Home

Last edited by **javagyak** 2 weeks ago

Clone repository (/javagyak/02/wikis/git_access)

Home (/javagyak/02/wikis/home)

Tantárgyi követelmények (http://kto.web.elte.hu/hu/oktatas/java/) Java SE 8 Documentation (http://docs.oracle.com/javase/8/docs/) Feladatok

Emlékeztető

```
$ javac HelloWorld.java
$ java HelloWorld
Hello World!
```

Osztályok definiálása és példányosítása

A korábbiakban már láthattuk, hogy Java nyelvben alkalmazható típusok alapvetően két csoportra bonthatóak: primitív típusokra és objektumtípusokra. Ezek közül az előbbi lényegében néhány előre definiált, beépített eszköz (pl. `int` az egész számok ábrázolásához), az utóbbi pedig a szabványos könyvtárban definiált típusok halmaza. A objektumtípusok halmazát mi magunk is tudjuk (sőt, tulajdonképpen kell is) bővíteni saját osztályok definiálásával.

Például tegyük fel, hogy szeretnénk kétdimenziós pontokat ábrázolni és megadni a rájuk értelmezhető műveleteket.

```
// Kétdimenziós pontok ábrázolása
class Point2D {
    // Koordináták mint mezők
    double x;
    double y;

    // Pont eltolása egy vektorral
    void translate(double dx, double dy) {
        x += dx;
        y += dy;
    }

    // Pont középpontos nagyítása egy skalárral
    void scale(double c) {
        x *= c;
        y *= c;
    }

    // Pont forgatása adott szöggel
    void rotate(double phi) {
        double nx = x * Math.cos(phi) - y * Math.sin(phi);
        double ny = x * Math.sin(phi) + y * Math.cos(phi);
        x = nx;
        y = ny;
    }

    // Pont átalakítása szöveggé
    String show() {
        return String.format("{ x = %.3f, y = %.3f }", x, y);
    }

    // Két pont távolsága
    static double distance(Point2D p0, Point2D p1) {
        double xd = p0.x - p1.x;
        double yd = p0.y - p1.y;
        return (Math.sqrt((xd * xd) + (yd * yd)));
    }

    // A második pont tükrözése az elsőre
    static Point2D mirror(Point2D c, Point2D p) {
        Point2D result = new Point2D();
        result.x = (2 * c.x) - p.x;
        result.y = (2 * c.y) - p.y;
        return result;
    }
}

// Szögekkel kapcsolatos segédműveletek
class Angle {
    // Fok -> radián
    static double degreeToRadian(double x) {
```

Clone repository (/javagyak/02/wikis/git_access)

Home (/javagyak/02/wikis/home)

```
        return x * Math.PI / 180;
    }

    // Radián -> fok
    static double radianToDegree(double x) {
        return x * 180 / Math.PI;
    }
}

// "Főprogram"
public class Point2DDemo {
    public static void main(String[] args) {
        Point2D p0 = new Point2D();
        p0.x = Math.E;
        p0.y = Math.PI;
        p0.translate(0.1, 12.34);
        p0.scale(0.78);
        p0.rotate(Angle.degreeToRadian(90));
        System.out.println("p0 = " + p0.show());

        Point2D p1 = new Point2D();
        p1.x = 2 * Math.PI;
        p1.y = Math.E / 2;
        p1.scale(1.445);
        p1.rotate(Angle.degreeToRadian(-33));
        System.out.println("p1 = " + p1.show());

        System.out.println("distance(p0, p1) = " + Point2D.distance(p0, p1));

        Point2D p2 = Point2D.mirror(p1, p0);
        System.out.println("mirror(p1, p0) = " + p2.show());
    }
}
```


Érdemes megemlíteni, hogy elegendő a főprogramot, vagyis a `Point2DDemo` osztályt lefordítani. A fordító ennek lefordítása közben ugyanis az összes többi hivatkozott osztályt is megpróbálja lefordítani.

