I. Fejezet

Java alapú intelligens kártya

A Java Galaxisban utazva a következő fuvar a „kisszámítógép-világba" ad rövid betekintést. Az előforduló szakmai kifejezések vagy rövidítések magyarázata megtalálható az anyagban, de érdemes felkeresni az on-line betűrendes angol szótárt is. [Peth 99]

I.1. Vissza a gyökerekhez: az eszközbe épített Java

A Java webes, majd önálló programozási nyelvként elért sikere után hamarosan újraindultak a különböző eszközökbe építendő Java fejlesztések. Az ilyen rendszereket nevezzük beágyazott rendszereknek. Ilyenkor az eszközben egy célprogram kerül elhelyezésre, mely célprogram képessége széles skálán mozoghat. A JavaSoft felelős a Java alkalmazások menedzseléséért. A JavaSoft [SM-Sou 01] a Java eszközökben megvalósított alkalmazásait három területre osztja:

- Personal Java - televíziókészülékekhez, intelligens telefonkészülékekhez és különböző személyes digitális eszközökhöz
- Embedded Java - nyomtatókba, fénymásolókba és hasonló eszközökbe építve
- Card Java: intelligens kártyák (IK) részére, azaz smart card-okhoz

Ebben a fejezetben ez utóbbiról lesz szó, de röviden említünk olyan kapcsolódó területeket is, mint a Wireless Application Protocol (WAP) és a Jini (ejtsd: dzsini), melyek együttesen meghatározó szerepet játszanak majd a jövő mobil és eszközfüggetlen kommunikációjában. A Java Card 2.1.1-es szabvány is kiemeli, hogy az IK mellett a kis memóriakapacitással rendelkező eszközökre is gondoltak a szabvány megtervezésénél és lehetséges felhasználásánál. A Java 2 Platform, Micro Edition (J2ME) kiegészítéseként a Java Card 2.1.1 egyszerűsíti az alapvető fogyasztói technológiák beépítését egy teljes Java szoftveres környezetbe.

I.2. Visszatekintés és felhasználási területek

2001. január 24-én volt 27 éves a francia Roland Moreno ötlete a kis integrált áramkörös eszközről (eleinte még nem plasztikkártyába képzelte a chip beépítését, így a kártya kifejezés nem jelent meg), és három hónappal később bejelentette szabadalmát. Ez a szabadalom tekinthető az intelligens kártyák közös ősének. 1978-ban az ugyancsak francia Michel Ugon a francia bankok részére készített egy chipes bankkártya megoldást (itt már megjelenik a kártya forma is, az ekkor már régóta ismert hitelkártya méretű lapkaként). Michel Ugon ezért a megoldásért később a Francia Becsületrendet is elnyerte. Ezeknek a történelmi gyökereknek köszönhetően az intelligens kártyával foglalkozó legnevesebb cégek Franciaországban vannak.

Fele ennyi idős a szabványosítás kezdete, melynek köszönhetően az IK alapja az ISO/IEC 7816-os szabvány a maga alegységeivel, melyek a fizikai paraméterektől kezdve a kártyába épített modul érintkezőin át a chip belső utasításkészletéig több paraméter szabványait foglalják magukba. Az egyes gyártók erre alapozzák termékeiket, de mindegyik tartalmaz saját megoldásokat, kiegészítő parancsokat, melyek a különböző operációs

rendszerekhez készült, és különböző programozási nyelveket támogató kártyák inkompatibilitásához vezettek. Az IK területén a széleskörű terjedésnek gátat szabó nehézkes programozási lehetőségek és a platformfüggő megoldások világát váltja fel a Java filozófiáján alapuló *Java kártya* kezdeményezés.

Az intelligens kártya attól intelligens, hogy a külvilág felől érkező kéréseket saját operációs rendszere és beépített biztonsági mechanizmusa dolgozza fel, és reagál ezekre a kérésekre. A kártya testében található a chipet körülvevő modul a RAM, ROM, és EEPROM memóriákkal együtt. Mivel ez a konstrukció egy kis számítógépnek felel meg, ezért a nagy számítógépek több funkcióját is el tudja látni. Ezen felül olyan kiegészítő biztonsági elemek is helyet kapnak egy kártyás rendszerben, melyek befogadására egy személyi vagy egy nagyobb számítógép felépítésénél fogva sem képes. Ilyen megoldás például a chipet tartalmazó modul fizikai védelme a kiemeléssel szemben. A számítógép részegységeinek mobilitásával ellentétben a kártya felületében lévő modul nem bővíthető, ami hátrányt jelenthet hosszabb távon, de a biztonság mindennapos feladatát így láthatja el könnyebben.

A kártya biztonságos adattárolást tesz lehetővé, és hordozhatósága révén biztosítja az egységes alkalmazások használatát. Elektronikus pénztárca esetén független a forgalomban lévő pénzegységektől (mindig tud visszaadni), kiemelt biztonsági feladatok esetén társkártya biztosíthatja a megosztott felelősséget, vagy éppen a nagyobb garanciát és biztonságot. Az alkalmazási területek a telekommunikáció (újratölthető telefonkártya, kábel-tv előfizetés, internetelérés, stb.), a hozzáférés-védelem (beléptetőrendszerek, számítógép-hozzáférés), az egészségügy (sürgősségi kártya, közgyógyellátás, háziorvosi rendszer, biztosítási rendszer) területeit is magába foglalhatja. A közös alap a személyazonosítás, mely a személyi igazolvány, az útlevél, a jogosítvány és egyéb okmányok esetén alapkövetelmény. Az ilyen módon tárolt információk a mai digitális világban a gyorsabb és konzisztensebb működést teszik lehetővé, és az információk hamisítása jóval nehezebb és drágább a mai papíralapú rendszerek hamisításánál.

A különböző felhasználási területek integrálhatók egyetlen eszközbe is, mely ezáltal több különböző feladat ellátására is képes. Ilyen példa az új diákigazolvány, mely a világon egyedülálló abban a tekintetben, hogy minden felsőoktatásban részt vevő hallgató ilyen eszközt használhat. Az új diákigazolványt körülvevő gazdasági és politikai körülmények sokszor negatív hatásai helyett érdemesebb a kártyát technológiai szempontból vizsgálni.

Az eszköz a francia Gemplus cég terméke, felületi biztonsága a korszerű lézergravírozott feliratozás által is szavatolt (a kártyákon levő képek sokat vitatott minősége nem a technológia hibája). Az eszköz felhasználható DES (Data Encryption Standard) és háromszoros DES (3DES) kódolásra, beléptető rendszerekhez, pénzt helyettesítő területre (elektronikus pénztárca), tanulmányi adatok kezelésére éppúgy, mint számítógépek hozzáférés-védelmére vagy könyvtári kölcsönzés nyilvántartására. Egy ilyen eszközzel olyan rugalmas felhasználás is elképzelhető, melyben a bankszámláról felvett elektronikus ösztöndíj nemcsak a fénymásoláshoz vagy a menzához használható, hanem a telefonáláshoz is, akár egy külön erre a célra fenntartott memóriarekeszben kezelve az újrafeltöltést. Mindez nem képzelgés (technológiai szempontból), mert van példa hasonló alkalmazásokra vagy szakdolgozatokra.

I.3. A kezdetektől a *Java kártyáig*

Az első „specifikáció” 1997 tavaszán jelent meg Java Card 1.0 néven. Az anyag néhány oldalból állt, de a Java-t és az IK világát ismerők körét megragadta az elképzelésben rejlő jövőbe mutató lehetőség. Ugyanebben az évben megjelent a 200 oldalas, három részből álló 2.0-ás specifikáció, és a boltokban is megjelentek az első fejlesztőkörnyezetek (ez többnyire azt jelenti, hogy a virtuális boltokon keresztül a fejlesztőrendszer megrendelhető). Mára több intelligens kártyával foglalkozó nagyvállalat rendelkezik saját *Java*

kártya implementációval. Az egyik 8 bites, a másik 32 bites RISC alapú processzort használ, és a specifikáció megvalósításai a gyártók érdekeit képviselik. Az eredeti célként kitűzött eszköz- és platformfüggetlenség elérése és koordinálása a Java Card Forum feladata. [JCF 01] A *Java kártya* esetén a ROM memória tartalmazza a szabvány alapján implementált virtuális gépet, amely a cardletet futtatja. A cardlet elnevezés a Schlumberger cég által védett. A továbbiakban az applet kifejezés a kártya appletet (cardletet) jelenti, míg a kártya szó alatt a kártya testébe helyezett modul, pontosabban a modulban lévő chip értendő.

1999 márciusában jelent meg Java Card 2.1-es szabvány, mely több ponton is eltért elődjétől. [SM-Card 01] A CD mellékleten található példaprogramok a 2.0-ás szabvány alapján készültek. Itt kell megjegyezni, hogy a 2.0-ás szabványra épülő megvalósítások közül egy sem alkalmazta a teljes szabványt, és ezen felül a szabványban nem definiált utasításokkal is bővítették az egyes implementációkat. Ezért nem biztosított minden esetben az egyszer megírt alkalmazások eszközfüggetlensége.

A Java újabb és újabb verzióinak követése és a Java Card szabvány fejlődése eredményeként 2000-ben jelent meg a Java Card 2.1.1 szabvány, és 2001-ben jelennek meg az első termékek, melyek ezt a szabványt támogatják.

A *Java kártya* is az ISO/IEC 7816-os szabványra épül, mely közös hivatkozási alapja minden processzoros IK alkalmazásnak. E mellett az Europay - MasterCard - Visa hármas által kidolgozott EMV szabványnak is megfelel, így lehetőség van elektronikus pénztárca alkalmazások beépítésére is. A nem teljes eszközfüggetlenség az ISO szabványra is jellemző, amikor egy adott kártya esetén a dokumentációban azt olvashatjuk, hogy az ISO szabványból mit valósít meg a kártya, és mivel egészíti ki azt. Az EMV szabvány az említett hitelkártyatársaságok összefogásával készült, majd később az American Express is csatlakozott a csoporthoz, így ezen a területen létezik a legszélesebb összefogás egy közös szabvány alkalmazásában. Az EMV2000 szabványkötetek elérhetők 2000 decembere óta a `http://www.emvco.com` lapról.

