Kávéfőzés lépésről lépésre (5. rész)

A programozó pedig ablakokat és nyomógombokat hozott létre. Így a felhasználó kedve szerint kattinthatott, az ő legnagyobb megelégedésére...

Ha a kedves Olvasó idáig figyelemmel kísérte a cikksorozatot, bátran állíthatja, hogy azokkal az objektumközpontúsággal kapcsolatos fogalmakkal, melyek sokaknak mindössze valamilyen titkos szekta misztikus varázsigéit jelentik, bizony már szoros baráti viszonyban áll. Az osztály, az objektum, az interfész, az öröklődés és a kivételkezelés hallatán szívéhez közel álló eszközök és módszerek jutnak eszébe, melyek segítségével egy alkalmazás tervezése, megvalósítása és karbantartása is egyszerűbbé és gyorsabbá válik.

Joggal jelenthetjük ki tehát, hogy mindent megtettünk saját magunk, programozók örömére. Azonban ne felejtsük el, hogy nincs programozó felhasználó nélkül, hiszen az sem jó színész, akit a rendezőn kívül más nem látott szerepelni. Ezért most, hogy már elhangzottak a kenetteljes szentbeszédek a programozókat kiszolgáló objektumközpontú technikák mellett, eljött az ideje, hogy a felhasználóknak is kedvezzünk. A felhasználó csak annak tud örülni, amit lát, adjunk tehát most a külcsínre.

Mivel mindig hangsúlyozom, hogy az informatikában is a célnak kell meghatároznia az eszközt, és nem fordítva, először gondoljuk meg, milyen programozási környezet lenne a legalkalmasabb grafikus felhasználói felület (*GUI, Graphical User Interface*) létrehozásához. A cikksorozat eddigi részei a hűséges Olvasóban talán már gyanút ébreszthetnek, ám mielőtt kimondanánk a boldogító igent a Java oldalán, mérlegeljük a mellette és az ellene szóló érveket.

A *Java* objektumközpontú, ami a jelenleg ismert leghatékonyabb módszer ablakozó rendszer programozására. Emellett a futtatókörnyezettől bájtkód szinten független, ezért könnyen hordozható. Ugyanakkor lehetőséget ad akár az operációs rendszer által nyújtott eszközkészlet, akár egy saját, minden rendszeren azonosan megjelenő eszközkészlet használatára. A képernyőn megjelenő vezérlőkhöz köthető eseménykezelők pedig teljes és jól átlátható rendszert adnak.

Ezzel szemben tudni kell, hogy egy *Java* alkalmazás virtuális gépet igényel (*JVM, Java Virtual Machine*). Ez minden szélesebb körben használt operációs rendszerre elérhető, de nem szükségszerű, hogy fel is telepítették. Erőforrásigénye sem nevezhető kimondottan alacsonynak. Így eleshetünk egyes, matuzsálemi korú berendezésekkel dolgozó felhasználóktól. Azt se felejtsük el, hogy bár létezik megoldás 3 dimenziós ábrák és animációk használatára, kevés ilyen elemeket tartalmazó Java alkalmazást láthatunk.

Ezek után a készítendő program dönti el, hogy a mérleg nyelve merre dől. Én személy szerint azért teszem erre a nyelvre a voksomat, mert így könnyen elérhetem, hogy programjaimat, melyeket a Microsoft által készített operációs rendszereken kell bemutatnom, szeretett ***Linuxom*** alatt írhatom meg. Mindenki maga döntsön, majd a helyes döntést követően olvassa tovább a cikket.

Ismerjük el, egy kissé kezd unalmas lenni, hogy mindenki állandóan a világot köszöntgeti. A sorozat első részében már mi is megtettük, első *Java* nyelven írt grafikus alkalmazásunk legyen ennél valamivel érdekesebb. Lássuk a kódot *(1. kód)*!