Metódusok (Függvények, eljárások)

Az előbbi kódban láthattuk, hogy az osztályokban szereplő műveleteket függvényekkel, vagy Java terminológia szerint metódusokkal adtuk meg. Ezek általános alakja:

```
<módosítószavak> <visszatérési érték> <név>( <paraméterek listája> ) [ throws <kivétel
    <utasítás1>;
    <utasítás2>;
    ...
}
```

Ezekről röviden:

- A módosítószavak a metódus viselkedését tudják befolyásolni. Ezek közül több is létezik, amelyekkel majd a félév során folyamatosan megismerkedünk. Jelenleg a `static` kulcsszót használjuk, amely azt adja meg, hogy a metódus osztályszintű, vagyis nem szükséges aktív objektumpéldány a meghívásához. Lényegében hagyományos függvénydefinícióként működik.
- A visszatérési érték lehet egy tetszőleges (létező) típus, valamint `void`. A `void` esetében lényegében eljárásról beszélünk.
- Metódusnév: az alprogram neve, formátuma `lowerCamelCase`.
- Paraméterlista: az alprogram formális paramétereinek listaszerű felsorolása, típussal és azonosítóval. Az elemei minden esetben érték szerint adódnak át.
- Kivételek listája: a metódus által kiváltható ellenőrzött kivételek, előbb is majd később lesz szó.

Csomagok

Egy csomag osztályok és interfészek gyűjteménye. Csomagok alkalmazásának általában a következő okai, céljai vannak:

- a programok fejlesztésének modularizálása,
- névütközések feloldása névterek létrehozásával,
- hozzáférés szabályozása.

A csomagoknak nevet kell adni, ezeket általában pontokkal tagolt azonosítósorozattal nevezzük el. Általános elnevezési konvenció, hogy a csomagnevek globális (világszinten) egyediek, és az interneten található domain névekhez hasonló felépítés szerint képződnek, pl. `org.ietf.jgss`.

A Java szabványos csomagjai többnyire a `java.` előtaggal rendelkeznek, illetve a gyakorlatok során a programjainkban is egyszerűsített csomagelnevezéseket fogunk alkalmazni.

Importálás

A Java nyelv alapvető objektumait és interfészeit (pl. `Object`, `String`, stb.) a `java.lang` csomag tartalmazza, amelynek elemei közvetlenül elérhetőek a forrásszövegekben. Minden más csomag elemeit egy ún. teljesen minősített név ("fully-qualified name") használatával tudjuk elérni. Például `java.util.Scanner` a `java.util` csomagban található `Scanner` osztályt nevezi meg.

```
public class Simple {
    public static void main(String args[]) {
        Scanner sc = new Scanner(System.in);
        System.out.print("? ");
        if (sc.hasNextInt()) {
            int x = sc.nextInt();
            System.out.println("x = " + x);
        }
        else System.out.println("No valid input.");
    }
}
```

Amennyiben nem akarjuk a külső neveket a programunkat minden alkalommal teljesen kiírni, úgy `import` utasítások segítségével beemelhetjük a csomagunk névterébe más csomagok neveit.

```
import java.util.Scanner;   // Egy konkrét típus esetén
import java.math.*;         // A csomag minden elemének láthatóvá tétele
```

Az fenti program importok segítségével:

```
import java.util.Scanner;

public class Simple {
    public static void main(String args[]) {
        Scanner sc = new Scanner(System.in);
        int x = sc.nextInt();
        System.out.println("x = " + x);
    }
}
```


Hasznos csomagok:

- `java.awt` : Grafikus felületek programozásához,
- `java.awt.event` : Grafikus felületek eseményekezelése,
- `javax.swing` : A grafikus felületek másik implementációja,
- `java.util` : Adatszerkezetek és egyéb hasznos segédeszközök,
- `java.util.regex` : Reguláris kifejezések,
- `java.io` : Input/Output kezelése.