A különböző implementációk összehasonlítása túlmutat a fejezet célján, és a 2.1.1-es szabványra épülő környezetek már nem is fognak ilyen jelentősen eltérni egymástól. Ez a kompatibilitás igény a Java nyelv tulajdonsága és a felhasználók igényei alapján egyaránt következik. Ezek mellett az egyik legnagyobb hatást a mobiltelefonok terjedése adja. A Java-alapú SIM-kártyák alkalmazása a dinamikus frissítés lehetőségét biztosítja, így ezen a területen gyors terjedés zajlik. Egy konferencián elhangzott adat szerint 1998-ban még nem volt jelentős a Java-SIM alkalmazások száma, de 1999-ben 20 millió, és 2000-ben 100 milliós volt a várható darabszám.

Az IK területén a kompatibilitás elérését két főbb irányzat képviseli. Az egyik a PC/SC [MS-PCSC 01] szabvány, mely mögött a Microsoft áll, míg a másik irányzat az OpenCard Framework [OC-OCF 99] mely az IBM-hez köthető és Java alapú. Mindkét irányzatnak több támogató tagja van, több cég mindkét csoport munkájában is részt vesz! Ezek és a többi létező szabvány és irányzat ismertetése már túlmutat a könyv és azon belül e fejezet céljain, de az érdeklődők az ajánlott WWW címekről indulva feltérképezhetik a területet, és elérhetik a szükséges információkat. A szélesebb értelemben vett függetlenséget az XML (Extensible Markup Language) nyelvből kialakított SML (Smart Markup Language) biztosítja. Az érdeklődők bővebb információt kaphatnak erről a forradalmian új kompatibilitási törekvésről az smartX honlapján. Mivel az elképzelés és a megvalósítás mesteri, ajánlatos e fejezet alapos ismerete után felkeresni az oldalt, így jobban értékelhető az ott elérhető információ! [SMX 00]

A francia Schlumberger cég az egyik élenjárója a Java Kártya technológia alkalmazásának. Terméküket a Cyberflex nevet viseli és Core változata mindössze 2,8 Kbájt EEPROM memóriával (szokták még az E^2PROM jelölést is használni), míg a Multi8k változat 7,5 Kbájt memóriával rendelkezik. [Sch-Cyb 01] Az egyik legújabb termék, mely Linux operációs rendszer felett is használható, az Open16K, 16 Kbájt memóriával rendelkezik.

A Linux rendszerekhez illeszthető IK alkalmazások fejlesztése a MUSCLE (Movement for Use of Smart Cards in Linux Environment) program keretében folynak, és a Linux hívőknek kiemelten ajánljuk a program lapjának felkeresését. [MUS 01] Az átlagos RAM memória a legtöbb kártya esetén 512 vagy a nagyobb kapacitásúak esetében 1024 bájt (nem Kbájt!). Az ezredfordulóra tervezett Cyberflex kártya tulajdonságai:

- 128 Kbájt flash ROM
- 4 Kbájt RAM
- 32 vagy 64 Kbájt EEPROM memória az appletek számára
- a Java Interpreter (a ROM-ban lévő Java Interpreter) mérete 16 Kbájt.

I.4. Felépítés, működés, szerepek

A *Java kártya* ismertetése előtt meg kell ismerkedni az IK működésével. Akinek ismerős az APDU-k és ATR-ek világa (lásd később), az ugorja át ezt a részt. Akit részletesebben is érdekel a terület, annak az egyik legjobb kiadvány, a *Smart Card Developer's Kit* című könyv ajánlható. [Guth Jur 99]

A kártya felületébe helyezett modul tartalmazza a chipet, mely az ISO szabvány szerint 8 lábbal csatlakozik a modult lefedő, arany vagy ezüst színű lapkához. A processzorhoz (lehet mellette még co-processzor is, mint például egy kriptográfiai számításokat végző kripto-processzor) egyik oldalról a lábak, míg másik oldalról a memóriák kapcsolódnak (RAM a futtatáshoz, ROM az operációs rendszerhez és *Java kártya* esetén a virtuális géphez, valamint EEPROM a kártyán tárolandó alkalmazásokhoz, adatokhoz).

Pár éve megjelentek az érintés nélküli kapcsolatfelvételre is alkalmas eszközök (contactless card - közelítő kártya), és ma már elérhető ezek kombinált változata is. A kontaktus nélküli (közelítő) kártyák működését az ISO/IEC 10536, 14443 és 15693 szabványok határozzák meg attól függően, hogy az adott kártyatípus milyen távolságon belül képes a tranzakcióra.

Az IK chiplábak szerepe a következő kiosztású lehet az ISO/IEC 7816 alapján:

- *clock*, azaz órajel kezelése (egyes eszközök rendelkeznek már belső órával)
- *reset*, azaz fel- vagy újraélesztő jel kezelése
- 0 V-os láb, ami a *GND*-nak (ground - föld) felel meg (a modult fedő lapkán a középre kivezetett rész; a kis képen a jobb felső sarok)
- 5 V-os láb a *Vcc* (tápfeszültség, ettől eltérő mértékű is lehet bizonyos határok között)
- 25 V-os láb a *Vpp* (programozáshoz szükséges feszültség, ami szintén lehet eltérő mértékű)
- I/O láb, amelyen az adatjelek közlekednek
- 2 tartalék láb jövőbeni alkalmazásokhoz (jelenleg kevés példa van használatukra)

A lábak pontos elhelyezkedését és más paraméterek meghatározását az ISO7816 szabvány első három alpontját leíró rész tartalmazza, mely a CD-ROM `JavaCard` alkönyvtárában is megtalálható `iso7816.txt` néven.

A kártyaolvasók a chipet fedő lapka megfelelő részére kapcsolódó fémlábakkal teremtenek kapcsolatot a kártyával. A kártya, az olvasó és a számítógép között megoszthatóaz intelligencia. Vannak olyan olvasók, amelyek kulcstartó méretűek, és csak bizonyos adatok kiolvasására alkalmasak (pl. a kártya egyenlegét képesek kiolvasni). Van billentyűzettel rendelkező olvasó is, így nem a számítógép billentyűzetén bevitt adatokkal kell dolgozni, hanem a biztonságosabb közvetlen bevitellel élhetünk, amikor sokkal nehezebb az elektromágneses kisugárzásból kinyerni a bevitt információt (pl. hozzáférési kódokat). [CR-DPA 99]

A PCMCIA-s csatlakozójú olvasók a drágább kategóriába tartoznak, de elérhetők a laptopok számára is, és elérhetők a különböző olvasók soros és párhuzamos portokra egyaránt minden széles körben használt operációs rendszerű személyi vagy szerver számítógéphez. A kulcstartó méretű eszközök a programozáshoz (íráshoz) szükséges feszültséget nem tudják előállítani, míg a számítógéphez csatlakoztatható eszközök általában két vonalon kapcsolódnak a számítógéphez. Az egyik vonalon az adatkommunikáció, míg a másikon a működéshez szükséges feszültség folyik. Ez utóbbi a számítógép és a billentyűzet közé illeszthető többnyire PS/2-es csatlakozóval.

A komplex szolgáltatást nyújtó olvasók rendelkezhetnek belső modemmel is, így például egy italautomata a begyűjtött elektronikus pénzt le tudja jelenteni a nap végén, vagy akár naponta többször is, ha a forgalom vagy a biztonság megkívánja. Egy modemen keresztül az automata lejelentheti a készlet állását is.

I.4.1. A biztonság

A mai chipek és memóriák élettartama több év is lehet, de az életciklus a használat intenzitásától is függ, ezért a tranzakciók száma az élettartam mérőszáma, ami a korszerű változatoknál százezres nagyságrendű. A legújabb termékek esetén nem ritka a milliós nagyságrend sem! Ez elméletileg azt jelenti, hogy az EEPROM memória napi 100 tranzakció (írás) esetén negyed század múlva is használható (újraírható), ha a műanyag lapka is bírja addig a gyűrődést, ami a használat során éri.

A tranzakció folyamán a chipben tárolt információk védelme többszintű lehet. A kártya felett rendelkezési joga lehet a gyártónak (egyedi sorszám módosíthatatlan beírása gyártáskor), a kibocsátónak (személyhez rendelés korlátozottan módosítható adatainak felvitele), a birtokosnak (személyes adatok kezelése kóddal védett hozzáféréssel). A különböző hozzáférési jogok ellenőrzése a Card Holder Verification (CHV) nevet viseli, amiből a CHV- bit kifejezés származik. Az ilyen biteket az egyes adatokat tároló állományok Access Control listája (ACL - hozzáférést felügyelő lista) tárolja egy ACL-bájtban.

A CHV-bit mellett az egyes állományokhoz való hozzáférést az azonosító bitek tárolják, amelyeket AUT-biteknek neveznek az authentication (hitelesítés) szó alapján. Adott CHV-bit adott AUT-bit mellett adott műveleteket (pl. írás, olvasás) határoz meg az állományokhoz tartozó ACL-bájt megfelelő beállításával. Az állományokon végezhető műveletek jogait a CHV- és az AUT-bitekből felépített mátrix tartalmazza, melyből az ACL-bájt kiszámítható. Például egy egészségügyi kártya esetén egy cukorbeteg CHV-bitje és a beadott inzulin mennyiségének tárolására szolgáló fájl AUT-bitje adja meg a fájl ACL-bájtját, ami alapján a cukorbeteg a fájlt írhatja és olvashatja. A kezelőorvos csak olvashatja ezt az adatot, és írhatja azt a fájlt, amibe az inzulin típusát tárolja (ezt a beteg csak olvashatja), mert az orvos és a beteg különböző CHV-bitjéből alakult ki a fájlok ACL-bájtja. A modell alapján a mátrix első oszlopában az AUT-biteket, első sorában a CHV-biteket, a kettő metszésében az ACL bájtokat tároljuk.

Összefoglalva: egy adott kódot megadó felhasználó („belépett a kártyába") adott állományba író parancs kiadásakor elutasítást kap (kivételt generál), ha a kártya operációs rendszere az állományhoz tartozó ACL-bájt értelmezésekor a megfelelő biten 0-át talál 1-es helyett.

A próbálkozásokat egy számláló jegyzi, és adott számot elérve a chip blokkolhat vagy kiemelt esetben visszaállíthatatlan adattörlést hajt végre. Ez utóbbi esetben a chip használhatatlan, míg a blokkolást egy mester kártya segítségével oldani lehet. Ekkor az említett számláló újraindul (reset-elődik). A mester kártya kinézetre ugyanolyan, mint a többi, de a chip tartalmazza a magasabb jogosultsághoz szükséges információkat. A blokkolás oldása mellett olyan szerepe is lehet, hogy a rendszer többi kártyája csak a mester kártya birtokában programozható át. A mester kártyát ezért nevezik sokszor SAM (Secure Access Modul) kártyának is.

