Kávéfőzés lépésről lépésre (7. rész)
Két folyamat beszélget. Az egyik azt mondja...

Az igazat megvallva még egy viccet sem hallottam, ami így kezdődött volna. Ez egyrészt szerencse, mivel némiképp alátámasztja azt a magamról alkotott képet, hogy nem vagyok arany fokozatú kocka. Másrészt furcsa, hogy a folyamatok közötti kommunikációt (IPC, inter process communication) biztosító eszközök gazdag tárháza eddig nem jelentett termékeny táptalajt az ismerten magas színvonalú informatikus humornak.

Az *IPC*, ahogy az az előző mondatból sejthető, egy gyűjtőfogalom. Az ide sorolható programozási eszközök nagyobb része ugyanazon a számítógépen futó folyamatok közötti kommunikációt tesz lehetővé. Ilyen például a jól ismert csővezeték, ami akár parancshéjból is könnyen létrehozható. Nem kisebb azonban azoknak az eszközöknek a jelentősége, melyek távoli folyamatok kommunikációjára használhatóak. Ezek közül a legszélesebb körben használt megoldás a foglalat.

A foglalat nagy népszerűségét leginkább annak köszönheti, hogy programozói szemszögből nem sokban különbözik egy közönséges állományleírótól. Az első lépés a megnyitás, ami a kapcsolat felépítését jelenti. Ezt követheti az írás és olvasás, azaz egy foglalat biztosítja mindkét irányban a kommunikációt. Végül le kell zárni, vagyis a kapcsolatot meg kell szüntetni. Ez tehát újabb ékes példája az unalomig ismételt bölcseletnek, mely szerint ***UNIX*** alatt minden fájl. Mi több, a módszer annyira jól működik, hogy nem kizárólag ***UNIX*** rendszereken honosodott meg. Valójában minden olyan operációs rendszer, ami képes ***TCP/IP*** alapú kommunikációra, lehetőséget ad valamilyen formában foglalatok létrehozására. Így nem véletlen, hogy a Java szabványos osztálykönyvtárában is találunk olyan osztályokat, melyekkel foglalatok hozhatók létre. A sorozatot megkoronázó, befejező cikkben ezek használatát vizsgáljuk meg közelebbről.

Mi az a foglalat?

A foglalat két folyamat közötti két irányú kommunikációnak egy végpontja. Jellemzően kiszolgáló-ügyfél felépítésű adatcserét biztosít. A kiszolgáló létrehoz egy ***foglalatot (socket)***, és ***hozzáköti (bind)*** egy úgynevezett kapuhoz (port). Ez egy egész szám, ami azonosítja a számítógépen futó folyamatot. A kiszolgáló ezután várakozik az ügyfélre. Amint egy ügyfél jelentkezik, annak kapcsolódási kérelmét ***elfogadja (accept)***, és ezzel kezdetét veszi az adatcsere. Az ügyfél ugyanúgy egy foglalatot hoz létre. Általában a kapuhoz való kötést az operációs rendszerre bízza, mivel ennek a foglalatnak a kapu száma érdektelen. Ismeri a kiszolgáló IP címét és a távoli folyamat kapu számát, ehhez ***kapcsolódik (connect)***. A kapcsolódással a kiszolgáló egy új foglalatot kap egy másik kapuhoz kötve, így a kommunikáció már egy másik kapun keresztül folyik. Eközben, mivel a kiszolgáló meghirdetett kapuja felszabadult, új ügyfélre várakozhat.

Java-ban mindez igen egyszerűen működik. A megoldáshoz két osztályt használhatunk a `java.net` csomagból. Az egyik a `Socket`, ami a fent vázolt kommunikációnak egy végpontját jelenti. A `ServerSocket` pedig egy olyan foglalat, amit kiszolgálók használhatnak ügyfelek csatlakozási kérelmeinek figyeléséhez és elfogadásához. Ezek az osztályok elrejtenek minden, operációs rendszertől függő megvalósítási kérdést, így a ***Java***-tól jól megszokott platformfüggetlen módon programozhatók.