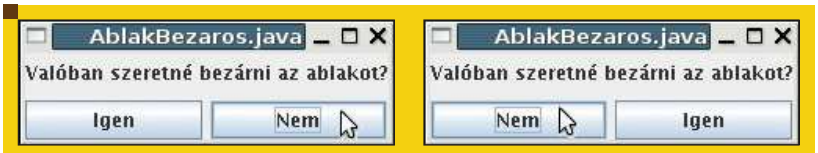
Minden szerénységem megtartása mellett meg kell jegyezzem, hogy tisztában vagyok azzal, mennyire lebilincselőek az írásaim. Ennek ellenére most arra kérném a kedves Olvasót, hogy mielőtt továbbolvasná a cikket, gépelje be az itt látható program kódját, fordítsa és futtassa azt.

Két nyomós indokom van arra, hogy arra kérjem, hogy olyan sorokat gépeljen, melyeknek jelentése nem azonnal adódik, s így ez esetleg fárasztó terhet jelenthet.

Az első, hogy az alkalmazás egyfajta feladvány, ugyan nem a komoly fejtörést okozók fajtájából, de mindenképpen tanulságos. A kód azonnali elem-

1. kód

```
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JButton;
import javax.swing.JPanel;
import java.awt.GridLayout;
import java.awt.Container;
import java.awt.event.MouseAdapter;
import java.awt.event.MouseEvent;
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;

/**
* Az AblakBezaros jatek megvalositasa.
*/
public class AblakBezaros extends MouseAdapter
➥implements ActionListener {

    /**
    * A teljes ablak
    */
    private JFrame ablak;

    /**
    * A kerdest tartalmazo cimke
    */
    private JLabel szovegCimke;

    /**
    * Bal- es jobboldali nyomogombok
    */
    private JButton balGomb;
    private JButton jobbGomb;

    /**
    * Igen es Nem allandok
    */
    private final String igenSzoveg = "Igen";
    private final String nemSzoveg = "Nem";

    /**
    * A konstruktor letrehozza az ablakot,
    * felteszi ra a vezerloket, es beallitja
    * az esemenykezeloket.
    */
    public AblakBezaros() {
  // ablak es vezerlok letrehozasa
  ablak = new JFrame("AblakBezaros.java");
  szovegCimke = new JLabel("Valóban szeretné
  bezárni az ablakot?",
      JLabel.CENTER);
  balGomb = new JButton(igenSzoveg);
  jobbGomb = new JButton(nemSzoveg);

  // gombok onallo panelt kapnak
  // az elrendezeskezelo miatt
  JPanel gombokPanelje = new JPanel(new
  GridLayout(1, 2, 5, 5));
  gombokPanelje.add(balGomb);
  gombokPanelje.add(jobbGomb);

  // az ablak tartalmanak feltoltese
  Container tartalom = ablak.getContentPane();
  tartalom.setLayout(new GridLayout(2, 1, 5, 5));
  tartalom.add(szovegCimke);
  tartalom.add(gombokPanelje);

  // esemenykezelok mind ebben az objektumban
  balGomb.addMouseListener(this);
  jobbGomb.addMouseListener(this);
  balGomb.addActionListener(this);
  jobbGomb.addActionListener(this);

  // vegso "simitasok"
  ablak.setDefaultCloseOperation
  (JFrame.DO_NOTHING_ON_CLOSE);
  ablak.pack();
  jobbGomb.requestFocusInWindow();
  ablak.setVisible(true);
    }

    /**
    * Akkor fut le, ha valamelyik gomb fole kerul
    * az egerkurzor. Ekkor felcsereli a ket gomb
    * szoveget es a fokuszt is athelyezi.
    */
    public void mouseEntered(MouseEvent esemeny) {
  Object forras = esemeny.getSource();
  if (((JButton) forras).getText().equals
  ➥(igenSzoveg)) {
      if (((JButton) forras).equals(balGomb)) {
     balGomb.setText(nemSzoveg);
     jobbGomb.setText(igenSzoveg);
     balGomb.requestFocusInWindow();
      } else {
     balGomb.setText(igenSzoveg);
     jobbGomb.setText(nemSzoveg);
     jobbGomb.requestFocusInWindow();
      }
  }
    }

    /**
    * Akkor fut le, ha valamelyik gomb benyomodik.
    * Ha a benyomott gomb felirata "Igen", az
    * alkalmazas bezarul.
    */
    public void actionPerformed
    ➥(ActionEvent esemeny) {
  Object forras = esemeny.getSource();
  if (((JButton)
forras).getText().equals(igenSzoveg)) {
      System.exit(0);
  }
    }

    /**
    * Belepesi pont, egy peldanyt hoz letre az
    * osztalybol.
    */
    public static void main(String[] args) {
  new AblakBezaros();
    }

}
```



