Kávéfőzés lépésről lépésre (4. rész)

Kivételkezelés és javadoc – avagy két remek érv Java- függőségünk magyarázatára

Miután az előző hónapban egy kalap alatt ismerhetett meg az Olvasó két olyan távol álló fogalmat, mint a tömbök és az interfészek, jelen írásban már nem érheti meglepetés. Ebben a cikkben egy nyelvi eszközt, a kivételkezelést, és egy, a programozásfejlesztéskor oly fontos dokumentálást segítő programot mutatok be.

A két eszköz talán egyetlen közös vonása az, hogy noha használatuk kisebb alkalmazások esetén mellőzhető, nagyobb projektek esetében nagy könnyebbséget jelentenek, és így fontosságukról ódákat lehetne zengeni.
A kivételkezelés nem kötődik szorosan az objektumközpontú filozófiához, ugyanakkor a legtöbb (ha nem az összes) objektumközpontú nyelvben megtalálható. Nincs ez másképp a Java esetében sem. Ennek a módszernek a célja a futásidőben keletkező hibák egyszerűen átlátható és kényelmes kezelése. Ezen hibák nagy részét interaktív alkalmazásokban szokás szerint a felhasználó váltja ki. Tipikus példája ennek az osztóprogram.

Divide et impera

Ebben az alkalmazásban az osztás műveletét valósítjuk meg. Egyelőre ne foglalkozzunk azzal, hogy a felhasználótól várjuk el a bemenő adatot, mindössze írjunk egy `osztas` nevű metódust.

```
/**
* Ez az osztaly az osztas muveletet valositja meg.
*/
public class Osztoprogram {

   /**
   * Az osztast vegzo metodus megjeleniti
   * a kepernyon az osztas eredmenyet.
   */
   private void osztas(double szamlalo, double
   ➥nevezo) {
      System.out.print(szamlalo + " / "
      ➥+ nevezo + " = ");
      System.out.println(szamlalo /
      ➥nevezo);
   }
   /**
   * A konstruktor harom osztast
   * vegez el.
   */
   public Osztoprogram() {
      osztas(1.0, 2.0);
      osztas(1.0, 3.0);
      osztas(1.0, 0.0);
   }

   /**
   * Az alkalmazas inditasakor letrejon egy
   * peldany az Osztoprogram osztalybol
   */
   public static void main(String[] args) {
      new Osztoprogram();
   }

}
```

Az alkalmazás belépési pontjánál egyetlen példányosítás történik. Az objektumközpontú programozásban ez megszokott gyakorlat. Ezzel a megoldással ugyanis rövid úton áthidalható az a probléma, hogy egy teljesen objektumközpontú környezetből hiányoznak a strukturált programozásban megszokott globális függvények. Ha nem hoznánk létre egyetlen objektumot sem, csak a `static` kulcsszóval minősített metódusokat használhatnánk. Természetesen megtehettük volna statikusnak az `osztas` nevű tagfüggvényt, kihagyhattuk volna a konstruktort, és törzsét átemelhettük volna a `main` függvénybe. Ekkor viszont teljes joggal feltehetnénk magunknak azt a kérdést is, hogy miért erőltetjük ezt a Java-t, ha *C*-ben szeretünk programozni.
A példányosításkor lefut az osztály konstruktora, ezért háromszor meghívásra kerül az `osztas` metódus. Ez két `double` típusú, azaz dupla lebegőpontosságú számot vár paraméterként, és ezek hányadosát íratja ki, miután a műveletet is jelezte egy sortörést mellőző `System.out.print` segítségével. Fordítás és futtatás után várakozásainkat felülmúló eredményt láthatunk a képernyőn:

```
1.0 / 2.0 = 0.5
1.0 / 3.0 = 0.3333333333333333
1.0 / 0.0 = Infinity
```

