Miskolci Egyetem
Általános Informatikai Intézeti Tanszék

Java kódolási ajánlások

Összeállította: Ficsor Lajos

Lektorálta: Dr Baksáné dr Varga Erika

Miskolc, 2020

Dokumentum verzió: 0.8

Tartalomjegyzék

1. Bevezetés

Ebben a segédletben olyan szabályokat foglalunk össze, amelyeket a Java nyelv specifikációja nem tartalmaz (de természetesen nem is ellenkezik azzal).

A szabályok betartásának célja a Java forráskód olvashatóbbá tétele, ami megkönnyíti a kód megértését. Miért van erre szükség?

- Egy elkészült kód tesztelése gyakran azzal kezdődik, hogy valaki (aki nem a kódot író programozó) átnézi azt. Ezzel gyakran kiszűrhetők olyan hibák, hogy bár a kód működik, de nem pontosan azt csinálja, amit kellene.
- Segíti a korábban megírt kódrészletek újra felhasználását.
- Segít a kódolás során bizonyos félreértésekből származó szemantikai hibák elkerülésében.
- Segíti az esetleges hibák kijavítását, vagy továbbfejlesztését, amit a gyakorlatban tipikusan nem is annak a programozónak kell elvégeznie, aki eredetileg a kódot írta. De – tapasztalatból mondom – ha régebben (akár néhány hete …) elkészített saját kódot is újra meg kell értenem, ha módosítani akarom.

Tekinthetjük ezeket a szabályokat egyfajta illemszabályoknak is – ha valaki olvassa a kódunkat, és nem látja ezek betartását, könnyen a hozzáértésünket is megkérdőjelezheti. Jelszó: amit nem tiltanak a „törvények” (a Java nyelvi specifikáció) még nem biztos, hogy szokás használni.

A gyakorló és házi feladatok elkészítése során tehát törekedjünk a segédletben leírt szabályok betartására. Egy idő után ez szinte automatikuson menni fog.

Megjegyzés:
Az oktatási anyagokban – főleg az előadás fóliákon – terjedelmi okokbók nem mindig lehet betartani ezeket az előírásokat.

2. Elnevezések

2.1. Általános szabályok

Bár a Java nyelv megengedi az UTF-8 kódtábla használatát az elnevezésekben, csak az angol ábécé betűit és a számjegyeket (az ASCII kódtábla elemeit), illetve a későbbiekben részletezett esetben az _ (aláhúzás) karaktert alkalmazzuk. Ezzel elősegítjük a forráskód hordozhatóságát különböző fejlesztési környezetek között.

Sok karakterkészletben az l (kis L betű) és az 1 (egyes számjegy) könnyen összetéveszthető. Például ennek a szövegnek az éppen használt karakterkészlete is ilyen: „l1”. Ezért ezeket ne használjuk olyan helyen, ahol az félreértésre adhat okot.

Az elnevezések nyelvére nincs megkötés. Ha idegen nyelvet választunk, az lehetőleg az angol legyen, de nem szokatlanok a gyakorlatban a „Hunglis” elnevezések sem (például „isParos” mint metódusnév).

Fontos, és a programszöveg megértéséhez alapvetően hozzájárul az, ha az elnevezések az általuk megnevezett nyelvi elem jelentésére, tartalmára utalnak. Ezt sokszor több szavas, nem túl rövid azonosítókkal tudjuk elérni. Ne féljünk ezek használatától, mert a gépelési munkát általában csak az első előfordulásukkor növelik, a fejlesztőeszközök kódkiegészítő funkciója általában az első néhány karakter begépelése után felajánlja a megfelelő azonosítót beszúrásra, vagy akár automatikusan be is szúrja azt a szövegbe.

2.2. Csomagnevek

A csomagnevek csupa kisbetűsek legyenek akkor is, ha több szóból állnak. Ne használjunk számjegyeket, sem _ (aláhúzás) karaktert.

Például:

```
package oop.elsogyakorlat.elsofeladat
```

Ne használjunk névtelen csomagot, ha egy egyszerű gyakorló feladat akár csak egyetlen osztályt igényel is, azt is tegyük csomagba.

Gyakorló feladatok esetén nem gyakori, de ha egy nagyobb fejlesztésen dolgozunk (főleg ha csapatmunkában), használjuk ki a csomagokat az osztályok csoportosítására. Egy csomagba logikailag összetartozó, esetleg egy fejlesztő által készített osztályok legyenek.

2.3. Osztálynevek

Az osztályokat nagy betűvel kezdődő, majd kisbetűkkel folytatódó elnevezéssel lássuk el. Ha osztálynévnek több szavas kifejezést használunk, a második és további szavak nagy kezdőbetűvel, de folyamatosan írandók. (Ezt az írásmódot hívják angolul „UpperCamelCase”-nek.) Ezt a szabályt kell alkalmazni az interfészek és a felsorolás típusok (enum-ok) elnevezésére is.