A *Java kártya 2.0* esetében egy „homokláda" (Java Card SandBox) felügyelt a biztonságra, de a Java 2-höz alkalmazkodva már a Java Card 2.1 szabvány kiemelten kezeli a biztonság témakörét. A kártyákon jelen van a tűzfal kétféle megvalósítása: az egyik a kártyán futó alkalmazások közötti tűzfal, melyet a kártya virtuális gépe valósít meg. A másikat a HW gyártók építik be fizikailag az egyre növekedő memóriakapacitások szegmentálása révén. Ez utóbbiban élenjáró az ST Microelectronics által tervezett chipek legújabb generációja.

A támadási lehetőségek között szerepel a külső áramforrás és/vagy órajel manipulációja. Ez az operációs rendszert olyan hibára késztetheti, melynek következtében a kártya titkos információi is hozzáférhetővé válnak. [And 01] Az ilyen támadások ellen úgy védekeznek a gyártók a kiemelt biztonsági funkciókra gyártott chipek esetén, hogy a kártyába építik a szükséges áramforrást is. Ilyen eszköz a *Java gyűrű* (lásd később), mely a belső áramforrás használata és az órajel előállítás mellett egy kristály segítségével random (bizonyos korlátok között véletlenszerű) frekvencián (MHz) működteti a processzort.

A legújabb támadási lehetőség a Differential Power Analysis nevet viseli. [CR-DPA 99] Ennek az a lényege, hogy a chipek energiafelvételéből (megfelelően érzékeny és drága eszközökkel) kikövetkeztethető a processzor által végzett művelet. Ez lehetőséget biztosít a kódoló eljárás algoritmusának meghatározására is, amennyiben az a „security by obscurity" (az eljárás biztonsága az alkalmazott algoritmus titokban tartása révén szavatolt) elven alapszik. Ilyenkor az algoritmus lépéseit próbálják „kihallgatni" a működés során felvett feszültségeket mérve. Ez ellen a gyártók szintén a random függvények alkalmazásával lépnek fel. Ezzel összezavarják az egyes műveletek által felvett energia méréséből levonható következtetések logikáját. És a verseny megy tovább: újabb biztonsági megoldások és újabb hackelések. Tökéletes biztonság nincs, csak tudatos kockázatvállalás!

I.4.2. Kommunikáció a kártyával

Az olvasó és a kártya master/slave viszonyban van egymással. Mindig az olvasó kezdeményez, és a kártya válaszol. Az adatjelek közlekedését protokollok írják le. A két alapprotokoll a T=0 (bájt alapú átvitel) és T=1 (blokk alapú átvitel), mindkettő fél-duplex (adat mindig csak egyik irányban közlekedhet). Az adatok a RAM memórián keresztül cserélődnek, így a RAM kis méretéből következik, hogy kis mennyiségű adat utazhat egyszerre, tehát nagyobb mennyiségű adat több ciklusban darabonként továbbítódik. Fontos a fél-duplex átvitelben, hogy egyrészt ne egyszerre küldjön adatot a kártya és az olvasó, mert ebben az esetben elvész az adat, másrészt ne várjanak egymásra, mert ez holtpontot eredményez.

Az adatokat jelenleg a fejlesztői környezet szintjén is sokszor hexadecimálisan kezelik és jelenítik meg. A közeli jövőben ez az ábrázolási mód adatkonverziókkal támogatva teljes elfedésre kerül, a programozók munkáját is megkönnyítve ezzel. A Gemplus cég GemXplore Case eszköze már olyan fejlesztőkörnyezetet biztosít a SIM programozáshoz,

melyben egyre inkább az egérkattintások veszik át a szerepet a billentyűzet használatától. A későbbiek során az adatok hexa-kódú megjelenéséről még lesz szó.

Pár éve még azt olvashattuk, hogy a kommunikáció során elméleti lehetőség létezik a 115.200 bps átviteli sebességre elérésére, de a gyakorlatban a 9600 bps az általános. A kevés adat átvitele közben a csatorna elég zajos, így a sebességnél sokkal fontosabb a hibamentes átvitel. Nem sokkal később az 1 Mbit/s-os sebesség is megvalósult, és a chipben lévő processzor számítási kapacitása terén is folyamatos és gyors a fejlődés, így a 20 MIPS-es 60 MHz-es RISC processzor is elérhető a piacon. Az egyik legújabb termék, a Tiny2J, melyet a *Java kártyák* részére fejlesztettek ki, egy 60 MIPS-es, 75 MHz-es, 32 bites RISC processzor.

Amikor egy kártyát az olvasóba helyeznek, akkor az olvasó egy RESET jelet küld a kártya felé. Erre a kártya egy ATR (Answer To Reset) jelet küld vissza az olvasónak, és az olvasó ebből a szabványosan felépített bitsorozatból tudja meg, hogy milyen módon kommunikálhat a kártyával. Természetesen a kapcsolatfelvétel sikeressége függ attól is, hogy a kártya képes-e az olvasóval kommunikálni, vagy az olvasó képes-e a kártyával „szót érteni" (fel tudja-e dolgozni az ATR jelet). A 3K Multiflex kártya esetén az ATR = `3B 02 14 50`, ezt küldi vissza az olvasónak. Ennek feldolgozása után „megegyeznek" a kommunikáció során használandó protokollban, ami jelen példánál a T=0 protokoll, és a kártya parancsra vár.

A két fél között közlekedő adatokat APDU-nak (Application Protocol Data Unit) nevezünk. Az APDU-k (parancs és válasz) felépítése szabványos.

A parancs APDU felépítése a következő:

CLA mező (Class byte) - az utasítás osztályát írja le

INS mező (Instruction byte) - az utasítás kódja

P1, P2 mezők (Parameter bytes) - az utasítás paraméterei/kapcsolói

Lc (Length counter) - az adatmező hosszát írja le, ha van adatmező, különben üres

Data - adatmező, mely opcionális az Lc-től függően, amikor nem adatot kezelő parancsról van szó

Le (Length expected) - a válasz APDU adatmezőjének elvárt hossza, ha van válasz

A válasz APDU felépítése:

Adat - a válasz adat, ha a parancs végrehajtása eredményezett ilyet (lásd parancs APDU-nál)

SW1, SW2 (Status word) - a válasz „állapotleíró" jelei (végrehajtás sikere, vagy sikertelenség esetén a hibakód)

A hexa hibakódokhoz tartozik egy lista, melyből kideríthető a hiba szöveges értelmezése, de ezek az alkalmazáson belül is kezelhetők a megfelelő konstansok definiálásával. A legkedvesebb visszajelzés egy kártyaprogramozó számára a 0x9000, ami a sikeres végrehajtást jelenti.

Az egyes hexa számokhoz minden környezetben definiálhatunk konstansokat, így csak erre az időre kell tisztában lennünk jelentésükkel, melyet a környezethez kapott programozói kézikönyv megfelelő táblázataiból ismerhetünk meg.

A *Java kártya* esetén a hibák kivételt váltanak ki, melyeket kezelni kell, mert a nem megfelelően kezelt kivételek a kártya időszakos blokkolását vagy adatainak végleges törlését jelenthetik. A kivételek és más hexa kódok saját vagy a fejlesztőkörnyezet által

biztosított (beépített) konstansok által válnak kezelhetőbbé és egyben beszédessé. A teljesség igénye nélkül néhány jellegzetes kód, a forrásban alkalmazott definíciós módon például *Java kártya* esetén:

```
public final static short SW_NO_ERROR = (short)0x9000;
    /* minden rendben volt */

public final static short SW_COMMAND_NOT_ALLOWED = (short)0x6986;
    /* nem engedélyezett a parancs */

public final static short SW_FILE_NOT_FOUND = (short)0x6A82;
    /* kiválasztásra szánt állomány nem található */

public final static short SW_INS_NOT_SUPPORTED = (short)0x6D00;
    /* az alkalmazott parancs osztálya nem támogatott */
```

A kommunikáció váratlanul is megszakadhat, ha a kártyát tápláló áramforrás kimarad, vagy a kártyát kihúzzák az olvasóból. Az ilyen események kezeléséről a perzisztencia résznél lesz szó.

I.4.3. A kártya fájlszerkezete

A memóriában található állományok fa-struktúrában helyezkednek el. A gyökérkönyvtár egy példányban létezik, Root Directory (RD) a neve, és egyedi hexa azonosítója 0x3F00. Ezen belül lehetnek az alkönyvtárak, melyeket Dedicated File-nak (DF) nevezünk (szokásos a Directory File elnevezés használata is). Az állományok megfelelői az Elementary File- ok (EF), amelyek nem tartalmazhatnak DF bejegyzéseket.

Az EF bejegyzések szolgálhatják a kártyában levő alkalmazásokat vagy a kártyán kívüli alkalmazásokat. Ezen belül többfélék lehetnek attól függően, hogy milyen a kezelési módjuk. Típusuk (transzparens vagy rekord) egyben a felhasználási területet is meghatározza az egyes alkalmazásokban. A transzparens típusúak szekvenciálisan tárolják az adatokat. A rekord típusúakban a rekordok lehetnek fix vagy változó hosszúságúak, ezen belül a rekordok lehetnek szekvenciálisan vagy ciklikusan összekapcsoltak. A ciklikus EF fix hosszú rekordokat tartalmaz.

Egy elektronikus pénztárca esetén az egyszeri alkalommal elkölthető összeg maximálva van, és a legutóbbi néhány tranzakciót a visszakereshetőség érdekében tárolni szokták. Erre a feladatra az elektronikus pénzt egy fix rekord hosszúságú ciklikus EF állományban ajánlatos tárolni.

Minden bejegyzés egy 2 bájtos FID (File IDentifier) azonosítóval érhető el. Adott DF-on belül egy EF azonosítható rövidebb (5 bites) azonosítóval is. Az ISO szabvány szerint az EF típusa összefügg az EF bejegyzéseket kezelő parancsokkal is. Ennek megfelelően a `Read Binary` és társai (`Write, Update, Erase`) csak transzparens állományokra alkalmazható, míg a `Read Record` és társai (`Write, Append, Update`) csak rekord típusú állományokra. Az egyes implementációkban ezek az alapok léteznek, a környezethez adott programozói kézikönyvek pedig leírják a megfelelő eljárások nevét és alkalmazhatósági paramétereit. A kézikönyv tartalmazhat több olyan állománykezelő eljárást is, amelyek csak az adott környezeten belül alkalmazhatók.