Nem meglepő a csonkított ***1/3***, ám az sokkal inkább, hogy ***1/0*** végtelen. Ezt az első ránézésre furcsa eredményt az alábbi egyenlet indokolja:

$$\lim_{n \to +0} \frac{1}{n} = +\infty$$

Azon kedves Olvasók, akik eddig sikeresen megúszták az analízis leírhatatlanul szép világának megismerését, elég ha elképzelik az ***1/n*** függvényt. Ha az ***x*** tengelyen jobbról közelítjük a nullát, a függvényérték a végtelenhez tart. Látható tehát, hogy vannak olyan matematikai eszközök, melyekkel a nullával való osztás kezelhető. Ennek ellenére az osztás műveletének ez a kivétele függvényértékként nem értelmezett, és a mindennapi gyakorlatban feltétlenül külön kell vele foglalkozni. Hogyan módosíthatnánk az `osztas` metódust, hogy megfeleljen kívánalmainknak?

```
private void osztas(double szamlalo, double nevezo) {
   System.out.print(szamlalo + " / " + nevezo + "
   ➥= ");
   if (nevezo == 0) {
      System.out.println("nem ertelmezett!");
   } else {
      System.out.println(szamlalo / nevezo);
   }
}
```

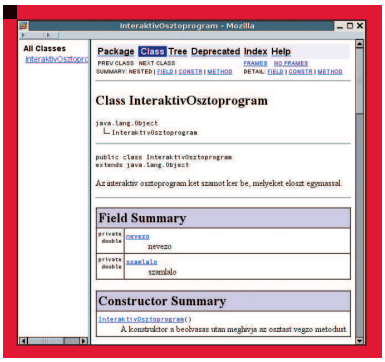
Ez a megoldás jól működik. Ám megszokhattuk, hogy ez önmagában kevés, ha a kód nem kellően beszédes. Ha egy olyan programozónak mutatnánk meg ezt a forrást, aki véletlenül nem ismeri az osztás műveletét, egy dolgot biztosan nem fog tudni eldönteni. Ebben a feltételes elágazásban melyik számít tipikus esetnek? Melyik ág az, ami normális működés mellett várhatóan lefut, és melyik az, amelyre csak egyfajta hiba előfordulásakor kerül a vezérlés? Jó volna valahogyan kifejezni ezt az ellentétet. Erre használhatjuk a `try - catch` párt.

```
private void osztas(double szamlalo, double nevezo) {
   System.out.print(szamlalo + " / " + nevezo + " =
   ➥");
   try {
      if (nevezo == 0) throw new Exception();
      System.out.println(szamlalo / nevezo);
   } catch (Exception kivetel) {
      System.out.println("nem ertelmezett!");
   }
}
```

A `try` blokk egy kritikus szakaszt hivatott jelölni. Ebben a szakaszban olyan műveletek vannak, amelyek miatt kivétel képződhet. Ez két esetben fordulhat elő. Egyrészt a programozó szándékosan dobhat kivételt a `throw` kulcsszó segítségével, amint az a példából is látszik. Másrészt vannak olyan műveletek, amelyek magukban rejtik ennek a lehetőségét. Például egész típusú (`int`) számok osztásakor, nulla értékű nevező esetén kivétel keletkezik.
A kivételek elkapása, azaz lekezelése egy `catch` ágban történik. Egy `try` blokkot több `catch` is követhet. A kivétel fellépésekor a vezérlés azonnal átkerül arra a `catch` ágra, amelyre a kivétel típusa először illeszthető. Jelen esetben egyetlen `catch` ág van, amely az `Exception`-ből, illetve az ebből örököltetett osztályokból származó példányokat kezelheti le. A `try` legelején, ha a nevező nulla, pont egy új `Exception` objektumot dobunk el. Így tehát, ha a `nevezo` változónk nulla, a `try` második sora már nem fut le, helyette az egyetlen `catch` ágban folytatódik a futás, ahol a megfelelő üzenet kerül kiíratásra.
Lássuk ennek a megoldásnak egy kicsivel pontosított változatát:

```
private void osztas(double szamlalo, double nevezo) {
   System.out.print(szamlalo + " / " + nevezo + " =
   ➥");
   try {
      if (nevezo == 0) throw new
      ➥ArithmeticException();
      System.out.print(szamlalo / nevezo);
   } catch (ArithmeticException kivetel) {
      System.out.print("nem ertelmezett!");
   } finally {
      System.out.print(System.getProperty
      ➥("line.separator"));
   }
}
```