Saját csomagok készítése

Csomagokon keresztül természetesen a saját programunkat is tudjuk strukturálni. Ennek két fontos lépése van:

- A forrásszövegben el kell helyeznünk a befoglaló csomag azonosítóját. Ezt a `package` kulcsszóval tehetjük meg, amely után megadhatjuk az azonosítót. Ennek a korábban említett teljesen minősített névnek kell lennie.
- A forráskódokat a csomag nevének megfelelő könyvtárhierarchiába kell szervezni és a fordítást, majd a futtatást az ún. csomagtőből ("package root") kell végrehajtani. Ennek hiányában nem fog rendesen működni.

Mindezek szemléltetéséhez a korábban elkészített, kétdimenziós pontokkal foglalkozó programunkat bontsuk csomagokra! A `Point2D` osztályt tegyük a `geo.basics` csomagba, az `Angle` osztályt a `geo.utils` csomagba, végül a főprogramként szolgáló `Point2DDemo` osztályt a `main` csomagba!

Ehhez először ki kell alakítanunk a megfelelő könyvtárszerkezetet, majd elhelyeznünk benne a forráskódokat:

```
$ md geo
$ md geo\basics
$ md geo\utils
$ md main
$ mv Angle.java geo\utils
$ mv Point2D.java geo\basics
$ mv Point2DDemo.java main
```

Tehát vizuálisabban:

```
csomagtő (befoglaló könyvtár)
  |
  |- geo
  |   |- basics
  |   |   Point2D.java
  |   |
  |   '- utils
  |       Angle.java
  |
  '- main
      Point2DDemo.java
```


Ezt követően pedig mindegyik forrásszöveg tetejére be kell illeszteni a neki megfelelő csomag azonosítóját. Például a `Point2D.java` elejére ezt kell odatenni:

```
package geo.basics;
```

Az `Angle.java` esetén:

```
package geo.utils;
```

Végül a `Point2DDemo.java` esetén:

```
package main;

import geo.basics.Point2D;
import geo.utils.Angle;
```

Valamint az osztályokat és a bennük szereplő összes tagot (mezőt, metódust) a `public` módosítóval kell ellátni. Ez engedélyezi, hogy kívülről is el lehessen érni ezeket. Alapértelmezés szerint ugyanis ezek láthatósága csak a csomagon belülre terjed ki.

Fordítás és futtatás

A fordító nem támogatja a források rekurzív fordítását, viszont megadható neki egy állomány a `@` karakterrel, amelyben a fordítani kívánt állományokat lehet felsorolni.

```
$ dir /s /B *.java > sources.txt
$ javac @sources.txt
```

A több csomagból álló alkalmazásunkat ezután a csomagtőből tudjuk futtatni, a főosztály (`Point2DDemo`) teljesen minősített nevének megadásával.

```
$ java main.Point2DDemo
```

Programok nyomkövetése

A programokban olykor hibát kell keresnünk, vagy meg kell értenünk, pontosan miként is működik. Ezt a folyamatot debuggolásnak, esetleg trace-elésnek ("trészelés"), magyarosabban pedig nyom(on)követésnek nevezik.

A program nyomonkövethetőségéhez gyorsabb kódot a `-g` kapcsolóval kell fordítani, a `javac` segítségével. Ez fogja engedélyezni az összes szükséges segédinformáció elhelyezését a létrehozott tárgykódban.

```
$ javac -g X.java
```

Clone repository (/javagyak/02/wikis/git_access)

Sikeres fordítást követően a `jdb` (mint "Java DeBugger") nevű segédprogrammal tudjuk a programban elhelyezett nyomkövetési segédinformációkat felhasználni. Ehhez el kell indítanunk a parancssorból, amelyre válaszul másik egy interaktív paranccsort kapunk.