Példák:

```
Alkalmazott
ElsoGyakorlatElsoFeladat
```

Az osztálynevek általában főnevek vagy főnévi szerkezetek, hiszen az osztályok valamilyen valóságbeli objektumot modelleznek.

Ha az osztálynév olyan rövidítést tartalmaz, amit csupa nagybetűvel szoktunk írni, azt is nagy kezdőbetűsre kell alakítani.

Például:

```
VideoDvd
HttpOldal
```

2.4. Metódusnevek, nem konstans adattagok és paraméterek

Az osztálynévvel egyező szabályokkal, de **kis kezdőbetűvel** kell az azonosítóikat képezni.

Például:

```
adatBeolvasas(...)
fizetesSzamitas(...)
private int szemelyiJovedelemAdo(...)
```

A metódusok valamilyen feladatot, cselekvést hajtanak végre, tehát logikus elnevezésük ige, vagy igei szerkezet.

Az adattagok neve általában főnévi jellegű.

2.5. Konstansok

Konstansnak tekintjük a `final` minősítésű adattagokat és a felsorolás típus mezőit.

A konstansok azonosítóit csupa nagybetűvel képezzük. Több szavas kifejezés esetén a szavakat _ (aláhúzás) karakterrel választjuk el.

Például:

```
static final int SORHOSSZ=80;
final int MEGENGEDETT_MAXIMALIS_ERTEK=2000;
static final String FEJLEC = ”Első oszlop\tMásodik oszlop”;
public enum TargyRoviditesek {PROGALAP, OOP, OPRENDSZER};
```

2.6. Lokális változók

Elnevezésükben alapvetően az adattagokra vonatkozó szabályok érvényesek, és itt is törekedni kell a „beszélő” nevek használatára.

Mivel azonban a lokális változók hatásköre korlátozottabb (erre törekedni is kell), ezért az azonosítóik képzésére alkalmazott szabályokat is lazábban kezelhetjük. Megengedettek rövidebb azonosítók, mint például `sum, osszeg` stb.

Szokásjog alapján a ciklusváltozókra használhatjuk akár a megszokott egybetűs `i, j, k` azonosítókat is, ha a ciklusváltozónak nincs önálló jelentése, csak a ciklus vezérlésére használjuk. Ilyenkor ezeket a ciklusváltozókat a ciklus fejében deklaráljuk, és ne felejtsük el, hogy a ciklustörzsön kívül nem használhatóak.

Ha egy lokális változó azonosítója megegyezik valamelyik adattagéval, elfedi azt (bár természetesen a `this` kulcsszó segítségével elérhető marad). Mivel azonban ez félreértésekre adhat okot, lehetőleg kerüljük.

2.7. Metódus paraméterek

Mivel ezek a metódus lokális változóiként tekinthetők, az előző pontban leírtak ezekre is érvényesek, azonban mindig „beszélő” azonosítót kell választanunk.

Néhány (később részletezett) esetben az olvashatóságot növeli, ha a paraméter és valamelyik adattag azonosítója megegyezik.

3. A forrásfile-ok felépítése

A Java nyelv szabályai szerint minden `public` osztályt egy külön forrásfile-ban kell elhelyezni. A fájl tartalma az alábbi legyen:

- A készítő nevét tartalmazó JavaDoc komment, ha van. Az Eclipse fejlesztőkörnyezet például generál ilyet.
- `package` utasítás (Nem tartalmazhat sortörést).
- `import` utasítások (Nem tartalmazhatnak sortörést).
- `public` osztálydefiníció (Megelőzheti egy JavaDoc komment).
- `private` osztálydefiníció(k) (ha vannak).

A fenti részeket egy üres sorral válasszuk el.

3.1. Az osztály definíció felépítése

Az osztály definíció értelemszerűen a fejléccel kezdődik. Vele egy sorban, tőle egy helyközzel elválasztva írandó a kezdő kapcsos zárójel. Maga az osztály definíció tipikusan az alábbi szekciókat tartalmazhatja, amelyeket egy üres sorral válasszunk el:

- `public` adattagok és konstansok (**kerülendő!**)
- `private` adattagok
- `static private` adattagok
- `final private` adattagok
- inicializáló blokk(ok)
- konstruktor(ok)
- adatelérő metódusok („setter – getter ” metódusok)
- `public` metódusok
- `private` metódusok
- belső osztály(ok) definíciói (ha vannak)

Ha több konstruktor, adatelérő metódus vagy azonos nevű metódus van, azokat egy csoportba kell rendezni, közöttük semmiféle más elem nem lehet.

