Kezdő Java programozók hibái

Azok a buktatók, amikről ha nem tudsz, napokat töprenghetsz hiába Java-tanulás közben

2016.03.01.

www.programozas-oktatas.hu
Pasztuhov Dániel

Miért írom ezt az útmutatót?

Programozás-oktatói pályafutásom alatt nagyon sokszor találkoztam olyanokkal, akik a nulláról akarnak megtanulni programozni. Egy idő után megfigyeltem, hogy szinte minden esetben belefut a tanítvány ugyanazokba a hibákba. Szinte kivétel nélkül mindenki, sőt, van, aki többször is. Az eredménye persze az, hogy utána órákat – ha nem napokat – a gép előtt görnyed és „guglizik”, hátha rátalál a megoldásra, de általában nem. Így aztán hozzáértő segítségét kell kérnie.

Így aztán arra gondoltam, hogy elkezdem gyűjteni ezeket a hibákat, és összegyűjtök egy csokorra valót tanulságul azoknak, akik éppen most kezdenek programozni tanulni, hogy megspóroljak nekik néhány napnyi szenvedést és guglizást.

Ez persze nem jelenti azt, hogy az anyag teljes lenne. Ezek csak azok a hibák, amiket az én feladataim kapcsán láttam rendszeresen előjönni, nyilván rengeteg másfajta hiba lehetséges még.

Az eredeti anyag a C-jellegű nyelvek (C, C++, PHP, Java) készült, de most közreadom annak kimondottan a Javához hozzáigazított, kibővített verzióját. Az elkövetkezendő 6 oldalon összesen 20 hibát, kérdést, problémás helyzetet mutatok be.

Tartalomjegyzék

1. Az alapok csapdái

a. A figyelem

Sajnos sokszor előfordul, hogy szóról szóra elmondom a megoldását az adott problémának, majd rákérdezek újra, és a tanítvány csak néz, mint Rozi a moziban. Az alábbi tulajdonságokra természetesen fel szoktam hívni a hallgatóság figyelmét, ellenben általában ez nem sokat ér.

A figyelem nagyon fontos, főleg, ha meg akarsz tanulni programozni!

b. Az egészosztásról

Adott ez a programrészlet:

```
int c = 45;
int f = 9 / 5 * c + 32;
```

… és a tanítvány nem tudja, miért kap 113 helyett 77-et. A magyarázat az osztás „kétszínűségében” rejlik. Ha mindkét paramétere egész, akkor az osztás matematikai eredményének egészrészét adja (egészként). Ha legalább az egyik paramétere törtszám (float, double), akkor pedig a matematikai eredmény lesz az értéke törtszámként. Tehát helyesen így lehetne (többek között):

```
int c = 45;
float f = 9.0f / 5 * c + 32;
```

c. String összeadás furán működik

Folytassuk az előző kódrészletet. Azt tanuljuk, hogy kiírni egy változó értékét így lehet:

```
System.out.println(”Hőmérséklet: ”+hom);
```

Változó érték helyén állhat biztosan kifejezés is, akkor legyen:

```
System.out.println(”Hőmérséklet: ”+ 9.0/5.0 * c + 32);
```

Hogyan értelmezi a Java? (c=1)

- 9.0 / 5.0 * c = 1.8
- ”Hőmérséklet: ” + 1.8 = ”Hőmérséklet: 1.8”
- ”Hőmérséklet: 1.8” + 32 = ”Hőmérséklet: 1.832”

d. a = b, azaz b felveszi a értékét…

Nagyon sokan nem tudják elsőre megjegyezni, hogy az értékadás merről merre zajlik. Nos, nem balról jobbra, hanem JOBBRÓL BALRA.

e. if (a = b)…

Az elágazás (if) tanulásánál nagyon gyakori ez a gubanc. A tanítvány feltehetőleg össze akarta hasonlítani a-t és b-t és ha megegyeznek, akkor akart valamit csinálni. Helyesen úgy lett volna, hogy

```
if (a == b)…
```

de a tanítvány még nem tudja, hogy az = és a == operátorok között mi a különbség. Az = értékadás, míg az == összehasonlítás.