Mennyi az idő?

Helló világ helyett ismét kísérletet teszünk valami olyasmi készítésére, aminek haszna is van. Az *1. Listában* egy egyszerű idő-kiszolgáló kódja látható.

1. Lista Egy egyszerű időkiszolgáló

```
import java.net.ServerSocket;
import java.net.Socket;
import java.io.PrintWriter;
import java.io.IOException;
import java.util.Date;

/**
* Egy ido kiszolgalot hoz letre, ami egy egyetlen szalon futo
* TCP szolgaltatast jelent. A csatlakozo ugyfel foglalatara
* kiirja a pillanatnyi datumot, majd lezarja a kapcsolatot.
* Jelenleg a kiszolgalot csak Ctrl+C-vel lehet leallitani.
* @see #run
*/
public class IdoSzolgaltato implements Runnable {

        /**
        * A kiszolgalo foglalata.
        */
        private ServerSocket kiszolgaloFoglalat;

        /**
        * A kiszolgalo egyetlen szala.
        */
        private Thread kiszolgaloSzal;

        /**
        * Letrehozza a parameterkent kapott kapun a szolgaltatast.
        * Kivetelt dob, ha a muvelet sikertelen volt. Ezt kovetoen
        * letrehoz egy szalat, es elinditja a szolgaltatast.
        * @param kapu
        * a TCP kapu szama
        * @throws IOException
        * ha a foglalatot nem sikerul letrehozni
        */
        public IdoSzolgaltato(int kapu) throws IOException {
                // foglalat letrehozasa
                kiszolgaloFoglalat = new ServerSocket(kapu);
                // szal letrehozasa, inditasa
                kiszolgaloSzal = new Thread(this);
                kiszolgaloSzal.start();
        }

        /**
        * A kiszolgalo szalat jelento metodus. Ez egy vegtelen ciklus,
        * amelybol a kilepes jelenleg nem megvalositott. Ezert a
        * <code>kiszolgaloFoglalat</code> soha nem kerul lezarasra!
        * A szal varakozik egy ugyfelre, majd sikeres kapcsolodas utan
        * kiirja annak foglalatara a pillanatnyi datumot, es lezarja
        * a kapcsolatot.
        */
        public void run() {
                while (true) { // vegtelen ciklus
                        try {
                                System.out.print("Varakozas ugyfelre...   ");
```

1. Lista (folytatás)

```
                        // a varakozas az ugyfel foglalatara
                        // blokkolja a szalat!
                        Socket ugyfelFoglalat =
                                kiszolgaloFoglalat.accept();
                        // az ugyfel foglalatanak kimeneti adatfolyama
                        PrintWriter iro = new PrintWriter(
                                ugyfelFoglalat.getOutputStream(), true
                        );
                        // pillanatnyi datum kiiratasa
                        iro.println(new Date());
                        // kimeneti adatfolyam lezarasa
                        iro.close();
                        // ugyfel foglalatanak lezarasa
                        ugyfelFoglalat.close();
                        System.out.println("siker");
                } catch (IOException kivetel) {
                        System.out.println("hiba");
                }
        }
    }

    /**
    * Az alkalmazas belepesi pontja letrehoz egy uj peldanyt az
    * osztalybol, <code>6666</code> parameterrel (kapu szam). Elkapja
    * a konstruktor kivetelet es hibauzenetet jelenit meg a stderr
    * csatornan, ha hiba tortent.
    */
    public static void main(String[] args) {
            try {
                    // uj ido szolgaltato a 6666-os kapun
                    new IdoSzolgaltato(6666);
            } catch (IOException kivetel) {
                    // reszletes hibauzenet egyebkent
                    System.err.println(
                            "Nem sikerult letrehozni a foglalatot: '" +
                            kivetel.getLocalizedMessage() + "'."
                    );
            }
    }

}
```

