?> getColumnClass(int columnIndex) {
        return String.class;
    }

    public void remove(int idx) {
        list.remove(idx);
        fireTableRowsDeleted(idx, idx);
    }

    public CatalogEntry get(int idx) {
        return list.get(idx);
    }

    public void set(int idx, CatalogEntry entry) {
        list.set(idx, entry);
        fireTableRowsUpdated(idx, idx);
    }

    public void clear() {
        list.clear();
        fireTableDataChanged();
    }

    public void add(CatalogEntry entry) {
        list.add(entry);
        fireTableRowsInserted(list.size() - 2, list.size() - 2);
    }

    public void newCatalog(List<CatalogEntry> list) {
        this.list = list;
        fireTableDataChanged();
    }
```

```
    public List<CatalogEntry> getCatalog() {
        return list;
    }
}
```

A táblázatot ezután az alábbi kódrészlet által bemutatott módon hozzuk létre. Beállítjuk, hogy a táblázat kitöltse a konténer területét, és megadjuk, hogy egyszerre csak egy sort lehessen kiválasztani. Ezután a táblázathoz rendezőt adunk meg, hogy az oszlopok szerint a sorokat növekvő vagy csökkenő sorrendbe tudjuk rendezni. Végül görgetősávval is ellátjuk a táblázatot.

```
public class MainWindow implements Runnable {

    private JFrame mainWindow;
    CatalogTableModel tableModel = new CatalogTableModel();
    JTable table = new JTable(tableModel);
    ...

    @Override
    public void run() {
        ...
        JPanel panel = new JPanel(new GridBagLayout());
        panel.setMinimumSize(new Dimension(750, 550));
        table.setFillsViewportHeight(true);
        table.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);
        table.setRowSorter(new TableRowSorter<CatalogTableModel>(tableModel));
        JScrollPane scrollPane = new JScrollPane(table);
        panel.add(scrollPane, new GridBagConstraints(0, 0, 4, 1, 4, 80, GridBagConstraints.FIRST_LINE_START, GridBagConstraints.HORIZONTAL, new Insets(0, 0, 0, 0), 10, 10));
        ...
    }
}
```

A Swing keretrendszer rendelkezik még néhány kiemelt fontosságú komponenssel, amelyek nem fordulnak elő a példaprogramban. A 10.2. táblázat foglalja össze ezeket.

10.2. táblázat: További gyakori Swing-komponensek

Név	Leírás
`JPasswordField`	A szövegmező olyan változata, amelyben a begépelt karakterek csillag karakterekként látszanak, ezért jól használható jelszavak beviteléhez.
`JFormattedTextField`	Olyan szövegmező, amely objektumot képes formázottan megjeleníteni.
`JEditorPane` és `JTextPane`	Többsoros formázott szöveg megjelenítésére és szerkesztésére alkalmas komponensek.
`JCheckBox`	Jelölőnégyzetet valósít meg.
`JComboBox`	Legördülő listát valósít meg, ez tetszőleges szöveg bevitelét is lehetővé teszi.
`JList`	Hagyományos legördülő listát valósít meg.
`JColorChooser`	Szín kiválasztásához használható.
`JProgressBar`	Folyamatjelző sáv, tehát azt jelzi, hogy a folyamatban lévő művelet mekkora része van hátra.
`JSlider`	Minimális és maximális érték között adott lépésközzel állítható csúszka.
`JSplitPane`	Konténer kettéosztására használható úgy, hogy a felhasználó a felezővonalat mozgathatja, tehát a megjelenített komponensek területeinek aránya futásidőben változtatható.
`JTabbedPane`	Fülek létrehozására használható komponens.
`JToolBar`	Eszköztárat valósít meg, amely le is csatolható a tartalmazó ablakról. Ekkor az eszköztár külön ablakban jelenik meg.
`JTree`	Fastruktúrájú lista.

10.6. A párbeszédablakok megjelenítése

Párbeszédablakokat a `JDialog` osztály segítségével hozhatunk létre. Ennek az elrendezésmenedzsere is alapértelmezésben `BorderLayout`, akárcsak az ablakok esetén. A párbeszédablak is konténer, ezért a megszokott módon, az `add()` metódus használatával helyezhetjük el rajta az elemeket. A párbeszédablak lehet normál vagy modális. Utóbbi azt jelenti, hogy a főablak nem kaphatja vissza a fókuszt egészen addig, ameddig a párbeszédablakot be nem zárjuk. A példaalkalmazás is ilyen párbeszédablakokat használ új lemezek felviteléhez. Alapértelmezésben a párbeszédablak nem modális, a konstrukorban adható meg, hogy az legyen. Szintén a konstruktorban adjuk meg a párbeszédablak tulajdonosát. Modális párbeszédablak esetén a `setVisible(true)` hívás csak akkor tér vissza, ha a párbeszédablakot bezártuk, ezért eb-

ben az esetben nem szükséges eseménykezelőt írni az ablak bezárásának kezeléséhez. A példaprogramban ezért a párbeszédablaktól rögtön lekérhető a felhasználó által megadott adat. Ezt az alábbi kódrészlet szemlélteti.

```
case NEW_ENTRY:
    EditDialog newDialog = new EditDialog(mainWindow);
    newDialog.setVisible(true);
    CatalogEntry newEntry = newDialog.parseForm();
    tableModel.add(newEntry);
    break;
```

Nem modális párbeszédablakok esetén, vagy ha más típusú esemény kezelésének igénye lép fel, `WindowListener` típusú eseménykezelőt regisztrálhatunk, ennek metódusai pedig `WindowEvent` objektumban kapják meg az események részleteit.

A megnyitás és a mentés művelethez használt fájlválasztó párbeszédablakot gyakran használjuk, ezért a Swing kész komponenst kínál a megjelenítésére. Ezt a `JFileChooser` osztály valósítja meg. Az osztály példányának a `showOpenDialog()` és a `showSaveDialog()` metódusát hívva rendre a megnyitás és a mentés párbeszédablakait tudjuk megjeleníteni. A metódusok visszatérési értéke az osztályban definiált `CANCEL_OPTION`, `ACCEPT_OPTION` és `ERROR_OPTION` konstansok egyike, attól függően, hogy a felhasználó megszakította a fájlválasztást, kiválasztotta a fájlt, vagy hiba történt. Ha a felhasználó választott, akkor a `getSelectedFile()` metódussal érhető el a kiválasztott fájl. A komponensnek megadható, hogy fájlok, könyvtárak vagy mindkettő kiválasztását engedélyezze. Ha több kiválasztott érték van, akkor azok a `getSelectedFiles()` metódussal kaphatók meg. A példaprogram a katalógus mentéséhez a `JFileChooser` osztályt használja az alábbi kódrészletben látható módon.

```
case SAVE_DOCUMENT:
    JFileChooser saveChooser = new JFileChooser();
    int retSave = saveChooser.showSaveDialog(mainWindow);
    if (retSave == JFileChooser.APPROVE_OPTION) {
        String saveFile = saveChooser.getSelectedFile().↺
getAbsolutePath();
        try {
            CatalogUtil.saveCatalog(tableModel.getCatalog(), ↺
saveFile);
        } catch (IOException e1) {
            e1.printStackTrace();
        }
    }
    break;
```

10.7. Menüsor készítése

A menüsor elkészítésében több osztály is szerepet játszik. A `JMenuBar` osztály a teljes menüsort képviseli, ehhez adjuk hozzá az egyes menüket. A menüket a `JMenu` osztály példányai reprezentálják. Ebbe kerülhetnek menüpontok és elválasztó vonalak. A `JMenuBar` olyan speciális konténer, amely menüsorként jelenik meg. A menüsoron más komponens is elhelyezhető, de általában csak menüket helyezünk el rajta. A menüelemek az `add()` metódussal vehetők fel a menübe.

A `JMenu` osztály menüt valósít meg, amelyet a menüsoron helyezhetünk el. A menühöz címke tartozik, ez jelenik majd meg a menüsoron. A címkén kattintva a menü legördül, a tartalma megjelenik. A menübe elemeket az `add()` metódussal vehetünk fel. A menüelemek `JMenuItem` típusú objektumok, és a rajtuk való kattintás a gombokhoz hasonló módon, `ActionListener` eseménykezelő regisztrálásával történik. Az akció is a gombnál megismert módon állítható be. A menübe a menüelemek közé elválasztó vonalakat is felvehetünk az `addSeparator()` metódussal vagy egy `JSeparator` objektum hozzáadásával.

A példaalkalmazásban csak a főablak rendelkezik menüsorral. Ezen három menü van: Fájl, Beállítások és Segítség. A Fájl menüből végezhető új katalógus kezdése, katalógus megnyitása, valamint a programból való kilépés. A kilépés menüpontot elválasztó vonal különíti el a többitől. A Beállítások menüben végezhető a megjelenés beállítása, a Segítség menüben pedig a program névjegyét nézhetjük meg. A menüelemekhez közös `ActionListener` tartozik, ez az akció alapján végzi el a kívánt műveletet. A menü létrehozását a következő kódrészlet mutatja be.

```
ActionListener actionListener = new CatalogActionListener();

JMenu fileMenu = new JMenu("Fájl");
JMenuItem fileNewMenu = new JMenuItem("Új");
fileNewMenu.setActionCommand(ActionCommand.NEW_DOCUMENT.toString());
fileNewMenu.addActionListener(actionListener);
fileMenu.add(fileNewMenu);

JMenuItem fileOpenMenu = new JMenuItem("Megnyitás...");
fileOpenMenu.setActionCommand(ActionCommand.OPEN_DOCUMENT.toStringo
());
fileOpenMenu.addActionListener(actionListener);
fileMenu.add(fileOpenMenu);

JMenuItem fileSaveMenu = new JMenuItem("Mentés...");
fileSaveMenu.setActionCommand(ActionCommand.SAVE_DOCUMENT.toStringo
());
fileSaveMenu.addActionListener(actionListener);
fileMenu.add(fileSaveMenu);

fileMenu.addSeparator();

JMenuItem fileExitMenu = new JMenuItem("Kilépés");
```

```
fileExitMenu.setActionCommand(ActionCommand.EXIT.toString());
fileExitMenu.addActionListener(actionListener);
fileMenu.add(fileExitMenu);

JMenu optionsMenu = new JMenu("Beállítások");
JMenuItem optionsLFMenu = new JMenuItem("Megjelenés...");
optionsLFMenu.setActionCommand(ActionCommand.OPTIONS_LF.toString());
optionsLFMenu.addActionListener(actionListener);
optionsMenu.add(optionsLFMenu);

JMenu helpMenu = new JMenu("Súgó");
JMenuItem helpAboutMenu = new JMenuItem("Névjegy...");
helpAboutMenu.setActionCommand(ActionCommand.ABOUT.toString());
helpAboutMenu.addActionListener(actionListener);
helpMenu.add(helpAboutMenu);

JMenuBar menuBar = new JMenuBar();
menuBar.add(fileMenu);
menuBar.add(optionsMenu);
menuBar.add(helpMenu);
mainWindow.setJMenuBar(menuBar);
```

10.8. A megjelenés lecserélése

A Swing keretrendszer által nyújtott összes komponens pehelysúlyú, tehát rajzolt komponens. Ez lehetővé teszi, hogy a kinézetüket könnyen testre szabjuk. A Swing ezt a *lecserélhető megjelenések (pluggable look and feels)* segítségével támogatja. Az egyes megjelenéseket a `LookAndFeelInfo` egy-egy példánya írja le. A telepített megjelenések leíróinak tömbjét az `UIManager` osztály statikus `getInstalledLookAndFeels()` metódusával érhetjük el. Az aktuálisan beállított megjelenés leíróját a `getLookAndFeel()` metódussal kapjuk meg. A leíróból lekérdezhető a megjelenés neve, illetve a megjelenést megvalósító osztály, amely a `LookAndFeel` osztály példánya. A megjelenést az `UIManager` osztály statikus `setLookAndFeel()` metódusával állíthatjuk át, ennek vagy `LookAndFeel`-példányt vagy osztálynevet kell megadni. Ha a megjelenés beállítása előtt már megjelenítettünk grafikus komponenseket, akkor az összes felső szintű konténerre meg kell hívni a `SwingUtilities` osztály statikus `updateComponentTreeUI()` metódusát, különben a megjelenés nem frissül megfelelően.

A példaprogramban egy `LFUtil` segédosztályt alkalmazunk, ez a Properties API formátumának megfelelő (lásd 6.1. alfejezet) konfigurációs fájlban tárolja a beállított megjelenést. A megjelenés beállítását párbeszédablakban tesszük lehetővé, ebben listázzuk a telepített megjelenéseket. Alapértelmezésben az aktuális megjelenés van kiválasztva. Ha ezt megváltoztatjuk, akkor a beállítás elmentődik, és a következő induláskor betöltődik. A megjelenések listáját a következő kóddal töltjük fel.

```
LookAndFeelInfo[] plafs = UIManager.getInstalledLookAndFeels();
List<String> plafList = new ArrayList<>();
for (LookAndFeelInfo li : plafs) {
    plafList.add(li.getClassName());
}

final JList<String> list = new JList<>(plafList.toArray(new ↵
String[0]));
list.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);
list.setSelectedValue(LFUtil.getLF(), true);
```

TIZENEGYEDIK FEJEZET
Szálkezelés és időzítés

Gyakran szükséges, hogy különböző feladatok egymás mellett, párhuzamosan fussanak. A párhuzamosítás lehetővé teszi a jobb erőforrás-kihasználást, mert amíg egyes feladatok valamilyen eseményre várakoznak, addig mások futhatnak. Szintén a párhuzamosítás előnye, hogy a háttérben végrehajtódó műveletek nem bénítják meg a felhasználói felületet, ezért a program reagálni tud a felhasználói interakcióra. A párhuzamosítás ugyanakkor növeli a program komplexitását, és a versenyhelyzetből adódó problémák miatt a párhuzamosan futó szálak megfelelő szinkronizációjára is ügyelni kell. A fejezet bemutatja a Java szálkezelését és a szinkronizációs eszközeit. Mivel az időzítés is szálak segítségével történik, azt is ez a fejezet tárgyalja.

11.1. A Thread és a Runnable

Java nyelven a szálakat osztályokban valósítjuk meg, ezek példányaiból jönnek létre a ténylegesen futó szálak. Szálak megvalósításához implementálni kell a `Runnable` interfészt, amely egy `public void run()` szignatúrájú metódust ír elő. A metódus a szál belépési pontja, a szál létrejötte után a végrehajtás itt fog elkezdődni. A szálat azonban egy `Thread` objektum segítségével kell elindítani, a `run()` metódus egyszerű meghívása nem hoz létre új szálat. A `Thread` osztály példányosításakor a konstruktorban át kell adni a szálat megvalósító osztály egy példányát, illetve a második paraméterben megadható egy opcionális szálnév. Ezután a `Thread` objektum `start()` metódusával indítható a szál.

A `Thread` osztály maga is megvalósítja a `Runnable` interfészt. Szálat tehát ennek specializálásával is létre lehet hozni. Ekkor a leszármazott osztályt egyszerűen példányosítjuk, majd meghívjuk a `start()` metódusát. Szoftvertervezési szempontból szebb megoldás azonban a `Runnable` interfész használata, mert ekkor szétválasztjuk a szál által megvalósított logikát a futtatásától. Ebben az esetben tehát öröklés helyett delegálással valósítjuk meg a szálat.

A `start()` metódus hívásakor a szál megkezdi futását a főszállal párhuzamosan, és egészen addig él, amíg a `run()` metódus be nem fejeződik. Eközben a szál futása meg is szakadhat, ha az ütemező más szálaknak ad futási jogot. A Java virtuális gép specifikációja nem tesz megkötéseket az ütemező működésére, de a legtöbb implementáció prioritásos preemptív ütemezést alkalmaz. A szálak végrehajtási sorrendje tehát nem jósolható meg. A `Thread` osztály a futtatott szál prioritásának beállítására a `setPriority()` metódust kínálja. A maximális, minimális, illetve az alapértelmezett prioritást az osztály `MAX_PRIORITY`, `MIN_PRIORITY`, illetve `NORM_PRIORITY` konstansai tárolják. A prioritások használata hatékonyabbá teheti a program működését, ha a virtuális gép ütemezője prioritásos alapon működik. Tehát hacsak lehetséges, célszerű a prioritási szintek használata. Fontos azonban, hogy a prioritásra csak mint optimalizálási lehetőségre tekintsünk, és a program írása során ne hagyatkozzunk arra, hogy

a szálak valóban a prioritások szerint fognak lefutni. Ezt ugyanis a szabvány nem írja elő.

A fejezetben ismertetett példa banki tranzakciók feldolgozását szimulálja. Egy periodikusan futó taszk tranzakciókat hoz létre a rendszerben regisztrált bankszámlákhoz. A tranzakciók jóváírást vagy terhelést kezdeményezhetnek. Az összes kérés várakozási sorba kerül. A tranzakciókat egy másik szál dolgozza fel. A program a tranzakciók érkezését és feldolgozását is kiírja a szabványos kimenetre, ezáltal végigkövethető a számlatörténet. A tranzakciókat feldolgozó szálat az alábbi kóddal indítjuk.

```
Runnable consumer = new RequestConsumerTask(queue, accounts);
Thread t = new Thread(consumer);
t.start();
```

A szálak közt megkülönböztetünk felhasználói és démonszálakat. A program addig fut, ameddig vannak futó felhasználói szálak. Ha már csak démonszálak maradtak, akkor a Java virtuális gép befejezi futását. Alapértelmezésben a szálak felhasználói szálak, azokat a `setDaemon(true)` metódushívással tehetjük démonszállá.

11.2. A szálak állapotai

A szálak öt fő állapotban lehetnek. `NEW` állapotban azok a szálak vannak, amelyek létrejöttek, de futásukat még nem kezdték meg. Amikor a szálat elindítjuk, az futásra kész, azaz `RUNNABLE` állapotba kerül. A `RUNNABLE` állapotban lévő szálak közül az ütemező választja ki, melyik szál fusson. Ha lefoglalt erőforrást szükséges használnia, akkor a szál blokkolódik, `BLOCKED` állapotba kerül. A szál ebben az állapotban marad, és amíg az erőforrás fel nem szabadul, addig az ütemező nem ütemezheti. A blokkolt állapottól különböző a várakozó állapot, ebbe akkor kerül a szál, ha adott időre felfüggesztjük a futását vagy ha eseményre várakozik. Attól függően, hogy a várakozásnak van-e időkorlátja `WAITING`, vagy `TIMED_WAITING` állapotban lesz a szál. Az ötödik állapot a `TERMINATED` . Ebbe futásra kész állapotból kerülnek a szálak, miután `run()` metódusuk lefutott. A `Thread` osztály `stop()`, `suspend()` és `resume()` metódusai elavultak, ugyanis a szál futásának tetszőleges állapotban történő megszakítása veszélyes. A metódusok használata ezért nem célszerű. Helyette olyan saját megoldást kell kifejleszteni a szálak leállítására, amellyel biztonságosan le tudjuk állítani, és a program konzisztens futási állapotát nem veszélyezteti. Sokszor a szálban egyetlen ismétlődő feladat fut végtelen ciklusban. Ebben az esetben például használhatunk feltételellenőrzést a végtelen ciklus helyett. Az alábbi példa mutatja a megoldást.