Itt már `ArithmeticException` típusú objektumot dobunk, és kapunk el. Ez a változtatás beszédesebbé teszi a forráskódot, könnyebben kideríthető, milyen típusú hiba miatt keletkezett a kivétel. A Java kivételosztályok egész tárházát nyújtja, melyeknek közös ősosztályuk az `Exception`. Ha ezekkel nem lennénk megelégedve, készíthetnénk saját kivételosz-



tályt is, mindössze az `Exception` osztályt kell kiterjeszteni, és máris kivételhez jutunk, amit kedvünkre dobálhatunk.
A másik változtatás a `finally` blokk bevezetése. Ebből a `try` blokkhoz hasonlóan egy lehet egy szerkezetben, ám ennek a végére kell kerülnie. Attól különleges, hogy függetlenül attól, hogy a megszokott módon, a `try` blokkban, vagy valamelyik `catch` ágban ért véget a kiértékelés, a legvégén ez kötelező érvénnyel lefut. Ebben az ágban a rendszertulajdonságok közül lekérdezzük a soremelés karakterét, és kiíratjuk. Ezzel megszabadultunk a `println` műveletektől. Így már kellően olvasmányos kódhoz jutottunk, melyet próbáljunk meg felolvasni:
Az osztás műveletéhez két számra van szükség, a számlálóra és a nevezőre. Először írassuk ki a képernyőre az elvégzendő műveletet. Majd tegyünk egy próbát a művelet elvégzésére. Ha a nevező nulla, aritmetikai hiba van, mellyel később foglalkozunk. Normális esetben el kell osztani a számlálót a nevezővel, és kiíratni. Aritmetikai hiba esetén ki kell íratni, hogy a művelet nem értelmezett. Végezetül sort kell emelni.
Sajnos egyes, a strukturált programozást sem ismerő, mégis művelő emberek azt mondják a kivételkezelésre, hogy a modern nyelvekbe így hozták vissza a `goto` parancsot. A feltétel nélküli ugrás már a strukturált programozási nyelvekben sem volt szeretett megoldás, a *Java*-ból pedig már teljesen hiányzik. Foglalt kulcsszó, tehát például `goto` nevű változót nem hozhatunk létre, de nem nyelvi elem. A kivételkezelés nem egy tiltott elemnek egy új köntösben megjelenő változata, hanem egy olyan eszköz, amivel átláthatóbbá válnak a forráskódjaink. Nem láttam még kivételkezelést, ami összezavart volna, feltétel nélküli ugrást használó spagettikódot viszont igen.
Most tegyük fel, hogy egy metódusunkban felléphet valamilyen hiba, amit nem helyben szeretnénk lekezelni. Jelen példára vetítve, az osztás műveletet kivitelező tagfüggvénytől azt várjuk el, hogy ne írjon semmit a képernyőre, hanem visszatérési értékként szolgáltassa az eredményt, és mellesleg dobjon kivételt, ha a művelet sikertelen. Lássuk a teljes forrást!

```
/**
* Ez az osztaly az osztas muveletet valositja meg.
*/
public class Osztoprogram {

   /**
   * Az osztast vegzo metodus megjeleniti
   * a kepernyon az osztas eredmenyet.
   */
   private void osztas(double szamlalo, double
   ➥nevezo) {
      System.out.print(szamlalo + " / " + nevezo +
      ➥" = ");
      try {
         System.out.print(oszt(szamlalo, nevezo));
      } catch (ArithmeticException kivetel) {
         System.out.print("nem ertelmezett!");
      } finally {
         System.out.print(System.getProperty
         ➥("line.separator"));
      }
   }

   /**
   * Ez a metodus elosztja a szamlalot a nevezovel.
   * Nulla nevezo eseten kivetelt dob, amit
   ➥a hivonak
   * kell lekezelnie.
   */
   private double oszt(double szamlalo, double
   ➥nevezo) throws ArithmeticException {
      if (nevezo == 0) throw new
      ➥ArithmeticException();
      return (szamlalo / nevezo);
   }

   /**
   * A konstruktor harom osztast vegez el.
   */
   public Osztoprogram() {
      osztas(1.0, 2.0);
      osztas(1.0, 3.0);
      osztas(1.0, 0.0);
   }

   /**
   * Az alkalmazas inditasakor letrejon egy
   * peldany az Osztoprogram osztalybol
   */
   public static void main(String[] args) {
      new Osztoprogram();
   }

}
```