I.4.4. A kriptológia szerepe

A kártya és az olvasó kölcsönös megismerkedése után egymás szigorúbb azonosítása is következhet akár DES alapú titkos kulcs, RSA alapú nyilvános kulcs, vagy Challenge-Response alapú kriptográfiai szabályok alapján. Az IK csökkentett memóriája miatt a

kisebb erőforrást igénylő Elliptic Curve Cryptology (ECC elliptikus görbéken alapuló kriptográfia) alapú kódolást is használják az RSA helyett. [Gal 99] Az elliptikus görbéken alapuló kriptográfiai algoritmusok alkalmazásának realitását az az összehasonlítás is jól mutatja, melyben az ugyanolyan erősségű kulcshosszak alapján az 1024 bites RSA a 175 bites ECC, a 2048 bites RSA a 876 bites ECC kulccsal egyenértékű. Egy 1024 bites kulcsú aláírás ECC megfelelője 350 bites.

A DES megfelelő erőforrással egy napon belül törhető az akadémiai szféra számára, miután megépítették a DESCRACKER nevű számítógépet közel 250.000 amerikai dollárból. [EFF-DES 99] Ezek után fontossá vált a DES helyett erősebb algoritmus kidolgozása. A DES cseréjét célzó AES (Advanced Encryption Standard) program eredményeképpen 2000-ben kihirdették a nyertes pályázó algoritmust, a Rijndael-t (ejtsd: rájndahl).[NIS-AES 01] Az algoritmus fokozatos alkalmazása a DES helyett nagy jelentőséggel bír az IK területén. Az algoritmust két fiatal kutató dolgozta ki: a 30 éves leuveni kutató Vincent Rijmen, és a 35 éves kriptográfus Joan Daemen, aki az IK területen jól ismert belgiumi Proton World International cég szakembere.

A biztonságot fokozza, hogy a korszerű chipek a kulcsgenerálást belső utasításokkal végzik, így a birtokos számára sem ismert a titkos kulcs, de alkalmazása lehetséges a kulcs generálásakor beállított jelszó ismeretében. Egy bájtsorozat esetén a chipen áthaladva kerül kódolásra az adott információ, így a titkos kulcs nem hagyja el az eszközt, és a memória tartalma egyszerű eszközökkel nem olvasható ki.

I.4.5. A biometria szerepe

A kártya elvesztése esetén a biztonságot szolgálhatja egy PIN (Personal Identification Number) kód is, de az ujjlenyomat- vagy íriszalapú biometriai azonosítás nagyobb biztonságot nyújthat. Ujjlenyomat esetében létezik csak élő ujjat azonosító/elfogadó rendszer, és a közeli jövőben az azonosítást nem egy külső eszköz, hanem maga a kártya is képes lesz elvégezni! Ilyen eszköz ma is létezik, de nem terjedt el még széles körben használata. E mellett fontos kiemelni, hogy a kártyában tárolt biometriai azonosító minta, amit a vett mintával hasonlít össze a rendszer, csak arra használatos, hogy a kártya birtokosát maga a kártya azonosítsa. A sikeres azonosítás után a minta nem játszik szerepet a kommunikációban, és nem hagyja el a kártyát. Rosszindulatú alkalmazások megpróbálhatják kinyerni a kártyából a mintát, hogy a birtokostól ellopott kártya felé ezzel azonosítsák magukat. Ezt az alkalmazások minőségbiztosításával, a digitális aláírások megfelelő hierarchiájával és a minta HW-es védelmével lehet megakadályozni. Egy ilyen rendszerben az alkalmazások hitelessége, és megbízhatósága nyújt garanciát és extra biztonságot.

Az adatvédelem szempontjából is jelentős egy ilyen rendszer alkalmazása, mivel a tárolt információk közül sok adat tartozhat a védendők közé. Ebben az esetben az adatok a szekvenciális kiolvasás ellen a biometria és a kriptográfia vegyítésével tárolandók. Az ujjlenyomatból képzett kód maga az adatok között fennálló kapcsolatot felfedő kulcs. Ebben az esetben is fontos az ujjlenyomatminta kiolvashatatlan tárolása. A *Java kártya* rendszerben, ahol az egyes alkalmazások a kártyára töltődve futnak, fontos a rosszindulatú alkalmazások elől is védeni ezeket az információkat!

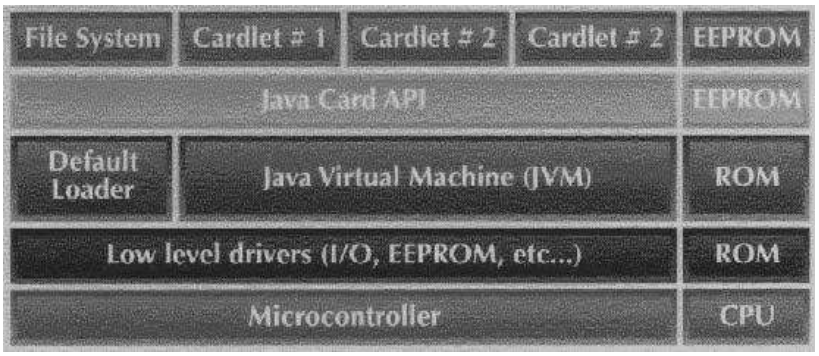
I.5. *Java kártya*

A Java alkalmazások közel 30-szor lassabban futnak egy intelligens kártyán, mint gépi kódban (assembly) megírt társaik. Az IK korlátozott memóriakapacitása miatt sem lehetséges, hogy a teljes JVM rákerüljön egy *Java kártyára* és a Java minden képessége kihasználható legyen. A Java Kártya egy kisebb JVM-et tartalmaz, és az egyes fejlesztőkörnyezetek a szabványban meghatározott csomagokat biztosítják ahhoz, hogy a Java

nyelven megírt alkalmazások ezen csomagok segítségével tudjanak kommunikálni a kártyával.

A `.java` forrás a *Java kártya* fejlesztőkörnyezethez adott JDK alá illesztett csomagok kiegészítésével és a Java fordítóval `.class` bájtkóddá alakítható. Ezek után következik a kártyára tölthető applet elkészítése (fejlesztőkörnyezettől függő kiterjesztésű állomány a fejlesztőkörnyezetet tartalmazó számítógépen), majd a kártyára töltés, melyben a fejlesztőkörnyezethez adott segédprogramok segítenek. A két műveletet végző segédprogram a *linker* és *loader* nevet kapta, az általuk elvégzett feladat jellegéből adódóan, de az egyes környezetekben ettől eltérő nevekkel találkozhatunk (pl. MakeSolo és LoadSolo a Cyberflex kártya esetében).

A Cyberflex esetében alkalmazott ábrázolás általánosan is szemlélteti a memóriák és tartalmuk közötti összefüggéseket:



Amikor az alkalmazás a kártyán van, akkor aktivizálható, és ekkor már a kártya virtuális gépe gondoskodik az alkalmazás futtatásáról.

Itt fontos megjegyezni, hogy a *Java kártya* nem a Java nyelv butításából jött létre. A Java Card VM a Java VM-nek alrésze (subset), de alkották, és nem a Java VM-ből hozták létre. A Java logikáját követve az intelligens kártyák fizikai paramétereinek figyelembevételével készült el a *Java kártya* specifikációja, és ennek alapján az egyes cégek megvalósítása, fejlesztőkörnyezete.

Elképzelhető, hogy a kis memóriakapacitás miatt nem fér el az alkalmazás az eszköz szabad memóriájában. Ilyenkor vagy a memóriából szabadítunk fel, vagy az alkalmazást próbáljuk meg optimalizálni. A *loader* eszközök többnyire GUI alapúak, és néhány parancs a grafikus felhasználói felületről az egér használatával egyszerűen elérhető. Ilyen többek között a szabad memóriát lekérdező parancs. Az érkező válaszból kiszámítható a rendelkezésre álló memória, amelybe alkalmazásunkat be kell szorítani.

A CD-ROM mellékleten a `JavaCard/wrankl` alkönyvtárban található szimulációs programot nézve (Wolfgang Rankl munkája, német vagy angol menükkel telepíthető Windows 3.1/95 operációs rendszerre) elképzelhető, hogy a közeli jövőben megjelennek a jelenleginél is egyszerűbben használható alkalmazások. A Java Card Commander vagy a Smart Card Commander ötlete adott, a megvalósítás várható.

A biztonság területén az egyes alkalmazások sokszor a „security by obscurity” elvét alkalmazzák, miszerint a titoktartás biztonságában bíznak. Ez nem alkalmazható a Java esetén, mivel a Java alkalmazások forráskódja is visszanyerhető, így a már elkészített alkalmazásokból is lehet tanulni, sőt módosításokkal újabb alkalmazások készíthetők. Természetesen ebben az esetben tiszteletben kell tartani az ide vonatkozó szerzői jogi szabályokat. A könyv CD-mellékletén a `JavaCard` alkönyvtárban található anyagok is ebbe a kategóriába tartoznak.

A biztonságot matematikailag igazolt helyességű eljárások szolgálhatják, de nem állunk messze attól sem, hogy az információ (mit tud? - jelszót) vagy az eszköz (mivel rendelkezik? - kártya) azonosítása helyett az egyént azonosító (ki ő?) biometrikus eljárások széleskörűen elterjedjenek a *Java kártya* rendszerekben. Az egyik ilyen lehetőség a mobiltelefonokban alkalmazott *Java kártya* alapú hangazonosító rendszer alkalmazása, mely a mesterséges intelligencia segítségével öntanuló rendszert is eredményezhet. Minél többet beszél az eszköz tulajdonosa, annál nagyobb biztonsággal ismeri fel a hang alapján a jogos kártyabirtokost. A *Java kártya* a Java nyelv platformfüggetlenségétől az eszközfüggetlenség felé tarthat, ha az egyes implementációk is kompatibilissé válnak.