```
public class MyThread implements Runnable {
    private boolean isStopped = false;

    public void stop() {
        isStopped = true;
    }
```

```
    public void run() {
        while (!isStopped) {
            ...
        }
    }
}
```

Az osztálykönyvtár a fentihez hasonló megoldást ad a *megszakításjelzéssel (interrupt flag)*. A megszakításjelzést az `interrupt()` metódussal állíthatjuk be a szálon. A `Thread` osztály statikus `interrupted()` metódusának hívásával vizsgálhatjuk, hogy az éppen futó szál meg lett-e szakítva. Ha igen, akkor elvégezhetjük a szálban a leállásához szükséges lépéseket. A metódus hívása törli a megszakításjelzést, ezért minden `interrupt()` hívás után csak az `interrupted()` legelső hívása ad vissza igaz értéket. A fenti példa megszakításjelzéssel az alábbi módon valósítható meg.

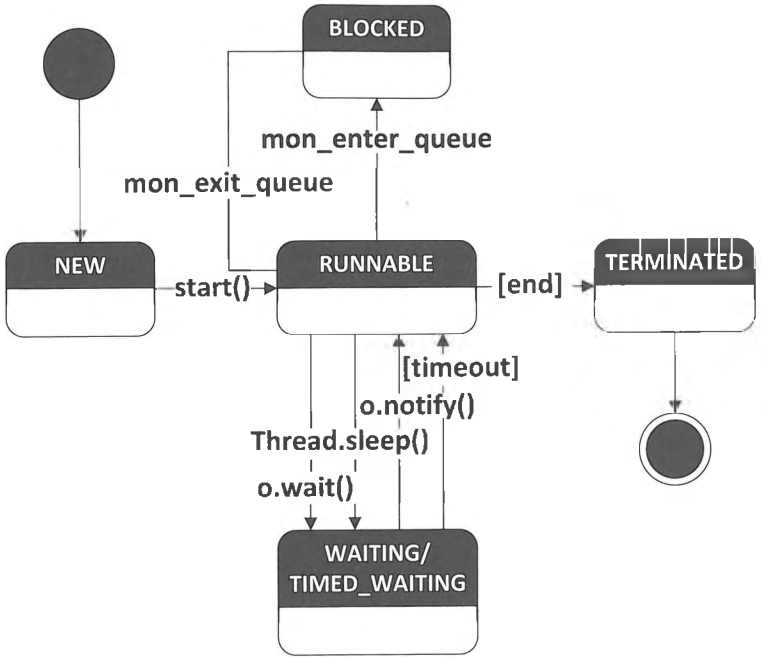
```
public class MyThread implements Runnable {
    public void run() {
        while (!Thread.interrupted()) {
            ...
        }
    }
}
```

A szálak állapotait és a köztük történő átmeneteket a 11.1. ábra mutatja be. A fejezet további részében bővebben is megismerjük az állapotátmeneteket.

11.3. Futásra kész és várakozó szálak

A szálak futása több ok miatt is megszakadhat. Az egyik ok, hogy az ütemező szakítja meg a szál futását, és egy másik szálnak adja a futás jogát. A `Thread` osztály statikus `yield()` metódusa jelzést küld az ütemezőnek, hogy a jelenlegi szál kész önként lemondani a futási jogáról, hogy ideiglenesen más szál futhasson. A metódus többnyire azt is eredményezi, hogy más szál kezd el futni. Az ütemezési algoritmus azonban nincs előírva a Java-szabványban, ezért nincs garancia arra, hogy a metódus hívása után az ütemező nem ugyanazt a szálat választja-e ki.

A szintén statikus `sleep()` metódus a jelenlegi szálat a megadott időre várakozó állapotba küldi (`TIMED_WAITING` állapot), így addig nem lehet ütemezni. A metódusnak `long` típusú paraméterben kell megadni, hogy a szálat hány milliszekundumig kívánjuk elaltatni. A metódusnak van kétparaméteres változata is, ez a második, `int` típusú paraméterben nanoszekundumok megadását is lehetővé teszi. A pontosság azonban az operációs rendszertől is függ, tehát tulajdonképpen bizonytalan: a megadott idő csak megközelítőleg fog teljesülni. Ezen kívül a `sleep()` metódus hívása csak azt garantálja, hogy a szál a megadott ideig nem fog futni, azt nem, hogy az idő letelte után az ütemező rögtön ütemezi. Tehát a futás nélkül eltöltött idő igen változó lehet.



11.1. ábra: A szálak állapotai és állapotátmenetei

Egy adott szál befejeződésére is várakozhatunk. Ehhez a szálat reprezentáló `Thread` objektum `join()` metódusát kell meghívni, ez addig nem tér vissza, ameddig a szál be nem fejeződött (`WAITING` állapot). A metódusnak opcionálisan megadható egy időlimit is milliszekundumokban (`TIMED_WAITING` állapot). Ha az időlimit letelt, akkor a metódus visszatér, még ha a szál futása nem fejeződött is be. A `sleep()` és a `join()` metódus esetén is előfordulhat, hogy várakozás közben a szálat egy másik szál megszakítja. Ezt a metódusok az `InterruptedException` kivétellel jelzik. A kivételt célszerű kezelni, nem továbbadni.

11.4. Az időzített feladatok

Az időzített feladatok kezelése is szálakkal történik. A `TimerTask` a `Runnable` interfészt implementáló absztrakt osztály. Ebből leszármazott osztályt létrehozva, a `run()` metódusában adjuk meg az időzítetve végrehajtandó kódot. Az osztály ezen kívül két újabb metódust is definiál:

`boolean cancel()`
: Felfüggeszti az időzített feladatot. Többször is hívható, a további hívásoknak nem lesz hatása. Akkor tér vissza `true` értékkel, ha a hívás nélkül a feladat még végrehajtódott volna néhányszor.

`long scheduledExecutionTime()`
A legutolsó megkezdett végrehajtás időzítés szerinti idejét adja vissza a referenciaidő óta eltelt milliszekundumokban (lásd 4.4. alfejezet). A valós végrehajtási idő ettől jelentősen eltérhet, az időzítő ugyanis nem garantálja a pontosságot.

Miután elkészült az időzítendő feladat implementációja a `TimerTask` osztály segítségével, a `Timer` osztállyal hozhatunk létre időzítőszálat, ez egy vagy több időzített feladatot futtat egymáss után, az ütemezésük szerint. Ezért ha a feladatok egymáshoz közeli időpontra vannak időzítve, vagy sokáig futnak, akkor az időzítés pontossága romlik. A `Timer` osztálynak négy konstruktora van:

`Timer()`
Időzítőszálat hoz létre alapértelmezett névvel, nem démonszálként.

`Timer(boolean isDaemon)`
Időzítőszálat hoz létre alapértelmezett névvel, démonszálként, ha a második paraméter értéke `true`.

`Timer(String name)`
Időzítőszálat hoz létre a megadott névvel, nem démonszálként.

`Timer(String name, boolean isDaemon)`
Időzítőszálat hoz létre a megadott névvel, démonszálként, ha a második paraméter értéke `true`.

A `Timer` osztály példányosításakor létrejön az időzítőszál, ez azonban egyelőre nem tartalmaz egyetlen időzített taszkot sem. A `TimerTask` osztály segítségével megvalósított taszkokat időzíthetjük egyszeri vagy periodikus végrehajtásra. Periodikus végrehajtás esetén kétféle megközelítés alkalmazható aszerint, hogy az időzítő hogyan kezelje a pontatlanságokat. Választhatunk, hogy a két végrehajtás között eltelt időt a legutolsó végrehajtás tényleges vagy ütemezett idejétől számítsa az időzítőszál. Előbbi esetben a taszk lefutásai nagyjából egyenletesek lesznek, utóbbi esetben pedig a késéseket igyekszik korrigálni, hogy azok minél inkább az eredeti ütemezés ideje szerint fussanak. Az alábbi lista ismerteti a `Timer` osztály metódusait:

`void schedule(TimerTask task, Date time)`
Egyszeri futásra időzíti a taszkot a megadott időpontban. Késés esetén a taszk azonnal futni fog.

`void schedule(TimerTask task, long delay)`
Egyszeri futásra időzíti a taszkot a milliszekundumokban megadott idő letelte után.

`void schedule(TimerTask task, Date firstTime, long period)`
Periodikus futásra időzíti a taszkot. A taszk először a második paraméterben megadott időpillanatban fog futni, majd a harmadik paraméterben milliszekundumokban megadott késleltetéssel fut újra. A metódus a késleltetést próbálja konstans értéken tartani, azaz a tényleges utolsó futástól számítja azt.

`void schedule(TimerTask task, long delay, long period)`
Periodikus futásra időzíti a taszkot. A taszk először a második paraméterben milliszekundumban megadott idő letelte után fut, majd a harmadik paraméterben megadott késleltetéssel fut újra. A metódus a késleltetést próbálja konstans értéken tartani, azaz a tényleges utolsó futástól számítja azt.

`void scheduleAtFixedRate(TimerTask task, Date firstTime, long period)`
Periodikus futásra időzíti a taszkot. A taszk először a második paraméterben megadott időpillanatban fog futni, majd a harmadik paraméterben milliszekundumokban megadott késleltetéssel fut újra. A metódus az ismételt futás késleltetését a legutolsó időzített futástól számítja, tehát megpróbálja az esetleges késéseket behozni.

`void scheduleAtFixedRate(TimerTask task, long delay, long period)`
Periodikus futásra időzíti a taszkot. A taszk először a második paraméterben milliszekundumban megadott idő letelte után fut, majd a harmadik paraméterben megadott késleltetéssel fut újra. A metódus az ismételt futás késleltetését a legutolsó időzített futástól számítja, tehát megpróbálja az esetleges késéseket behozni.

`void cancel()`
Befejezi az időzítő működését. Leállítja az összes időzített taszkot, és nem fogad el újakat. Az aktuálisan futó taszk még befejeződik, majd az időzítőszál kilép. Taszkok `run()` metódusából is hívható, ekkor az éppen futó taszk lesz az utolsó. A metódus többször is hívható, de a további hívásoknak nem lesz hatása.

`int purge()`
Törli a már leállított taszkok referenciáit, így azok alkalmassá válhatnak a szemétgyűjtésre, ha külső referencia sem hivatkozik rájuk. A törölt taszkok számát adja vissza.

A példában a banki tranzakciókat időzített taszk hozza létre. Az időzített taszk váza az alábbi kódrészletben olvasható.

```
public class PeriodicRequestProducerTask extends TimerTask {
    private Queue<TransactionRequest> queue;
    private Map<Long, BankAccount> accounts;
    private Random rnd = new Random(new Date().getTime());

    public PeriodicRequestProducerTask(Queue<TransactionRequest> ↺
queue, Map<Long, BankAccount> accounts) {
        this.queue = queue;
        this.accounts = accounts;
    }

    @Override
    public void run() {
        ...
    }
}
```

11.5. A szinkronizáció

Ha többszálú környezetben dolgozunk, számolni kell vele, hogy a szál futása bármelyik pillanatban megszakadhat. Mire a szál újra futási időt kap, a program állapotát más szál már megváltoztathatta. Ezért a több szálban is használt változók és objektumok elérését koordinálni, szinkronizálni kell, hogy egyszerre csak egy szál férhessen hozzájuk, és a művelet befejezéséig más szál ne módosíthassa őket. Ha ezt nem tesszük meg, akkor a programban a párhuzamos végrehajtás eredményeként inkonzisztens állapotok alakulhatnak ki. Képzeljünk el egyet a példában kezelt bankszámlák közül, amelyről pénzt akarunk kivenni. Megadjuk a felvenni kívánt mennyiséget, majd a program ellenőrzi, hogy rendelkezésünkre áll az egyenleg. Eközben egy másik, nagyobb prioritású folyamat ugyanezeket a lépéseket végzi el, és sikeresen levesz a számláról egy adott összeget. Ekkor a vezérlés visszatér az eredeti programhoz. Az eredeti program a pénzfelvétel előtt kérdezte le az egyenleget, így azt hiszi több pénz van a számlán, mint a tényleges összeg. Így tehát több pénzt is ki lehetne venni a számláról, mint amennyi valójában rendelkezésre áll. Feltételezhetjük, hogy a program a pénzfelvétel végén elmenti az új egyenleget, de ez is a rosszul kiszámolt egyenleg lesz, tehát a két felvétel során kevesebb pénz tűnik el a számláról, mint amennyit valójában felvettünk. A szálak ehhez hasonló problémákat okozhatnak, ha ugyanazokon az adatokon dolgoznak. A fenti példához hasonlóan ez néha jelenthet biztonsági kockázatot is, de a program helyes működését és stabilitását mindenképpen veszélyezteti. Fontos tehát a szinkronizáció alapos átgondolása és helyes megvalósítása.

A Java nyelv objektumszinten támogatja, hogy kritikus kódrészletek közül egyszerre csak egy fusson. Minden objektumpéldány rendelkezik egy *monitorral*, ez a zárakhoz hasonló szinkronizációs primitív. Egy monitort egyszerre csak egy szál szerezhet meg. Ha a monitor már foglalt, akkor az arra igényt tartó szál blokkolódik (`BLOCKED` állapot), amíg a monitor fel nem szabadul. Ha több szál is blokkolódott ugyanazon a monitoron, nem biztos, hogy a monitort legkorábban igénylő szál kapja meg először a hozzáférést. A monitorok tehát alkalmazhatók a szinkronizációra úgy, hogy a megosztott adathoz hozzáférő kódrészletek előtt lefoglalunk egy monitort, majd a kritikus műveletek elvégzése után visszaadjuk. Ez az monitor praktikusan a több szálból használt objektum monitora, primitív példányváltozó esetén pedig a tartalmazó osztályé. Ha minden szál, amelyik hozzáfér az objektumhoz, betartja ezt a konvenciót, akkor nem léphetnek fel a fentihez hasonló esetek. A monitort a `synchronized`-blokk segítségével tudjuk lefoglalni, ennek zárójelben adjuk meg az objektumot, amelynek a monitorát le kívánjuk foglalni. A példában előállított banki tranzakciókat várakozási sorban tároljuk. Amíg a sorba írunk, vagy abból olvasunk, a sor monitorát használjuk szinkronizációhoz. A sorba történő írást az alábbi kódrészlet mutatja.

```
// elkészül a feldolgozási kérés
TransactionRequest req = new TransactionRequest(accNo, sum);

// betesszük a sorba, és felébresztjük a feldolgozót
synchronized (queue) {
    queue.offer(req);
    System.out.println("Tranzakció érkezett a(z) " + accNo
        + " számlához " + sum + " összeggel.");
    queue.notify();
}
```

Vannak olyan műveletek is, amelyet a fenti megfontolások miatt mindig szinkronizáltan szeretnénk végrehajtani. Ezekben az esetekben célszerű ezt kikényszeríteni, és nem hagyatkozni arra, hogy más programozók vagy akár saját magunk mindig emlékezni fogunk ezekre a megkötésekre. A Java nyelvben teljes metódusokat is szinkronizálttá tehetünk a nevük előtt megadott `synchronized` módosítóval. Ekkor a metódus saját objektumának monitorát foglalja le mielőtt futni kezdene, statikus metódus esetén pedig az osztályhoz tartozó `Class<T>` (lásd 12.1. alfejezet) objektum monitorát. Ez a megoldás tulajdonképpen azzal egyenértékű, mint ha a metódus teljes törzsét a `this` referenciával szinkronizáltuk volna.

Fontos észrevenni, hogy a `synchronized` kulcsszó használata szigorúan véve nem teszi atomivá a kódot, más szálak továbbra is megszakíthatják. Csupán azt garantálja, hogy más szálak ne férjenek hozzá a monitorral védett adatstruktúrákhoz addig, amíg a kritikus művelet be nem fejeződik. Az azonos monitoron hívott `synchronized` blokkok ezért egymás számára oszthatatlannak tűnnek.

Bonyolultabb programok esetén ügyelni kell az ún. *holtpontok (deadlock)* elkerülésére is. Holtponton azt értjük, hogy *A* szál *b* monitorára vár, de azt *B* szál már lefoglalta. *B* szál viszont *a* monitorára vár, amelyet *A* foglalt le. Ekkor egyik szál sem tud továbbmenni, mert mindkettő a másikra vár. A holtpontok ellen úgy védekezhetünk egyszerűen, hogy a monitorokat mindig megadott sorrendben foglaljuk le.

11.6. Várakozás eseményekre

A monitorok segítségével a szálak egy adott objektumon várakozhatnak. A várakozás az `Object` osztályban definiált `wait()` metódussal történik, tehát bármely Java-objektum használható a várakozáshoz. A metódus paraméter nélküli változatát meghívva a szál várakozni kezd (`WAITING` állapot), amíg egy másik szál fel nem ébreszti az adott objektumon várakozó szálakat. A `wait()` metódusnak időlimit is megadható milliszekundumokban, ekkor a szál ébresztésig vagy az időlimit leteltéig várakozik (`TIMED_WAITING` állapot). A `wait()` hívásához először meg kell szereznünk az objektum monitorát, ez tehát csak szinkronizált kontextusból hívható, különben `IllegalMonitorStateException` kivétel váltódik ki. A várakozás során a szál kilép a monitorból, majd ébresztés után újra belép abba. Adott objektumon várakozó szálak felébresztéséhez szintén meg kell szerezni az objektum monitorát. A `notify()` egyetlen várakozó szálat ébreszt fel. A szál ekkor `RUNNABLE` állapotba kerül, de nem biztos, hogy rögtön futni fog, ez az ütemezőtől függ. Arra sincs garancia, hogy az ébresztett szál az

első szál lesz, amelyik várakozásba kezdett. A `notifyAll()` felébreszti az összes várakozó szálat. A `wait()` metódus `InterruptedException` kivételt válthat ki, akárcsak a `Thread` osztály `sleep()` metódusa, a várakozó szálat ugyanis másik szál megszakíthatja. A várakozás jól használható termelő-fogyasztó problémánál. A problémára a naiv megoldás, hogy végtelen ciklusban vizsgáljuk az adat érkezését. Ez nagyon költséges megvalósítás. Hatékonyabb az ellenőrzések között a `sleep()` metódus hívása valamilyen rövid ideig, de az idő megválasztása nem könnyű. Ha túl kicsire választjuk, akkor továbbra is erőforrás-pazarló lesz a program, ha túl nagyra, akkor pedig a termékek előállítása és feldolgozása közt szükségtelenül sok idő telik el. A legjobb megoldás ezért, hogy a fogyasztó várakozik, amikor nincs feldolgozandó termék, a termelő pedig felébreszti, miután új terméket helyezett el a sorban. Így a fogyasztó ténylegesen csak akkor fut, amikor arra szükség van. A számlatranzakciók előállítása és feldolgozása is termelő-fogyasztó probléma. Az előző kódrészletben látható, hogy a várakozási sorba történő írás után a sor `notify()` metódusát hívjuk. A fogyasztóban a sorból történő olvasás az alábbi kódrészlettel történik.

```
// várakozunk, amíg nincs kérés, utána kivesszük a sorból
synchronized (queue) {
    if (queue.isEmpty())
        queue.wait();
    req = queue.poll();
}
```

11.7. A szálbiztos osztályok

Az osztálykönyvtár olyan osztályokat is kínál, amelyek másokéval megegyező funkcionalitást valósítanak meg, de metódusaik szinkronizálva vannak, ezért praktikusan használhatók többszálú környezetben. A `StringBuffer` a `StringBuilder` helyett használható, a `Vector` szinkronizált listát valósít meg, a `Hashtable` pedig a `HashMap` szinkronizált megfelelője. Kollekciók esetén a `Collections` osztály `synchronized` szóval kezdődő metódusai is használhatók (lásd 5.3.8. alfejezet). Ezek bármilyen típusú kollekciót képesek becsomagolni olyan objektumba, amely szinkronizálttá teszi őket. A szinkronizált osztályok jó szolgálatot tehetnek többszálú környezetben, de ha nem használunk szálakat, vagy az adott adatstruktúrát csak egyetlen szálból érjük el, akkor ne használjuk őket. A szinkronizáció ugyanis többletköltséggel jár.

Fontos belátni, hogy ezek az osztályok nem nyújtanak teljes védelmet az inkonzisztens adatok ellen. Például ha egy lista utolsó elemét próbáljuk lekérdezni, az elemszám és az utolsó elem lekérdezése közben más szál módosíthatja a listát. Lehetséges ezért, hogy az index már nem lesz érvényes, vagy nem az utolsó elem indexe lesz. A `Vector` osztály metódusai szinkronizálva vannak, jelen esetben mégsem segít annak használata, az elemszám lekérdezésének és az utolsó elem kiolvasásának ugyanis együtt kell megszakíthatatlan egységet alkotnia. Ezért fontos mindig átgondolni, hogy pontosan meddig terjednek a kritikus műveletsorok.

A példaprogramban a számlákról történő tényleges levonás és jóváírás abból áll, hogy lekérdezzük az egyenleget, kiszámoljuk az új értékét, majd visszaírjuk. Ez a három lépés kezelendő egy egységként. Ha ugyanis az egyenleg kiolvasása után más szál módosítja azt, akkor a módosítás hatása elvész, mert a feldolgozás alatt lévő tranzakció hatását a régi érték alapján számoljuk ki. Az alábbi kódrészlet mutatja ezt a részt:

```
// elvégezzük a feldolgozást egy menetben
synchronized (acc) {
    double balance = acc.getBalance();
    balance += req.getSum();
    acc.setBalance(balance);
    System.out.println("Tranzakció feldolgozva " + acc.getAccountNo()
        + " számlához " + req.getSum() + " összeggel, új egyenleg "
        + balance + '.');
}
```

11.8. Szálkezelés a Swing-alkalmazásokban

A grafikus felhasználói felülettel rendelkező alkalmazások szálkezelése különös körültekintést igényel. Ennek két fő oka van. Az egyik, hogy a Swing keretrendszer osztályai általánosan nem szálbiztosak, ezért az eseménykezelés és az UI-elemek manipulációja egyetlen kitüntetett szálban, az eseménykezelő szálban történik. Van néhány szálbiztos osztály is, ezek bármely szálból hívhatók. A Javadoc-dokumentáció ezeket az eseteket egyértelműen említi. Az osztályok általánosan azért nem szálbiztosak, mert a Swing keretrendszer komplexitása mellett nem lehet hatékony szálbiztos megvalósítást készíteni. A Swing keretrendszer az eseménykezelő szálat maga hozza létre, és abban taszkokat az `EventQueue` osztály `invokeLater()` statikus metódusával tudunk futtatni. Ez az oka, hogy az UI-elemeket nem a főszálban hozzuk létre, hanem az `invokeLater()` metódust használjuk.

A másik ok, amiért a többszálúság nehezen kezelhető a grafikus felülettel rendelkező programokban, hogy az eseménykezelő szálban futó hosszú taszkok megbénítják a felhasználói felületet. Ezért indokolt, hogy külön szálban fussanak, abból azonban nem tudják frissíteni az UI-elemeket. Az UI-elemek frissítése ugyanis csak az eseménykezelő szálból végezhető el biztonságosan. Ehhez tehát szálak közötti kommunikációt kell megvalósítani. Erre a problémára a Swing *munkaszálakat (worker threads)* kínál, ezeket a `SwingWorker` absztrakt osztály származtatásával valósíthatjuk meg. Ezek több kommunikációs mechanizmust is nyújtanak, hogy kommunikálhassunk az eseménykezelő szállal. Rövid taszkok, amelyek nem bénítják meg a felhasználói felületet, az `invokeLater()` segítségével az eseménykezelő szálban is hívhatók. Hosszabb taszkok esetén ajánlott a `SwingWorker` használata. A Swing-alkalmazások tehát a funkciójuk szerint háromféle szálat alkalmaznak:

- Inicializáló szál vagy szálak: az UI összeállítását és megjelenítését indítják el az eseménykezelő szálban. Tipikusan ez a főszál, ebben fut a `main()` metódus.

- Eseménykezelő szál: az a kitüntetett szál, amelyben az eseményeket kezeljük, és az UI-elemeket manipuláljuk. Az `EventQueue` osztály statikus `invokeLater()` metódusával tudunk benne taszkokat futtatni.

- Munkaszálak: ezek segítségével tudunk hosszú lefutású taszkokat végrehajtani a háttérben. A `SwingWorker` absztrakt osztály származtatásával hozhatók létre.

Az alfejezethez külön példa készült, ez egy szövegdobozt frissít három különböző módon. A frissítéshez szükséges adatok előállítása hosszú lefutású folyamat. Ezt a példában úgy szimuláljuk, hogy az adatok létrehozása közben a szálat rövid időkre elaltatjuk. Az első módszer az eseménykezelő szálban állítja elő az adatokat. Ez a helytelen megoldás csupán annak megfigyelésére szolgál, hogy a felhasználói felület valójában megbénul a frissítés ideje alatt. A frissítést gomb megnyomására tudjuk elvégezni. Az alábbi kódrészlet valósítja meg a frissítést. A kódot a példaprogram az `EventQueue` osztály `invokeLater()` metódusával hajtja végre.

