Stilul folosit la scrierea programelor

Cele ce urmează constituie o versiune prescurtată a documentului Coding Style Draft de Doug Lea.

1. Structura şi documentarea

1.1. Pachetele

Creaţi un pachet java nou pentru fiecare proiect auto-conţinut sau fiecare grup cu funcţionalitate înrudită. Creaţi şi folosiţi directoare corespunzător convenţiilor pentru pachetele Java.
Luaţi în considerare scrierea unui fişier `index.html` în fiecare director, fişier care să descrie pe scurt scopul si structura pachetului.

1.2. Fişiere cu programe

Puneţi fiecare clasă într-un fişier separat. Acest lucru se aplică chiar şi la clasele care nu sunt publice (pentru care compilatorul Java permite amplasarea în acelaşi fişier cu clasa principala care le foloseşte) cu excepţia cazului în care se folosesc o singura dată, unde despre clasa ne-publică nu se poate concepe a fi folosită în afara contextului său.

Începeţi scrierea fiecărui fişier cu un comentariu care să cuprindă:

- Numele fişierului şi/sau informaţie de identificare inclusiv, dacă este aplicabil, informaţii despre copyright.
- Un table istoric care să cuprindă datele, autorii şi rezumatele modificărilor.
- Dacă un fişier conţine mai mult de o clasă, o listă a claselor împreună cu o descriere foarte pe scurt a fiecăreia.
- Dacă fişierul introduce un punct de intrare principal pentru un pachet, descrieţi pe scurt motivaţia construcţiei pachetului.

Puneţi imediat după fiecare antet:

- Numele pachetului
- Lista de importuri.

Exemplu:

```
/*
   Fisier: Exemplu.java
   Data      Autor      Modificari
   Sep 1  95 Doug Lea    Creat
   Sep 13 95 Doug Lea    Adaugat conventii pentru documente noi
*/

package demo;
import java.util.NoSuchElementException;
```

1.2.1. Clase şi interfeţe

Scrieţi toate comentariile /** ... **/ **folosind convenţiile `javadoc`**. (Chiar dacă javadoc nu cere, încheiaţi fiecare comentariu /** cu **/ pentru a-l face mai uşor de citit şi verificat.)

Prefaţaţi fiecare clasă cu un comentariu / ... **/** care să descrie scopul clasei, invarianţii garantaţi, instrucţiuni de folosire şi/sau exemple de folosire. De asemenea includeţi orice elemente pentru aducere aminte sau elemente care să prevină reclamaţii ulterioare (disclaimers) despre îmbunătăţiri solicitate sau dorite. Folosiţi formatul HTML cu etichete suplimentare:
`@author numele autorului`

```
@version numărul de versiune al clasei
@see sir de caractere (vezi…)
@see URL
@see numeclasa#numemetoda

Exemplu:

/**
 * O clasa care reprezintă o fereastra pe ecran.
 * Spre exemplu:
 * <pre>
 * Window win = new Window(parinte);
 * win.show();
 * </pre>
 *
 * @see awt.BaseWindow
 * @see awt.Button
 * @version 1.2 31 Jan 1995
 * @author Clovnul Bozo
**/
class Window extends BaseWindow {
...
}
```

1.2.2. Variabile clasă

Folosiţi convenţiile javadoc pentru a descrie natura, scopul, constrângerile şi modul de utilizare al variabilelor instanţă şi al variabilelor statice. Folosiţi formatul HTML, cu etichetele suplimentare:

```
@see sir
@see URL
@see numeclasa#numemetoda
```

Exemplu:

```
/**
 * Numarul curent de elemente
 * trebuie sa fie ne-negativ si mai mic sau egal cu capacitatea.
**/
protected int count;
```

1.2.3. Metode

Folosiţi convenţiile javadoc pentru a descrie natura, scopul, precondiţiile, efectele, note despre algoritmi, instrucţiuni de folosire, elemente pentru aducere aminte etc. Folosiţi formatul HTML cu etichete suplimentare:

```
@param numeParametru descriere.
@return descrierea valorii returnate
@exception numeExceptie descriere
@see sir
@see URL
@see numeClasa#numeMetoda
```

Fiţi cât mai precişi (rezonabil) în documentarea efectelor. Iată câteva convenţii şi practici pentru specificaţii semi-formale:

@return conditie: (conditie)

Descrie postcondiţiile şi efectele adevărate la revenirea dintr-o metodă.

@exception numeExceptie IF (conditie)

Indică ce condiţii sunt necesare pentru aruncarea fiecărei excepţii. Include condiţiile sub care se pot arunca excepţii ne-comune, neverificate (nedeclarate).

@param numeParametru WHERE (conditie)

Indică restricţii asupra valorilor argumentelor. Alternativ, dacă aşa este implementat, listaţi restricţiile împreună cu excepţiile care rezultă; spre exemplu IllegalArgumentException. În particular, indicaţi dacă se permite ca argumentele referinţa să fie null.