I.5.1. A *Java kártya* jelentősége más rendszerekben

Az intelligens kártyát alkalmazó területek szaporodásával szaporodnak a kártyák is. Ennek több hátránya is lehet, a több PIN kód megjegyzésétől kezdve a sok kártya állandó nyilvántartásán keresztül kényelmetlen hordozásukig. Kezdetben az intelligens kártyák memóriájuk korlátozott méretei miatt inkább egy-egy célfeladatra voltak alkalmasak, de más okoknál fogva az egyes kibocsátók napjainkban is jobban kedvelik a célfeladatra alkalmas kártyákat, míg a felhasználók a több célra használható (multifunkcionális) kártyákra szavaznak.

A *Java kártyán* nem szükséges tárolni az alkalmazásokat, mivel azok letölthetők a kártyára, innen az applet alapján a *cardlet* név is. Ennek köszönhetően nagyobb memória all rendelkezésre különböző adatok tárolására. Ennek következménye, hogy ezek az adatok több, egymástól különböző alkalmazást képesek kiszolgálni, és ez vezet a multifunkcionális kártyához.

Az intelligens kártya terjedését elsősorban az elfogadóhelyek (a kártyakezelő terminálok) száma, kompatibilitása és terjedése befolyásolja. Figyelembe véve a forgalomban lévő eszközök számára vonatkozó adatokat és a közeli jövőre vonatkozó becsléseket, a legalkalmasabb kártyakezelő eszköz a mobiltelefon lesz! Már több cég is rendelkezik SIM *Java kártyával* és fejlesztőkörnyezettel, és egy Michel Ugonnak tulajdonított mondás szerint a mobiltelefon egy olyan intelligens kártyaolvasó eszköz lesz, amivel mellesleg telefonálni is lehet. Akit ez a fejlődési irány is érdekel, az ismerkedhet a *J2ME Wireless Toolkit 1.0* csomaggal (`http://java.sun.com/products/j2mewtoolkit`).

A *Java kártya* szabvány sokoldalú alkalmazhatóságának egyik kiváló bizonyítéka a Dallas Semiconductor terméke, az iButton (interactive button - párbeszédes gomb) alapú *Java gyűrű*. A különböző paraméterekkel rendelkező gombok és felhasználási területeinek ismertetése egy külön fejezetet is kitöltene. Az érdeklődők keressék fel a cég és a termék honlapját. [DSC-JRing 00]

A cég által több éve gyártott gomb alakú memóriák és mikroprocesszoros termékek több millió példányban léteznek a világ több pontján futó alkalmazásokban. A megoldás alapja az úgynevezett „1-wire logic" (1 huzalos logika), ami azt jelenti, hogy a gomb formába ágyazott elektronika egyetlen kapcsolódási vonalon kommunikál az író/olvasó eszközzel. Ezen a vonalon keresztül kapja a működéséhez szükséges feszültséget és a kommunikáció során használt adatok is ezen a vonalon utaznak. Egy hatásvadász megfogalmazás szerint az eszköz „úgy boot-ol, hogy sok egyest kap". A Java-alapú iButton gyűrű formában jelent meg először a Java közösségben.

A gyűrűbe ágyazott gombszerű fémkorong tartalmazza az aktív részt. A vízhatlan acélmagba ágyazott elektronika működésében a *Java kártya* 2.0-ás szabványt támogatja, de létezik a 2.1-es szabványt támogató gomb is.

A „blue dot receptor" (kék pontos/gombos érzékelő) nevet viselő író/olvasó eszköz segítségével valósítható meg a kommunikáció a gyűrű és a számítógép között. A gomb átmérője megegyezik a kék gomb átmérőjével. Elég a gyűrűt a kék gombhoz érinteni, de

lehetőség van a gyűrűt az író/olvasóban rögzítve is tartani hosszabb ideig tartó használat esetén. Egy író/olvasóban lehetőség van két gyűrű használatára is.

A gyűrű tartalmaz egy elemet, melynek köszönhetően a nem felejtő memória (nonvolatile RAM) 10 évig megőrzi adattartalmát. Minden egyes gyűrű a gyártók által garantált egyedi 64 bites azonosítóval rendelkezik, de a gyűrű adatainak védelmét külön biztonsági megoldások is szolgálják. Az Egyesült Allamok és Kanada 1998. július 21-én a FIPS 140-1-es (FIPS: Federal Information Processing Standards, 140-1: "Security Requirements For Cryptographic Modules") minőségi tanúsítvánnyal látta el a cég Crypto iButton termékét, mely 1024 bites RSA kódolásra is képes. A termék honlapjáról többféle operációs rendszeren is használható fejlesztőkörnyezet és egy szimulációs alkalmazás is ingyenesen elérhető. A jelenleg elérhető verziók a CD-ROM `JavaCard/javagyuru` alkönyvtárában is megtalálhatók. A Java 2-höz való változatok 1999. június közepén jelentek meg. Egy évre rá már volt újabb változat, így várható 2001-nyarára is egy új generáció megjelenése.

A *Java kártya* jövőjéről csak szűk időintervallumokban lehet becsléseket megengedni. A fejlődés az alig négy éve megjelent 1.0-ás ötlet óta minden képzeletet felülmúlt.

A fejezet elején említett WAP fórumnak van egy külön munkacsoportja, a Smart Card Expert Group (SCEG). Ez a csoport dolgozik azokon a szabványosítási javaslatokon, amelyek a WAP tagoktól érkeznek, hogy a mobil eszközökön megvalósítandó WAP szabványban a smart card szerepét is pontosan definiálják. [WAP 01]

Eközben a Jini szabvány megjelenése az eszközök Java- alapú összekapcsolását határozza és valósítja meg. [SM-Jini 01] A Jini segítségével hardvertől és szoftvertől függetlenül bármilyen digitális berendezéseket hálózatba köthetünk, így azok megoszthatják az információt és együttműködhetnek. Ennek köszönhetően a jövőben megvalósítható egy *Java kártya* alapú mobil kommunikációval is használható integrált komplex eszközök feletti informatikai struktúra. Az elképzeléseknek csak a képzelet szab határt, a megvalósításoknak a piaci igények. A felhasználók részéről az igények és az elképzelések is megvannak.

I.6. A *Java kártya* 2.1.1-es szabványa

Mit tartalmaz a hivatalos *Java kártya*-specifikáció? Röviden összefoglalva: a kártyával történő kommunikáció szabályait, a kártyán kezelhető állományok típusait, hierarchiáját (ez főként az első verzióra volt igaz, a 2.1-es változatban már nem annyira, hiszen a szükséges állományok kezelése Java-s alapon is megoldható), és az ezen állományok kezeléséhez szükséges utasításokat, valamint a biztonsággal kapcsolatos utasításokat (ide érthető a különböző titkosítási algoritmusokat támogató utasítások halmaza is).

Érdemes a szabvány struktúráját áttekinteni, mivel egy adott fejlesztőkörnyezet használatához (bármennyit is térjen el a szabványtól) szükséges ismerni az alapkoncepciót. A szabvány három fő részből tevődik össze, és leírásuk együttesen meghaladja az 500 oldalt. A virtuális gép specifikációja 270, a *Java kártya* API leírása 200 oldalnyi, de a „Runtime Environment" című 70 oldalnyi anyag elolvasása elégséges az ismerkedés első lépéseihez. A könyv CD-ROM-melléklete tartalmazza ez utóbbi két anyag PDF formátumú verzióját. A 2.1.1-es szabvány változatával elérhető egy WinNT 4.0 és Solaris 7 alatt 1.2.2-es Sun JDK-val tesztelt szimulációs környezet (JCWDE - Workstation Development Environment), mely a 9025 port segítségével használható. Legalább 17 inch-es monitorral kényelmesen használható a két ablakot igénylő szerver-kliens kommunikáció szimulálására, az appletünk tesztelésére, így kártya nélkül is tesztelhető a szabványnak megfelelően megírt applet helyessége és futása.

A kártya chipek korlátozott memóriakapacitása miatt már a szabvány szintjén is léteznek nem támogatott és tiltott eszközök vagy típusok. E két kategória mellett van egy megszorításokat tartalmazó halmaz is, melyben a legfeljebb alkalmazható metódusszámtól a legfeljebb 255 bájt hosszú adatmezőjű hivatkozásokon át a maximális tömbelemszám meghatározásáig több megkötés szerepel. Az APDU puffer mérete 37 bájtban van maximálva (5 bájt az APDU fejléc és 32 bájt adat). Az adattípusok korlátozott száma miatt a típuskényszerítés (castolás) használata szükséges. Ez a forráskód olvashatóságát és a kódolás hatékonyságát csökkenti, de a következő generációs környezetekben egyszerűbbé teszi a bővítést.

Egyes kártyákon 32 bájtos verem áll rendelkezésre, így az eljárások hívásával is takarékoskodni kell, mivel a verem megteltekor kivételdobás történik, és át kell gondolnunk az alkalmazás struktúráját. A kódismétlés használata segíthet ezen a problémán, de hátrányként megjelenik a forrás méretének növekedése és az olvashatóság romlása.

Az egyre nagyobb kapacitással és a különböző processzorarchitektúrákkal megjelenő környezetek külön engedményeket vagy korlátozásokat is bevezethetnek. Ilyen szempontból jelentős eltérések mutatkoznak egy 8 bites kis memóriával rendelkező és egy 32 bites RISC processzorral ellátott nagyobb memóriakapacitással rendelkező eszközre épített fejlesztőkörnyezet között, mint például a használható típusok és adatstruktúrák bővülése egy 32 bites felépítés esetében.

I.6.1. Az alapcsomagok

Tekintsük át a szabványban leírt négy csomagot és a köztük lévő hierarchiát, néhol egy rövid magyarázattal kiegészítve.

A négy alapcsomag: a `java.lang` (a Java nyelvhez való kiegészítéseket tartalmazza), a `javacard.framework` (a kártyával való kommunikációs eljárások), a `javacard.security` (az exportkorlátozás nélkül elérhető biztonsági csomag) és a `javacardx.crypto` (a kiegészítő biztonsági csomag, mely az erős, és ezért különböző korlátozások alá vont kriptográfiai eljárásokat foglalja magában).