3.2. A metódusok felépítése

3.2.1 Sorhossz

Egy sor hossza ne haladja meg a 80 vagy 100 karaktert, hogy minden sor vízszintes görgetés nélkül is olvasható legyen.

3.2.2 Minden utasítás külön sorban

Ez a szabály vonatkozik a lokális változók deklarációira is.

Ha egy utasítás túl hosszú ahhoz, hogy az ajánlott sorhosszba „beleférjen”, az utasítást (ha lehetséges) valamilyen „logikus” helyen célszerű megtörni. A folytatósor az kezdősor alatt plusz egy behúzással kezdődjön.

Sokszor a túl hosszú sort elkerülhetjük, ha előtte metódus hívásokat kiemelünk külön változókba.

3.2.3 Szelekciós és ciklusutasítások írásmódja

Alapszabály, hogy az egyes ágakban mindig legyen kapcsos zárójel pár, akkor is, ha csak egy utasítást tartalmaz, vagy esetleg üres. Ezzel elkerülhetjük azt a hibát, hogy ha egy későbbi módosítás során újabb utasítást szúrunk be, azzal felborítjuk a szerkezetet.

Az egyes ágakban vagy a ciklustörzsben írt utasítások egy behúzással beljebb kezdődjenek.

Példák:

```
if (feltétel) {
      utasítások
}


if (feltétel) {
      utasítások
} else {
      utasítások
}


if (feltétel1) {
      utasítások
} else if (feltétel2) {
      utasítások
} else if (feltétel3) {
      utasítások
} else {
      utasítások
}
```

```
for (ciklusfej) {
      cuklustörzs
}

while (feltétel) {
      cklustörzs
}

do {
      utasítások
} while (feltétel);
```

Kivétel:

Egy `else` ág nélküli `if` utasítás esetén ha az igaz ágban csak egy rövid utasítás szerepel, a kapcsos zárójelek elhagyhatók, és az egészet egy sorba kell írni.

Példa:

```
int i=…;
boolean paros;
if (i % 2 == 0) paros = true;
```

3.3. Konstruktorok és „setter” metódusok

Szokásos kódolási stílus, hogy ha egy konstruktor paraméter vagy egy „setter” metódus egy adott adattag beállítását szolgálja, akkor neve egyezzen az adattagéval. Ilyenkor az adattagra a `this` segítségével hivatkozhatunk. Ha a fejlesztőeszközzel generáltatjuk ezeket, ezt a stílus követő eredményt kapunk.

3.4. Tömbdeklaráció

A tömböt jelző [] zárójelpárt mindig a típusnév után írjuk:

`String[] nevsor;`
(És nem `String nevsor[]` - bár szintaktikailag ez is helyes.)

4. Összefoglaló példa

A fenti szabályok alkalmazását az alábbi egyszerű osztálydefinícióval szemléltetjük.

```java
/**
 * @author ficsor
 *
 */

package osszefoglalopelda;

public class VizsgaZh {
	private String hallgatoNeve;
	private int pontszam;
	private static final int MAXIMALIS_PONTSZAM = 100;

	public VizsgaZh(String hallgatoNeve, int pontszam) {
		super();
		this.hallgatoNeve = hallgatoNeve;
		this.pontszam = pontszam;
	}

	public String getHallgatoNeve() {
		return hallgatoNeve;
	}

	public void setHallgatoNeve(String hallgatoNeve) {
		this.hallgatoNeve = hallgatoNeve;
	}

	public int getPontszam() {
		return pontszam;
	}

	public void setPontszam(int pontszam) {
		this.pontszam = pontszam;
	}

	public static int getMaximalisPontszam() {
		return MAXIMALIS_PONTSZAM;
	}

	@Override
	public String toString() {
		return "VizsgaZh [hallgatoNeve=" + hallgatoNeve + ", pontszam=" + pontszam + "]";
	}

// Eddig ezeket mind generálta az Eclipse

	public boolean megfeleltE() {
		return pontszam > MAXIMALIS_PONTSZAM /2 ? true : false;
	}

	public static int maximalisPontszam(VizsgaZh[] zarthelyik) {
		int max = -1;
		for(VizsgaZh zh : zarthelyik) {
			if (zh.getPontszam() > max) max = zh.getPontszam();
		}
		return max;
	}


/**
* Csak a teszt kedvéért
*/

package osszefoglalopelda;

import java.util.Random;

public class VizsgaZhTeszt {

	public static void main(String[] args) {

		final int letszam = 10;
		VizsgaZh[] megirtZhk = new VizsgaZh[letszam];
		// hallgatók neve Bela1, Bela2 stb, pontszám vélelenszerűen generálva
		Random r = new Random();
```

```
                for (int i=0; i<letszam; i++) {
                        megirtZhk[i] = new VizsgaZh("Bela" + (i+1),
                                        r.nextInt(VizsgaZh.getMaximalisPontszam() + 1));
                }

                // Adatok visszaírása
                String eredmeny;
                for (VizsgaZh zh : megirtZhk) {
                        eredmeny = zh.getPontszam() > VizsgaZh.getMaximalisPontszam() / 2 ?
                                        "Megfelelt" : "Bukta";
                        System.out.println(zh + " - " +  eredmeny);
                }

                // Maximális pontot elérők
                int maxElertPontszam = VizsgaZh.maximalisPontszam(megirtZhk);
                System.out.println("A maximális pontszám:" + maxElertPontszam +
                                " Maximális pontot írtak:");
                for (VizsgaZh zh : megirtZhk) {
                        if (zh.getPontszam() == maxElertPontszam) {
                                System.out.println(zh);
                        }
                }

        }

}
```