WHEN (conditie)

Indică ce acţiuni folosesc wait gardate până când ţine condiţia.

RELY (conditie)

Descrie supoziţii despre contextul de execuţie. În particular, a te baza pe faptul că alte acţiuni din alte fire de lucru (threads) să se termine sau sa ofere notificări.

GENERATE T

Pentru a descrie entităţi noi (pentru exemplul main, Threads) construite în cursul metodei.

ATOMIC

Indică dacă se garantează că acţiunile nu se interferează cu alte acţiuni din alte fire de lucru (normal fiind implementate via metode sau blocuri synchronized).

PREV(obj)

Se referă la starea unui obiect la începutul unei metode.

OUT(message)

Descrie mesaje (inclusiv notificări cum este notifyAll) trimise altor obiecte ca aspecte necesare ale funcţionalităţii sau la care se face referire în descrierea efectelor altor metode.

foreach (int i in lo .. hi) predicat

Înseamnă că predicatul ţine pentru fiecare valoare a lui i.

foreach (Obiect x in e) predicat

Înseamnă că predicatul ţine pentru fiecare element al unei colecţii sau la unei enumerări.

foreach (Tip x) predicat

Înseamnă că predicatul ţine pentru fiecare instanţă a lui Tip.

-->

Înseamnă ‚implică'.

unique

Înseamnă că valoarea este diferită de oricare alta. Spre exemplu, o variabilă instanţă unică care se referă întotdeauna la un obiect care nu este referit de nici un alt obiect.

fixed

Înseamnă că valoarea nu mai este schimbată după ce a fost iniţializată.

EQUIVALENT to { segment de cod }

Documentează metode de convenienţă sau specializate care pot fi definite în termenii a câteva operaţii folosind alte metode.

Exemplu:

```
/**
 * Insereaza un element la inceputul secventei
 *
 * @param element elementul de adaugat
 * @return conditie:
 * <PRE>
 *  size() == PREV(this).size()+1 &&
 *  at(0).equals(element) &&
 *  foreach (int i in 1..size()-1) at(i).equals(PREV(this).at(i-1))
 * </PRE>
**/

public void  addFirst(Object element);
```

1.2.4. Declaraţii locale, instrucţiuni şi expresii

Folosiţi comentarii /* ... */ pentru a descrie detaliile algoritmilor, note şi documentaţie legată de ceva care se întinde pe mai mult de câteva instrucţiuni.

Exemplu:

```
/*
 * Strategie:
 *    1. Gaseste nodul
 *    2. Cloneaza-l
 *    3. Cere lui inserter sa adauge clona
 *    4. Daca operatia reuseste, sterge nodul
 */
```

Folosiţi comentarii // pentru a clarifica porţiuni de cod care nu sunt evidente. Dar nu vă obosiţi să adăugaţi asemenea comentarii la ceva evident; în loc de asta, încercaţi să faceţi codul să fie evident!

Exemplu:

```
int index = -1; // -1 serveste ca fanion insemnand ca indexul nu este valid
```

Sau, adesea mai bine:

```
static final int INVALID= -1;
int index = INVALID;
```

Folosiţi orice set consistent pentru aranjarea sursei, inclusiv:

- Numărul de spatii la indentare.
- Amplasarea acoladei stânga ("{") la sfârşitul liniei sau la începutul liniei următoare.
- Lungimea maximă a unei linii.
- Indentarea suplimentare pentru divizarea liniilor lungi.
- Declararea tuturor variabilelor clasei într-un singur loc (prin convenţia obişnuită, la începutul clasei).

1.2.5. Convenţii de nume

Pachete: litere mici.
Luaţi în considerare convenţiile recomandate de denumire pe baza numelui de domeniu descrise în Java Language Specification, pagina 107 ca prefixe. (Spre exemplu, edu.oswego.cs.dl.)

Fişiere
Compilatorul Java impune convenţia ca numele de fişiere să aibă acelaşi nume de bază ca şi clasa publică pe care o definesc.

Clasă: `CuLitereMariCuvinteleInterneSiEleCULitereMari`

Clasă excepţie: `NumeleClaseiSeTerminaInException`.

Interfaţă. Atunci când e nevoie să se distingă de clase denumite asemănător: `NumeInterfataSeterminaInIfc`.

Clasă. Atunci când trebuie distinsă de interfeţe numite similar: `NumeClasaSeTerminaInImpl` sau `NumeClasaSeTerminaInObject`
Constante (finale): `LITERE_MARI_CU_SUBLINIERI`

Variabile: `primulCuvintLiereMiciCelelalteCuLiteraMare`

Metode: `primulCuvintLiereMiciCelelalteCuLiteraMare` ()

Metoda de fabricare pentru obiecte de tipul X: newX

Metodă de convertire care returnează obiecte de tipul X: toX

Metodă care raportează un atribut x de tip X: X getX().

Metodă care modifică un atribut x de tip X: void setX(X value).