A Java fordító aláhúzza, a tanítvány meg nem érti, miért?

f. if, while, for törzsében több utasítás

```
if (a == b)
     egyutasítás;
     mégegyutasítás;
```

A tanítvány azt szerette volna, hogy ha a és b értékei megegyeznek egymással, akkor lefusson mind egyutasítás, mind mégegyutasítás. De a blokkjelet elfelejtette körülük. Általában akkor jelentkezik ez a hiba, amikor először csak egy utasítás áll az if igaz ágában, majd hozzátesz még egyet, csak megfeledkezik a { } párról. És nem érti, miért nem fut le a második utasítás. Azért mert az if (és for, while) esetén a törzs vagy egy utasításból áll, vagy ha több utasításból áll, akkor kapcsos zárójelek közé kell tenni, azaz helyesen:

```
if (a == b) {
     egyutasítás;
     mégegyutasítás;

}
```

g. for végére pontosvessző?

Adott a következő sor:

```
for (i = 6; i < 10; i++); { és a műveletek }
```

… és a tanítvány nem érti, hogy miért csak egyszer fut le, amit a blokkba tett. Hiszen már blokkban van! A rejtély kulcsa a pontosvessző, ami jelen esetben egy üres utasítást jelképez. a for-ciklus lefut négyszer, és lefuttatja az üres utasítást, majd hozzákezd a blokk végrehajtásához – egyszer…

h. Ellenőrzés problémái

A do-while ciklust szoktam arra javasolni, hogy ha egy bevitelt nem jól ad be a felhasználó, akkor kéressük be a programmal újra.

```
System.out.println(”Kérem a számot!”);
Scanner sc = new Scanner(System.in);
int szam;
szam = sc.nextInt();
while (szam < 0 || szam > 5);
```

Minden jól működne, ha a 3. és 4. sor közül nem hiányozna a „do {” (lletve a while elöl a bezáró kapcsos zárójel („}”).

Így viszont végtelen ciklust okoz, hiszen az utolsó sor egy önálló while-ciklus, amely addig hajtja végre az üres utasítást („;”), míg hamis nem lesz a feltétel. A feltétel meg sosem lesz hamis… ☺

i. do-while feltétele

Előfordul, hogy ilyesmivel hoz össze a sors:

```
System.out.println(”Kérem a számot!”);
Scanner sc = new Scanner(System.in);
do {
    int szam = sc.nextInt();
} while (szam < 0 || szam > 5);
```

A tanítvány nem érti, hogy a `while` feltételében miért van aláhúzva a „`szam`”, hiszen létrejött a ciklus törzsében.

Aha, de meg is szűnt a `while` előtti bezáró kapcsos zárójelnél. ☺

A helyes programkód:

```
System.out.println(”Kérem a számot!”);
Scanner sc = new Scanner(System.in);
int szam;
do {
    szam = sc.nextInt();
} while (szam < 0 || szam > 5);
```

j. switch-case

A leggyakoribb probléma az, hogy a case-ágak végén lemarad a break.

k. Hol van a laptopon a * billentyű?

AltGr+ SHIFT melletti ’-’ (mínusz) jel.

2. Függvények nyalánkságai

a. Függvény meghívása és a deklarációja

Sokan összekeverik a kettőt. Adott a függvény deklarációja:

```
public static int fuggveny(int a, long b);
```

valamint a függvény meghívása:

```
x = fuggveny(c, d);
```

Néha természetesen összekeveredik, így a meghívásba tesznek típusnevet… Az ökölszabály: ha függvényt deklarálunk, van típusnév, ha meghívjuk, nincs.

b. A függvény meghívásakor, ha változót használunk, akkor annak nevének meg kell egyeznie a (formális) paraméter nevével.