Az osztály látható módon példányosítás után létrehoz egy új szálat, és ebben kezeli a kapcsolódó ügyfeleket. Ez viszont nem azt jelenti, hogy a kiszolgáló több szálon fut! Egy ügyfél kiszolgálása alatt más ügyféllel nem tud foglalkozni. Jelen helyzetben a szálkezelés akár el is maradhatott volna, ám fontos hangsúlyozni, hogy a `ServerSocket` osztály `accept()` metódusa addig nem adja vissza a vezérlést, amíg nem érkezett új ügyfél. Az ilyen tulajdonságú műveleteket pedig érdemes külön szálba helyezni, így később könnyebben bővíthető az alkalmazás.

A belépési pont egy példányt hoz létre a kiszolgálóból a 6666-os kapun. Mivel a konstruktor kivételt dobhat, ezt itt lekezeljük. Hiba esetén a szabványos hibacsatornára írunk ki egy üzenetet. Megjegyzem, hogy egy kivétel `getLocalizedMessage()` metódusától ne várjunk sokat. Általában ugyanazt adja vissza, mint a `getMessage()`. Például az alkalmazást kétszer indítva nekem az alábbi jelent meg:

```
Nem sikerult letrehozni
➥a foglalatot: 'Address
➥already in use'.
```

A konstruktor egy `ServerSocket` objektumot hoz létre. Ezzel egy lépésben létrehoztunk egy foglalatot és hozzákötöttük a megadott kapuhoz. Ez a művelet okozhat

2. Lista A finalize működése

```
public class Alma {

        public Alma() {
                System.out.println("letrejott egy alma");
        }

        protected void finalize() throws Throwable {
                System.out.println("megszunt egy alma");
        }

        public static void main(String[] args) {
                new Alma();
        }

}
```

`IOException` kivételt, aminek a kezelését a hívóra bízzuk. Ezt jelezzük a metódus fejlécében. Továbbá létrehozzuk a fő szálat is, és azonnal el is indítjuk. A szál kódja jelen osztály `run()` metódusában található.
A `run()` egy végtelen ciklust tartalmaz. Ez nem programozási hanyagság, egy folyamatosan figyelő kiszolgálót így szokás megvalósítani. A ciklus minden lefutásban lekezel egy ügyfelet. Előbb fogadja a csatlakozási kérelmet, majd a kapott foglalathoz készít egy `PrintWriter` objektumot a kényelmes íráshoz. Ennek konstruktorában a második paraméterként megadott `true` jelenti az ***azonnali írás (autoflush)*** bekapcsolását.
A foglalatra kiíratjuk a pillanatnyi dátumot, majd lezárjuk az adatfolyamot és a foglalatot.
Ebben a helyzetben nincs szükségünk arra, hogy külön szál foglalkozzon minden egyes ügyféllel, mert egy ügyfél lekezelése nagyon rövid ideig tart. Komolyabb alkalmazások esetén ez azonban elkerülhetetlen. Ilyenkor a kapcsolat fogadása után azonnal el kell készíteni egy új szálat az ügyfélnek, majd annak átadni a kapott foglalatot. Ehhez egy másik osztály készítése szükséges, aminek a feladata az, hogy egyetlen ügyféllel foglalkozzon. Ez esetben pusztán az alkalmazás leállítása is szinkronizációs kérdéseket vet fel, melyek megoldása túlmutat a cikk keretein.

Az alkalmazás leállításánál maradva, be kell ismernem, hogy ez az egyetlen szállal működő program sem tökéletes. Látható, hogy a `kiszolgaloFoglalat` objektum `close()` metódusát sehol sem hívjuk meg. Ez nem jelent nagy problémát, mert a foglalatokat a nyitva maradt állományokhoz hasonlóan a ***JVM*** kilépéskor lezárja. Mégsem kimondottan szép ez a megoldás.