1. ábra Ön mit tenne az adott szituációban? (2 pont)

zésével a megoldás magáért beszélne. A másik, hogy nehéz úgy egy grafikus alkalmazást megérteni, hogy az eredményét pusztán a képzelőerőnkre bízzuk. Különösen igaz ez akkor, ha még nem sok tapasztalattal rendelkezünk a grafikus felhasználói felületek készítése területén. Ezért mindenképpen javaslom a játék azonnali kipróbálását. A felejthetetlen játékélményt követően nézzük meg, hogyan épül fel az alkalmazás. Az előző részben is használtuk az `import` kulcsszót, most viszont már-már ijesztő méretekben fordul elő. Az `import` a megadott osztályt hozza be az alkalmazás névterébe. Ennek révén rövidebben hivatkozhatunk ugyanarra az osztályra, vagyis a behozatal után nem kell minden egyes helyen kiírni, hogy `javax.swing.JButton`, elegendő a `JButton` használata. Ennek természetesen megvan az a hátulütője, hogy nem hozhatunk létre saját `JButton` nevű osztályt az alkalmazásban, de erre legtöbbször nincs is szükség.

Miután csak három csomagból használunk osztályokat, nevezetesen a `javax.swing`, a `java.awt` és a `java.awt.event` csomagokból, az első tíz sort lerövidíthettük volna így is:

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
```

Így a csomagok összes osztálya a névtérbe került volna. Ez viszont azért nem javasolt, mert ha a későbbiekben akár nekünk, akár másnak elemeznie kell a kódot, nem egyértelmű, hogy melyik csomag miért kerül behozatalra, más szóval milyen osztályokból építkezik a program. Nagy valószínűséggel hibát nem követhetünk el, mert az adott helyen létrehozott osztályok felülbírálják a behozott neveket, mégsem jó programozói gyakorlat olyannal szennyezni a névteret, ami nem oda való.

Behoztunk tehát tíz olyan osztályt, amivel a sorozatban eddig nem találkoztunk. Az aggodalom azonban csak addig tarthat míg fény nem derül a ***Swing*** és az ***AWT*** mibenlétére. A kezdetekben grafikus felhasználói felületek létrehozásához ***Javaban*** az ***AWT (Abstract Windowing Toolkit)*** állt csak rendelkezésre. Ez egy olyan osztályokból és interfészekből álló halmazt biztosított a programozónak, mellyel ablakokat hozhatott létre, rajta vezérlőket, és kezelhette a különféle eseményeket.

Az egyik legnagyobb hátránya ennek a megoldásnak pont az absztraktságában rejlett. Miután az ***AWT*** használatával a programozó teljes mértékben a futtatókörnyezetre bízta a megjelenítést, csak olyan elemek voltak elérhetőek számára, amelyek minden rendszeren megtalálhatók. Így az ***AWT*** a különféle ablakozó rendszerek beépített grafikus eszközkészletének legszűkebb keresztmetszetét kínálta csupán. Ráadásként az sem volt minden esetben elfogadható, hogy nehezen megjósolható a program kinézete is, hiszen mindent a futtatókörnyezet dönt el.

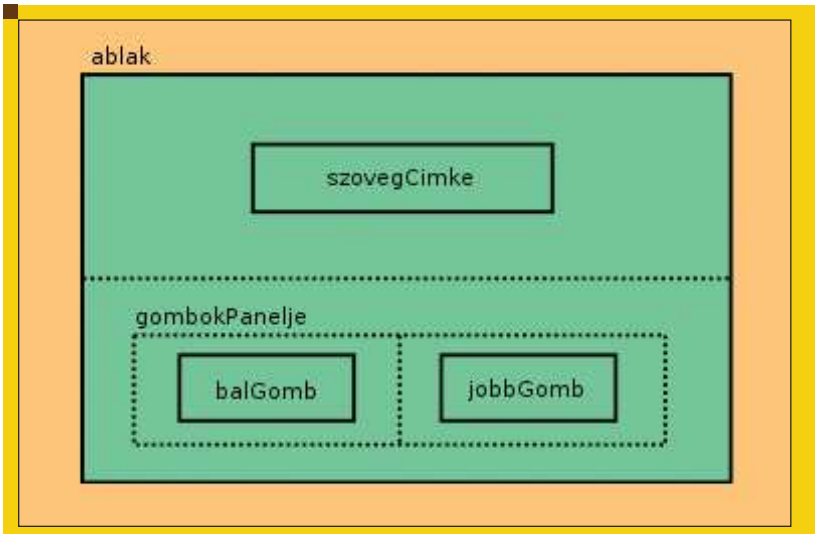
A ***Java 1.3*** megjelenésével megérkezett a ***Swing***, amely már egy önálló eszközkészlet a nyelvben. Teljesen *Java*-ban íródott, ami jópár előnnyel jár. A grafikus alkalmazás így nem veszít hordozhatóságán, ugyanakkor több és nagyobb tudású vezérlőket használhat, mivel az ablak nem függ a futtatókörnyezettől. A kinézet egységes minden rendszeren, sőt, témákat is alkalmazhatunk a még egyedibb megjelenítés érdekében.

A ***Swing*** hátterében az ***AWT*** áll, így nem lehet azt állítani egy alkalmazásról, hogy „tisztán swinges". A vezérlőknél még igen szembetűnően minden osztály neve elé oda kell ragasztani egy `J` betűt (`Button` helyett `JButton`), az eseménykezelő osztályok és interfészek viszont a jó öreg ***AWT*** csomagból valók. Ez nem baj, mert eddig is kifogástalanul működtek, és nem lett volna értelme újraírni őket. A két csomag remekül kiegészíti egymást, és jól használhatók, ha nem hanyagoljuk el az ***API*** tanulmányozását.

Ugorjuk át most az osztály fejlécét és nézzük meg, mi van a törzsében. Először a tagváltozók felsorolását látjuk. Egy ablakban egy szövegcímkét és két nyomógombot jelenítünk meg, ez négy önálló objektumot jelent. Itt találhatunk továbbá két állandót, melyek a gombok feliratait adó szövegfüzérek. A `final` kulcsszóval biztosítjuk, hogy a változók értéke a későbbiekben ne legyen módosítható. Az ilyen mezők életükben egyszer kaphatnak értéket. Ez a művelet kerülhetett volna a konstruktorba is, ekkor azonban feltétlenül ügyelni kellett volna arra, hogy még az első állandóra való hivatkozás előtt ellássuk azt a megfelelő kezdőértékkel.

Az ezután következő konstruktor hozza létre az ablakot. Első sorában a `JFrame` osztályt példányosítjuk, és a kapott referenciát az ablak tagváltozóban tároljuk. A `JFrame` egy ***Swing*** összetevő, amely egy ablakot képvisel. Több konstruktora is van, mi most azt a változatot használtuk, mellyel rögtön meg is lehet adni az ablak címét. Fontos tudni, hogy ettől a sortól az ablak még nem jelenik meg. Alapértelmezésként a láthatósága hamis, ami azért jó, mert így a háttérben felépíthetjük az ablakot, amit csak akkor jelenítünk meg, ha már teljesen készen van.