5. Egyéb kódolási ajánlások

Ebben a pontban olyan tanácsokat foglalunk össze, amelyek nem formai előírások, hanem inkább a helyes – és biztonságos - kódolási stílust segítik, és sokszor nehezen felderíthető szemantikai hibákat kerülhetünk el segítségükkel.

5.1. Kivételek lekezelése

Ha egy kivételt figyelünk (akár azért mert kötelező, akár a saját választásunk szerint) soha ne hagyjuk üresen a `catch` blokkot, azaz érdemben kezeljük le a kivételt. Ha ez a program adott pontján nem lehetséges, használjuk a kivétel „továbbdobásának” lehetőségét. Ha a kivételes eseményt a program egyáltalán nem tudja kezelni, a futás leállítása előtt kapjon a felhasználó megfelelően informatív hibaüzenetet. A programok gyakran készítenek naplófile-t a működésük során, ilyenkor ide is megfelelő bejegyzést kell készíteni a futás leállása előtt.

5.2. Hivatkozás a statikus elemekre

A statikus (osztály szintű) elemekre mindig az osztály nevével hivatkozzunk. Az első objektum példány létrehozása előtt nem is tehetünk mást. Ha már van objektum példány, akkor szintaktikailag lehetséges azzal hivatkozni az osztályszintű elemre, de nem hasznos. (A legtöbb fejlesztőeszköz warning-gal honorálja az ilyen kísérletet.)

5.3. A finalize metódus használata

A finalize metódus pontos végrehajtási ideje nem ismert, hiszen azt a szemétgyűjtő mechanizmus hívja meg, amikor úgy dönt, hogy megszünteti az adott objektumot. Ezért használata nehezen felderíthető hibákhoz vezethet.

Ha az objektum feleslegessé válása esetén szükséges bizonyos műveleteket elvégezni (például fájlt lezárni, adatbázis kapcsolatot megszüntetni stb) akkor használjunk általunk megírt, és a szükséges helyen és időben meghívott metódust. Ennek neve lehet például `destroy`.

5.4. Ne példányosítsunk fölöslegesen objektumokat

A példányosítás erőforrás igényes művelet lehet, ezért fölösleges példányok létrehozása az erőforrások pazarlásával jár. Ez ugyan egy gyakorló feladat szintű programban nem okoz problémát, de jó,ha gondolunk rá. Például egy ciklustörzsben példányosított objektum a ciklus minden lefutásakor létrejön, és a ciklustörzs végén megszűnik. Ha szemantikailag lehetséges, az ilyen példányosítást emeljük ki a ciklus elé.

5.5. Használjuk az `@Override` annotációt

Ha egy örökölt metódust akarunk felüldefiniálni, a definíció elé mindig tegyük ki az @Override annotációt („megjegyzést”). Ezzel elkerülhetjük, hogy véletlenül felüldefiniálás helyett függvény overloadingot hozzunk létre.

Például nem véletlenül néz így ki a generált toString definíciója:

```
@Override
public String toString() { … }
```

Ha megpróbáljuk az annotáció nélkül másképp definiálni a toString metódust (nem ajánlott, csak a példa kedvéért):

```
public String toString(String header) { … }
```

akkor nem kapunk semmilyen jelzést, de – szándékunk ellenére – a string konverzióhoz nem ez a metódus, hanem az örökölt toString fog meghívódni.

Az

```
@Override
public String toString(String header) { … }
```

formátum esetén hibajelzést kapunk, mert ez nem szabályos felüldefiniálás.

6. Felhasznált források

1. Google Java Style Guide:

https://google.github.io/styleguide/javaguide.html

(Link utoljára ellenőrizve 2020. 02. 09.)

2. Joshua Bloch: Hatékony Java

Kiskapu, Budapest, 2008.