Azért hagytam mégis így, mert nem találtam egyszerű és rövid utat a foglalat bezárásához.
Egy rövid kitérő erejéig bemutatom, hogy milyen módszereken gondolkoztam, mielőtt erőt vett rajtam a lustaság. Pontosabban fogalmazva, mielőtt úgy döntöttem, hogy a foglalatok használatát bemutató példaprogramot nem szeretném annyira elbonyolítani, hogy a megvalósítás apró részleteinek sokasága eltakarja a lényeget.
Mivel a `ServerSocket` objektum a konstruktorban jön létre, adódik, hogy azt egy destruktor metódus zárja le. Java-ban nincs destruktor, mivel a ***szemétgyűjtő (garbage collector)*** eljárás gondoskodik az objektumok megszüntetéséről. Van azonban egy destruktor jellegű metódus, amelyet minden osztály ősosztálya, az Object definiál. Ez a finalize(), melynek működését a *2. Listában* olvasható rövid példa mutatja be.
A programot futtatva az alábbi kimenetet láthatjuk:

```
letrejott egy alma
```

Hogy-hogy nem szűnt meg? A szemétgyűjtő láthatólag nem volt hajlandó meghívni a `finalize()` metódust.

3. Lista Saját ügyfélprogram

```
import java.net.Socket;
import java.io.InputStreamReader;
import java.io.BufferedReader;
import java.io.IOException;

/**
* Ugyfel alkalmazas az
* <code>IdoSzolgaltato</code>-hoz.
*/
public class IdoUgyfel {

        /**
        * A tavoli szamitogep neve
        *(lehet IP cim is).
        */
        private String szgepNev;

        /**
        * A szolgaltatas kapu szama.
        */
        private int kapuSzam;

        /**
        * Beallitja a szamitogep nevet es
        * a szolgaltatas kapu szamat.
        * Nem vegez semmilyen muveletet
        * a halozaton.
        * @param szgep
        * a szamitogep neve (lehet IP cim is)
        * @param kapu
        * a szolgaltatas kapu szama
        */
        public IdoUgyfel(String szgep,
        ➥int kapu) {
                szgepNev = szgep;
                kapuSzam = kapu;
        }

        /**
        * Lekerdezi a pillanatnyi idot
        * a beallitott szolgaltatotol.
        * Kivetelt dob, ha a muvelet kozben hiba
        * tortent.
        * @throws IOException
        * ha a lekerdezes iras/olvasas miatt
        * meghiusult
        */
        public void lekerdez() throws
        ➥IOException {
                // foglalat letrehozasa,
                // csatlakozas
                Socket foglalat = new
                ➥Socket(szgepNev, kapuSzam);
                // pufferelt olvaso keszitese
                BufferedReader olvaso = new
                ➥BufferedReader(
                   new InputStreamReader
                   ➥(foglalat.getInputStream())
                );
                // lekerdezes, kiiratas
                System.out.println
                ➥(olvaso.readLine());
                // olvaso lezarasa
                olvaso.close();
                // foglalat lezarasa
                foglalat.close();
        }

        /**
        * Az alkalmazas belepesi pontja letrehoz
        * egy uj peldanyt az
        * osztalybol, <code>"localhost"</code>
        * es <code>6666</code>
        * parameterekkel. A lekerdezes igy
        * a helyi szamitogeptol, a
        * 6666-os kapun tortenik. Hibauzenetet
        * jelenit meg a stderr csatornan,
        * ha a muvelet sikertelen volt.
        */
        public static void main(String[] args) {
                // ugyfel letrehozasa
                IdoUgyfel ugyfel = new IdoUgyfel
                ➥("localhost", 6666);
                try {
                        // lekerdezes
                        ugyfel.lekerdez();
                } catch (IOException kivetel) {
                        // reszletes hibauzenet,
                        // ha nem sikerult
                        System.err.println(
                            "A lekerdezes
                            ➥nem sikerult: '" +
                              kivetel.
                              ➥getLocalized
                              ➥Message() + "'."
                        );
                }
        }
}
```