A kód immár komoly esztétikai élvezetet nyújt a szakértő szemeknek. Bevezettünk egy `oszt` nevű tagfüggvényt, ami átvette az `osztas` metódus válláról a számolás terhét. Utóbbi csak kiírat, előbbi csak az eredményt számolja. Mivel az `oszt` metódusban kivétel képződhet, amit helyben nem kezelünk le, a függvény fejlécében jelezni kell, hogy a hívó felelőssége a megfelelő `try - catch` szerkezetről gondoskodni. Természetesen az is továbbháríthatja ezt a felelősséget fentebbre, ha a fejlécében szerepel a kivétel megnevezése.
Eljött az idő, hogy elkészítsük az alkalmazás felhasználóbarát változatát. Ebben már a kezelő adhatja a bemenő adatokat, vagyis nem rögzítjük a forráskódban a művelet operandusait.

```
import java.io.*;
```

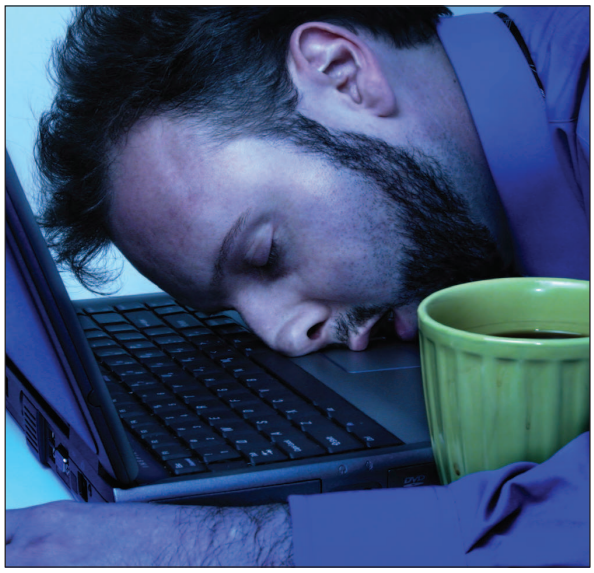
```
/**
* Az interaktiv osztoprogram ket szamot
* ker be, melyeket eloszt egymassal.
*/
public class InteraktivOsztoprogram {

   /**
   * szamlalo
   */
   private double szamlalo;
   /**
   * nevezo
   */
   private double nevezo;

   /**
   * Beolvassa az adatokat a kezelotol, es
   * azonnal double tipusuva is alakitja.
   */
   private void beolvasas() {
      BufferedReader olvaso = new
BufferedReader(new InputStreamReader(System.in));
      try {
         System.out.print("szamlalo? ");
         szamlalo = Double.parseDouble
         ➥(olvaso.readLine());
         System.out.print("nevezo? ");
         nevezo = Double.parseDouble
         ➥(olvaso.readLine());
      } catch (IOException kivetel) {
         System.out.println
         ➥("Hiba tortent az adatbevitel soran!");
         System.exit(1);
      }
   }

   /**
   * Az osztast vegzo metodus megjeleniti
   * a kepernyon az osztas eredmenyet.
   * @param szamlalo
   * az osztas szamlaloja
   * @param nevezo
   * az osztas nevezoje
   */
   private void osztas(double szamlalo, double
   ➥nevezo) {
      System.out.print(szamlalo + " / " + nevezo +
      ➥" = ");
      try {
         System.out.print(oszt(szamlalo, nevezo));
      } catch (ArithmeticException kivetel) {
         System.out.print("nem ertelmezett!");
      } finally {