Az osztályok hierarchiája csomagonként (a `java.lang.*` osztályokat a könyv fő részeiben mutattuk be):

```
1. class java.lang.Object
class java.lang.Throwable
  class java.lang.Exception
    class java.lang.RuntimeException
      class java.lang.ArithmeticException
      class java.lang.ArrayStoreException
```

```
class java.lang.ClassCastException
class java.lang.IndexOutOfBoundsException
  class java.lang.ArrayIndexOutOfBoundsException
class java.lang.NegativeArraySizeException
class java.lang.NullPointerException
class java.lang.SecurityException
```

2. `class java.lang.Object`

`class javacard.framework.AID`
/* az appletek kezelése az Application Identifier azonosítóval */

`class javacard.framework.APDU`
/* az APDU-k és a kommunikációs protokollok kezelése */

`class javacard.framework.Applet`
/* applet kezelése, pl. telepítés, kiválasztás, feldolgozás */

`class javacard.framework.JCSystem`
/* applet végrehajtás, erőforrás, atomi tranzakció és appletközi objektummegosztás kezelése és ellenőrzése */

`class javacard.framework.OwnerPIN` (implements javacard.framework.PIN)
/* PIN kezelése, pl. hibás próbálkozások számolása, blokkolás feloldása */

```
class java.lang.Throwable
  class java.lang.Exception
    class javacard.framework.CardException
      /* a kivételek őse, a metódusok reason változója írja le a kivételdobás okát */
      class javacard.framework.UserException
        /* a programozó által definiált kivételek */
    class java.lang.RuntimeException
      class javacard.framework.CardRuntimeException
        /* egyenes kivétel-alosztályai: APDU, Crypto, ISO, PIN, System, Transaction
        */
        class javacard.framework.APDUException
          /* hibás szerkezetű/tartalmú APDU */
        class javacard.framework.ISOException
          /* szabványban rögzített állapotkóddal (SW) tér vissza */
        class javacard.framework.PINException
          /* az OwnerPIN által kiváltott kivétel */
        class javacard.framework.SystemException
          /* a JCSystem osztályhoz tartozó kivétel, de
          a javacard.framework.Applet.register() is dobhatja például utasítás
          végrehajtásához nem elégséges erőforrás esetén a NO_RESOURCE kóddal */
        class javacard.framework.TransactionException
          /* a JCSystem osztályba tartozó és tranzakciót kezelő metódusok dobhatják,
          pl. BUFFER_FULL kóddal a tranzakció műveleteit a tranzakció befejezéséig
          tároló commit puffer megtelte esetén */
```

`class javacard.framework.Util`
/* tömbökön vagy tömb elemeivel végezhető műveletek, pl. másolás, összehasonlítás, elemkonkatenáció */

`interface javacard.framework.ISO7816`
/* a 3. és 4. alpontokban definiált hexa konstansokhoz rendelt statikus beszédes konstansok gyűjteménye, pl. public static final short `SW_RECORD_NOT_FOUND`, melynek értéke: 0x6A83 */

`interface javacard.framework.PIN`
/* implementálandó többek között a PIN mérete, értéke, próbálkozások száma */

`interface javacard.framework.Shareable`
/* megosztott objektumok azonosítása; csak azok az objektumok oszthatók meg a kártya tűzfalán keresztül, amelyek közvetve vagy közvetlenül implementálják ezt az interfészt */

`3. class java.lang.Object`

`class javacard.security.KeyBuilder`
/* a `buildKey(byte keyType, short keyLength, boolean keyEncryption)` metódussal különböző kriptográfiai kulcsokat készít az aláíró és kódoló algoritmusok (DSA, DES, RSA) számára, akár 2048 bites RSA kulcshosszal is */

`class javacard.security.KeyPair`
/* egyedüli új osztály a Java Card 2.1 szabványhoz képest; kulcsgeneráló metódussal rendelkezve titkos és nyilvános kulcspárok tartály osztálya, mely nem érvényesít semmilyen biztonságot (inicializáláskor `PrivateKey`-ként kell kezelni) */

`class javacard.security.MessageDigest`
/* hash algoritmusok alaposztálya, beépítetten támogatott algoritmusok: MD5, RIPE MD-160, SHA */

`class javacard.security.RandomData`
/* véletlenszám-generálás osztálya */

`class javacard.security.Signature`
/* aláíró algoritmusok osztálya */

`class java.lang.Throwable`
`class java.lang.Exception`
`class java.lang.RuntimeException`
`class javacard.framework.CardRuntimeException`
`class javacard.security.CryptoException`
/* a kriptográfiai algoritmusokkal kapcsolatos kivételek osztálya */

`interface javacard.security.DSAKey`
/* DSA alapú digitális aláíráshoz szükséges kulcsok készítése és kezelése */
`interface javacard.security.DSAPrivateKey`
`interface javacard.security.DSAPublicKey`

`interface javacard.security.Key`
/* minden *Key interfész alapja */

`interface javacard.security.PrivateKey`
/* aszimmetrikus kódolásban használt privát kulcsokhoz */
`interface javacard.security.DSAPrivateKey`
`interface javacard.security.RSAPrivateCrtKey`
/* a kínai maradéktétel alkalmazásával történt RSA alapú aláírásra vonatkozó interfész */
`interface javacard.security.RSAPrivateKey`

```
interface javacard.security.PublicKey
  /* aszimmetrikus kódolásban használt nyilvános kulcsokhoz */
  interface javacard.security.DSAPublicKey
  interface javacard.security.RSAPublicKey

interface javacard.security.SecretKey
  /* szimmetrikus kódolásban használt titkos kulcsokhoz */
  interface javacard.security.DESKey
```

```
4. class java.lang.Object

class javacardx.crypto.Cipher
  /* titkosító algoritmusok alaposztálya, feladatai közé tartozik a kulcskezelés, a titkosító
  és megoldó algoritmusok kezelése */
interface javacardx.crypto.KeyEncryption
  /* a setKeyCipher(Cipher keyCipher) állítja használatba a Cipher objektumot, míg
  a getKeyCipher() a dekódoláshoz szükséges Cipher objektumot adja vissza; alapér-
  telmezés szerint null értékűek, ekkor nincs titkosítás */
```

Az RSA alapú titkosításnál a kínai maradéktétel (CRT - Chinese Remainder Theorem) alapján az egyes számítások gyorsabban elvégezhetők. Egy 512 bites aláírás esetén akár háromszor annyi időbe is telhet az aláírás elvégzése. A tétel ismertetésétől most tekintsünk el, az irodalomban megtalálható a matematikai leírása. [Schne 96] A könyvben a matematikával és a biztonsággal foglalkózó fejezetek is foglalkoznak a kriptográfiával. Az ELTE programozóképzésében az elsős Bevezetés a matematikába című jegyzetben pedig megtalálható a részletes ismertető is.

Fontos kiemelni a szabványban lévő általánosságokat, így az egyes kriptográfiai funkciókat leíró interfészek és osztályok azon tulajdonságát, amely alapján az egyes implementációk bővíthetők (pl. a már említett ECC alapú kriptográfia alkalmazására).

Az osztályok hierarchiája egymáshoz viszonyítva:

```
class java.lang.Object
  class javacard.framework.AID
  class javacard.framework.APDU
  class javacard.framework.Applet
  class javacardx.crypto.Cipher
  class javacard.framework.JCSystem
  class javacard.security.KeyBuilder
  class javacard.security.KeyPair
  class javacard.security.MessageDigest
  class javacard.framework.OwnerPIN
  class javacard.security.RandomData
  class javacard.security.Signature
  class java.lang.Throwable
    class java.lang.Exception
      class javacard.framework.CardException
        class javacard.framework.UserException
      class java.lang.RuntimeException
        class java.lang.ArithmeticException
        class java.lang.ArrayStoreException
        class javacard.framework.CardRuntimeException
          class javacard.framework.APDUException
          class javacard.security.CryptoException
```

```
          class javacard.framework.ISOException
          class javacard.framework.PINException
          class javacard.framework.SystemException
          class javacard.framework.TransactionException
        class java.lang.ClassCastException
        class java.lang.IndexOutOfBoundsException
          class java.lang.ArrayIndexOutOfBoundsException
        class java.lang.NegativeArraySizeException
        class java.lang.NullPointerException
        class java.lang.SecurityException
  class javacard.framework.Util

interface Hierarchy

 interface javacard.security.DSAKey
   interface javacard.security.DSAPrivateKey
   interface javacard.security.DSAPublicKey
 interface javacard.framework.ISO7816
 interface javacard.security.Key
   interface javacard.security.PrivateKey
     interface javacard.security.DSAPrivateKey
     interface javacard.security.RSAPrivateCrtKey
     interface javacard.security.RSAPrivateKey
   interface javacard.security.PublicKey
     interface javacard.security.DSAPublicKey
     interface javacard.security.RSAPublicKey
   interface javacard.security.SecretKey
     interface javacard.security.DESKey
 interface javacardx.crypto.KeyEncryption
 interface javacard.framework.PIN
 interface javacard.framework.Shareable
```

I.6.2. Tűzfalak, kontextusok, objektumok megosztása

Egy kártya több különböző alkalmazás futtatására is képes, ezért az alkalmazások által elérhető adatok biztonsága érdekében tűzfal megoldás alkalmazható. A tűzfal futási időben biztosít védelmet az appletek számára, de megengedett a futtató környezet többlet biztonsági ellenőrzése is. Egy alkalmazás csak akkor érheti el egy másik alkalmazás adatait vagy objektumait, ha az a másik alkalmazás engedélyezte ezt. Az objektummegosztás kezelésére a `Shareable` interfész szolgál. A kártya biztonsági felügyelete ezen felül is végez hozzáférési ellenőrzéseket a hibás PIN-megadástól az illetéktelen hozzáférés-próbálkozások kezeléséig minden szinten.

A PIN-megadásokat egy számláló kíséri a `getTriesRemaining() byte` típusú metódusban, és ha a logikai típusú `check(byte[] pin, short offset, byte length)` metódus `false` értékkel tér vissza, akkor ezt a számlálót csökkenti. Ha a számláló értéke 0, akkor blokkolja a PIN-t, de az egyes implementációktól függ, hogy mennyi a számláló maximuma, és a blokkolás időszakos vagy végleges-e. Az időszakos blokkolást a mester PIN (magasabb jogosultságot biztosító PIN-kód) segítségével feloldhatjuk. A végleges blokkolás esetében a kártya operációs rendszere visszaállíthatatlan memóriatörlést is végezhet az „inkább elvesztem, mint felfedjem!” elv alapján! Ha a `check()` eljárás `true` értékkel tér vissza és a PIN nincs blokkolva, akkor a számlálót maximumra állítja. Itt érdemes megjegyezni annak az elképzelésnek a gyengeségét, amikor a kártya a PIN-kód alapján

kerül alacsonyabb vagy magasabb biztonsági szintre attól függően, hogy a mester PIN-t írták be, vagy a felhasználói PIN-kódot. Idő és türelem kérdése a mester PIN kinyerése, hiszen az ismert maximális próbálkozási számot elérve a számlálót maximumra állítjuk a helyes PIN megadásával, és így előbb- utóbb a hálóba akad a másik helyes megoldás is, a mester PIN.