Egészítsük most ki az `Alma` osztály `main(String[])` metódusát az alábbi sorral:

```
System.runFinalizersOnExit(true);
```

Fordításkor az alábbi figyelmeztető üzenetet kapjuk:

```
Note: Alma.java uses or
➥overrides a deprecated API.
Note: Recompile with -Xlint:
➥deprecation for details.
```

A második futtatás eredménye már a várt eredményt adja:

```
letrejott egy alma
megszunt egy alma
```

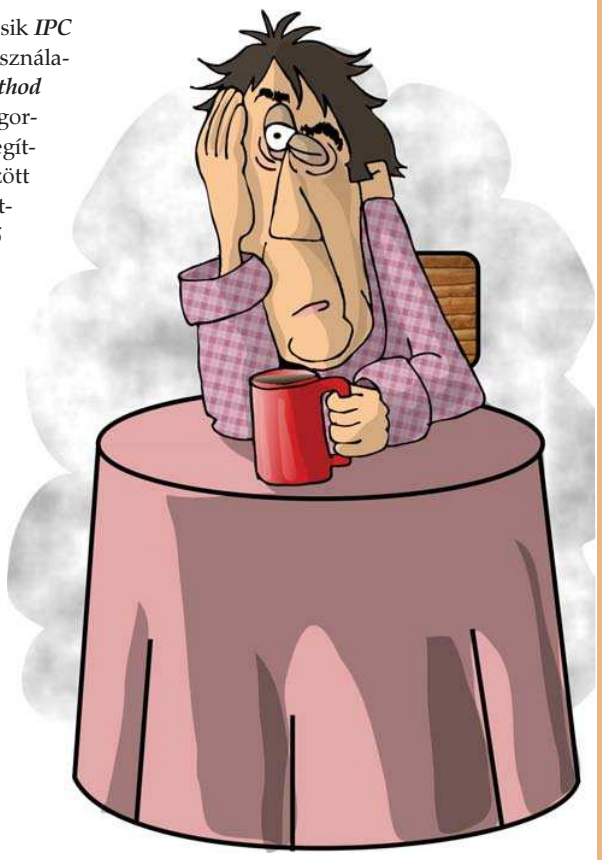
Ettől azonban nem lehetünk teljesen elégedettek. A figyelmeztető üzenetet azért kaptuk, mert a `runFinalizersOnExit(boolean)` metódus az új Java változatokban érvényét vesztette (deprecated). Használható ugyan, de nincs garancia arra, hogy a jövőbeni változatok is tartalmazzák, mivel nem minősül biztonságosnak. Egy többszálas alkalmazásban előfordulhat ugyanis, hogy a `finalize()` metódus még élő objektumok esetén is meghívódik, melyeket más szálak még használnak, ami akár holtpontot is okozhat.
A destruktor megközelítés tehát nem hozott eredményt. Nyilvánvaló, hogy valamilyen egyéb eseményt kell találni, aminek fellépésekor a foglalat bezárható. Egy vad megközelítésben rábízhatnánk az ügyfelekre a bezárást. Ha az ügyfél foglalatáról olvasnánk is, figyelhetnénk, hogy kapunk-e olyan üzenetet, ami a szolgáltatás bezárását kérelmezi. Ez nem csak nem biztonságos, de felveti azt a fontos kérdést is, hogy miért van szükség hálózatra egy szolgáltatás leállításához.
Linux alatt jellemzően ***jelzésekkel (signal)*** adhatjuk tudtára egy démonnak, ha be szeretnénk zárni. A jelzések mellesleg szintén az *IPC* csoportjába tartoznak. Ezt a módszert viszont tisztán a ***Java API***-ra támaszkodva nem használhatjuk. Valószínűleg a Java fejlesztői úgy ítélték meg, hogy a jelzések túlságosan platformfüggők ahhoz, hogy helyet kapjanak a szabványos osztálykönyvtárban. Szerény véleményem szerint létezik olyan értelmes keresztmetszete a különböző rendszereken használt jelzéseknek, amelyre alapozva nincs elvi akadálya a jelzéskezelés megvalósításának. Jelenleg csak különböző kerülő utakon, jellemzően nem teljesen platformfüggetlen módon lehetne ezt megoldani.
További megoldás a billentyűzetről, vagy valamilyen *GUI* felületről kezdeményezett bezárás. Egy ilyen alapokon nyugvó alkalmazás azonban soha nem kerülhet igazán háttérbe, ezért az ilyen megoldásokat szokás a programozói szakzsargonban gagyinak hívni. A félreértések elkerülése végett megjegyzem, hogy a probléma nem megoldhatatlan. Egy másik *IPC* eszközzel, a *Java*-ban használatos *RMI*-vel (***Remote Method Invocation***) átvágható a gordiuszi csomó. Az *RMI* segítségével két folyamat között távoli eljáráshívás valósítható meg. Így készíthető olyan alkalmazás, ami bezárja a főprogramot. Ez érzésem szerint egy kicsit hasonlít arra, amikor ágyúval lövöldözünk verebekre. Fogadjuk tehát el, hogy a ***Ctrl+C*** billentyűkombinációval lépünk ki. Ez egyébként egy *SIGINT* jelzés a *JVM*-nek, csak az nem hajlandó erről értesíteni a futtatott alkalmazást. Búsulásra viszont semmi ok, hiszen a virtuális gép kilépéskor lezár minden nyitott állományt, illetve foglalatot.
Térjünk vissza az időkiszolgálónkhoz. Fordítás és futtatás után a következőt láthatjuk:

```
Varakozas ugyfelre...
```