         System.out.print(System.getProperty
         ➥("line.separator"));
      }
   }

   /**
   * Ez a metodus elosztja a szamlalot a nevezovel.
```



```
   * Nulla nevezo eseten kivetelt dob, amit
   ➥a hivonak
   * kell lekezelnie.
   * @param szamlalo
   * az osztas szamlaloja
   * @param nevezo
   * az osztas nevezoje
   * @return az osztas eredmenye
   */
   private double oszt(double szamlalo, double
   ➥nevezo) throws ArithmeticException {
      if (nevezo == 0) throw new
      ➥ArithmeticException();
      return (szamlalo / nevezo);
   }

   /**
   * A konstruktor a beolvasas utan
   * meghivja az osztast vegzo metodust.
   */
   public InteraktivOsztoprogram() {
      beolvasas();
      osztas(szamlalo, nevezo);
   }

   /**
   * Letrehoz egy peldanyt az
   * InteraktivOsztoprogram osztalybol.
   */
   public static void main(String[] args) {
      new InteraktivOsztoprogram();
   }

}
```

Az alkalmazás igen egyszerűen működik. A `beolvasas` metódus az objektum két tagváltozójának, a `szamlalo`, és a `nevezo` tagoknak ad értéket. Ezeket felhasználva hívja meg a konstruktor az `osztas` metódust, melyet az `oszt`

segítő függvénnyel együtt a meglévő `Osztoprogram.java` forrásból emeltük át. A `beolvasas` jó példája annak, hogyan használhatjuk a kivételkezelésre építő ***Java*** könyvtári osztályokat. A `readLine` metódus `IOException` típusú kivételt dob, ha az olvasás sikertelen volt, egyébként visszaadja a beolvasott sort `String` objektumként. A `beolvasas` különlegessége még az eddig látottakhoz képest néhány új osztály használata, így többek között a `BufferedReader` és az `InputStreamReader` alkalmazása. Ezek a `java.io` csomag részei, ezért volt szükség a program legelején jelezni a csomag összes elemének behozatalát az alkalmazás névterébe. A csomagokról a későbbiekben még szó lesz, most legyen elég annyi, hogy egy osztály használatához be kell hozni azt a névtérbe. A `System` és az `Exception` osztályok esetén ezt azért kerülhettük el, mert azok a `java.lang` csomag részei, és ezért önműködően a névtér elemeit képezik.