Home (/javagyak/02/wikis/home)

```
$ jdb
Initializing jdb ...
>
```

A nyomkövető programban, vagyis a debuggerben használható parancsokat a `help` parancs segítségével lehet listázni. Ezek közül most csak a legfontosabbakat fogjuk áttekinteni.

```
> help
** command list **
[..]
run [class [args]]        -- start execution of application's main class
[..]
print <expr>              -- print value of expression
[..]
eval <expr>               -- evaluate expression (same as print)
set <lvalue> = <expr>     -- assign new value to field/variable/array element
locals                    -- print all local variables in current stack frame
[..]
stop at <class id>:<line> -- set a breakpoint at a line
[..]
step                      -- execute current line
[..]
cont                      -- continue execution from breakpoint
[..]
exit (or quit)            -- exit debugger
[..]
```

Az első, és talán legfontosabb parancs a `stop at`, amellyel a program futását majd meg tudjuk állítani egy ponton. Ennek segítségével ún. töréspontokat ("breakpoint") lehet a program kódjában elhelyezni. Amikor a program futása (a debuggeren belül) elér egy aktív töréspontot, akkor megáll és a visszakapjuk a debugger parancssorát.

```
> stop at Point2DDemo:3
Deferring breakpoint Point2DDemo:3.
It will be set after the class is loaded.
```

A töréspontokat az osztály betöltődése nélkül is be lehet állítani, aktiválódni akkor fognak, amikor az osztály adott részére adódik a vezérlés. A betöltött osztály `main()` metódusának futtását a `run` paranccsal tudjuk kérni.


```
> run Point2DDemo hello
run Point2DDemo hello
Set uncaught java.lang.Throwable
Set deferred uncaught java.lang.Throwable
>
VM Started: Set deferred breakpoint Point2DDemo:3

Breakpoint hit: "thread=main", Point2DDemo.main(), line=3 bci=0
3            Point2D p0 = new Point2D();
```

Clone repository (/javagyak/02/wikis/git_access)

Home (/javagyak/02/wikis/home)

A töréspont elérésekor a debugger meg is mutatja az érintett sort, annak sorszámával együtt. Ilyenkor a program futása felfüggesztődik. Ekkor van lehetőségünk megvizsgálni például az éppen elérhető (scope-ban levő) változók értékeit, illetve ezekhez kifejezéseket kiértékeltetni.

A `locals` parancs ezek felderítésében segít.

```
main[1] locals
Method arguments:
args = instance of java.lang.String[1] (id=386)
Local variables:
```

Látható, hogy ezen a ponton még csak a parancssori paramétereket tároló `args` változó érthető el. Ezt a `print` paranccsal tudjuk megvizsgálni.

```
main[1] print args
 args = instance of java.lang.String[1] (id=386)
main[1] print args.length
 args.length = 1
main[1] print args[0]
 args[0] = "hello"
```

A forráskódban szereplő `p0` változó csak akkor fog létrejönni, ha engedjük lefutni azt a sort. A soronkénti egyenkénti futtatását léptetésnek ("step") hívjuk, ezért ezt a `step` paranccsal tudjuk megtenni.

```
main[1] step
>
Step completed: "thread=main", Point2D.<init>(), line=1 bci=0
1    class Point2D {
```

Ekkor a vezérlés a `Point2D` osztály példányának inicializálására adódik, mivel a példányosítás során ennek is le kell futnia. Léptessük addig a végrehajtást, amíg a `p0` valóban létre nem jön:

```
main[1] step
>
Step completed: "thread=main", Point2DDemo.main(), line=3 bci=7
3           Point2D p0 = new Point2D();
```



```
main[1] step
>
Step completed: "thread=main", Point2DDemo.main(), line=4 bci=8
4           p0.x = Math.E;
```

Clone repository (/javagyak/02/wikis/git_access)

Home (/javagyak/02/wikis/home)

Ekkor már a `p0` is megvizsgálható:

```
main[1] print p0
 p0 = "Point2D@faa824"
```