Természetesen nem. Például az alábbi kóddal jól szemléltethető:

```
public static int max(int a, int b) { return a > b ? a : b; }
```

…

```
int a,b,c,d, ab,cd;
```

…

```
ab = max(a,b);
cd = max(c,d);
```

Hát már miért ne számolhatnánk ki c és d maximumát?

c. Visszatérési érték

Éppen megtanulta a tanítvány a függvények használatát, máris gondban lehet. Nem érti, hogy mikor meghívja a függvényt a visszatérési érték az mi a csuda, és hogy hogy kell azt kezelni. Gyakran láthatunk ilyesmit:

```
egyfuggveny(p1, p2, p3);
valtozo = egyfuggveny(p1, p2, p3);
```

Azt gondolja, hogy az `egyfuggveny` valahogy magába szívja a visszatérési értéket, és bent tartja, amíg le nem írjuk a függvény nevét (rosszabb esetben teljes meghívását) újra, és akkor végre kiereszthetí magából. Látszólag egyébként jól is működik a dolog. A szépséghiba az, hogy a függvény kétszer fut le.

A másik lehetséges probléma, ha nem is tárolják a visszatérési értéket, csak gondolják, hogy majd visszajött valahogy, és majd felhasználjuk… valahogy.

d. Visszatérés logikai értékkel

Ha egy függvénynek meg kell vizsgálnia egy feltételt és annak megfelelően vissza kell térnie egy igaz vagy hamis értékkel, azt sokan így csinálják:

```
if (a<b)
      return true;
else
      return false;
```

Pedig egyszerűbben is lehetne:

```
return a < b;
```

e. Hányszor lehet leírva a return egy függvényben?

Van, aki szerint egyszer, de ő téved. A return akárhányszor le lehet írva a függvényben, de ha a vezérlés ráfut, akkor kilép a függvényből. Olyan, mint a rexasztal. Több lyuk is van rajta, de ha arra jár egy golyó, beleesik.

f. Függvényen belül lehet függvényt definiálni?

Nem. Bár jogos a felvetés, mert pl. Pascalban ez megengedett. Javaban is meg lehet oldani, de ez nem kezdőknek való téma.

g. Prímtulajdonság vizsgálata

Gondoljunk arra a feladatra, hogy el kell dönteni, hogy egy kártyacsomagban van-e piros lap. Végignézzük az összes lapot. Ha az, amit nézünk piros, akkor kész vagyunk, tudjuk, hogy van a pakliban piros lap. Ha nem piros, akkor még nem tudunk semmit, hiszen később lehet még piros. A tanítványok legtöbbje ezt elrontja, és azt írja a programba, hogy ha piros, akkor kész vagyunk, mert van a pakliban piros, ha nem piros, akkor pedig tudjuk, hogy nincs benne, valahogy így:

```
for (i = 0; i < max; i++) {

      if (piros)
            return true;
      else
            return false;

}
```

A prímtulajdonság vizsgálata címet pedig azért kapta, mert ugyanez a helyzet akkor, ha meg kell állapítani egy számról, hogy prím-e. Ha találunk osztót 2 és a szám-1 (vagy a szám fele vagy a szám négyzetgyöke) között, akkor azonnal tudjuk, hogy nem prím. Ha az adott szám nem osztója, akkor bizony tovább kell menni.

3. Az osztályok szépségei

a. Egy objektum metódusának meghívása

… esetén nehezen esik le az aktuális objektum fogalma. Mármint az az objektum, amire a tagfüggvény futása esetén a `this` mutatni fog, pl.

```
Diak feri = new Diak(”Kiss Ferenc”);
feri.kiir();
```

esetén a `kiir()` metódusban „`this`” a „`feri`” által mutatott objektumra mutat.

b. Osztály tagfüggvényének átadni a tagváltozókat is

Az objektum-orientált programozás nem tartozik a gyorsan megérthető részek közé. Az egyik gyakori hiba, hogy – struktúrált programozásból váltva – a tanítvány át akarja adni a tagfüggvénynek a tagváltozókat is, valahogy így:

```
public class Valami {
     private int belsovaltozo;
     public void tagfuggveny(int belsovaltozo, int parameter) {…}
};
```

Természetesen a tagfüggvény hozzáfér a tagváltozóhoz, így erre semmi szükség… sőt, mindenféle objektum-orientált tervezési elvet sért, ha így teszünk.