Térjünk vissza a `Reader` osztályokhoz. Ezek segítségével olvastunk be egy-egy sort a billentyűzetről. Most ahelyett, hogy a cikk írója által adott magyarázatokból eredő kényelem miatt a kedves Olvasó hagyná, hogy elhatalmasodjon rajta a lustaság, nézzünk utána együtt ezen osztályok leírásának az API-ban. Ebből kiderül, hogy a `System.in`, ami a szabványos bemeneti csatorna, `InputStream` típusú. Ennek van ugyan `read` metódusa, ám amint az a leírásból kitűnik, bájtokat olvas és nem karaktereket. Például egy olyan terminálon, ami ***Unicode*** kódolást használ, ez nem adna használható eredményt. Ezért beburkoljuk egy `InputStreamReader`-rel, ami technikailag azt jelenti, hogy készítünk egy ilyen objektumot, és a konstruktorának átadjuk a `System.in`-t paraméterként. Ez már valóban karaktereket olvas, tehát elvégzi a szükséges átalakításokat, hogy használható bemenetet kapjunk. Ahhoz, hogy a sorvége kezelésével és a karakterek összeragasztásával se kelljen vesződnünk, ezt még beburkoljuk a `BufferedReader` osztállyal, amelynek példányai rendelkeznek az áhított `readLine` metódussal. A megoldásban az a szép, hogy a `System.in`-t egy másik adatfolyamra cserélve is tökéletesen működik, így tehát fájlból is ugyanezzel a módszerrel dolgozhatunk.

Vessünk egy pillantást a másik újdonság, a `Double` osztály leírására. Az osztálynak a `parseDouble` nevű metódusát használtuk, mindennemű példányosítás nélkül. Ezt azért tehettük meg, mert a nevezett tagfüggvény statikus. Paraméterként egy `String` típusú objektumot vár, és egy `double` típusú számot ad vissza. Meglepő ravaszsággal a szövegfüzérből egy lebegőpontos számot állít elő.

Az `osztas`, illetve `oszt` műveletek megvalósításai nem változtak, mindössze a megjegyzések egészültek ki egy-két érdekes elemmel. Ennek oka abban keresendő, hogy az Olvasóban az ***API*** tüzetes tanulmányozása után biztosan felmerült a kérdés, hogy miként tudna hasonlóan jól áttekinthető, egységes leírást mellékelni ***Java*** alkalmazásaihoz. A válasz egyetlen parancssorral megadható:

```
$ javadoc -d docs -private
➥InteraktivOsztoprogram.java
```

A parancsot abban a könyvtárban kell kiadni, ahol a forráskód található. A `-d` kapcsoló azt a célkönyvtárat határozza meg, amelyben a ***HTML*** alapú dokumentációt szeretnénk látni. A `-private` hatására a személyes láthatóságú adattagok is bekerülnek a leírásba, ami azért fontos, mert nem ez az alapértelmezés. A kapcsolók után azon megfelelő formátumú megjegyzésekkel tűzdelt forrásokat kell felsorolni, amelyekből ki szeretnénk nyerni az értékes információt. Jelen esetben ez egyetlen állomány.

Rövid várakozást követően elkészül a `docs` könyvtár, melyből az `index.html` lapot megnyitva kedvenc böngészőnkből a képen látható tartalom tárul elénk. Hasonlítsuk össze a leírást a forráskód megjegyzéseivel, és máris látni fogjuk, miből lesz a cserebogár. A dokumentációban is megjelenő megjegyzések kötelezően `/**` és `*/` között szerepelnek, és mindig a hivatkozott elem előtt. Metódusok esetén érdemes kihasználni a címkék nyújtotta lehetőségeket. A címkéket `@` jel vezeti be, és így kényelmesen megadható egy tagfüggvény paramétereinek és visszatérési értékének rövid magyarázata.

Ennyit terveztem erre a hónapra. Ha összehasonlítjuk az alcímben is szereplő két témának a súlyát a cikkben, meglepődve tapasztalhatjuk, mennyire részrehajlóan sok szó esett az elsőről. Ennek oka az, hogy a ***javadoc*** egy az egyszerű, és mégis nagyszerű eszközök közül. Érdemes egy kis időt eltölteni a kapcsolók listájának böngészésével, ám kezdetnek tökéletesen elég, ha gondosan karban tartjuk megjegyzéseinket, mert így pillanatok alatt kiváló dokumentációt varázsolhatunk.

Sok örömet kívánok a kísérletezéshez, a sorozattal kapcsolatban pedig várom az észrevételeket, javaslatokat.

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