A következő sorban hozzuk létre a szövegcímkét. Konstruktorában megadjuk a szöveget, továbbá, hogy a rendelkezésre álló helyen középre igazítsa azt. Ez a címke még nem az ablak része. Semmilyen módon nem fejeztük ki, hogy a vezérlő, amit létrehoztunk, hol jelenjen meg. Ugyanez a helyzet az ezután álló sorokban létrehozott `Igen` és `Nem` feliratú gombokkal. Az első sorokban tehát még csak különálló objektumokat hoztunk létre. Ahhoz, hogy ezek az ablakban megjelenjenek, ki kell választanunk egy elrendezéskezelőt. Az elrendezéskezelő az a láthatatlan kéz, amelyik kirakosgatja az ablakhoz hozzáadott elemeket az ablak területére. Mi a `GridLayout` nevű elrendezéskezelőt használjuk, amely egy táblázatként értelmezi az ablakunkat. A `GridLayout` példányosításakor megadhatjuk, hogy



2. ábra Az ablak felépítése

hányszor hányas táblázatot szeretnénk használni. Ha például kétszer ketteset választunk, akkor az első hozzáadott elemet az ablak bal felső sarkába helyezi, a másodikat a jobb felsőbe, a harmadikat a bal alsóba, a negyediket pedig a jobb alsóba.

Most egy olyan képet szeretnénk elérni, ahol a címke az ablak felső részében van, a gombok pedig alul, egymás mellett. Ennél az elrendezéskezelőnél maradva ezt úgy tudnánk elképzelni, hogy a kétszer kettes tábla felső sorában álló cellákat összevonjuk. Ez azonban itt nem megoldható. Ehelyett úgy tudjuk ugyanazt a hatást kelteni, hogy az ablakot két részre osztjuk csupán. A felső rész tartalmazza majd a szövegcímkét, az alsó pedig egy új panelt. Az új panel egy bal és egy jobboldali cellából fog állni, ezekbe kerülnek a gombok.

A vezérlők objektumainak létrehozása után ezt a panelt készítjük el.

A `JPanel` kosntruktorának rögtön át is adunk egy új `GridLayout` objektumot, amely 1 sorból, 2 oszlopból álló táblázatban rendezi el az elemeket, és mind vízszintes, mind függőleges irányban 5 képpontot hagy ki a cellák között. Miután a panel rendelkezik egy elrendezéskezelővel, az elemek hozzáadása egy egyszerű add metódussal történik. Az ablakon belüli tényleges, x és y koordinátákkal történő pozicionálás már a `GridLayout` dolga.

Megjegyzem, hogy a `JPanel`-nek van olyan konstruktora, amely nem vár paramétert, tehát nem kell megadnunk az elrendezéskezelőt. Ilyenkor az alapértelmezett `FlowLayout` végzi a vezérlők helyének meghatározását. Ez balról jobbra folyamatosan tölti fel az ablakot a vezérlőkkel, és ha a következő már nem fér ki az adott sorba, akkor a következőben folytatja. Ez a viselkedés nagyon közel áll egy szövegszerkesztőben megtekintett bekezdésre. Az ablak mérete alapvetően befolyásolja itt a vezérlő helyét, mi több, az ablak átméretezésével az egész megjelenés megváltozhat, ezért csak kevés értelmes felhasználása van ennek az elrendezéskezelőnek.

Megváltoztatható az elrendezéskezelő egy `JPanel` esetében a `setLayout` metódussal. Viszont mivel a `JPanel` paraméter nélküli konstruktora létrehoz egy példányt a `FlowLayout` osztályból, a felülbírálással egy felesleges objektum létrehozását okozzuk. Ezért, amikor csak lehet, érdemes megadni még a konstruktorban a saját elrendezéskezelőt, hogy elkerüljük az értelem nélküli példányosítást.

Miután elkészültünk a panellel, felrakhatjuk az elemeket az ablakra. Ehhez közvetlenül nem adhatjuk hozzá a vezérlőket, viszont az ablaktartalom egyszerűen lekérdezhető a `getContentPane` metódussal. Ehhez először hozzárendelünk egy `GridLayout` elrendezéskezelőt, 2 sorral, 1 oszloppal, és 5 képpontos hézagokkal. `JFrame` esetében nem is lett volna lehetőségünk a konstruktorban ezt meghatározni, így ezt itt tesszük meg. Ezt követően az add metódussal felhelyezzük a címkét és a gombok paneljét az ablakra.

Mielőtt megjelenítenénk az ablakot, beállítjuk az eseménykezelőket. Elsőként az egérrel kapcsolatos eseményekkel foglalkozunk. Bármely osztály, amely megvalósítja a `MouseListener` interfészt, hozzárendelhető egy vezérlőhöz. Így, ha a vezérlő területére belép az egérkurzor, vagy kilép onnan, kattintás történik, esetleg a felhasználó lenyomja az egér valamelyik gombját, vagy felengedi azt, az interfészt megvalósító osztály megfelelő metódusához kerül a vezérlés.