Vagy a programban is alkalmazott formában:

```
main[1] print p0.show()
 p0.show() = "{ x = 0.000, y = 0.000 }"
```

Szükség esetén a programban szereplő változók értékei meg is változtathatóak. Ez leginkább akkor hasznos, ha valamilyen rossznak vélt értéket akarunk helyreállítani, vagy csak kísérletezni akarunk velük.

```
main[1] print p0.x
 p0.x = 0.0
main[1] set p0.x = 1.0
 p0.x = 1.0 = 1.0
main[1] print p0.x
 p0.x = 1.0
```

Ha folytatni szeretnénk a program futtatását a megszokott tempóban, akkor a `cont` parancssal tudjuk továbbengedni. Így a program futni fog normál módban egészen addig, amíg el nem ér egy újabb töréspontot, vagy be nem fejeződik.

```
main[1] cont
> p0 = { x = -12.076, y = 2.198 }
p1 = { x = 8.684, y = -3.298 }
distance(p0, p1) = 21.474953930229827
mirror(p1, p0) = { x = 29.444, y = -8.794 }

The application exited
```

Feladatok

- Hozzunk létre egy nemek ábrázolásához használt `Gender` nevű osztályt! Ebben szerepelje két osztályszintű konstans, amelyek rendre `Gender.MALE` (férfi) és `Gender.FEMALE` (nő).
- Készítsünk `Person` névvel egy olyan osztályt, amelyben személyi adatokat tudunk eltárolni! A rögzíteni kívánt adatok: a személy vezeték- és keresztneve (mind a kettő `String`), neme (`Gender`), születési éve (`int`).

- Legyen a `Person` osztálynak egy olyan statikus metódusa `makePerson()` névvel, amely ezeket az adatokat paraméterként bekéri és ezekből összeállít egy `Person` típusú objektumot!

  A létrehozás előtt azonban ellenőrizzük, hogy minden megadott paraméter eleget tesz a személyi adatokra érvényes megszorításokra:

  - A vezeték- és keresztnév nem lehet üres, valamint legalább kétkarakteresnek kell lennie és nagybetűvel kell kezdődnie. Ezek kifejezéséhez nyugodtan használjuk a `java.lang.String` (http://download.oracle.com/javase/8/docs/api/java/lang/String.html) osztályt!
  - A születési évnek 1880 és 2014 között kell lennie.

  Amennyiben a megadott paraméterek nem tesznek eleget a fenti megszorításoknak, úgy a metódus adjon vissza üres, vagyis `null` referenciát! (Vagyis ilyenkor objektum nem jön létre.)
- Egészítsük ki a `Person` osztályt egy `show()` metódussal, amely `String` típusú értékké alakítja az adott objektum belső állapotát!
- Adjunk a `Person` osztályhoz egy `isAdult()` metódust, amely egy `boolean` típusú értékkel tér vissza, és attól függően igaz (`true`) vagy hamis (`false`), hogy az illető személy 18 évesnél idősebb vagy sem!
- Készítsünk egy `equals()` nevű metódust a `Person` osztályhoz! Ennek az a feladata, hogy eldöntse a paraméterként megadott másik `Person` objektumról, hogy megegyezik-e az aktuális példánnyal. Ennek megfelelően a visszatérési értéke egy `boolean` lesz, amely igaz, ha megegyezik, ellenben hamis.

  Vigyázzunk arra, hogy mivel referenciát adunk át paraméterként, az lehet (többnyire véletlenül) `null` érték is! Ilyenkor értelemszerűen az eredménye hamis lesz.
- Tegyük az eddigi osztályokat a `person` csomagba és készítsünk hozzá egy főprogramot, amelyben szabványos bemenetről bekérve létrehozunk egy `Person` objektumot és kiírjunk a szabványos kimenetre! A főprogram kerüljön a `main` csomagba!

Linkek