```
class EventHandlerUpdate implements Runnable {
    @Override
    public void run() {
        try {
            Thread.sleep(5000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        textArea.setText("");
        for (int i = 1; i < 10; i++)
           textArea.append("Adat" + i + '\n');
    }
}
```

A másik két esetben a `SwingWorker` által kínált kommunikációs mechanizmusokat alkalmazzuk. A `SwingWorker` két típusparaméterrel (lásd 5.2. alfejezet) rendelkező generikus osztály. Az osztály esetében nem a `run()`, hanem a `doInBackground()` metódus definíciója adja a szálban futtatandó kódot. Az első típusparaméter ennek a visszatérési értéke, a munkaszálak ugyanis gyakran valamilyen adatot állítanak elő, amelyet a felhasználói felületen meg kell jeleníteni. A legegyszerűbb módja a felhasználói felület frissítésének a munkaszál implementációjában a `done()` metódus újradefiniálása. Ezt a metódust az eseménykezelő szál hívja meg, miután a `doInBackground()` lefutott. A metódusban ezért szabadon frissíthetjük a felhasználói felületet. A `doInBackground()` által visszaadott értéket a `done()` metódusban a `get()` hívásával kapjuk meg. Az alábbi példa ezt a frissítési módot szemlélteti. A `DoneUpdate` osztály az ablakot megvalósító osztály belső osztálya, ezért hozzá tud férni a példányváltozóihoz, a `done()` metódus tehát frissíteni tudja a szöveget.

```
class DoneUpdate extends SwingWorker<String, Void> {
    @Override
    protected String doInBackground() throws Exception {
        StringBuilder sb = new StringBuilder();
        for (int i = 1; i < 10; i++) {
            sb.append("Adat");
            sb.append(i);
            sb.append('\n');
            Thread.sleep(500);
        }
        return sb.toString();
    }

    @Override
    protected void done() {
        try {
            textArea.setText(get());
        } catch (InterruptedException | ExecutionException e) {
            e.printStackTrace();
        }
    }
}
```

Egy másik lehetséges megközelítésben a munkaszál folyamatosan részeredményeket tesz elérhetővé az eseménykezelő szál részére. A második típusparaméterben a részeredmények típusa adható meg. A fenti példában nem használtunk részeredményeket, ezért a `Void` helykitöltő típust adtuk meg. A részeredmények publikálása a `publish()` metódus hívásával történik. A feldolgozást a munkaszál osztályában implementált `process()` metódus végzi, ezt az eseménykezelő szál hívja meg. Lehetséges, hogy mire a `process()` metódust az eseménykezelő szál meghívja, a munkaszál már több részeredményt is publikált. A metódus ezért a részeredmények listáját kapja meg. A megoldás előnye, hogy a felhasználói felület folyamatosan frissíthető, ezért a felhasználó nagyobb folyamatosságot érzékel a program használata során. Az alábbi kódrészlet erre a mechanizmusra mutat példát:

```
class PublishUpdate extends SwingWorker<Void, String> {
    private boolean firstResult = true;

    @Override
        protected Void doInBackground() throws Exception {
            Thread.sleep(2000);
            for (int i = 1; i < 10; i++) {
                publish("Adat" + i);
                Thread.sleep(2000);
            }
            return null;
```

```
        }
    @Override
    protected void process(List<String> chunks) {
        if (firstResult) {
            textArea.setText(chunks.get(0));
            for (Iterator<String> it = chunks.listIterator(1); it.↺
hasNext();)
                textArea.append('\n' + it.next());
            firstResult = false;
        } else {
            for (String s : chunks)
                textArea.append('\n' + s);
        }
    }
}
```

TIZENKETTEDIK FEJEZET

A reflection API

A reflection API osztályok, objektumok és a tagjaik programból történő elérésére használható. Segítségével lekérdezhetjük az osztályok tagjait, és a metódusokat meg is hívhatjuk. Még a privát tagokhoz is hozzá tudunk férni. Ezt a technikát keretrendszerek használják generikus funkcionalitás megvalósítására. Például a fejlesztőkörnyezetekben gyakran elérhető vázlatnézet is így valósítható meg, vagy a JavaBeans-komponensek tulajdonságainak feltérképezésére is használható. A getter és setter metódusok ugyanis jól meghatározott elnevezési konvenciót követnek. A reflectiont azonban a hétköznapi alkalmazásokban ritkán használjuk. A reflection igen erőteljes eszköz lehet, de megsérti az objektumorientált programozás egységbe zárás elvét, ezért használata csak kellő körültekintés után célszerű. A fejezet rövid betekintést nyújt a reflection használatába, de nem részletezi mélyebben.

12.1. Az osztályok felderítése

A reflection API a `java.lang.reflect` csomagban található, illetve használja a `java.lang` csomag `Class<T>` osztályát. Ez utóbbi egy Java-osztályt reprezentál, és típusparamétere is a reprezentált osztályra hivatkozik. Egy osztályt reprezentáló `Class`-példányt a következő két módon szerezhetünk:

```
Class<String> c1 = String.class;
Class<String> c2 = Class.forName("java.lang.String");
```

Miután megszereztük az osztályt reprezentáló objektumot, metódusain keresztül gyakorlatilag az egész osztályt feltérképezhetjük. Lekérdezhetjük a konstruktorokat, a metódusokat, a tagváltozókat, a tagváltozóként használt összes osztályt, az osztály csomagját, és az osztályon elhelyezett annotációkat is. Ezeket az elemeket mind saját osztály reprezentálja, amelyeken keresztül lehetséges az elemek jellemzőinek további lekérdezése.

12.2. A tagváltozók lekérdezése

A tagváltozókat a `Field` osztály reprezentálja. Az összes tagváltozót elérhetjük egy tömbben a `Class` osztály `getFields()` metódusával. Egyes tagváltozókat a `getField()` metódussal kérdezhetünk le, ez a tagváltozó nevét várja paraméterben. Ha ilyen nem létezik, akkor `NoSuchFieldException` váltódik ki.

A `Field` objektum `String getName()` metódusa visszaadja a reprezentált tagváltozó nevét, a `Class<?> getType()` metódus pedig a típusát. Primitív típusonként létezik lekérdező metódus, ennek paraméterül adhatunk egy objektumpéldányt, és kinyeri belőle a reprezentált tagváltozó értékét. Például, ha a tagváltozó `long` típusú, akkor a `long getLong()` metódust használhatjuk a kiolvasására.

12.3. A metódusok lekérdezése

A metódusokat a `Method` osztály reprezentálja. A `Class` objektumtól az összes metódust tartalmazó tömböt a `getMethods()` metódussal érhetjük el. Név szerint is lekérdezhetünk metódusokat, de a túlterhelés miatt egy névvel több, különböző paraméterlistájú metódus is létezhet. Ezért a metódus egyenkénti lekérdezése sokkal bonyolultabb, ugyanis a paraméterlistát is meg kell adni a lekérdezés részeként. Az erre szolgáló metódus szignatúrája a következő:

```
Method getMethod(String name, Class<?>... parameterTypes)
```

Az így lekérdezett metódusokon ezután szintén számos művelet végezhető: lekérdezhető a visszatérési érték típusa, a paraméterek típusa és a metódus annotációi is. A metódus meg is hívható egy adott objektumpéldányon. Ennek a módját nem részletezzük.

12.4. Egy példa

Az alábbi példaprogram parancssori paraméterben vár egy csomagnévvel kvalifikált osztálynevet, majd feltérképezi és kiírja az osztály tagváltozóit és metódusait a szignatúrájukkal együtt. A program nem kezeli az összes módosítót, és a típusparamétereket sem.

```
public class Reflector {

    public static void reflectModifier(int m) {
        if (Modifier.isPublic(m))
            System.out.print("public ");
        if (Modifier.isProtected(m))
            System.out.print("protected ");
        if (Modifier.isPrivate(m))
            System.out.print("private ");
        ...
    }

    public static void reflectField(Field f) {
        reflectModifier(f.getModifiers());
        System.out.print(f.getType().getSimpleName());
        System.out.print(' ');
```

```
        System.out.print(f.getName());
        System.out.println(';');
    }

    public static void reflectMethod(Method m) {
        reflectModifier(m.getModifiers());
        System.out.print(m.getReturnType().getSimpleName());
        System.out.print(' ');
        System.out.print(m.getName());
        System.out.print('(');
        Class<?>[] paramTypes = m.getParameterTypes();
        for (int i = 0; i < paramTypes.length; i++) {
            System.out.print(paramTypes[i].getSimpleName());
            System.out.print(" arg" + i);
            if (i < paramTypes.length - 1)
                System.out.print(", ");
        }
        System.out.println(");");
    }

    public static void reflectClass(Class<?> c) {
        for (Field f : c.getFields())
            reflectField(f);
        for (Method m : c.getMethods())
            reflectMethod(m);
    }

    public static void main(String[] args) {
        if (args.length < 1) {
            System.out.println("Adja meg a csomagnévvel kvalifikált ↺
osztálynevet paraméterben!");
            System.exit(-1);
        }
        try {
            Class<?> c = Class.forName(args[0]);
            reflectClass(c);
        } catch (ClassNotFoundException e) {
            System.out.println("Az osztályt nem található!");
            e.printStackTrace();
        }
    }
}
```

TIZENHARMADIK FEJEZET

A naplózás

A legtöbb alkalmazásban szükség van naplózásra. A naplóüzenetek segítenek végigkövetni az alkalmazás futásának folyamatát. Ez több célt szolgálhat. Ha a program futása valamilyen nem kívánatos hatást okozott, például fontos fájlok tűntek el, vagy inkonzisztens állapotba került az adatbázis, akkor a naplóüzeneteket *auditálási* célra használhatjuk. A napló ugyanis felfedheti a hiba okát, amely akár egy biztonsági visszaélés is lehetett. A naplók a *hibakeresést* is megkönnyítik, mert segítenek megállapítani, hogy a program mely ponton kezdett el hibásan működni. A felhasználók általában nem rendelkeznek mély ismeretekkel a programokról, és nincsenek is tisztában az adott alkalmazás sajátosságaival, ezért a hibajelentések gyakran kevés információt tartalmaznak ahhoz, hogy a fejlesztők megtalálják a jelentett hiba okát. A naplóüzenetek ebben az esetben is segíthetnek. A fejezet bemutatja a naplózás használatát Java nyelven. Főként a JDK saját naplózórendszerét tárgyaljuk, ez ugyanis kielégíti a legfontosabb naplózási igényeket, és használatához nem szükséges külső osztálykönyvtár letöltése. Végül bemutatjuk, hogyan lehet keretrendszertől független módon megvalósítani a naplózást.

13.1. A JDK 1.4 Logger API

A JDK Logger az 1.4-es verziótól része a Java nyelvnek. Előtte leginkább a külön projektként fejlesztett nyílt forrású naplózó keretrendszert, a Log4j-t használták a Java-fejlesztők. A keretrendszer valójában bővebb funkcionalitással rendelkezik, mint a JDK saját megoldása, de a többletfunkcionalitásra ritkán van szükség. Ennek ellenére a Log4j még mindig népszerű, sok fejlesztő megszokásból ezt használja az új alkalmazásokban is. Szerencsére a JDK Logger API ismeretében a Log4j is könnyen megérthető, ugyanis nagyon hasonlóan működik. Ezért a könyv csak a JDK saját naplózó megoldását ismerteti.

13.1.1. A naplózórendszer áttekintése

A naplózórendszer a `java.util.logging` csomagban található. A keretrendszer több alapvető fontosságú osztályból áll. A `Logger` metódusainak adjuk át a naplóüzeneteket, ezen keresztül történik a tényleges naplózás. A keretrendszeren belül a `LogRecord` hordozza a naplóüzenetet, de ehhez csak akkor kell hozzáférnünk, ha saját kimeneti formátumot, szűrőt vagy formázót készítünk. A `Handler` példányai dolgozzák fel a naplóüzeneteket, és a konkrét leszármazott osztálynak megfelelő kimenetre küldik. Ez a kimeneti lehet `OutputStream`-példány (lásd 4.5.1. alfejezet), a konzol szabványos kimeneti folyama, fájl vagy socket. Természetesen a `Handler` osztály specializálásával saját kimenethez is készíthető támogatás. A `Level` osztály naplózási szinteket támogat. Tulajdonképpen úgy működik, mint egy enumeráció (lásd 3.13. alfejezet), de osztály-

ként van megvalósítva, mert a naplózórendszer bevezetésekor a Java nyelv még nem támogatta az enumerációkat. Példányai naplózási szinteket reprezentálnak. Az üzenetek és a Logger példányai egyaránt megadhatnak naplózási szintet, és a Logger csak azokat az üzeneteket naplózza, amelyek szintje nem kisebb az övénél. A 13.1. táblázat csökkenő sorrendben felsorolja a naplózási szinteket és a funkciójukat.

13.1. táblázat: A naplózási szintek és funkciójuk

Szint	Leírás
SEVERE	hibajelzés-értékű naplóüzenet
WARNING	figyelmeztetés; nem hibaértékű, de fontos naplóüzenet
INFO	információs üzenet, például a program futása során bekövetkezett lényeges események jelzése
CONFIG	konfigurációval kapcsolatos információt hordozó üzenetek, például a beolvasott konfigurációs paraméterek, illetve futás közbeni változásuk
FINE	szabadon alkalmazható részletes naplóüzenetekhez
FINER	szabadon alkalmazható még részletesebb naplóüzenetekhez
FINEST	szabadon alkalmazható nagy részletességű naplóüzenetekhez

A Logger és a Handler is rendelkezhet szűrővel, ez üzenetszinteken túli szűrést is végezhet. Szűrőt a Filter interfész megvalósításával készíthetünk. Ez az isLoggable() metódust írja elő. Ennek LogRecord-példányt kell átadni, és ebben implementálható a saját szűrési feltétel. A metódus true értéket, ha az üzenetet naplózni kell. A naplózórendszer nem teszi lehetővé, hogy szűrőket egymáshoz láncoljunk (Chain of Responsibility tervezési minta, lásd [4]), de kézileg készíthetünk ilyen megvalósítást, például úgy, hogy olyan Filtert implementálunk, amely listában tárolt szűrőknek delegálja a döntést, és kiértékeli az eredményt. Az utolsó fontos osztály a Formatter, ez az üzenetek kiírás előtti formázását végzi. Az osztálykönyvtár egyszerű szöveget (SimpleFormatter) és XML-formátumot (XMLFormatter) támogat, de saját implementáció is készíthető.

A naplózás használatához először példányt kell szereznünk a Logger osztályból. Ezt annak statikus metódusaival tehetjük meg. A getAnonymousLogger() névtelen példányt ad vissza. Jó gyakorlat azonban a naplózó objektumoknak nevet adnunk, amely utal a csomagra vagy az osztályra, ahonnan az üzeneteket kiírjuk. Ehhez a getLogger() metódust használjuk, amely karakterlánc paraméterben kapja meg a naplózó objektum nevét. Ezután két megközelítést választhatunk a naplóüzenetek kiírásához. Az első megközelítés szerint a naplóüzenet mellett az osztály és a futó metódus nevét is megadjuk. Ekkor ezek is megjelennek a kimenetben, és segítenek azonosítani a hiba helyét. A másik megközelítésben nem csatolunk ilyen kiegészítő információt az üzenetekhez. Természetesen a két módszer együttesen is alkalmazható. Ezen kívül a Logger osztály rendelkezik általános és kényelmi metódusokkal a naplózáshoz. Utóbbiak a gyakori esetekben alkalmazható egyszerűsített metódusok. A metódusok tehát négy csoportba oszthatók:

- Ha a naplóüzenetet el akarjuk látni az osztály és a metódus nevével is, akkor a `Logger` osztály `logp()` metódusát hívjuk, ennek sorban megadjuk a naplózási szintet, az osztály nevét, a metódus nevét, valamint a naplóüzenetet karakterláncként. Az opcionális ötödik paraméter típusa `Object`, `Object[]` vagy `Throwable` lehet, és a naplóüzenet mellé csatolhatunk vele paramétereket.

- Az osztály- és metódusnév nélküli üzenetek kiírására a `log()` metódus szolgál. Ennek tehát csak a naplózási szintet és a naplóüzenetet kell megadni. Az opcionális harmadik paraméter az előző metódus ötödik paraméterének felel meg.

- Kényelmi metódusok a metódushívások be- és kilépési pontjának, valamint a kivételek naplózására. Ezeknek a metódusoknak tehát nem kell megadni a naplózási üzenetet, csak az osztály- és a metódusnevet, az opcionális paramétert vagy a kivételt.

- Kényelmi metódusok naplózási szintenként. Ezeknek nem kell külön megadni a naplózási szintet, de osztály- és metódusnevet sem adhatunk meg. Az egyetlen paraméter a naplóüzenet.

A metódusok nem támogatják, hogy a naplóüzeneteket egyedi azonosítóval lássuk el, ezért ha ez szükséges, akkor az azonosítók létrehozásához saját algoritmust kell kifejleszteni. Az alábbi kód egy egyszerű példán mutatja be az alapvető naplózási hívások használatát:

```
public class Main {
    private static final Logger logger = Logger.getLogger("logging");

    public static void main(String args[]) {
        logger.entering("Main", "main");

        String name = "";

        logger.info("A konfiguráció beolvasása");
        Properties props = new Properties();
        File f = new File("settings.properties");
        try {
            if (!f.exists()) {
                f.createNewFile();
            }
            props.load(new FileReader(f));
            name = props.getProperty("Name", "Névtelen felhasználó");
        } catch (IOException e) {
            logger.severe("Nem sikerült a konfigurációt olvasni.");
            e.printStackTrace();
        }
        logger.config("A konfiguráció beolvasva.");

        logger.fine("A felhasználó üdvözlése");
```

```
        System.out.println("Üdvözlöm, " + name + '!');
        logger.exiting("Main", "main");
    }
}
```

13.1.2. A naplózás konfigurációja

A `Logger` osztály példányai a megadott név alapján hierarchiákba szerveződnek. A neveket konvencionálisan a csomagnevekhez hasonló módon adjuk meg, ezért a naplózórendszer a pont karaktert tekinti elválasztónak. Például a `net` nevű naplózó objektum szülője lesz a `net.clientnek`, ha mindkettő létezik. Ha nem hozzuk létre a névtér szerinti szülőt, vagy nincs pont a naplózó objektum azonosítójában, akkor az üres karakterlánccal azonosított gyökérnaplózót kapja meg szülőnek. A hierarchia segíti a konfigurációt, a naplózók ugyanis alapértelmezésben öröklik szülőjük naplózási szintjét, illetve a szülő `Handler` eibe is írnak. Az JRE alapértelmezett beállítása szerint a gyökérnaplózó szintje `INFO`, és az a konzolra írja a naplóüzeneteket. A többi naplózó is ezt a beállítást örökli alapértelmezésben.

A beállítások megváltoztatásának két módja van. Az egyik, hogy az osztálykönyvtár hívásaival programból beállítjuk a naplózási szintet, `Handler` eket stb. Ehhez a naplózórendszer automatizált megoldást is nyújt. Ez úgy működik, hogy készítünk egy osztályt, amelynek statikus inicializációs blokkjaiban, illetve alapértelmezett konstruktorában végezhetjük el az inicializációt. Ezt követően a program indításakor a `java.util.logging.class` rendszerbeállításban meg kell adni az inicializációs osztály csomagnévvel kvalifikált nevét. Az alábbi kódrészlet ad példát arra, hogyan szabható testre a naplózás programból. A kódrészlet először letiltja a szülők handlereinek használatát a `Loggeren`. Ezután `FINEST`, és XML formátumú fájlba íratja a naplóüzeneteket. Ha a naplófájl megnyitása meghiúsul, akkor a konzolra ír mindent.