Az alapkoncepció szerint a kártya kezelőjében/birtokosában sem szabad bízni, aki tudatosan adhat maximális hozzáférést a kártyán tárolt adatokhoz egy rosszindulatú applet számára, így nyerve információt hasonló kártyák megtörésére, vagy éppen a mester PIN kiderítésével akarja manipulálni a kártyában tárolt adatokat. Több lehetőség is létezik a koncepciót kikerülni szándékozók ellen. Egy megoldás, amikor a magasabb biztonsági szint eléréséhez egy külön kártya, egy mester kártya szükséges, és csak ez használható a blokkolt PIN oldásához. Az ilyen kártyát szokták SAM (Secure Access Module) kártyának nevezni. Ez a mester kártya szintén rendelkezik PIN-kóddal és más kiegészítő biztonsági elemekkel és ellenőrzésekkel is, mert egy ilyen kártya biztosíthatja a rendszer összes kártyájához való hozzáférést, ami nagy biztonsági kockázatot jelent. Képzeljünk el egy olyan szállodai kulcsot, ami a szálloda összes szobájának ajtaját nyitja.

A biztonság alulról építkezik, vagyis a minimális jogokat növeli szükség esetén, de csak annyi időre, ameddig arra szükség van. Futásának idejére az applet egy biztonsági burokba kerül, amit kontextusnak (context) nevezünk. Ez egy egyedi védett teret biztosít az objektumok és az adatok számára. Egy időben csak egy kontextus lehet aktív, az éppen kiválasztott applet kontextusa. Egy adott applet a vele egy kontextusban lévő appletek objektumait képes csak elérni, így az egy csomagban lévő appletek elérhetik egymás objektumait. A kontextusok váltása esetén az aktuális vermelődik, majd a másik applet befejeztével (saját kontextusa él futásának idején) a visszaváltás következik, így biztosítva az egymásba ágyazhatóságot. Itt is figyelembe kell venni a verem méretét. A fő/alap kontextus a futtató környezet (JCRE - Java Card Runtime Environment) kontextusa, amely a verem legalsó eleme vagy éppen aktív, és a kontextus váltásával hív egy appletet.

Az éppen aktuális applet által az aktuális kontextust túllépni szándékozó esemény a `java.lang.SecurityException` kivétel dobását eredményezi. A kontextusokon keresztül megvalósítható objektumelérést általános elvek szabályozzák, így erősen implementációtól függő a konkrét megvalósítás. Objektum megosztásnál egy másik applet objektumára történő hivatkozás esetén kontextusváltás történik a hivatkozott objektum kontextusába, majd a verem mutató alapján visszatéréskor kontextus visszaváltás történik.

Az egyes alkalmazások egy gyártófüggő `install(byte[], short, byte)` metódussal kerülnek a chipbe, és a metódus sikeres visszatérésének eredménye (a 0x9000 SW páros mellett) a chipben elhelyezett és futtatásra kiválasztható alkalmazás. Ezt nevezhetjük cardlet-nek is, de a kártyán belül ez egy applet. Az applet és az őt értelmezni képes virtuális gép életciklusának függvényében történik az applet felhasználása. Ez az adott implementációtól függ, de egy betartandó megkötés adott a *Java kártya* szabvány szerint: az `install()` metódust meghívó paraméterek együttes hossza legfeljebb 32 bájt hosszú lehet.

I.6.3. A virtuális gép életciklusa

A kártya virtuális gépe (Java Card Virtual Machine) a kártya testében lévő modulban elhelyezett chip ROM memóriájában van, és így életciklusa megegyezik a chip életciklusával.

A perzisztens memóriában (EEPROM) lévő adatok akkor is megmaradnak, ha megszűnik a kártya tápellátása. Ebben az esetben a JCVM csak időszakosan áll le, mert újbóli használat esetén újraindul, és megpróbálja befejezni megszakított munkáját. A kártyát befogadó egységben (Card Acceptance Device - CAD) töltött idő megegyezhet az életcik-

lussal. Egyes esetekben be szokták állítani egy perzisztens változóban a következő reset alkalmával alapértelmezett appletet, így ha a kártyát legközelebb az olvasóba helyezzük, akkor ez az applet kerül kiválasztásra. Szűkebb értelemben az olvasóban APDU-k forgalmazásával töltött idő az életciklus, de ebben az esetben a kártya olvasóba helyezése és kivétele között elképzelhető a teljes passzivitás is. Ez utóbbi esetben a kártya nem kap reset jelet sem, így nem is válaszol, tehát nincs használatban.

Egyes kiemelt biztonságú tranzakciók végrehajtása az egymáshoz képest master/slave viszonban lévő kártyákhoz köthető, amikor az egyik kártyán végzett műveletek csak akkor hajthatók végre, ha a másik kártya is jelen van. A mester kártyát (SAM kártya) a rendszer a tranzakció elején azonosítja, majd a szolga kártyán folynak a műveletek. A mester kártya jelenlétének folyamatos időközönkénti ellenőrzése is megoldható, de egyszerűbb a kártya kihúzását jelző kivétel dobás/elkapás. Ez a szolga kártya kihúzása után már nem ugyanolyan jelentőségű, mint azelőtt, tehát a két kártya kihúzási sorrendjét is figyelembe kell venni! A mester kártya több szolga kártya használata során is maradhat az olvasóban, míg egy szolga kártya csak addig használható, amíg a mester kártya is jelen van. Mindezekből érzékelhető a kártya, vagyis a JCVM életciklusának pontos meghatározásának fontossága az alkalmazás *tervezésekor*.

I.6.4. Az applet (cardlet) életciklusa

A kártyán lévő appleteknek van egy azonosítójuk, amit az AID (Applet IDentifier) jelöl. Adott DF-en belül használhatók az 5 bites rövid azonosítók (short AID) is. A hibák esetén dobott kivételek egy részét már a Java virtuális gépe is jelzi (Java nyelv specifikus kivételek), és ezt egészíti ki az applet futtatásakor a JCVM (kártya specifikus kivételek). A maradék kivételeket az API kezeli, ilyen lehet az alkalmazott metódusok paraméterellenőrzése.

Egy a kártyára telepített applet életciklusa szűkebb értelemben a rá alkalmazott `select()` metódus meghívásával kezdődik, és a `deselect()` meghívásával végződik. A metódus hexa felépítése: CLA=0x00, INS=0xA4, P1=0x04, de ha P1 más értékű, akkor az utasítás nem kiválasztást jelent, így az APDU parancsként hajtódik végre. Kiválasztáskor a rendszer kinullázza az APDU puffert. Természetesen a fejlesztőkörnyezet segítségével a programozónak nem kell tudnia ezeket a kódokat, de amennyiben tudja, úgy egy-egy parancsot is kiadhat a GUI (grafikus felhasználói felület) megfelelő paraméter ablakain keresztül, így az elemi utasítások szintjén elemezhetnénk tovább az életciklusok alakulását. Nem tesszük, inkább térjünk vissza a kiválasztáshoz.

Ha a `select()` metódus sikeres volt (az applet nem utasította el a kiválasztást `false` visszajelzéssel, és nem dobódott kivétel sem), akkor a `true` visszatérési érték alapján a `process()` metódust hívja meg a rendszer az APDU-k közvetítésére vagy végrehajtására. Ekkor az előzőleg kiválasztott appletre meghívódik a `deselect()` metódus, mely memóriatakarításra jogosít fel a következő applet részére.

A `process()` metódus `ISOException(short sw)` kivételt dobhat, ahol `sw` a kivételdobást okozó esemény kódja. Itt kell kiemelni, hogy a `throw()` helyett a `throwIt(sw)` használatos a kivételek eldobására. A kódhoz rendelt esemény az ISO/IEC 7816-4 alpontban definiált eseményhalmaz eleme, de az egyes fejlesztőkörnyezetekhez adott kézikönyv, illetve a beépített statikus változók segítséget nyújthatnak az értelmezésükben. Egy példával szemléltetve mindezt: a 0x6999 kód jelentése `SW_APPLET_SELECT_FAILED`).

I.6.5. Perzisztencia

Az applet életciklusa alatt objektumok jöhetnek létre, és ezek többnyire perzisztens objektumok. Egy objektum akkor perzisztens, ha az applet `register()` metódusát meghívták, vagy ha az objektumra történő hivatkozást tárolta bármely perzisztens objektum

vagy osztály egy saját mezőben. Ez a belső adatstruktúra integritásának megőrzését is szolgálja (emlékezzünk a reset jelre újjáéledő JCVM tulajdonságára).

A perzisztens adatokkal szemben a tranziens adatok a reset jeltől a következő reset jelig, vagy a tápellátás megszakadásáig élnek. Amikor megszakad a tápellátás, vagy a kártya reset jelet kap, akkor a tranziens adatok az alapértéket veszik fel, a folyamatban lévő tranzakció abortál és az éppen kiválasztott applet implicit kiválasztatlan lesz, amikor a kártya újraéled. A tranzakciók olyan műveletsorok, melyek közül vagy mindegyik végrehajtódik, vagy egy sem és ekkor a tranzakció előtti eredeti állapot áll vissza.

A *Java kártya* nem kezeli a Java transient kulcsszavát. A tranziens adatokat biztonsági megfontolásokból sohasem szabad „nemfelejtő" memóriákban tárolni, csak a RAM memóriában. Ebben az esetben a tranzakció megszakadása nem okozza az előző adatok visszaállítását (a tranziens adatokon végzett műveletek nem tranzakció alapúak). A tranziens objektumok ideálisak kis mennyiségű és sűrűn változó adatok tárolásához. A kártyában lévő titkos adatokat, mint például a titkos kulcsokat szintén tranziens módon kezeljük, és az „eseményjelző" a `CLEAR_ON_RESET` vagy a `CLEAR_ON_DESELECT` beállítással jelzi, hogy milyen eseményre törlődjenek a memóriából.