Elsőre elég, ha saját gépünkről próbáljuk ki a hálózati alkalmazást. Ha tűzfalunk nem akadályoz meg ebben, egy egyszerű ***telnet*** ügyféllel csatlakozhatunk a 6666-os kapura. Ekkor az alábbit olvashatjuk:

```
Trying 127.0.0.1...
Connected to
➥localhost.localdomain.
Escape character is '^]'.
Tue Nov 01 22:07:58 CET 2005
Connection closed by foreign
➥host.
```

A 4. sor bizonyítja, hogy a kiszolgáló kifogástalanul működik. Mi több, a kiszolgáló konzolján megjelenik a `siker` felirat, majd ismét várakozó állásba helyezkedik. Viszont ne érjük be ennyivel, írjunk saját ügyfélprogramot! Ennek kódja látható a *3. Listában*.
Az osztály kódja igazából önmagáért beszél. A konstruktor beírja a paraméterként kapott számítógép nevet és kapu számot a tagváltozókba.



A `lekerdez()` metódus kapcsolódik a távoli folyamathoz, létrehoz egy pufferelt olvasó folyamot, kiírat egy olvasott sort, majd bezárja a folyamot és a foglalatot.
A `main(String[])` metódus egy példányt hoz létre az osztályból, majd egy lekérdezést hajt végre. Ha ez meghiúsult, a keletkezett kivételt elkapja és hibaüzenetet jelenít meg a szabványos hibacsatornán.
A *Java* sorozat ezennel véget ért. Remélem, hasznos olvasmányt jelentett mind a kezdők, mind a haladóbbak számára ez az utolsó rész is. Ha a kedves Olvasónak bármilyen *Java*-val kapcsolatos kérdése van, írjon bátran!

Fülöp Balázs
(bigwig42@gmail.com)
21 éves, imádja a Túró Rudit, a Debian Linuxot és a teheneket. Kedvenc írója Slawomir Mrožek. Leginkább a számítógépes hálózatok biztonsága érdekli. A BME VIK műszaki informatikus szak hallgatója.