```
boolean consoleLog = false;

logger.setUseParentHandlers(false);
logger.setLevel(Level.FINEST);
Handler h;
try {
    h = new FileHandler("application.log");
} catch (SecurityException | IOException e1) {
    h = new ConsoleHandler();
    consoleLog = true;
}
h.setLevel(Level.FINEST);
h.setFormatter(new XMLFormatter());
logger.addHandler(h);
if (consoleLog)
    logger.warning("Nem lehet a naplófájlt inicializálni, a napló a ↺
konzolra kerül.");
```

A másik lehetőség konfigurációs fájl használata a Properties API szerinti formátumban (lásd 6.1. alfejezet). Ennek elérési útja a `java.util.logging.config.file` rendszerbeállítás segítségével adható át a rendszernek, de az így megadott beállítások csak akkor töltődnek be, ha nem adunk meg inicializációs osztályt. Az alábbi példa mutatja a fájlban történő konfiguráció szintaxisát. A konfiguráció törli a gyökérnaplózó handlereit, majd a XML-formátumú üzenetek fájlba mentését állítja be `FINEST` szinttel, akárcsak a fenti példa.

```
# Ez a gyökérnaplózó beállítása
handlers =

# A logging naplózó beállításai
logging.handlers           = java.util.logging.FileHandler
logging.useParentHandlers  = true
logging.level              = FINEST

# Az összes FileHandler beállításai
java.util.logging.FileHandler.level     = FINEST
java.util.logging.FileHandler.filter    =
java.util.logging.FileHandler.formatter =
java.util.logging.FileHandler.encoding  =
java.util.logging.FileHandler.limit     =
java.util.logging.FileHandler.count     =
java.util.logging.FileHandler.append    = false
java.util.logging.FileHandler.pattern   = application.log
```

Mint látható, egy adott típusú handlerhez csak egyféle beállítást tudunk megadni, a példányok nem kezelhetők külön. Ha bonyolultabb konfigurációra van szükség, akkor a beállításokat kódból kell elvégezni, egyébként ajánlott a konfigurációs fájl használhata, azzal ugyanis egyszerűen és újrafordítás nélkül változtathatjuk meg a beállításokat. Megjegyzendő az is, hogy a naplózórendszert úgy tervezték, hogy a naplózás költsége minimális legyen. A nem naplózott üzeneteket a keretrendszer már a folyamat elején kiszűri, hogy a további feldolgozás elkerülhető legyen. A formázók hívása is csak a lehető legkésőbb történik. Ezért a programokban bátran megvalósíthatunk gazdag naplózást. Ez megkönnyíti a hibakeresést, de kikapcsolt állapotban gyakorlatilag nem befolyásolja a teljesítményt.

13.2. Az slf4j keretrendszer

A JDK Logger, a Log4j és egyéb naplózórendszerek saját programozói interfésszel rendelkeznek. Adott keretrendszer választása tehát azt eredményezi, hogy a program az adott keretrendszertől fog függeni, lecserélése pedig a kód módosítását igényli. Az ehhez hasonlóan erős technológia- és gyártófüggés sosem előnyös, mert ha az alkalmazott technológia vagy termék támogatása megszűnik, akkor a programban nehezen lehet jobban támogatott technológiára vagy termékre cserélni. Különösen előnytelen ez a függés akkor, ha egy osztálykönyvtár belső naplózásáról van. Ez ugyanis azt eredmé-

nyezi, hogy az osztálykönyvtárra épülő programok mind függeni fognak annak naplózási keretrendszerétől, és ha a naplókat egységesen akarják kezelni, akkor rá vannak kényszerítve arra, hogy ugyanazt a naplózórendszert alkalmazzák. Ez a probléma szorgalmazta a naplózás egységesítését. Az egyik ilyen megoldás az Apache Commons Logging, amelynek alapötlete, hogy saját API-t nyújt a naplózáshoz, de a színfalak mögött a tényleges naplózást egy másik naplózórendszernek delegálja. Ez a naplózórendszer a program indításakor konfigurálható, tehát később szabadon lecserélhető. Ezért a program nem fog attól függeni, csupán az Apache Commons Logging pehelysúlyú osztályaitól. Az Apache Commons Logging alapötlete jó, azonban nehézkesen konfigurálható. Ezért született egy másik megoldás is, a Simple Logging Facade for Java, vagy röviden slf4j. Az slf4j tovább egyszerűsíti a konfigurációt. A programozói interfésze nagyon egyszerű, mindent delegál ugyanis egy tényleges naplózórendszernek, így a kimeneti formátum, a szűrők, a formázó stb. konfigurációja abban történik. A keretrendszer programozói interfészét a könyv ezért nem tárgyalja.

A fenti megfontolások alapján osztálykönyvtárak fejlesztésénél az slf4j használata javasolt. Ez a legnagyobb rugalmasságot eredményező választás, de ha az osztálykönyvtár a JDK Loggert, a Log4j-t vagy a kevésbé rugalmas Apache Commons Loggingapplication>ot használja, az slf4j projekt akkor is kínál áthidaló (bridging) megoldást a naplózás egységesítésére. A JDK Logger a JDK osztálykönyvtárának része, ezért nem cserélhető le. Az sl4j az `SLF4JBridgeHandler` osztályt biztosítja, ez a naplózás kimenetét az slf4j keretrendszerhez küldi, utóbbi viszont delegálni tudja a kizárólagosnak választott naplózórendszernek. Csupán úgy kell konfigurálni a JDK Loggert, hogy a `Logger` példányai ezt a `Handler`t használják. A másik két keretrendszer nem a JDK része, ezért az osztályok egyszerűen lecserélhetők a JAR-csomag kicserélésével. Az slf4j ezekhez a naplózórendszerekhez is kínál olyan implementációt, amely a naplózást neki delegálja. Az áthidaló megoldások segítenek abban, hogy a naplózást utólag egységesítsük, még akkor is, ha eredetileg nem volt lehetséges, mert az egységesített naplózás nem merült fel igényként a fejlesztés kezdeti szakaszában. Ennek ellenére hosszú távú megoldásként megfontolandó az slf4j keretrendszerre történő teljes átállás.

TIZENNEGYEDIK FEJEZET
Nyelvek és kultúrák

Egy alkalmazást sokszor különböző nyelvű, kultúrájú személyek is használnak. A felhasználói komfort érdekében fontos lehet az internacionalizáció és a lokalizáció. Előbbin az adott kultúra konvencióihoz való igazodást értjük, így például a megfelelő pénznem és dátumformátum használatát; utóbbi a program adott nyelvre történő lefordítását jelenti. A fejezet ezt a két témakört ismerteti Java nyelvű környezetben.

14.1. Az internacionalizáció

Az internacionalizáció és a lokalizáció egyik alaposztálya a `Locale`. Az osztály földrajzi, kulturális vagy politikai régiót azonosít, magában foglalja a nyelvet, a betűrendet és az országot. A lokalizáció szó ezeknek a paramétereknek a halmazát is jelenti, nemcsak a honosítási folyamatot. Az internacionalizációt támogató osztályoknak a `Locale` megfelelő példányával adhatók át a kulturális paraméterek. Az osztályból példányokat háromféleképpen szerezhetünk:

- Némely nyelvekhez vagy országokhoz kapcsolódó példányokat konstansként elérhetünk az osztályból, például `Locale.JAPANESE` vagy `Locale.JAPAN`.

- Használhatjuk az osztály konstruktorait. Az egyparaméteres konstruktornak csak a nyelv *ISO-639* szabvány szerinti kódját adjuk meg, a kétparaméteresnek az ország *ISO-3166* kódját is.

- A statikus `getAvailableLocales()` tömbként visszaadja az összes elérhető példányt.

14.1.1. A számok formázása

Számokat és pénznemet a `NumberFormat` absztrakt osztállyal és a leszármazott osztályaival formázhatunk. Az osztályból a formázás rendeltetése szerint négyféle példányt készíthetünk. Mind a négy típusú példányt statikus factorymetódusokkal hozhatjuk létre. Ezeknek paraméterben adhatjuk meg a kívánt beállításokat hordozó `Locale`-példányt. Ha nem adunk meg paramétert, akkor az alapértelmezett lokalizáció szerinti formázót kapjuk meg. A négy formázót a következőképpen hozzuk létre:

- A `getInstance()` és a `getNumberInstance()` általános célú számformázót ad vissza.

- A `getIntegerInstance()` által visszaadott példány egész számok formázásához használható.

- A `getPercentInstance()` százalékos formátumot készítő példányt ad vissza. Ez a számokat arányként értelmezi, és azokat százalékos formátumban adja vissza, például `0.25` helyett 25%.

- A `getCurrencyInstance()` pénznemek kiírásához alkamazható formázót készít.

A `NumberFormat` osztály is rendelkezik `getAvailableLocales()` metódussal. Ez csak azokat a `Locale`-példányokat adja vissza, amelyek a számok formázását is támogatják. Nem biztos ugyanis, hogy a rendszeren létező összes regisztrált lokalizációhoz meg van valósítva a számformázás. A példány létrehozása után a `format()` metódus adja vissza karakterláncként a formázott számot. A metódusnak `long` és `double` paramétert fogadó változata is van. Az alábbi kódrészlet Portugália konvenciói szerint formáz meg egy számot:

```
Locale loc = new Locale("pt", "PT");
double d = 128.35;
System.out.println(NumberFormat.getInstance(loc).format(d));
System.out.println(NumberFormat.getIntegerInstance(loc).format(d));
System.out.println(NumberFormat.getPercentInstance(loc).format(d));
System.out.println(NumberFormat.getCurrencyInstance(loc).format(d));
```

A kódrészlet a következő eredményt adja:

```
128,35
128
12.835%
128,35 €
```

14.1.2. A dátumok formázása

A dátumok formázásához a `DateFormat` osztály használható. Az osztály `Date` objektumot formáz. A `Calendar` típussal reprezentált dátumokat ezért először erre kell konvertálni `getTime()` metódussal. Az elnevezések megtévesztők, a `getTime()` metódus ugyanis `Date` objektumot ad vissza, amely nemcsak a dátumot, hanem az időt is tárolja.

Akárcsak a `NumberFormat`, a `DateFormat` osztály is absztrakt, és példányosításához a statikus factorymetódusok használhatók. Szintén többféle példányt hozhatunk belőle létre:

- A `getDateInstance()` által visszaadott példány csak a dátumot írja ki formázva. Opcionális paraméterében megadható a `DateFormat` osztály `SHORT`, `MEDIUM`, `LONG` vagy `FULL` konstansa. Ez a formázott dátum stílusát adja meg. Ha nem adunk meg stílust, akkor az alapértelmezett formázást kapjuk.

- A `getTimeInstance()` által visszaadott példány csak az időt írja ki formázva. A fenti konstansok itt is megadhatók opcionális paraméterben.

- A `getDateTimeInstance()` olyan példányt ad vissza, amely a dátumot és az időt is kiírja. Az első paramétere a dátum formázását, a második az időét állítja be.

A fenti metódusok az alapértelmezett lokalizációhoz készítenek formázót. Megadható a kívánt lokalizáció is a formázási stílusok után, de olyan factorymetódus nincs, amelynek csak `Locale`-példányt kell adni. A formázónak a `setTimeZone()` metódussal az időzónát is megadhatjuk. A tényleges formázás a `format()` metódussal végezhető el, ez `Date` objektumot vár, és a formázott dátumot karakterláncként adja vissza. Az alábbi kódrészlet Portugália konvenciói szerint formázza meg az aktuális dátumot.

```
Date date = new Date();
Locale loc = new Locale("pt", "PT");
System.out.println(DateFormat.getDateInstance(DateFormat.SHORT,
    loc).format(date));
System.out.println(DateFormat.getDateInstance(DateFormat.MEDIUM,
    loc).format(date));
System.out.println(DateFormat.getDateInstance(DateFormat.LONG,
    loc).format(date));
System.out.println(DateFormat.getDateInstance(DateFormat.FULL,
    loc).format(date));
System.out.println(DateFormat.getTimeInstance(DateFormat.SHORT,
    loc).format(date));
System.out.println(DateFormat.getTimeInstance(DateFormat.MEDIUM,
    loc).format(date));
System.out.println(DateFormat.getTimeInstance(DateFormat.LONG,
    loc).format(date));
System.out.println(DateFormat.getTimeInstance(DateFormat.FULL,
    loc).format(date));
```

A kimeneten ezt kapjuk:

```
21-09-2013
21/Set/2013
21 de Setembro de 2013
Sábado, 21 de Setembro de 2013
16:08
16:08:18
16:08:18 CEST
16H08m CEST
```

Ha a `DateFormat` által kínált formázási lehetőségek nem elegendők, akkor a factorymetódusok által visszaadott példányt konvertálhatjuk `SimpleDateFormat` típusra. Ez gazdagabb funkcionalitással rendelkezik. A `SimpleDateFormat` osztály közvetlen is példányosítható. Konstruktoraiban formátumspecifikációt is megadhatunk. Ennek segítségével a formázás teljesen testre szabható.

14.2. A lokalizáció

A lokalizáció alapja a `ResourceBundle` osztály, amely nyelvfüggő erőforrásokat fog össze. Az erőforrások kulcs-érték párokként kérdezhetők le. A kulcs mindig `String` típusú, az érték típusa elméletileg tetszőleges osztály lehet. A fejezetben csak `String` típusú kulcsokkal foglalkozunk, a lokalizáció során ugyanis a felhasználói felületen megjelenített szöveges üzenetekkel dolgozunk. A `PropertyResourceBundle` leszármazott osztály a szöveges kulcs-érték párokat a Properties API formátumának megfelelő erőforrásfájlban (lásd 6.1. alfejezet) tárolja. Az osztály tulajdonképpen ilyen fájlok egy csoportjával dolgozik. A csoportot névvel azonosítjuk, például: *MessageBundle*. Az ezzel egyező nevű `.properties` kiterjesztésű fájl tárolja az alapértelmezett üzeneteket. A lefordított üzeneteket új erőforrásfájlokban helyezzük el. A fájlnév végéhez mindig hozzáfűzzük a nyelv és opcionálisan az ország azonosítóját. Például a *MessageBundle_hu.properties* fájlban menthetjük el a magyarra, a *MessageBundle_es_AR.properties* fájlban pedig a spanyol nyelv Argentínában beszélt változatára lefordított üzeneteket. A `ResourceBundle` osztályból példányt a `getBundle()` statikus factorymetódussal szerzünk, ennek meg kell adni az erőforráscsoport nevét, valamint a használni kívánt lokalizációt. Az erőforrásfájloknak a classpathban elérhetőknek kell lenniük, hogy az osztálykönyvtár megtalálja őket. Ezután az értékeket a `getString()` metódussal érhetjük el. Ha a kulcshoz nincs megadva érték, akkor `MissingResourceException` kivétel váltódik ki.

Az üzenetek kezeléséhez gyakran használjuk a `MessageFormat` osztályt is. Az üzenetek ugyanis gyakran sablonjellegűek, és a megjelenítés előtt paramétereket kell belejük helyettesíteni. A szövegben elhelyezett `{0}`, `{1}` stb. jelölések az első, második stb. paramétert jelölik. A paraméterek alapértelmezésben szövegesek, ellenkező esetben a pozíciók után meg kell adni az adat típusát is, például: `{1, number}` vagy `{2, date}`. Egyes esetekben további módosító is megadható, például: `{1, number, currency}` vagy `{2, time, full}`. Az érvényes kombinációk az osztály Javadoc-oldalán olvashatók. Az osztály a színfalak mögött a `NumberFormat` és a `DateFormat` osztályoknak delegálja a nem szöveges paraméterek formázását. A `MessageFormat` rendelkezik egy statikus `format()` metódussal, ennek karakterláncként megadható a minta, majd változó hosszúságú paraméterlistában rendre a behelyettesítendő paraméterek. Ez a metódus azonban az alapértelmezett lokalizáció szerint formáz, így a számok és a dátumok nem az elvárt formában íródnak ki. Sajnos eltérő lokalizáció használata ennél jelentősen bonyolultabb, mert ahhoz példányosítani kell az osztályt a minta és a lokalizáció megadásával, majd a `format` metódus másik, nem statikus változatát kell használni. Ez `Object[]` tömbben várja a paramétereket, és `StringBufferbe` írja az eredményt.

A lokalizációt egy egyszerű példaprogramon próbáljuk ki, amely a napszaknak megfelelő üdvözlést ír ki, majd megjeleníti az aktuális dátumot és időt. Ha az erőforrásfájlban meg van adva, hogy egy euró mennyit ér az ország valutájában, akkor azt is kiírja. A program kódját az alábbi listában olvashatjuk:

```
public class Main {
    private static final String GOOD_MORNING = "GoodMorning";
    private static final String GOOD_AFTERNOON = "GoodAfternoon";
    private static final String GOOD_EVENING = "GoodEvening";
    private static final String CURRENT_DATE = "CurrentDate";
```

```
    private static final String TODAY_IS = "TodayIs";
    private static final String CURRENT_TIME = "CurrentTime";
    private static final String EUR_PRICE = "EurPrice";
    private static final String EUR_PRICE_VAL = "EurPriceVal";

    public static void main(String[] args) {
        Locale locale = new Locale("en", "US");
        if (args.length == 2)
            locale = new Locale(args[0], args[1]);
        ResourceBundle bundle = ResourceBundle.getBundle↺
("MessageBundle", locale);

        Calendar c = Calendar.getInstance(locale);

        if (c.get(Calendar.HOUR_OF_DAY) < 10)
            System.out.println(bundle.getString(GOOD_MORNING));
        else if (c.get(Calendar.HOUR_OF_DAY) < 18)
            System.out.println(bundle.getString(GOOD_AFTERNOON));
        else
            System.out.println(bundle.getString(GOOD_EVENING));

        MessageFormat mf = new MessageFormat(bundle.getString↺
(CURRENT_DATE), locale);
        StringBuffer sb = new StringBuffer();
        mf.format(new Object[] { c.getTime() }, sb, null);
        System.out.println(sb.toString());

        String day = MessageFormat.format(bundle.getString↺
(TODAY_IS), c.getDisplayName(Calendar.DAY_OF_WEEK, Calendar.LONG, ↺
locale));
        System.out.println(day);

        mf = new MessageFormat(bundle.getString(CURRENT_TIME), ↺
locale);
        sb = new StringBuffer();
        mf.format(new Object[] { c.getTime() }, sb, null);
        System.out.println(sb.toString());

        if (bundle.containsKey(EUR_PRICE_VAL)) {
            String currPrice = bundle.getString(EUR_PRICE_VAL);
            double price = Double.parseDouble(currPrice);
            mf = new MessageFormat(bundle.getString(EUR_PRICE), ↺
locale);
            sb = new StringBuffer();
```

```
            mf.format(new Object[] { price }, sb, null);
            System.out.println(sb.toString());
        }
    }
}
```

A program első és második paramétere a nyelv és az ország kódja. Ezek alapján a program a megfelelő nyelvi beállításokat olvassa be, ha azok erőforrásfájlként elérhetők. Példaként álljon itt a spanyol fordításhoz tartozó erőforrásfájl:

```
GoodMorning=Buenos días.
GoodAfternoon=Buenas tardes.
GoodEvening=Buenas noches.
CurrentDate=La fecha de hoy es el {0, date, long}.
TodayIs=Hoy es {0}.
CurrentTime=Son las {0, time, full}.
```

TIZENÖTÖDIK FEJEZET
A tesztelés

A tesztelés a szoftverfejlesztés elengedhetetlen része. A hagyományos felfogás szerint a tesztelés a szoftver elkészülésének késői fázisában kezdődik, amikor már nagyrészt működnie kell. Az újabb módszertanok egyre inkább hangsúlyozzák a tesztelés fontosságát és korai elkezdését. A *test-driven development (TDD)* pedig még a komponensek kifejlesztése előtt ösztönöz a tesztesetek megírására. A tesztelés tehát olyan témakör, amely mellett nem mehetünk el szó nélkül. A tesztelésnek több szintje van. A fejezet csak az assertionöket, valamint a komponensek mint egységek tesztelését, azaz közismert néven a unittesztelést ismerteti. Röviden ismertetjük a mock-technikát is.

15.1. Az assertionök

Az assertionök hétköznapi nyelven feltételezéseknek nevezhetők. A program írása során sokszor feltehető, hogy bizonyos körülmények fennállnak, illetve bizonyos esetek sosem következnek be. Az assertiönök segítségével ezeket a program lassítása nélkül tesztelhetjük. Az asserionök ugyanis csak akkor értékelődnek ki, ha a programot úgy indítjuk, hogy explicit módon engedélyezzük őket. Egyébként az alkalmazás úgy fut, mint ha bele sem írtuk volna ezeket a feltételezéseket.

Az assertionök a Hoare-logika szerint fogalmazhatók meg jól. A Hoare-logika szerint a műveleteknek csak akkor kell helyesen működniük, ha az *előfeltételek (precondition)* igazak. Ha az előfeltételek igazak, akkor a művelet hatásait vizsgálva logikailag érvelhetünk a programállapotról, és megfogalmazhatunk *utófeltételeket (postcondition)*, amelyeknek a művelet lefutása után fenn kell állniuk. Ezen kívül esetleg felfedezhetünk olyan törvényszerűségeket, amelyek a program minden pillanatában fennállnak. Utóbbiakat *invariánsoknak (invariant)* nevezzük. Ez a három fogalom rendszerszerű keretet ad ahhoz, hogy az assertionöket a programban megfogalmazzuk. A metódusok elején jelezzük, hogy az előfeltételek fennállnak, a végén pedig azt, hogy az utófeltételek igazak. Ne használjuk azonban az assertionöket a felhasználó (vagy más programozó) által szolgáltatott bemenet validálására. Ne bízzunk meg a felhasználóban, inkább validáljuk a bemenetet és váltsunk ki `IllegalArgumentException` kivételt, ha az érvénytelen. Nem publikus metódusokban viszont alkalmazhatjuk az assertionöket a paraméterben megadott értékeken.

Az invariánsoknak minden pillanatban fenn kell állniuk, néha ezeket is megfogalmazhatjuk, de természetesen ésszerűtlen lenne őket minden utasítás után megismételni. Például a metódusok végén tesztelhetjük azokat az invariánsokat, amelyekben szerepel metódus által módosított változó. Az assertionök megadásásnak két formája van. Az egyszerűbb forma így néz ki:

```
assert kif;
```

Itt *kif* egy logikai kifejezés, ezt igaznak véljük. Ha az assertion kiértékelésekor a kifejezés értéke mégis `false`, akkor `AssertionError` hibát kapunk. Az assertionök másik szintaxisával megadhatunk egy második kifejezést is, ez diagnosztikai üzenettel jelzi, hogy pontosan milyen feltevés hiúsult meg:

```
assert kif : diag_kif;
```

A diagnosztikai kifejezésnek vissza kell adnia valamilyen értéket, nem lehet például visszatérési érték nélküli metódus meghívása. A kifejezés értéke ekkor átadódik az `AssertionError` megfelelő konstruktorának, és az assertion meghiúsulásakor a virtuális gép azt is ki fogja írni.

Az assertionök alapértelmezésben ki vannak kapcsolva. Ha a megadott feltevéseket ellenőrizni szeretnénk, akkor külön be kell kapcsolni őket. Erre a **java** parancs **-ea** parancssori opciója szolgál. Ha az opciót paraméter nélkül adjuk meg, akkor a rendszer osztályain kívül az összes osztályban engedélyezi a feltételek ellenőrzését. A **-ea:*csomagnev*...** paramétermegadás csak az adott csomag összes osztályára vonatkozik, a **-ea:...** pedig az alapértelmezett csomagra. Osztályt is megadhatunk a **-ea:*OsztalyNeve*** formában. A **-da** opció segítségével tilthatjuk az assertionöket. A **-ea:*csomag*...** **-da:*csomag.Osztaly*** parancssori opciók például az egész csomagban engedélyezik a feltételek kiértékelését, de tiltják a külön megadott osztályban.

15.2. Unittesztek a JUnittal

Unittesztelésen a program jól behatárolható, elemi komponenseinek izolált tesztelését értjük. Ha fókuszált tesztesetet készítünk egy konkrét metódus különböző végrehajtási eseteinek tesztelésére, akkor a teszteset meghiúsulása esetén jól behatárolható a hibás programrész. Érdemes ezért a teszteseteket elemi komponensekhez elkészíteni. Az izoláció célja, hogy más komponensek esetleges hibái ne befolyásolják a teszt kimenetelét. Ez szintén a hiba helyének felderíthetőségét segíti elő, a tesztesetek így ugyanis csak a ténylegesen hibás komponensnél hiúsulnak meg, a hiba nem terjed tovább a rájuk épülő komponensekre. A komponenseket úgy tudjuk izolálni a függőségeiktől, hogy a polimorfizmust alkalmazva valamilyen egyszerű osztályt adunk meg azok helyett. Az egyszerű osztályt úgy írjuk meg, hogy az adott környezetben biztosan a megfelelő eredményt szolgáltassa. Az ilyen osztályok példányait *mockobjektumnak* nevezzük.

Unitteszteket könnyen írhatunk úgy, hogy egy tesztosztályban példányosítjuk a tesztelendő osztályt, meghívjuk néhány metódusát, és minden egyes hívás után ellenőrizzük, hogy ezek a várt eredményt adják-e. A JUnit keretrendszer azonban nagyban leegyszerűsíti a folyamatot. A keretrendszernek megfelelően elkészített tesztosztályokat adunk át, az pedig automatikusan végrehajtja a teszteket, és kiértékeli az eredményt. A unittesztelés elvei szerint egy tesztosztály egy funkcionális osztályt tesztel, és a tesztosztály minden metódusa egyetlen metódus egyetlen esetét kezeli. A tesztmetódusok publikusak, és nincs visszatérési értékük, valamint a `@Test` annotációval kell ellátni őket. Ez jelzi a JUnit felé, hogy tesztként kell futtatnia őket. A keretrendszer minden tesztmetódus végrehajtása előtt új példányt készít a tesztosztályból, hogy az előző teszt kimenetele semmiképpen se befolyásolja a teszt kimenetelét. Ez a mecha-

nizmus is az izolált tesztelést szolgálja. A tesztmetódusban a JUnit `Assert` osztályának statikus metódusait használva a Java-assertionökhöz hasonló feltételezéseket fogalmazhatunk meg. Az `Assert` osztály metódusait statikusan szokás importálni, a példaprogramokban is ezt tesszük. A metódusokat az alábbi lista foglalja össze.

```
static void assertArrayEquals(byte[] expected, byte[] actual)
...
static void assertArrayEquals(String message, byte[] expected,
byte[] actual)
...
```

A két tömböt azonosnak feltételezzük. Megadható diagnosztikai üzenet is, ez akkor jelenik meg, ha mégis eltérnek.

```
static void assertEquals(double expected, double actual)
...
static void assertEquals(String message, double expected,
double actual)
...
```

A két paramétert azonosnak feltételezzük. Megadható diagnosztikai üzenet is, ez akkor jelenik meg, ha mégis eltérnek.

```
static void assertFalse(boolean condition)
static void assertFalse(String message, boolean condition)
```

A feltételt hamisnak feltételezzük. Megadható diagnosztikai üzenet is, ez akkor jelenik meg, ha mégis igaz.

```
static void assertNotNull(Object expected, Object actual)
static void assertNotNull(String message, Object expected,
Object actual)
```

A két objektumot egyezőnek feltételezzük. Megadható diagnosztikai üzenet is, ez akkor jelenik meg, ha mégsem egyeznek.

```
static void assertSame(Object object)
static void assertSame(String message, Object object)
```

Az objektumot nem `null` értékűnek feltételezzük. Megadható diagnosztikai üzenet is, ez akkor jelenik meg, ha mégis `null`.

```
static void assertTrue(boolean condition)
static void assertTrue(String message, boolean condition)
```

A feltételt igaznak feltételezzük. Megadható diagnosztikai üzenet is, ez akkor jelenik meg, ha mégis hamis.

```
static void fail()
static void fail(String message)
```

Hívásakor meghiúsul a tesztmetódus. Ha megadtunk diagnosztikai üzenetet, akkor azt kiírja. Használhatjuk például lehetetlennek vélt eseteknél.

Egy tesztmetódus sikeres, ha a megadott assertionök mind teljesülnek, valamint a futása nem eredményez kivételt. Az érvénytelen bemenettel kapcsolatos eseteket is fontos tesztelni, ezért olyan tesztmetódust is készíthetünk, amely egy bizonyos

kivételtípus esetén teljesül. Ehhez az annotáció `@Test(expected=Exception.class)` formáját kell használnunk a megfelelő kivételosztállyal. Időlimitet is megadhatunk a tesztmetódushoz. Ha a tesztmetódus ezt túllépi, akkor szintén meghiúsul. Ezzel a mechanizmussal egyszerű teljesítményteszteket is készíthetünk a kritikus metódusokhoz. A teszteset gyakori lefuttatásával ellenőrizhetjük, hogy a fejlesztés során a metódus teljesítménye nem romlik-e. Ehhez a `@Test(timeout=200)` formában használjuk az annotációt, ahol az időlimitet milliszekundumokban adjuk meg.

A tesztkörnyezet beállításait (test fixture), mint a tesztelendő objektum példányosítását, az esetleges mockobjektumok létrehozását, valamint egyéb inicializációs beállításokat, külön metódusban végezhetjük el. A metódust a `@Before` annotációval kell megjelölni, publikusnak kell lennie, és nem lehet visszatérési értéke. Ez a külön metódus minden tesztmetódus lefutása előtt végrehajtódik. Ehhez hasonlóan készíthetünk tisztogatómetódust, ez felszabadítja az esetlegesen lefoglalt erőforrásokat. Ezt az `@After` annotációval kell megjelölni. Osztályszintű inicializációs és tisztogatómetódusokat is készíthetünk, ezekre ugyanazok a megkötések vonatkoznak, és a `@BeforeClass` , valamint az `@AfterClass` annotációkkal jelölhetők meg. Az alábbi lista összefoglalja a JUnit annotációit:

`@Test`
: Tesztmetódus, amely akkor sikeres, ha nem eredményez kivételt, és az összes assertion teljesül.

`@Before`
: A tesztmetódus meghívása előtt lefutó objektumszintű inicializációs metódus.

`@After`
: A tesztmetódus meghívása után lefutó objektumszintű tisztogató metódus.

`@BeforeClass`
: Osztályszintű inicializációs metódus.

`@AfterClass`
: Osztályszintű tisztogatómetódus.

`@Ignore`
: Kihagyja a tesztmetódust. Praktikus, ha egy nagyobb refaktorálás után még nem frissítettük a tesztet, hogy megfelelően együttműködjön az új kóddal.

A fejezethez készült példaprogram egy bankszámlát megvalósító osztályhoz nyújt tesztosztályt. A bankszámlának a `withdraw()` metódusa végzi a levonást, és akkor ad vissza `true` értéket, ha sikerült levonni. Ha negatív értéket adunk meg levonandó összegnek, akkor kivételt kapunk.

```
/*
 * Ez az első tesztosztály, még mockobjektum nélkül.
 */
public class BankAccountTest {
    private BankAccount account;

    @Before
```

```
    public void setUp() throws Exception {
        account = new BankAccount(1234567812345678l, 30000.0,
            3630111111l);
    }

    @Test
    public void testSuccessfulWithdraw() {
        boolean result = account.withdraw(1000.0);
        assertTrue(result);
        assertEquals("A levonás utáni összeg nem egyezik a ↺
várttal", 29000.0, account.getBalance(), 1.0);
    }

    @Test
    public void testFailedWithdraw() {
        boolean result = account.withdraw(100000.0);
        assertFalse(result);
        assertEquals("A sikertelen levonás utáni összeg nem egyezik ↺
a várttal", 30000.0, account.getBalance(), 1.0);
    }

    @Test(expected = IllegalArgumentException.class)
    public void testInvalidWithdraw() {
        account.withdraw(-1000.0);
    }
}
```

A JUnit-tesztosztályok lefordításához le kell tölteni a keretrendszer weboldaláról[1] a `junit.jar` fájlt, és a classpathhoz kell adni. Futtatáskor szintén szükség van rá. Az Eclipse automatikusan támogatja a JUnit keretrendszert, így ennek használata esetén nincs szükség további beállításokra. Egyébként a fordítás és futtatás így történik:

```
javac -cp junit.jar  TesztOsztaly.java
java -cp junit.jar org.junit.runner.JUnitCore   TesztOsztaly
```

15.3. Az EasyMock használata

Tegyük fel, hogy a fenti példában a bankszámlát megvalósító program SMS-t küld a tulajdonosnak, ha a számláról akkora összeget próbálunk levonni, hogy a fedezet nem elég a művelet elvégzéséhez. A bankszámlaosztályt az SMS-t küldő komponenstől izoláltan szeretnénk tesztelni, hogy az esetleges hibái ne befolyásolják a bankszámlaosztály tesztjeinek kimenetelét. Ráadásul az SMS-küldéshez hálózati kapcsolat és a szolgáltató által előírt beállítások is szükségesek. A fejlesztést és a tesztelést igen bonyolulttá tenné, ha ezeket a beállításokat a fejlesztők gépén is előírnánk, illetve minden

[1] http://junit.org/

bizonnyal az SMS-szolgáltatóhoz használható hozzáférési adatokat sem szeretnénk a fejlesztőknek megadni. Ezért az alábbi példában olyan mockobjektumot gyártunk, amely csupán naplózza az SMS-küldés tényét, így nem kíván előzetes konfigurációt, sosem hiúsul meg, de a kimeneten jelzi, hogy valóban meg lett-e hívva. Ez a teszt gyorsabban lefut, mivel az eredeti SMS-küldőben késleltetés szerepelt, hogy valóságosabbnak tűnjön. Ebben az esetben a helyettesíteni kívánt objektum példányosítása a tesztelendő osztályon belül van. Ez sokszor előfordul. Lehetőséget kell teremteni arra, hogy kívülről adjunk meg helyette egy példányt, és így a mockobjektumra tudjuk cserélni. Ezt megtehetjük például egy új konstruktorral vagy egy setterrel. A tesztosztályokat konvencionálisan a tesztelendő osztállyal azonos csomagba tesszük de különböző könyvtárba (az Eclipse fejlesztőkörnyezet és a build rendszerek ezt támogatják), így ez az új konstruktor vagy metódus lehet csomagszintű, tehát nem jelent veszélyt a program számára. Talán szokatlanul hangzik, hogy a kódot csupán a teszteset miatt módosítjuk, de a tesztek kiemelt fontossága miatt ez megengedhető. A megközelítés tehát nem szokatlan. Az új konstruktor és a mockobjektum bevezetése után a tesztosztály így módosul:

```
// mockobjektum, amely csak kiírja, hogy meg lett hívva
class SMSSenderMock implements SMSSender {

    @Override
    public boolean send(long number, String msg) {
        System.out.println("send(" + number + ", " + msg + ")");
        return true;
    }
}

/*
 * Ez a második tesztosztály, ez már kézzel írt
 * mockobjektumot használ.
 */
public class BankAccountTest2 {
    private BankAccount account;
    private SMSSender sender;

    @Before
    public void setUp() throws Exception {
        sender = new SMSSenderMock();
        account = new BankAccount(12345678123456781, 30000.0,
            3630111111l, sender);
    }
    ...
}
```

Mockobjektumok létrehozása nem mindig triviális feladat. Az EasyMock keretrendszer segítséget nyújt ebben. A keretrendszer legfontosabb osztálya az `EasyMock`, ezzel a mockobjektumokat létrehozhatjuk, és a viselkedésüket specifikálhatjuk. Az osztály

metódusait statikusan szokás importálni. Az EasyMock keretrendszer alapelve, hogy a mockobjektumokat a programozónak nem kell a fenti módon teljesen megírni, csupán meg kell adnia a típusát. Szintén meg kell adni, milyen metódushívásokat várjon, és milyen visszatérési értékkel reagáljon rájuk. Az így készített mockobjektum a meghatározott értékeket fogja visszaadni, valamint könyveli, hogy tényleg a megadott metódusait hívták-e a megadott paraméterekkel és a definíció sorrendjében. Alább látható a fenti példa EasyMock keretrendszerre átdolgozott változata. A statikus `expect()` metódust használjuk a várt metódushívások specifikálására. A metódus paraméterébe a meghívni kívánt metódust kell írni annak paramétereivel vagy helyettesítésekkel. A helyettesítéseket statikus metódusokkal adjuk meg, például az `anyString()` tetszőleges karakterláncot jelent. Az `expect()` visszatérési értékén az `andReturn()` vagy `andThrow()` metódus hívásával adható meg, hogy a mockobjektum metódusának milyen értékkel kell visszatérnie, vagy milyen kivételt kell eredményeznie. Ez a fajta megadási mód elsőre szokatlannak tűnhet, mert nem konvencionális módon használja a metódushívásokat, de valójában nagyon közel áll az emberi gondolkodásmódhoz, ezért az elsajátítása után könnyen használható. A `replay()` metódus hívásával jelezzük, hogy a mockobjektum működésének specifikációját befejeztük. Ezután már hívhatók a mockobjektum metódusai. Végül a `verify()` metódussal tudjuk ellenőrizni, hogy tényleg a specifikációban megadott metódusok hívódtak-e meg:

```
public class BankAccountTest3 {
    private BankAccount account;
    private SMSSender sender;

    @Before
    public void setUp() throws Exception {
        sender = createMock(SMSSender.class);
        account = new BankAccount(1234567812345678l, 30000.0,
            3630111111l, sender);
    }

    @Test
    public void testFailedWithdraw() {
        // a mockobjektum bármilyen long és String értékkel fogad
        // metódushívást a send() metóduson, és true
        // értékkel tér vissza
        expect(sender.send(anyLong(), anyString())).andReturn(true);

        // beállítás kész, innentől lehet hívni a mockobjektumot
        replay(sender);
        boolean result = account.withdraw(100000.0);
        assertFalse(result);
        assertEquals("A sikertelen levonás utáni összeg nem egyezik ↺
a várttal", 30000.0, account.getBalance(), 1.0);
```

```
        // ellenőrizzük, hogy tényleg meg lett-e hívva, amit
        // előírtunk
        verify(sender);
    }
    ...
}
```

Ha az EasyMock keretrendszert is használjuk a tesztosztályban, akkor ezt is le kell tölteni[2], valamint fordításkor és futtatáskor az easymock-*x.y*.jar fájlt is a classpath-hoz kell adni. Itt *x.y* a keretrendszer verziószáma. A fejezet példaprogramja Maven-projekt, tehát a Maven keretrendszerrel a függőségeket könnyen le tudjuk tölteni. Az Eclipse fejlesztőkörnyezetbe importáláskor ez automatikusan megtörténik.

[2] http://easymock.org/Downloads.html

TIZENHATODIK FEJEZET
Az alkalmazások terjesztése

A fejlesztő az alkalmazást általában kényelmesen tudja futtatni a fejlesztőkörnyezet segítségével. A szoftver lefordítása után a **java.exe** hívásával vagy parancsfájl segítségével is futtatható. A felhasználók számára ezek a módszerek nem elégségesek, a felhasználók ugyanis nem feltétlen tudják, hogyan lehet a `.class` fájlokból álló Java-alkalmazást futtatni. Ebben a fejezetben ezért ismertetjük az alkalmazások terjesztésének felhasználóbarátabb módjait. Általában a szoftverrel együtt tölthető le a dokumentáció is, ezért a fejlesztői dokumentáció készítését is tárgyaljuk.

16.1. A Javadoc

A Javadoc technológia a dokumentációnak a forráskódhoz való csatolását teszi lehetővé, hogy később fejlesztői dokumentációt hozzunk létre belőle. Fejlesztői dokumentáción azt a szöveget értjük, amely programozási szempontból dokumentálja a program működését. Ez a dokumentáció tehát a csomagok, az osztályok, a konstruktorok, a metódusok és a tagváltozók szerepét írja le. Konstruktorok és metódusok esetén dokumentálhatók a paraméterek és a kiváltott kivételek, illetve metódusnál a visszatérési érték is. A forráskódhoz csatolt dokumentációt olyan speciális többsoros megjegyzésben adjuk meg, amely `/**` karakterekkel kezdődik, azaz egy helyett két csillagot tartalmaz az elején. Ezeket a megjegyzéseket mindig az elé az elem elé írjuk, amelyre vonatkoznak. A Javadoc-megjegyzésekbe alapvetően egyszerű szöveget írunk, de használhatók ún. Javadoc- és HTML-jelölőelemek is. Előbbiek `@` jellel kezdődnek, és valamilyen információt közölnek az utánuk következő paraméterről. Így adjuk meg például a készítő programozó nevét az `@author` jelölőelemmel. Egyes jelölőelemeket kapcsos zárójelben kell megadni, ilyen például a `@link` jelölőelem, ezzel hivatkozásokat helyezhetünk el a szövegben. A Javadoc-megjegyzésekből szinte mindig HTML-formátumú dokumentációt állítunk elő, ezért megadhatók HTML-jelölőelemek is, ezek bekerülnek a kimenetbe. Az alábbi példa mutat be egy osztályhoz kapcsolt Javadoc-megjegyzést:

```
/**
 * Ez a program főosztálya.
 *
 * @author Kövesdán Gábor
 *
 */
public class Main {
    ...
}
```

Metódusok esetén elengedhetetlen a paraméterek szerepének, a visszatérési értéknek, illetve a kiváltott kivételeknek a dokumentációja. Az alábbi programrészlet a metódusok dokumentációját szemlélteti, megfigyelhetők benne a leggyakrabban használt Javadoc-jelölőelemek is:

```
/**
 * A metódus a <em>napszaknak megfelelő</em> üdvözletet készít a
 * megadott felhasználó számára. Az időt {@link Calendar} objektum
 * segítségével adjuk meg. Ha a név <code>null</code>, akkor kimarad
 * a megszólítás.
 *
 * @param time
 *            az aktuális idő.
 * @param name
 *            a felhasználó neve.
 * @return a személyre szabott üdvözlő szöveg.
 *
 * @throws NullPointerException
 *             ha az idő helyett <code>null</code>t adunk meg.
 *
 * @see Calendar
 */
public static String greet(Calendar time, String name) {
    ...
}
```

Csomagokhoz is írhatunk dokumentációt. A csomagok azonban nem egy helyen vannak definiálva, hanem azokat a hozzájuk rendelt interfészek, osztályok és enumerációk alkotják. Ezért a Java nyelv a `package-info.java` fájlt vezette be, amelyet a csomag könyvtárában helyezhetünk el. A tartalma csupán a `package` utasításból és az azt megelőző Javadoc-megjegyzésből áll. Erre alább látunk példát:

```
/**
 * Ez egy nagyon egyszerű csomag, amelyben Helló, Világ! programot
 * valósítottunk meg.
 *
 * @author Kövesdán Gábor
 * @version 1.0
 */
package hello;
```

Miután megírtuk a Javadoc-megjegyzéseket, a komplett fejlesztői dokumentációt a **javadoc** segédprogrammal vagy az Eclipse fejlesztőkörnyezettel (lásd B függelék) hozhatjuk létre. Az alábbi példa bemutatja a **javadoc** segédprogram futtatását. Célkönyvtárnak a `html` könyvtárat adjuk meg, a forráskódok pedig az `src` könyvtárban vannak. Alapértelmezésben a program `public` és `protected` láthatóságú tagok do-

kumentációját állítja elő, a példa azonban a **-public** parancssori paraméterrel csak a publikus tagokat választja ki. Előírjuk még, hogy a verziószámot és a szerzőt tartalmazó Javadoc-jelölőelemeket is vegye figyelembe. Végül megadjuk, hogy a `hello` csomag dokumentációja jöjjön létre:

```
C:\__work\javabook_ws\F16>javadoc.exe -d html -sourcepath src -public
 -version -author hello
Loading source files for package hello...
Constructing Javadoc information...
Standard Doclet version 1.7.0_25
Building tree for all the packages and classes...
Generating html\hello\Main.html...
Generating html\hello\package-frame.html...
Generating html\hello\package-summary.html...
Generating html\hello\package-tree.html...
Generating html\constant-values.html...
Building index for all the packages and classes...
Generating html\overview-tree.html...
Generating html\index-all.html...
Generating html\deprecated-list.html...
Building index for all classes...
Generating html\allclasses-frame.html...
Generating html\allclasses-noframe.html...
Generating html\index.html...
Generating html\help-doc.html...
```

A **javadoc** segédprogram lehetséges opcióit megtekinthetjük, ha a programot parancssori paraméterek nélkül hívjuk, de ezeket az online dokumentációban[1] is olvashatjuk.

16.2. A Java Archive (JAR)

A Java-alkalmazások és -osztálykönytárak szabványos *JAR formátumú* csomagban terjeszthetők. A JAR-csomag valójában egyszerű ZIP formátumú tömörített fájl, amelynek `.jar` kiterjesztést adunk. A JAR-fájl tartalmazhatja a lefordított osztályok `.class` fájljait, a forráskódjukat, illetve a Javadoc-dokumentációt is. A csomag tartalmazhat opcionális leírót is, ez a csomagról közölhet információt, például a készítőjét, a függőségeit, illetve ha futtatható csomagot készítünk, akkor a futtatandó osztály nevét. Ezt a leírót a csomagon belül a `META-INF` alkönyvtárban `MANIFEST.MF` néven kell elmenteni. Ha van a csomagban leíró, akkor a tömörített fájlhoz ezt a fájlt kell elsőként hozzáadni. Alább láthatunk egy példát a leíró formátumára:

[1] http://docs.oracle.com/javase/7/docs/technotes/tools/windows/javadoc.html

```
Manifest-Version: 1.0
Main-Class: hello.Main
```

Mivel a JAR-csomag valójában ZIP-fájl, elméletileg elkészíthetnénk bármilyen tömörítőprogrammal, amely képes ilyen formátumban tömöríteni. A JDK azonban a **jar** parancssoros segédprogramot biztosítja a JAR-csomagok könnyű elkészítéséhez. Az Eclipse fejlesztőkörnyezet is támogatja projektek JAR-csomagként való exportálását (lásd B függelék). A **jar** segédprogram parancssori opciói hasonlítanak a UNIX-típusú operációs rendszerek **tar** segédprogramjához. Először a parancssori opciókat adjuk meg, majd sorban azok paramétereit. Például az **f** opció a JAR-fájl nevét várja, az **e** pedig a futtatandó osztályt, hogy ha futtatandó JAR-fájlt kívánunk készíteni. Az alábbi parancssor elkészíti a fejezetben bemutatott példa JAR-csomagját. A lefordított `.class` fájloknak a `hello` könyvtárban kell lenniük, és a futtatandó osztályt is a csomagnévvel kvalifikált formában adjuk meg. A **v** opció hatása, hogy a segédprogram részletesen listázza, mi került a JAR-fájlba.

```
jar.exe cvfe hello.jar hello.Main hello
added manifest
adding: hello/(in = 0) (out= 0)(stored 0%)
adding: hello/Main.class(in = 1362) (out= 796)(deflated 41%)
adding: hello/package-info.class(in = 111) (out= 95)(deflated 14%)
```

A JAR-fájlokat digitális aláírással is el lehet látni, hogy hitelességüket tanusítsuk. Az aláíráshoz először egy kulcsra van szükségünk, amellyel a fájlt aláírjuk. Üzleti környezetben általában olyan kulcsot használunk, amelyet tanúsító hatóság is aláírt, az aláíró személyazonossága ugyanis csak így garantálható. Ez a megoldás azonban drága, ezért sokszor még vállalatoknál sincs a kulcs hitelesítve. Appletek és WebStart alkalmazások futtatásánál hiteles kulcs esetén a program alapértelmezésben megkapja a biztonsági engedélyeket, hitelesítetlen kulcs esetén pedig egy ablak ugrik fel, amely figyelmezteti a felhasználót, és felajánlja a kulcs elfogadását vagy elutasítását. Az alábbi példában is ilyen kulcsot használunk. Ehhez először egy *kulcstárat (keystore)* kell létrehoznunk:

```
keytool.exe -genkey -keystore testStore -alias testKey
Enter keystore password:
Re-enter new password:
What is your first and last name?
  [Unknown]:  Gábor Kövesdán
What is the name of your organizational unit?
  [Unknown]:  n.a.
What is the name of your organization?
  [Unknown]:  n.a.
What is the name of your City or Locality?
  [Unknown]:  Budapest
What is the name of your State or Province?
  [Unknown]:  Budapest
```

```
What is the two-letter country code for this unit?
  [Unknown]:  HU
Is CN=Gábor Kövesdán, OU=n.a., O=n.a., L=Budapest, ST=Budapest, C=↺
HU correct?
  [no]:  yes

Enter key password for <testKey>
        (RETURN if same as keystore password):
```

Először a jelszót kell megadnunk, amellyel a kulcstárat a későbbiekben elérhetjük, majd a kulcs tulajdonosának adatait kell beírni. Végül a kulcs védelméhez is megadható jelszó, de ha nem írunk be semmit, akkor egyezni fog a kulcstár jelszavával. Ezután a fájl aláírása a következő paranccsal történik:

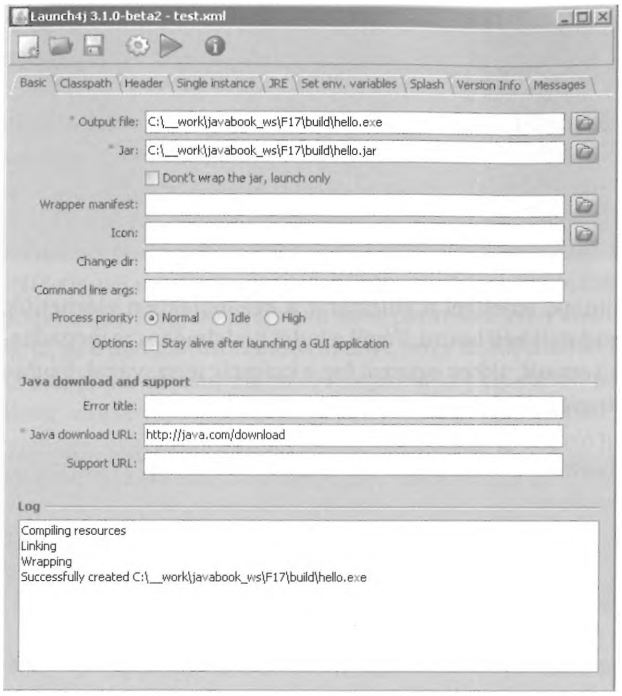
```
jarsigner.exe -keystore testStore hello.jar testKey
Enter Passphrase for keystore:

Warning:
The signer certificate will expire within six months.
```

A parancs futtatása után meg kell adnunk a kulcstár jelszavát. Az aláírás ezután elkészül, de figyelmeztetést kapunk, hogy a tanúsítvány hat hónap múlva lejár. Ez ugyanis az elkészített kulcs alapértelmezett érvényességi ideje. Ha ennél hosszabb időre van szükség, akkor a kulcs létrehozásánál azt is meg kell adni.

16.3. EXE-fájlok készítése

A JAR-csomagok futtathatóvá tehetők, ha a leíróban megadjuk a futtatandó osztályt. Megfelelő beállítások esetén a Windowsban dupla kattintással is futtathatjuk az ilyen JAR-fájlokat. Fejlesztőként azonban nem mindig bízhatunk abban, hogy a felhasználók gépén az ehhez szükséges beállítások el vannak végezve, sőt abban sem, hogy a felhasználó tudja, hogy mire szolgálnak a `.jar` kiterjesztésű fájlok. Az EXE-fájlok mindig indítható programként jelennek meg, és a programot azonosító ikont is magukban foglalhatják. Az osztálykönyvtárak a Java-fejlesztőknek szólnak, ezért esetükben jól alkalmazhatók a JAR-csomagok. Futtatható program esetén azonban megfontolandó a program EXE-formában történő terjesztése. A Launch4j[2] nyílt forráskódú segédprogram JAR-csomagokból képes futtatható EXE-fájlokat készíteni. A segédprogram kiegészítő funkciókat is bele tud építeni a futtatható fájlba, mint például a telepített Java-verzió ellenőrzését vagy betöltő képernyő megjelenítését. A futtatható fájl elkészítéséhez meg kell adni a JAR-fájlt, illetve a kimeneti EXE-fájl helyét, ahogyan azt a 16.1. ábra mutatja. Ezen kívül szükséges a minimális Java-verzió beállítása a *JRE* fülön. A verziószámot `1.7.0` formában kell megadni.

[2] http://launch4j.sourceforge.net/

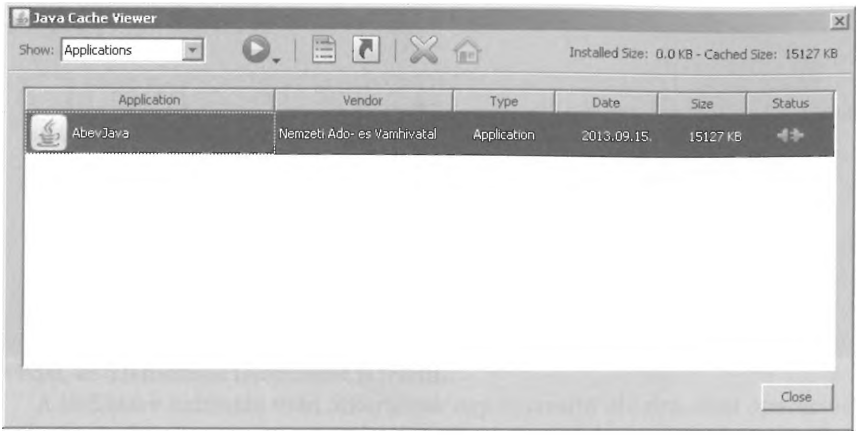


16.1. ábra: EXE-fájl készítése a Launch4j segédprogrammal

16.4. A Java WebStart

A terjesztés másik egyszerű módja a WebStart használata. A WebStart technológia XML-leírót használ az alkalmazás adatainak tárolására, amelyek segítségével az interneten fellelhető és futtatható. Ezért a leírót a program weboldalán elhelyezve a felhasználó egy kattintással elindíthatja a programot. A JRE a leíró alapján megkeresi az osztályokat, lokális tárba tölti le őket, majd elindítja a programot. Az osztályok a tárban maradnak, ezért az alkalmazás hálózati kapcsolat nélkül is futtatható marad. Az alkalmazástárat a Java Cache Viewer segédprogrammal tudjuk megtekinteni. Windows operációs rendszeren ez a Vezérlőpulton keresztül érhető el, a Java által internetről letöltött fájloknál. Ezt a 16.2. ábra szemlélteti.

A segédprogramban a fenti ikonok segítségével futtathatjuk az alkalmazást, megnézhetjük a leírófájlt, illetve parancsikont készíthetünk az asztalon. A WebStart használatához csupán JAR-formátumba kell csomagolni az alkalmazást, meg kell írni az XML-formátumú leírót, majd elérhetővé kell tenni őket a Weben. A leírófájlt *JNLP-fájlnak* is szokás nevezni, mert a feldolgozó protokoll neve *Java Network Launching Protocol (JNLP)*. A fájlt `.jnlp` kiterjesztéssel mentjük el, és a webszerver pedig `application/x-java-jnlp-file` MIME-típussal küldi el a böngészőnek. Az operációs rendszer ugyanis a MIME-típus alapján azonosítja be, milyen programmal kell az internetről származó fájlokat kezelnie.



16.2. ábra: Java Cache Viewer

Az alábbi JNLP-fájl a konfigurációs lehetőségeket szemlélteti. A fájl alapján könnyen elkészíthető a saját alkalmazás leírója is. Alább láthatjuk a JNLP-fájl főbb elemeit, a teljesség igénye nélkül. Az egyes elemek szerepét XML-megjegyzések írják le.

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

<!-- A saját webcímet kell beírni, illetve a JNLP-fájl nevét -->
<jnlp spec="1.0+" codebase="http://example.com/" href="hellov.jnlp">

  <!-- Általános információk -->
  <information>
    <!-- A program neve -->
    <title>Helló, Világ!</title>
    <!-- Készítő -->
    <vendor>Kövesdán Gábor</vendor>
    <!-- A programhoz tartozó ikon -->
    <icon href="vilag.jpg"/>
    <!-- Indítható hálózati kapcsolat nélkül -->
    <offline-allowed/>
  </information>

  <!-- Függőségek -->
  <resources>
    <!-- A szükséges Java-verzió legalább 7-es -->
    <j2se version="1.7+" href="http://java.sun.com/products/autodl/
j2se"/>
    <!-- A futtatandó JAR-fájl -->
    <jar href="hellov.jar" main="true"/>
  </resources>

  <!-- A futtatandó osztály -->
```

```
  <application-desc main-class="Main"/>

  <!-- Háttérben, transzparensen végezzen frissítést -->
  <update check="background"/>

  <!-- Minden biztonsági engedélyt megadunk neki, ehhez alá kell ↺
írni a JAR-fájlt -->
  <security>
    <all-permissions/>
  </security>
</jnlp>
```

A FÜGGELÉK

A JDK telepítése

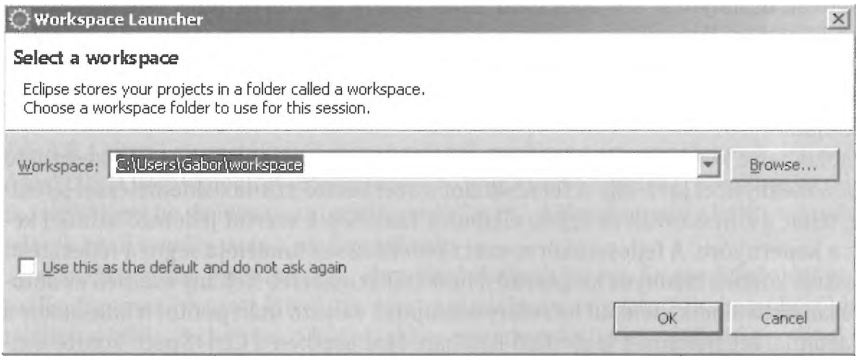
A Java aktuális verziója a `http://www.oracle.com/technetwork/java/javase/downloads/index.html` oldalról tölthető le. A könyv írásakor a legfrissebb verzió *Java Platform (JDK) 7u51*-ként szerepel a honlapon. A verziószám után az *u51* a frissítések számát jelenti, kritikus hiba vagy biztonsági sebezhetőség után ugyanis az Oracle kiad egy javítást, és a letölthető telepítőket is frissíti.

A letöltésre kattintás után átkerülünk egy összesítő oldalra, ahol operációs rendszer és platform szerint csoportosítva találjuk a letöltési lehetőségeket. Windows esetén futtatható telepítőt tölthetünk le, amely végigkísér a telepítés egészén, csupán futtatnunk kell. Linux és Mac OS X operációs rendszerekhez szoftvercsomagok tölthetők le, ezek az operációs rendszer csomagkezelő szoftverével telepíthetők a számítógépre.

A telepítés után érdemes a `JAVA_HOME` környezeti változóban beállítani a Java telepítési könyvtárának elérési útját, ugyanis számos program felhasználja ezt az információt. Ezek a programok nem fogják megtalálni a telepített JDK-t, ha a környezeti változó nincs beállítva. Szintén célszerű a `PATH` változóhoz hozzáfűzni a telepítési könyvtár `bin` alkönyvtárát. Ha ezt megtesszük, akkor parancssorban a Java segédprogramjainak hívásához nem kell begépelnünk a teljes elérési utat, csak a segédprogram nevét.

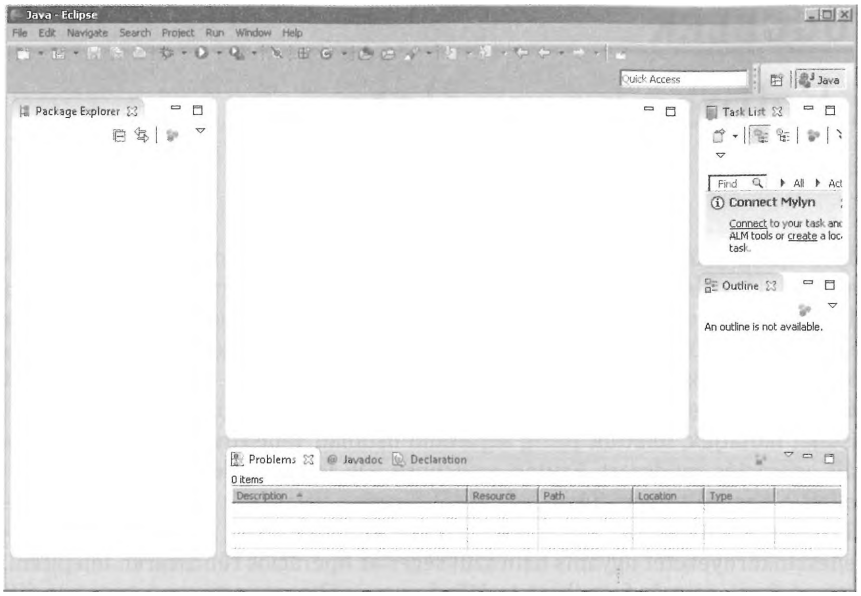
B FÜGGELÉK
Az Eclipse használata

Az Eclipse fejlesztőkörnyezet a `http://www.eclipse.org/downloads/` weboldalról tölthető le. A link a legfrissebb verzió letöltéséhez vezet, ez a könyv írásának pillanatában a 4.3.1-es, és a Kepler kódnevet viseli. A fejlesztőkörnyezet moduláris felépítésű, a bővítéseit később könnyen telepíthetjük és frissíthetjük, de a könnyebbség kedvéért a készítők több különböző disztribúciót kínálnak letöltésre. Ezek adott célokra összeállított bővítéseket tartalmaznak. A linket megnyitva a weboldalon a disztribúciók listáját láthatjuk, felettük pedig legördülő listában választhatjuk ki az általunk használt operációs rendszert. Ezután az Eclipse IDE for Java Developers disztribúció mellett a használt operációs rendszertől függően válasszuk ki a 32 vagy 64 bites változatot. A választástól függetlenül mindig egy tömörített fájlt tudunk letölteni. A fejlesztőkörnyezetet ugyanis nem szükséges az operációs rendszeren telepíteni, elég kitömöríteni, és már futtatható is. Windowsos változat esetén a futtatandó fájl neve `eclipse.exe`, a Linux és a Mac OS X esetén csupán `eclipse`. A fájl futtatása után hamarosan megjelenik a betöltő képernyő, majd a munkaterület-választó, ahogyan a B.1. ábra mutatja.