A kriptográfiai eljárások során a memóriában lévő titkos kulcsokat nem szeretnénk kiadni, ezért a kódolási eljárás közben megszakított működés (kártya kihúzása az olvasóból) után a reset jelre bekövetkező törlés esemény is beállítható az adott objektumra. Egy ilyen beállítás az adat létrehozásakor megtörténik, mint például ha egy tranziens bájt tömböt hozunk létre: `makeTransientByteArray (short length, byte event)`, ahol az `event` paraméter a `CLEAR_ON_*` értékek egyike lehet.

A tranzakciók lehetnek atomi és nem atomi műveletek a programozó igényei szerint, illetve a megfelelő módot igénylő műveletek szerint. Tranzakciós műveletet főként a perzisztens adatokkal végzünk, mely adatok kényességét perzisztens mivoltuk is jelzi. A művelet során az atomicitás biztosítja, hogy a tranzakciót végző utasítások megszakadása esetén a tranzakcióban szereplő adatmezők a tranzakció előtti állapotba kerüljenek vissza. Ekkor nem nyerhetők vissza a tranzakció adatai, így a külső manipuláció ellen is védett a művelet és az adatok integritása. Másolás esetén az `Util.arrayCopy()` metódus garantálja a mindent vagy semmit elvet: vagy átmásolja az összes adatot, amit paraméterül kapott, vagy nem másol át semmit. A programozó kikapcsolhatja ezt a szolgáltatást az `Util.arrayCopyNonAtomic()` használatával, különben a perzisztens objektum, osztály vagy tömb (array) adatmezőkkel végzett írás/frissítés atomi módú lesz.

A `JCSystem` osztály őrködik a kártyán végbemenő atomi tranzakciók felett, az ehhez szükséges erőforrások karbantartása és az appletek közötti objektummegosztás felett.

A tranzakciós modell szerint az atomi műveletek a `beginTransaction()` metódussal indulnak. A tranzakciók nem ágyazhatók egymásba, ilyen esetben kivételdobás történik. A műveletek a tranzakció kezdete után feltételesen mennek végbe, ami azt jelenti, hogy a beírt adatok visszaolvashatók, de a műveletek és a változók tartalma csak a `commitTransaction()` végeztével válnak perzisztenssé (kerülnek a perzisztens tárolóba). A tranzakciót az `abortTransaction()` megszakíthatja, amit az applet maga is meghívhat. A kontextusváltás nem befolyásolja a tranzakció menetét.

Tranziens objektumok esetén a `CLEAR_ON_RESET` típusúak perzisztens objektumként viselkednek. Ezek csak akkor férhetők hozzá, ha az aktuális kontextus megegyezik az objektumot létrehozó applet kontextusával.

A `CLEAR_ON_DESELECT` típusú tranziens objektumok csak akkor hozhatók létre vagy érhetők el, ha az aktuális kontextus megegyezik az aktuálisan kiválasztott applet kontextusával, ellenkező esetben `java.lang.SecurityException` kivétel dobódik a megfelelő esemény kódjával. Ezek az objektumok olyan adatokat tartalmaznak, melyeket törölni kell, ha az objektum kiválasztása megszűnik (nem maradhatnak a memóriában a kártya következő használatáig), és ez a típustól függően automatikusan megtörténik.

Az azonos csomagban lévő appletek mindkét típusú objektumot megoszthatják egymással. A `select()`, `deselect()`, `process()`, illetve `install()` eljárásokból visszatérve a folyamatban lévő tranzakciók is megszakadnak, az általuk létrehozott példányok megszüntetésre kerülnek, a rájuk való hivatkozások `null` értéket kapnak. Elég brutális a reakció egy kis próbálkozásra: felfüggesztés és takarítás. A vétlen próbálkozások elkerülésére leellenőrizhető, hogy fut-e tranzakció a `getTransactionDepth()` segítségével, amely jelenleg két értéket vehet fel: 0 akkor, ha nincs folyamatban tranzakció, 1 akkor, ha van. A *jelenleg* szó sejtet valamit a jövőre nézve, de egy többmélységű tranzakciókezelés esetén a biztonsági mechanizmus nagymértékű változása is borítékolható.

A commit műveletig a tranzakció kimenetét a commit puffer tárolja, melynek mérete az egyes implementációk memóriakapacitásától függ, de a szabvány meghatározza a `getMaxCommitCapacity() short` típusú metódust, mellyel lekérdezhető a rendelkezésre álló szabad terület.

I.7. Hello World a *Java kártyán*

A rendelkezésre álló fejlesztőkörnyezetben a meglévő eszközzel (pontosabban a ROM memóriában lévő operációs rendszerrel és virtuális géppel) és az eszköz memóriakapacitásának ismeretében a biztonsági szabályokat betartva elkészíthetjük az első *Java kártya* alkalmazásunkat. Ezt a támogatott Java fordítóval az adott környezethez kapott segédprogramokkal az eszközre tölthetjük, így a kártya applet használható is. Mindez bonyolultnak hangzik, de ez csak az e sorokat író hibája. Nem szabad meghátrálni, ki kell próbálni! Az első ismerkedő alkalmazás a HelloWorld szokott lenni.

A példa tesztelhető a már említett szimulátorral, ezt a következő címről lehet letölteni: `http://java.sun.com/products/javacard`. A csomagban több kisebb példa található, melyekkel elkezdhető a tesztelés, a magyarázó kommentekkel ellátott forráskódok olvasgatásával pedig a behatóbb ismerkedés. A 2.1.1 szabványnak megfelelő egyik alkalmazás, melynek eredeti szerzője Mitch Butler:

```
package com.sun.javacard.HelloWorld;
import javacard.framework.*; /* a kommunikációhoz szükséges csomag */
public class HelloWorld extends Applet {
   private byte[] echoBytes;
   private static final short LENGTH_ECHO_BYTES = 255;

   protected HelloWorld() {
      /* az osztály install() metódusa hozhatja létre az objektumot */
      register();
      echoBytes = new byte[LENGTH_ECHO_BYTES]; /* a kiírandó tömb */
   }

   public static void install(byte[] bArray, short bOffset, byte bLength) {
      new HelloWorld();
         /* installációs paraméterek, kezdőcím, adathossz, ami max. 32 bájt lehet */
   }

   public void process(APDU apdu) { /* az APDU, vagyis az adat küldése */
      byte buffer[] = apdu.getBuffer();
         /* az APDU-k egy puffer-en keresztül íródnak be és olvasódnak ki */
      short bytesRead = apdu.setIncomingAndReceive();
         /* alapeljárás az adatok küldésére választ is várva! */
      short echoOffset = (short)0;
```

```
      while ( bytesRead > 0 ) { /* ciklus, amíg van adat az APDU-ban */
         Util.arrayCopyNonAtomic(buffer, ISO7816.OFFSET_CDATA, echoBytes,
            echoOffset, bytesRead);
         echoOffset += bytesRead;
            /* lépked az olvasandó tömbön, a másolás nem atomi, azaz nem
               tranzakció alapú */
         bytesRead = apdu.receiveBytes(ISO7816.OFFSET_CDATA);
            /* a még olvasandó bájtok száma, vagy 0 */
      }
      apdu.setOutgoing();
         /* minden esetlegesen megmaradt bejövő adatot eldob */
      apdu.setOutgoingLength( (short) (echoOffset + 5));
         /* beállítja a küldendő adat hosszát */
      apdu.sendBytes( (short)0, (short) 5);
      apdu.sendBytesLong( echoBytes, (short) 0, echoOffset);
         /* kiírja az adatot */
   }
}
```

Eddig tartott a fuvar, de akiket megfogott a Java-galaxisnak ez a csillaga, és folytatni kívánják az ismerkedést, azoknak a szabvány áttekintése utáni szakaszra figyelmükbe ajánlom e két könyvet:

- Uwe Hansmann, Martin S. Nicklous, Thomas Schäck, Frank Seliger: *Smart Card Application Development Using Java*, Springer, 2000.
- Zhiqun Chen: *Java Card Technology for Smart Cards*, Java Series, Addison-Wesley, 2000

További szerencsés, élményekben gazdag utazást a Java galaxisában!

A fejezetben olvasható HTTP címeket a CD-ROM-on a `JavaCard` alkönyvtárban lévő `cimek.html` foglalja össze egy HTML oldalban.

Köszönetnyilvánítás: Magyari Péternek (a Schlumberger cég hazai képviselője) a Cyberflex Core Kit rendelkezésre bocsátásáért, Szenttornyai Lászlónak a *Java gyűrű* biztosításáért, a Neumann János Számítógéptudományi Társaságon belül működő Intelligens Kártya Fórum szervezetnek a szakmai forrásmunkák biztosításáért, több szerzőtárs észrevételeiért, ami a fejezet érthetőbb megfogalmazását eredményezte, és a szak- és TDK dolgozatot író hallgatóknak a türelmes hozzáállásért, ami a kezdeti ismerkedő lépések önálló megtételében volt nélkülözhetetlen.

Köszönet Chris Stanfordnak a C.A.F.E. demo CD-re írásának engedélyezéséért.

CD: A CD-ROM mellékleten a `JavaCard` alkönyvtárban található alkalmazások a fejezet anyagának példákkal történő jobb megértését segítik elő a teljesség igénye nélkül. Az alkalmazásokhoz szükséges fejlesztőkörnyezet is, de működésük tesztelhető eszközt nem igénylő szimulátorprogramokkal is. A *Java gyűrű* szimulátora (Integrated Development Environment) és a Java Card 2.1.1-hez elérhető szimulátor (a már említett JCRE) letölthető az Interneten keresztül. A CD-n található még néhány szimulációs alkalmazás, melyek az intelligens kártyák világával történő barátkozást segítik elő. A fejezetben szereplő Jini-ről, a biztonságról és a perzisztencia fogalmáról részletesebb anyag is szerepel a könyv más fejezeteiben vagy a weben.

A CD-ROM mellékleten található segéd- és önálló alkalmazások, dolgozatok szerzői:

- **Goldschmidt Balázs**, **Nagypál Gábor**: Út a javakártyáig. BME Műszaki Informatika szak - V. évfolyam
- **Ondi Attila**: Elektronikus pénztárca Java Gyűrűre, ELTE-TTK Programozó - matematikus szak - III. évfolyam
- **Szatmáry Botond**, és **Takács Mihály**: Beléptető rendszer. ELTE-TTK Programtervező - matematikus szak - IV. évfolyam
- **Szunyoghy Zsolt**: Cyberflex *Java kártya* alkalmazása. ELTE-TTK Programtervező - matematikus szak - IV. évfolyam