Viszont ne felejtsük el, hogy egy interfész megvalósítása az összes előírt metódus megvalósítását jelenti. Jelen esetben nekünk erre nincs szükségünk, hiszen csak az egér mozgását szeretnénk követni. Ha maradnánk a `MouseListener` interfésznél, a többi előírt metódust üres törzzsel ugyan, de szerepeltetni kellene a programban. Ennek kiküszöbölésére használható a `MouseAdapter`, amely egy olyan osztály, ami megvalósítja a `MouseListener` interfészt, üres függvénytörzsekkel. Ha ebből származtatjuk saját osztályunkat, csak azokat a metódusokat kell felüldefiniálnunk, melyeket ténylegesen használni szeretnénk.

Ezzel úgyszólván átvágtuk a gordiuszi csomót. Egyetlen osztályunk, az `AblakBezaros` kiterjeszti a `MouseAdapter`-t, ezt jelezzük is az osztály fejlécében. Elkészítjük a `mouseEntered` metódust, melynek meghívását az az esemény válthatja ki, ha az egérkurzor belép a vezérlő területére. Az eseménykezelővel később foglalkozunk, itt, a konstruktorban csak annyit állítunk, hogy a bal- és jobboldali gomb egérrel kapcsolatos eseményeivel jelen objektum (`this`) törődik. Ezt fejezik ki az `addMouseListener` hívások.

Gombok esetében ennél többet is figyelhetünk. Egy nyomógombbal kapcsolatos legfontosabb esemény az, ha lenyomják. Ez történhet egérrel is, de a billentyűzet segítségével is odalépkedhet a felhasználó, és rátenyerelhet a szóköz billentyűre. Ezt az eseményt is szeretnénk

kezelni, amit egy olyan osztály példányával tehetünk meg, amely megvalósítja az `ActionListener` interfészt. Az előzőekkel szemben nincs `ActionAdapter` osztály, de ez két ok miatt sem gond. Egyrészt az `ActionListener` csak egyetlen metódus megvalósítását írja elő, pont azt, amire szükségünk van. Másrészt többszörös öröklődés híján újabb osztályt már ki sem terjeszthetnénk.

A szemfüles Olvasó felfigyelhet arra a tényre, hogy a gombra történő kattintás egyszerre két eseménykezelőt is érint. Egyrészt az `ActionListener` interfészt megvalósító osztály példányának `actionPerformed` metódusa, másrészt a `MouseAdapter`-t kiterjesztő, vagy a `MouseListener`-t megvalósító osztály példányának `mouseClicked` tagfüggvénye. Ezt jelen esetben nem valósítottuk meg, de használata nem ütközött volna semmilyen problémába. Ilyen helyzetben mindkét függvény meghívásra kerül.

A „végső simítások” megjegyzéssel illetett részben elsőként megadjuk, hogy mi történjen, ha a felhasználó az ablakkezelőn keresztül próbálja

meg bezárni az alkalmazást. Ez legtöbbször a jobb felső sarokban található `X`-re való kattintást jelenti.

A `setDefaultCloseOperation` metódusnak a `JFrame` osztály megfelelő konstansai közül kell átadnunk egyet. Az alapértelmezett művelet az elrejtés (`HIDE_ON_CLOSE`). Ezt bíráljuk felül azzal, hogy ne történjen semmi (`DO_NOTHING_ON_CLOSE`). A leggyakrabban használt művelet minden bizonnyal a kilépés (`EXIT_ON_CLOSE`), ám itt nem erre van szükségünk.

A `pack` metódus meghívásával életre keltjük a láthatatlan kezet, amely szépen elrendezi az ablaktartalmat. Majd mivel fontos, hogy a fókusz ne a bal oldali gombon legyen, a jobboldalinak meghívjuk a `requestFocusInWindow` metódusát. Ez még nem biztosítja, hogy a vezérlő valóban meg is kapta a fókuszt, és egy komoly alkalmazásnál érdemes is megvizsgálni a visszatérési értéket e tekintetben. Viszont a legtöbb helyzetben elegendő, ha csak meghívjuk a függvényt. Végül láthatóvá tesszük az ablakot.

Az alkalmazás elindításával az imént tárgyalt konstruktor fut le. A további két tagfüggvény csak meghatározott események hatására lépnek életbe. Ezen végletekig felhasználóbarát program ablakában a felhasználó a kilépésért küzd. Ha az „Igen” feliratú gomb fölé viszi a kurzort, a gombok látszólag helyet cserélnek. Valójában csak a feliratuk változnak meg, és a vezérlők a helyükön maradnak. Ezt a cselt valósítja meg a `mouseEntered` metódus. Minden, a `MouseListener` által előírt metódus paraméterként egy

`MouseEvent` típusú objektumot kap. Az egérkurzor belépését figyelő függvény törzsének első sorában ennek az eseményobjektumnak a forrását kérdezzük le a `getSource` metódus segítségével. Tudnunk kell ugyanis, hogy melyik gomb fölé került a kurzor, hiszen mindkettőt figyeljük. Ám a `getSource` `Object` típusú referenciát ad vissza, nekünk pedig `JButton`-ra van szükségünk. Ezért később, ahol a `forras` változóra hivatkozunk, egy egyszerű típuskényszerítést hajtunk végre. Ugyanezt megtehettük volna már a lekérdezésnél is, így:

```
JButton forras = (JButton)
➥esemeny.getSource();
```

Ekkor a további típuskényszerítések feleslegesek lennének. Viszont hangsúlyozni szerettem volna a visszatérési érték típusát. A `(JButton) forras` tehát arra a gombra vonatkozik, amelyik kiváltotta az eseményt. Előbb meg kell vizsgálnunk, hogy a gomb felirata *Igen*-e, ellenkező esetben ugyanis nincs semmi dolgunk. Ehhez előbb lekérdezzük a feliratot a nyomógomb `getText` metódusával, majd ennek a `String` objektumnak hívjuk meg az `equals` tagfüggvényét az `igenSzoveg` paraméterrel. Tekintsük az alábbi példát:

```
if ( ((JButton) forras).getText()
➥== igenSzoveg ) { ... }
```

Helyes-e ez ebben a formában? Másképp fogalmazva, ugyanazt fejezi-e ki ez a sor, mint a forráskódban látható? Bár agyunknak az a része, ami a világot egyszerűbbnek szeretné látni, azt mondatja, hogy persze, sajnos nem így van. Az `==` operátor egyenlőséget vizsgál ugyan, de nézzük meg, mi áll a bal- és a jobboldalon. Mindkettő `String` referencia, azaz `String` típusú objektum hivatkozása. Ha két hivatkozást hasonlítunk össze, nagy valószínűséggel annak ellenére kapunk hamis eredményt, hogy az objektumok tartalma azonos. Miután meggyőződtünk arról, hogy az eseményt kiváltó gomb felirata *„Igen”*, hasonló eljárással megnézzük, hogy a baloldali gombról van-e szó. Ha igen, akkor a baloldali feliratát megváltoztatjuk *„Nem”*-re, a jobboldaliét *„Igen”*-re, és a baloldali kapja meg a fókuszt. Hasonló gondolatmenettel a jobboldali nyomógombra vonatkozó ág is értelmezhető.

Végezetül az `actionPerformed` metódus adja a menekülési útvonalat a felhasználónak. Ha sikerül lenyomnia a *„Nem”* feliratú gombot, az alkalmazás bezárul. Ez a metódus a korábbiak alapján magáért beszél. Az egyetlen újdonság a `System.exit(0)` használata. Az `exit` a `System` osztály egy statikus metódusa, melynek paramétere az a visszatérési érték, melyet az alkalmazás a kilépéskor szolgáltat. A függvény meghívásával a program bezárul.

Ebben a rövid programban szegény felhasználó kergeti az egérrel azt a gombot, amelytől azt várja, hogy be tudja zárni az ablakot. Hosszabb-rövidebb idő után rá kell jönnie arra, hogy ebben a csilli-villi ablakozós világban sem lehet elfelejteni a billentyűzet jelentőségét.

Bár ebben a hónapban még mindig nem teljes mértékben a felhasználó kényeztetése lebegett a szemünk előtt, következő alkalommal már hasznos és látványos animációkat fogunk készíteni, bepillantva a szálkezelés mesterségébe.

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.