B.1. ábra: Az Eclipse munkaterület-választó ablaka

A *munkaterület (workspace)* projekteket és hozzájuk tartozó beállításokat tartalmaz. Egyrészt összefogja a projekteket, amelyeken egyszerre vagy azonos beállításokkal kívánunk dolgozni. Például külön munkaterületet hozhatunk létre különböző cégeknek készülő fejlesztésekhez, valamint hobbiprojektjeinkhez. Másrészt a munkaterület a beállításokat is ezekhez a projektekhez rendeli. Például a céges verziókezelő rendszerek vagy a kód formázásának beállításai is a munkaterület részeként mentődnek. A munkaterület összes adata egyetlen könyvtárban tárolódik a lemezen. Ha új könyvtárat adunk meg, akkor egy új, üres munkaterület jön létre. Adjunk meg tehát egy nekünk tetsző elérési utat, és indítsuk el a fejlesztőkörnyezetet az **OK** gombra kattintva. A fejlesztőkörnyezet ablaka hamarosan megjelenik, és az üdvözlőképernyő fülét bezárva a B.2. ábra szerinti elrendezést láthatjuk.



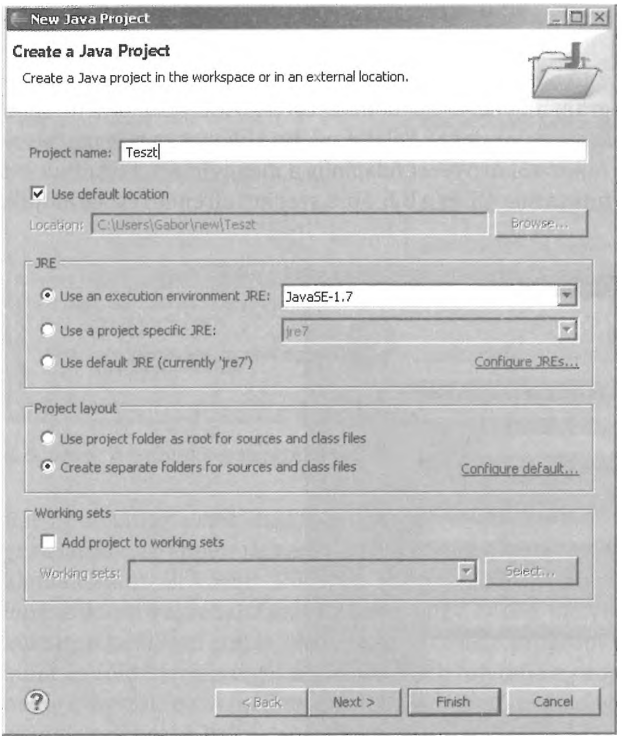
B.2. ábra: Az Eclipse fejlesztőkörnyezet ablaka

A fejlesztőkörnyezet ablakában kétoldalt és alul különböző *nézeteket (view)* látunk. A bal oldalon a *Package Explorer* nézet mutatja a projekteket és az azokban található csomagokat, osztályokat stb. Alul több nézet látszik egyszerre, füles elrendezésben. Többek között itt láthatók a projektek fordítási hibái és figyelmeztetései *(Problems nézet)*. Jobb oldalt található az *Outline* nézet, ez az éppen szerkesztett fájl elemeiről ad vázlatos nézetet. Ennek segítségével a fájlban könnyen megtalálhatjuk az elemek definícióját.

Az ablakban a központi helyet a *szerkesztő (editor)* foglalja el. Ebben szerkeszthető az éppen megnyitott Java-fájl. A forráskódot a szerkesztő szintaxiskiemeléssel jeleníti meg, tehát a kulcsszavak és egyéb elemek a funkciójuk szerint jellemző színnel kerülnek a képernyőre. A fejlesztőkörnyezet *Content Assist* funkciója segíti a fejlesztést: szerkesztés közben bizonyos kiegészítési funkciókat ajánl fel. Néhány esetben ez automatikusan aktiválódik, például ha referenciatípusú változó után pontot írunk, akkor a tagjai azonnal megjelennek legördülő listában. Más esetben a Ctrl+Space kombinációval kérhetünk kiegészítési felajánlást.

Új projekt létrehozása

Új projektet a **File** menü **New** pontjával hozhatunk létre. Válasszuk a **Java project** opciót. Ekkor egy ablak jelenik meg, amelyben megadhatjuk a projekt jellemzőit. Ezt a B.3. ábra mutatja. A projekt nevét mindenképpen ki kell tölteni. Ez a példában `Teszt`. A **Next >** gombra kattintva megadhatjuk a projekt függőségeit, jelen esetben erre azonban nincs szükség, ezért kattintsunk a **Finish** gombra.



B.3. ábra: Új projekt létrehozása

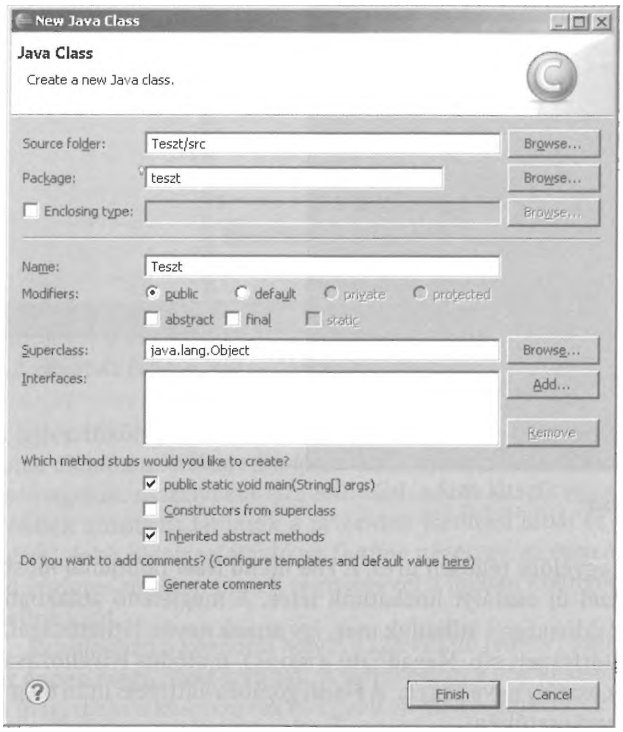
Ekkor a projekt létrejön, de egyelőre teljesen üres. A **File** menü **New** pontjánál most válasszuk a **Class** opciót, ezzel új osztályt hozhatunk létre. A megjelenő ablakban (lásd B.4. ábra) az osztály tulajdonságait adhatjuk meg, így annak nevét, láthatóságát, ősosztályát, implementált interfészeit stb. Megadható a `main()` metódus létrehozása is. Jelöljük ezt be, és legyen az osztály neve *`Teszt`*. A **Finish** gombra kattintás után megjelenik az új osztály váza a szerkesztőben.

A szerkesztőbe írjuk be a B.5. ábra által ábrázolt kódot. Ez egy üdvözlést és az aktuális dátumot hivatott kiírni, de most szándékosan elrontottuk, hogy később megtaláljuk a hibát. Ha beírtuk a kódot, akkor a szerkesztő felett található gombbal futtathatjuk a programot. A futtatás gomb zöld körben ábrázol egy fehér háromszöget. Ha a gombra kattintunk, akkor alul megnyílik a *Console nézet,* ebben a program kimenetét olvashatjuk.

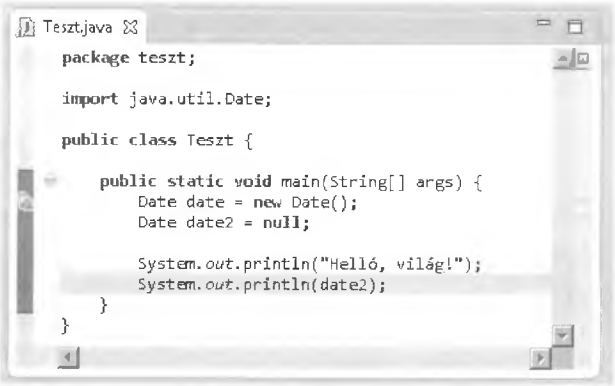
A hibakeresés

A fejlesztőkörnyezet a megjelenített nézeteket, szerkesztőket és ezek elhelyezkedését előre elkészített *perspektívákban (perspective)* fogja össze. A letöltött disztribúcióban alapértelmezésben a Java perspektívát látjuk, fent ezt tekintettük át. A megnyitott perspektívák között a jobb felső sarokban található perspektívaválasztóval válthatunk, illetve új perspektívát is itt nyithatunk meg. A *Debug perspektíva* például hibakeresésre szolgál, és a fejlesztett program hibakeresési módban történő futtatásakor

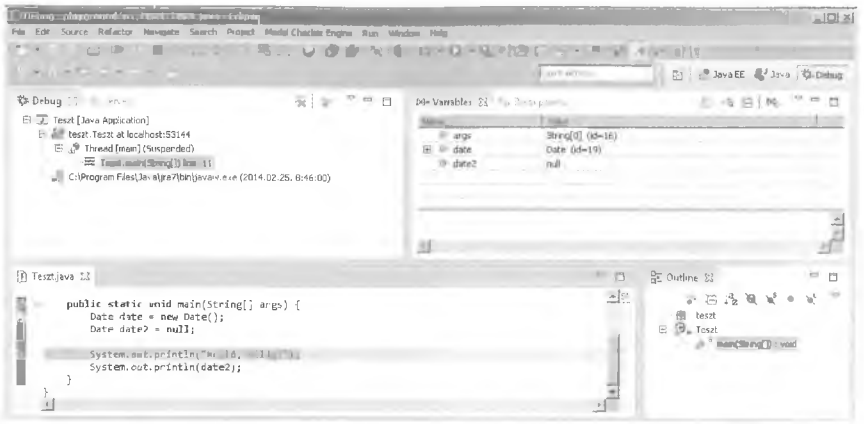
használható. Ennek kipróbálásához tegyünk először egy *töréspontot (breakpoint)* a program `main()` metódusára. Ezt úgy tehetjük meg, hogy a metódus első sorában jobb gombbal kattintunk a szerkesztő bal oldalán található szürke sávon, és a **Toggle Breakpoint** opciót választjuk. A program hibakeresési módban történő indításához a futtatás gombja melletti, bogarat ábrázoló gombra kell kattintani. Ha ekkor nem a *Debug perspektívában* vagyunk, akkor a fejlesztőkörnyezet felajánlja a megnyitását. Fogadjuk ezt el. A töréspontnál a program futása megáll, és a B.6. ábra szerinti elrendezést láthatjuk.



B.4. ábra: Új osztály létrehozása



B.5. ábra: A kódszerkesztő ablak



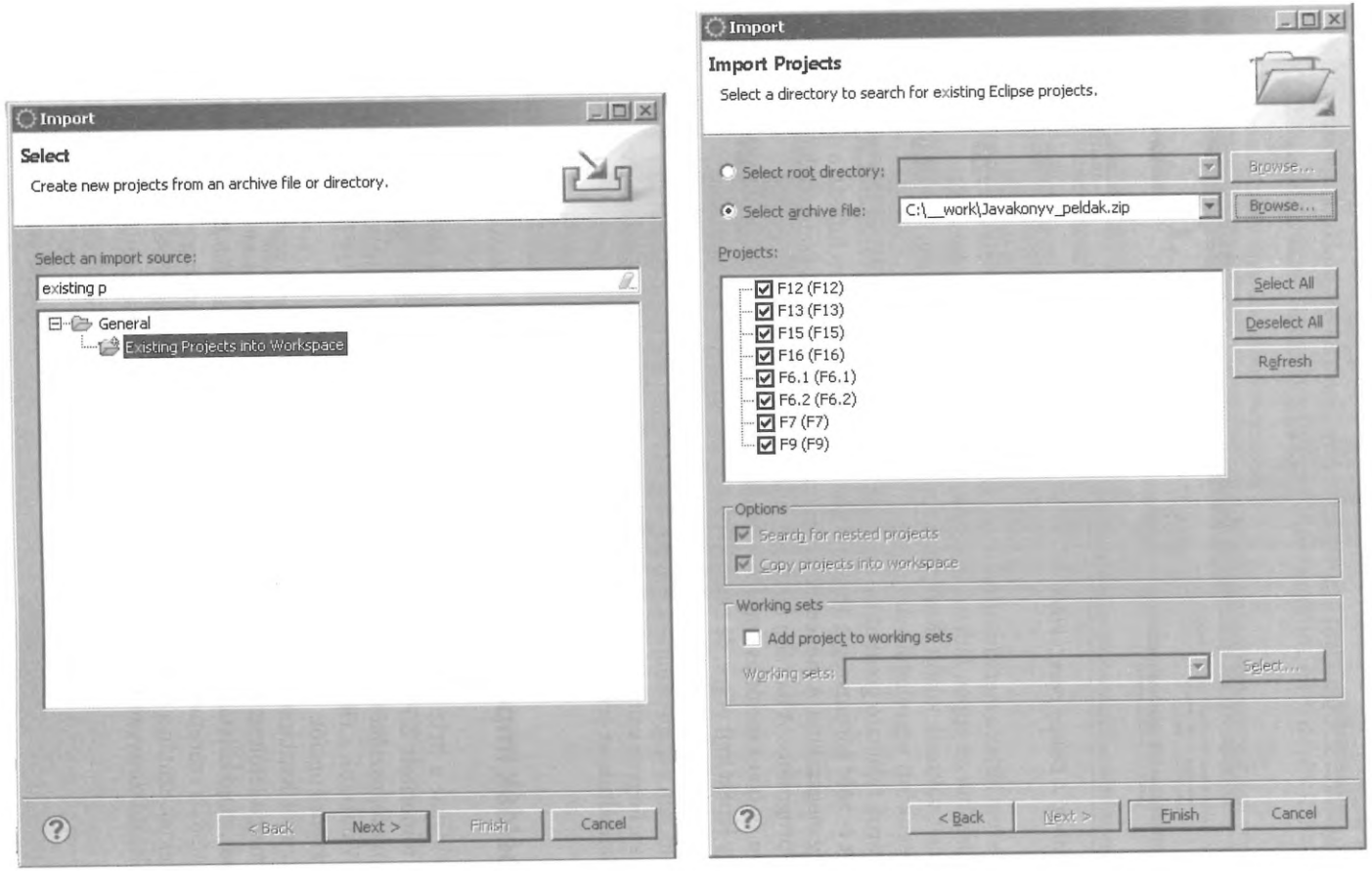
B.6. ábra: A Debug perspektíva

Balra fent látjuk a *metódushívási vermet (call stack)*, jobbra pedig az egyes füleken böngészhetünk az éppen látható változók, a beállított töréspontok, illetve a figyeléshez kiemelt változók között. Középen balra található a szerkesztő, ez töréspontnál vagy lépésenkénti végrehajtásnál mutatja, hogy éppen melyik kódsornál tartunk. Az eszköztáron található gombokkal vezérelhetjük a program végrehajtását. A sárga téglalapot és zöld háromszöget ábrázoló gomb folytatja a program futását, a piros négyzet pedig megszakítja. A mellettük jobbra található nyilakkal lépésenként hajthatjuk végre a programot. A balrább található, hurok nélküli sárga nyíl egy újabb utasítást hajt végre, és ha ez metódus- vagy konstruktorhívás, akkor bele is lép abba. A hurkos nyíl is egy lépést hajt végre, de nem lép bele a metódusok és a konstruktorok kódjába, hanem egyszerre végrehajtja azokat. Használjuk most a hurkos nyilat, hogy a program utasításain végiglépkedjünk. Minden egyes lépés után megfigyelhetjük a *Variables* nézetben, hogyan változik a program állapota. Itt feltűnik, hogy a dátumot valójában nem inicializáltuk, az értéke végig `null` volt. Ennek alapján már kijavíthatjuk a hibát.

A példák importálása

A példákat a `http://szak.hu/java/peldak.zip` weboldalról tölthetjük le. A példák egy tömörített ZIP-fájlban vannak. Ezt mentsük el a számítógépen. Indítsuk el az Eclipse fejlesztőkörnyezetet a munkaterülettel, amelybe a példákat importálni kívánjuk. Válasszuk a **File** menüből az **Import...** lehetőséget, majd az *Existing Projects into Workspace* opciót.

Ezután kattintsunk a **Next >** gombra, majd a következő párbeszédablakban adjuk meg a letöltött ZIP-fájl elérési útját. A fejlesztőkörnyezet megvizsgálja a fájl tartalmát, majd kilistázza az abban található projekteket. Itt kiválaszthatjuk, mely fejezetek példáit szeretnénk importálni. Végül kattintsunk a **Finish** gombra. A kiválasztott projektek ezután a munkaterületre másolódnak, és szerkeszthetők, illetve futtathatók a fejlesztőkörnyezetből. Az importálás menetét a B.7. ábra szemlélteti.



B.7. ábra: Projektek importálása

Néhány példa külső osztálykönyvtártól függ, ezért ezek típusa Maven-projekt. A Maven egy build rendszer, amely az alkalmazás függőségeinek letöltésére, az alkalmazás lefordítására, unittesztek futtatására stb. használható. A Maven-projektek jól integrálódnak az Eclipse fejlesztőkörnyezetbe, ezért a Maven rendszer ismerete nem szükséges a példák futtatásához.

Javadoc-dokumentáció létrehozása

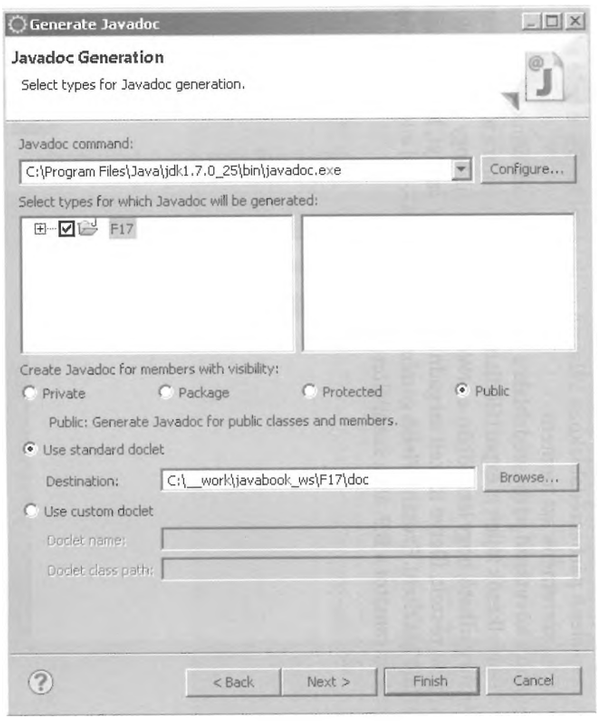
Kattintsunk jobb gombbal a projekten, amelynek a Javadoc-dokumentációját el szeretnénk készíteni, majd válasszuk az **Export...** lehetőséget. Keressük ki a *Javadoc* opciót, majd kattintsunk a **Next >** gombra. Itt meg kell adnunk a **javadoc** program helyét. Ennek a JDK telepítési helyén, a `bin` könyvtárban kell lennie. Megadhatjuk továbbá, hogy hova kerüljön az előállított dokumentáció, illetve hogy milyen láthatóságú tagokról jöjjön létre.

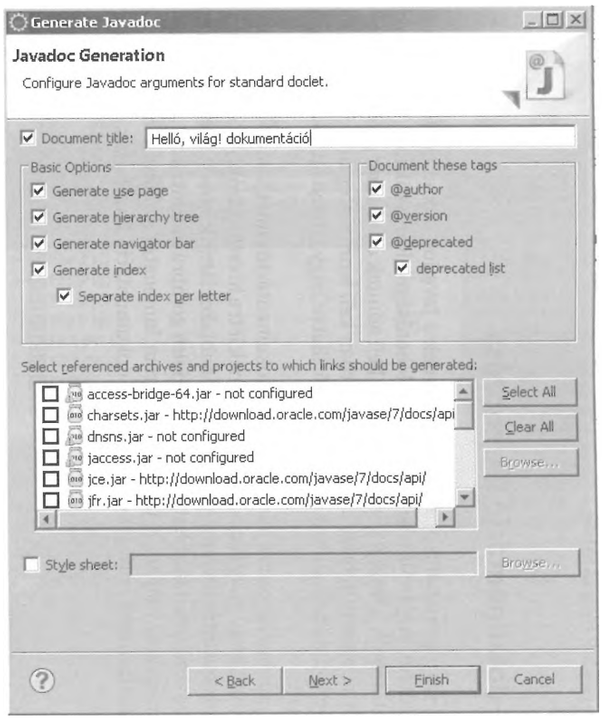
A **Next >** gombra kattintva megadhatjuk a dokumentáció címét, a létrehozandó elemeket, valamint hogy mely csomagokhoz jöjjenek létre hivatkozások. A **Next >** gombra történő újboli kattintás után még egy párbeszédablakot látunk, ebben a **javadoc** program hívási paramétereit szabhatjuk testre. A **Finish** gombra kattintva a dokumentáció elkészül a megadott helyen. A **javadoc** program kimenetét a *Console* nézetben láthatjuk, a hibaüzenetek is itt jelennek meg. A dokumentáció létrehozását a B.8. ábra mutatja be.

JAR-fájl készítése

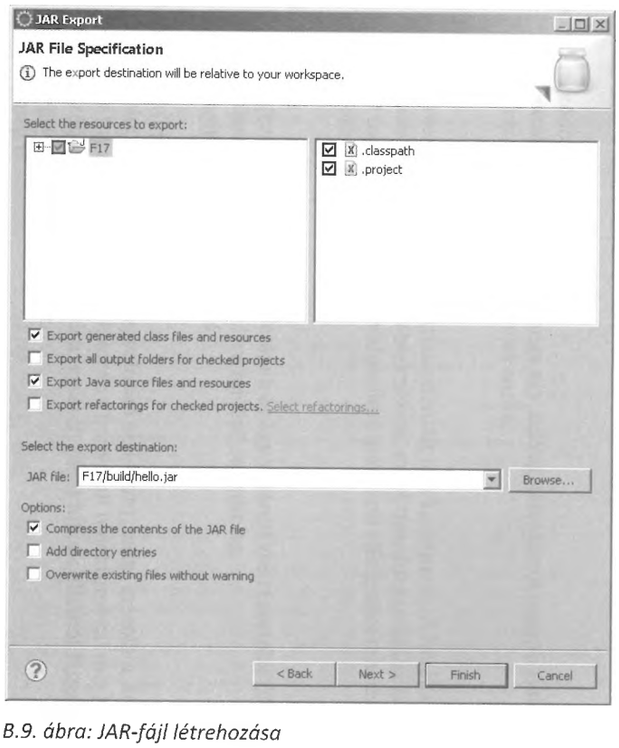
JAR-fájl készítéséhez szintén az **Export...** menüpontot kell választanunk, de most a *JAR file* opciót. A megjelenő párbeszédablakban választhatjuk ki, mi kerüljön bele a JAR-fájlba, illetve hova szeretnénk azt menteni.

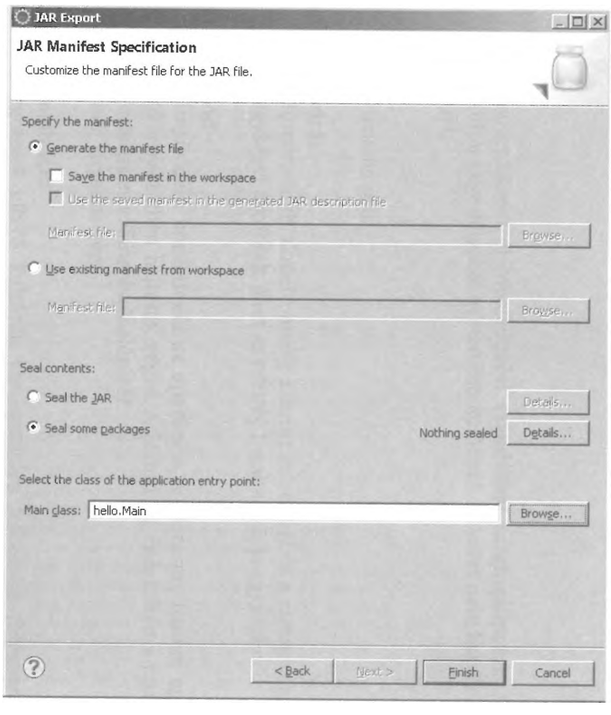
A **Next >** gomb a következő párbeszédablakra visz, ebben megadhatjuk, hogy a fordítási hibával rendelkezők fájlok is exportálódjanak-e. A **Next >** gombra történő újboli kattintás után megadható, hogy létrejöjjön-e a `MANIFEST.MF` leírófájl, hogy titkosítsuk-e a JAR-fájlt, vagy a részeit, illetve itt kell megadni a futtatandó osztályt, ha futtatható JAR-fájlt akarunk készíteni. Ezután a **Finish** gombra kattintva a JAR-fájl elkészül a megadott helyen. A folyamatot a B.9. ábra szemlélteti.





B.8. ábra: Javadoc-dokumentáció létrehozása





B.9. ábra: JAR-fájl létrehozása

Szójegyzék

assertion
: A programállapotról hibakeresési szándékkal megfogalmazott feltételezések. Csak külön engedélyezés esetén értékelődnek ki, ezért nem befolyásolják a teljesítményt.

bájtkód
: Lásd virtuális gép.

egységbe zárás
: Objektumorientált alapelv, amely szerint az adatok és a rajtuk végezhető műveletek egységet alkotnak, őket az osztály fogalma zárja egységbe.

felügyelt kód
: Kód, amely nem közvetlenül az operációs rendszeren, hanem egy abban futó futtatókörnyezetben hajtódik végre. A biztonság érdekében a futtatókörnyezet korlátozhatja az erőforrások elérését.
Lásd még virtuális gép.

internacionalizáció
: A programot eltérő kultúrák támogatására felkészítő folyamat. Idetartozik például az eltérő pénznemek és dátumformátumok támogatása.
Lásd még lokalizáció.

karakterkódolás
: Leképezés, amely megadja, hogy egy szöveg karaktereit hogyan rendeljük bájtsorozathoz a számítógépes tárolás során.

kivétel
: A hibakezelés típusos, objektumorientált mechanizmusa. A hibákat kivételekkel reprezentáljuk. A kivételek objektumok, ezért típushierarchiába sorolhatók, valamint tagváltozóikon és metódusaikon keresztül elérhetővé tehetnek a hibákhoz kapcsolódó információt.

lokalizáció
: A program felhasználói felületének és egyéb járulékos részeinek (dokumentáció, naplóüzenetek stb.) más nyelvre történő lefordítása.
Lásd még internacionalizáció.

mockobjektum
: Bonyolult funkcionalitást megvalósító objektum „buta" másolata, amely unitteszthez használható. A tesztek során a bonyolult objektumokat mockobjektumokra cseréljük le, hogy a tesztek a lecserélt objektumok hibáitól függetlenül, izoláltan végrehajthatók legyenek.
Lásd még unitteszt.

öröklés
: Objektumorientált alapelv, amely szerint a leszármazott osztályok örökölhetik őseik tagváltozóit és metódusait. A metódusok újradefiniálhatók.

polimorfizmus
: Objektumorientált alapelv, amely szerint bizonyos típusú interfész vagy osztály helyett annak tetszőleges leszármazottja is használható. A leszármazott osztályok specifikusabb működést valósíthatnak meg az ősök metódusainak újradefiniálásával.

reguláris kifejezés
: Szövegminták leírására használható számítógépes nyelv. A szövegminták alkalmasak például bonyolult keresési feltételek megfogalmazására, vagy a bemenet validálására. A POSIX-szabvány rögzíti a reguláris kifejezések szintaxisát és jelentését, de sok nem szabványos változat is létezik.

socket
: Logikai hálózati végpont, amelyen keresztül az alkalmazások adatokat küldhetnek és fogadhatnak.

sorosítás
: A programállapot bájtfolyammá történő alakítása, hogy az elmenthető, majd később visszaállítható legyen.

távoli metódushívás
: Programrészek hálózaton keresztül történő meghívása. A paramétereket és a visszatérési értéket is a hálózaton kell továbbítani. Ehhez sorosítani szükséges őket.
Lásd még sorosítás.

unitteszt
: A program egységnyi komponenseinek izolált tesztelése. Előnye, hogy az izolációnak köszönhetően a hiba helye jól beazonosítható, ugyanakkor nem fedi le a komponensek közti interakciót.

virtuális gép
: Szoftverréteg, amely a Java-programokat futtatja. A Java-terminológiában a kifejezés nem a hétköznapi értelemben használt virtuális gépet jelenti, amelyre teljes operációs rendszert telepíthetünk, hanem a futtatókörnyezetet, amely a lefordított Java-osztályokat futtatja. A Java-programok ugyanis nem az operációs rendszer által futtatható natív kódra fordulnak, hanem a virtuális gép nyelvére, az ún. bájtkódra. Ez a mechanizmus teszi lehetővé a Java-programok hordozhatóságát.

Tárgymutató

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

Irodalomjegyzék

[1] Balogh Péter, Berényi Zsolt, Dévai István, Imre Gábor, Soós István, és Tóthfalussy Balázs. *Szoftverfejlesztés Java EE platformon*. SZAK Kiadó. 2007.

[2] Ekler Péter, Fehér Marcell, Forstner Bertalan, és Kelényi Imre. *Android-alapú szoftverfejlesztés*. Az Android rendszer programozásának bemutatása. SZAK Kiadó. 2012.

[3] Adrian Kingsley-Hughes és Kathie Kingsley-Hughes. *Kezdőkönyv a programozásról*. SZAK Kiadó. 2006.

[4] Erich Gamma, Richard Helm, Ralph Johnson, és John Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley. 1994.

[5] Neil Bradley. *Az XML-kézikönyv*. SZAK Kiadó. 2005.

[6] Deepak Alur, John Crupi, és Dan Malks. *Core J2EE Patterns: Best Practices and Design Strategies*. Sun Microsystems Press. 2001.

[7] William Grosso. *Java RMI*. O'Reilly Media. 2001.

A szerzőről

Kövesdán Gábor

Okleveles mérnökinformatikus, a Budapesti Műszaki és Gazdaságtudományi Egyetemen végzett 2013-ban az Alkalmazott Informatika szakirányon, kitüntetéssel. Tanulmányai után az Automatizálási és Alkalmazott Informatikai Tanszéken jelentkezett doktori képzésre. Jelenleg is itt dolgozik, és a szakterületi modellezés és a szöveges szakterületi modellek témakörben végez kutatásokat.

Tanulmányai során több nyílt forráskódú projektben vett részt, illetve érdeklődésének megfelelően a tantárgyi kereteken túl is képezte magát. A Java nyelv témakörében megszerezte a Java SE Developer és a Java EE Enterprise Architect címeket, ezek az Oracle Certified Master program minősítései. Rendelkezik egyéb Oracle Certified Professional minősítésekkel is a Java nyelv egyes területeiről és a MySQL adatbázis-kezelő rendszerről. Doktori tanulmányai alatt is a Java programozáshoz kapcsolódó tárgyakat oktat az egyetemen, valamint kutatási eredményeit is Java nyelven valósítja meg.

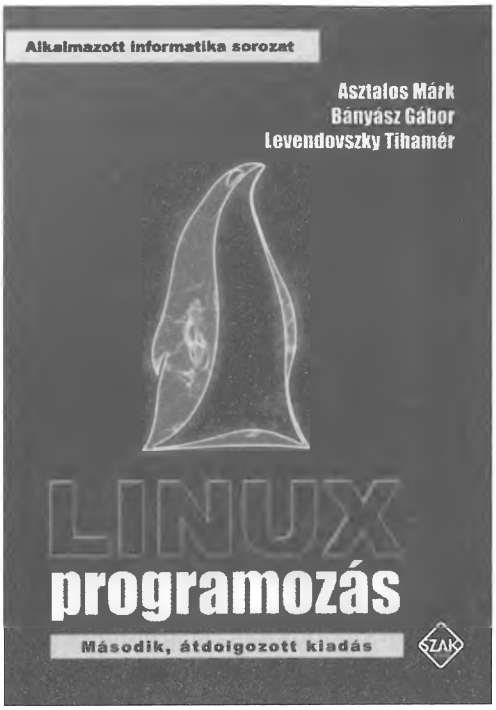
2006 óta a FreeBSD nyílt forráskódú operációs rendszer fejlesztői csapatának tagja. A projektben többféle területen dolgozott. Egyik kiemelkedő munkája az előállított dokumentáció és az előállításhoz felhasznált eszközök fejlesztése. Munkájának köszönhetően 2012 júniusában meghívást kapott a Documentation Engineering Teambe. Ez a csoport a dokumentációkészítésben és az ehhez használt XML-alapú technológiák használatában jártas szakemberekből áll. Ők felügyelik a FreeBSD dokumentációjának fejlődését. A könyvírás iránti motiváció is a könyvek és a dokumentáció szeretetéből adódott. Könyvét hallgatóinak, és főleg öccsének ajánlja, aki jelenleg szintén az egyetemen végzi tanulmányait.

Ekler Péter, Fehér Marcell,
Forstner Bertalan, Kelényi Imre

Android-alapú szoftverfejlesztés

Kartonált, B5
ISBN 978-963-9863-27-9
400 oldal, 6 000 Ft áfával

A Budapesti Műszaki és Gazdaságtudományi Egyetem Automatizálási és Alkalmazott Informatikai Tanszékén működő Alkalmazott Mobil Kutatócsoport (Amorg) kiemelt figyelmet fordít a hazai mobiltechnológia-oktatás folyamatos fejlesztésére. Munkájuknak köszönhetően lehetővé vált, hogy szélesebb rétegek ismerkedjenek meg a legnépszerűbb mobilplatformokkal, és naprakész tudásuk legyen a legújabb technológiákat illetően. Évente csak az egyetemről több mint 500 hallgató vesz részt a kurzusaikon. Az Amorg tagjai a tanszék népszerű Alkalmazott informatika sorozatának keretében ezúttal egy hiánypótló művel lepik meg az okostelefonok fejlesztése iránt érdeklődőket. Bátran állíthatjuk, hogy Android platform világszinten az egyik legnépszerűbb és legelterjedtebb mobilplatformmá nőtte ki magát, éppen ezért a könyv elolvasásával megszerezhető ismeretek hosszú távú tudást jelentenek a felhasználók számára. A könyv szerzői az Android platformmal már a legelső SDK megjelenése óta foglalkoznak, szaktudásukat állandóan szinten tartják, és folyamatos résztvevői a hazai és nemzetközi ipari kutatásoknak és fejlesztéseknek, így széles körű tapasztalattal rendelkeznek. A könyv gondosan válogatott témakörei, szakszerű szerkesztése és a válogatott példák lehetővé teszik, hogy a mindennapi munka során is alkalmazhassuk, és a részletes magyarázatok segítségével a technológia működését is könnyen megérthessük. A szerzők törekedtek arra, hogy a könyvben szereplő kódrészek használatát lehetőség szerint önálló alkalmazásokon keresztül is bemutassák, ezek a példák az előszóban található hivatkozáson keresztül érhetők el.



Asztalos Márk, Bányász Gábor, Levendovszky Tihamér

Linux programozás

Második, átdolgozott kiadás

Kartonált, B5
ISBN 978-963-9863-29-3
616 oldal, 7 800 Ft áfával

A Linux a közelmúltban volt húszéves. Ez a húsz év egyben sikertörténet is: a Linux-alapú szerverek népszerűsége töretlen, nagyon sok beágyazott eszköz futtat Linuxot, köztük a legnépszerűbb okostelefonok. A Linux-disztribúciók egyre közelebb kerülnek a felhasználókhoz, használatuk egyre egyszerűbbé válik. Jóllehet számos magas szintű programozási nyelv és környezet áll rendelkezésre, sok olyan feladat létezik, amelyet az operációs rendszer programozási felületén elérhető funkciókkal lehet csak megoldani. Ilyenkor szükség van az operációs rendszer működésének mélyebb ismeretére, ez beágyazott környezetben egyenesen elengedhetetlenné válik. Ezekhez a feladatokhoz kíván segítséget nyújtani a könyv. Az operációs rendszer C/C++-ban elérhető programozási felületének ismertetése során részletesen bemutatjuk a megvalósítási alapelveket, ennek tükrében érthetővé válik az egyes funkciók használata is. Egy sokkal teljesebb kép birtokában a magas szintű környezetek felhasználói is hatékonyabban tudják kihasználni környezetük lehetőségeit. Mivel a Linux a Unix operációs rendszerek családjának tagja, ezért a Linuxról elmondottak nagy része igaz a Unixra is. Mindezek alapján ajánljuk a könyvet mindenkinek, aki Linux/Unix környezetben tervezői, illetve programozói munkát végez, valamint azoknak, akik el szeretnék sajátítani az ehhez szükséges ismereteket.
A szerzők könyvüket annak első kiadása óta szinte teljesen átdolgozták, és számos új fejezettel bővítették.

Imre Gábor (szerk.)

Szoftverfejlesztés Java EE platformon

Kartonált, B5
ISBN 978-963-9131-97-2
460 oldal, 6 500 Ft áfával

A Java nyelv és a hozzá kapcsolódó technológiák folyamatos fejlődést mutatnak. Ez indokolttá tette, hogy a Java-t három különböző kiadásra (edition) osszák. A Java Standard Edition (Java SE) hagyományos desktop alkalmazások és appletek fejlesztését teszi lehetővé, a Java Micro Edition (Java ME) segítségével mobil eszközökre készíthetünk alkalmazásokat. Könyvünk témája a Java Enterprise Edition (Java EE), mely elosztott, sok felhasználóval rendelkező, vállalati méretű szoftverrendszerek fejlesztéséhez nyújt támogatást. A Java EE technológia a mögötte álló jelentős ipari támogatásnak köszönhetően napjaink egyik legnépszerűbb és legelterjedtebb szerver oldali megoldásává vált, így megismerése minden szoftverfejlesztő és -tervező hasznára válik.

A könyv két nagyobb részre tagolódik. Először a Java Enterprise Edition legfontosabb technológiái kerülnek bemutatásra. A tárgyalás a Java Enterprise Editionben kezdők számára is érthető, ugyanakkor a 2006-ban megjelenő Java EE 5 helyenként drasztikus újításai miatt a könyv azok számára is hasznos, akik már járatosak a J2EE korábbi verzióiban.

A fejezetek második fele az alkalmazásfejlesztés különféle kérdéseihez kapcsolódó jótanácsokat tartalmaz. Itt kapnak helyet a biztonsági és naplózási megfontolások, a szoftver életciklusához elengedhetetlenül hozzátartozó tesztelés automatizált megoldása, végül az integráció lehetőségei Java EE alkalmazások és más rendszerek között.

Szoftverfejlesztés JAVA SE platformon

Kövesdán Gábor

Napjainkra a szoftverfejlesztés egyik vezető platformjává a Java nyelv vált: széles körben használják ipari és kutatási feladatok megvalósítására. A könyv feltételezi, hogy az Olvasó már rendelkezik általános programozói alapismeretekkel, de a Java nyelvet az alapoktól kezdi. A téma ismertetése az aktuális programozási trendeket és technológiákat veszi alapul, és bemutatja a Java nyelv ezekhez való kapcsolódását is. A nyelvnek az íráskor aktuális, 7-es verzióját tárgyalja.

Az első öt fejezet a legalapvetőbb, a Java-programozók számára nélkülözhetetlen ismereteket nyújtja. Ezek stabil alapot jelentenek a platform használatához. Ide tartozik a változótípusok és az utasítások ismertetése, a fejlett objektumorientált eszköztár és az osztálykönyvtár áttekintése, valamint a generikus kollekciók bemutatása is. A későbbi fejezetek haladó témákat ismertetnek, végigviszik az Olvasót az összetett alkalmazások fejlesztése során felmerülő problémákon. Ezeket a fejezeteket az alapokat már jól ismerő programozók külön is feldolgozhatják.

A fejezetek során olyan témákat érintünk, mint a programállapot elmentése, az XML-formátum feldolgozása, a relációs adatbázisok használata, a hálózati kommunikáció, a grafikus felhasználó felület kifejlesztése, valamint a különböző nyelvek és kultúrák támogatása. Az utolsó két fejezet a tesztelést és a terjesztést mutatja be, ezek nélkülözhetetlen ismeretek minden valós alkalmazás fejlesztése során.

A szerző a Budapesti Műszaki és Gazdaságtudományi Egyetem Automatizálási és Alkalmazott Informatikai Tanszékén oktat programozást Java nyelven.

Ára: 5 900 Ft áfával
ISBN 978-963-9863-35-4

SZAK Kiadó a weben
http://www.szak.hu