2. Recomandări

1. Reduceţi la minimum formele de import *. Fiţi precişi referitor la ce importaţi. Verificaţi că toate importurile declarate sunt de fapt folosite.

2. Atunci când are sens, consideraţi scrierea unui main pentru clasa principala din fiecare fişier program. Metoda main ar trebui sa furnizeze un test simplu sau un demo.

3. Pentru aplicaţiile de sine stătătoare, clasa cu main ar trebui separată de cele care conţin clase obişnuite.

4. Luaţi în considerare scrierea de fişiere şablon pentru cele mai comune feluri de fişiere pe care le creaţi: applet-uri, clase de bibliotecă, clase de aplicaţie.

5. Dacă va gândiţi că altcineva va implementa funcţionalitatea clasei în mod diferit, definiţi o interfaţă, nu o clasa abstractă. În general, folosiţi clasele abstracte doar atunci când ele sunt "parţial abstracte", adică ele implementează o anumita funcţionalitate care trebuie partajată peste subclase.

6. Gândiţi-vă dacă vreo clasă trebuie să implementeze Cloneable şi/sau Serializable.

7. Declaraţi o clasă ca `final` doar dacă este o subclasă sau o implementare a unei clase sau a unei interfeţe care îşi declara toate metodele sale nespecifice implementării. (Similar pentru metode declarate `final`).

8. Nu declaraţi niciodată variabilele instanţă ca `public`.

9. Reduceţi la minimum baza pe care o puneţi pe iniţializările implicite ale variabilelor instanţă (cum este faptul că variabilele referinţă sunt iniţializate la `null`).

10. Reduceţi la minimum numărul de statice (cu excepţia constantelor `static final`).
    Motivaţie: Variabilele statice se comporta ca variabilele globale în limbajele non-OO. Ele fac metodele mai dependente de context, ascund efectele laterale posibile, uneori prezintă probleme legate de accesul sincronizat şi sunt sursa construcţiilor fragile, ne-extensibile. De asemenea nici variabilele statice nici metodele statice nu pot fi suprascrise în vreun fel util în subclase.

11. În general, preferaţi `long` lui `int`, şi `double` la `float`. Dar, folosiţi `int` pentru compatibilitate cu construcţiile şi clasele Java standard (ca un exemplu major, consideraţi indexarea tablourilor şi toate cele implicate de aceasta, spre exemplu dimensiunea maximă a tablourilor etc.).

12. Folosiţi `final` şi/sau convenţii de comentare pentru a indica dacă variabilele instanţă cărora nu li se schimbă valorile după construire au fost menite a fi constante (imutabile) pe durata de viaţă a obiectului (în raport cu cele care s-a întâmplat să nu le fie asignată vreo valoare într-o clasă, dar cărora li se atribuie valori într-o subclasă).

13. În general preferaţi `protected` lui `private`.
    Motivaţie: dacă nu aveţi cumva vreun motiv bun pentru a sigila o anumita strategie de folosire a unei variabile sau metode, s-ar putea la fel de bine să plănuiţi să modificaţi o clasa prin subclasare. Pe de alta parte, aceasta implică aproape întotdeauna un volum de muncă mai mare. A baza codul într-o clasă de bază în jurul metodelor şi variabilelor `protected` e mai greu, de vreme ce trebuie fie să relaxaţi fie să verificaţi supoziţiile despre proprietăţile lor. (Observaţi că în Java, metodele `protected` sunt accesibile din clase neînrudite din acelaşi pachet. Nu prea există vreun motiv de exploatare a acestui gând.)

14. Evitaţi metodele de acces şi actualizare a variabilelor care nu sunt necesare (accesori şi mutatori care nu sunt necesari). Scrieţi metode de tip get/set doar atunci când ele sunt aspecte intrinseci ale funcţionalităţii.

15. Reduceţi la minimum accesul intern la variabilele instanţă în interiorul metodelor. Folosiţi metode de acces şi actualizare `protected` în loc de aceasta (sau uneori pe cele publice dacă tot există).

16. Evitaţi să daţi unei variabile acelaşi nume cu al uneia dintr-o superclasă.

17. Preferaţi sa declaraţi tablourile ca Tip[] numeTablou în loc de Tip numeTablou[].

18. Asiguraţi-vă ca staticele ne-private au valori care au sens chiar dacă nu se creează vreodată instanţe. (În mod asemănător asiguraţi-vă ca metodele statice pot fi executate cu sens.) Folosiţi iniţializatori statici (static { ... }) dacă este necesar.

19. Scrieţi metode care fac doar "un singur lucru". În particular, separaţi metodele care schimba starea obiectului de cele care doar o consultă. Ca exemplu clasic într-o Stiva, preferaţi să aveţi două metode Object top() şi void removeTop() faţa de o singura metodă Object pop() care face ambele lucruri.

20. Definiţi tipul returnat ca `void` cu excepţia cazului în care metodele returnează rezultate care nu sunt (uşor) accesibile în alt fel. (adică, nu scrieţi vreodată "`return this`").

21. Evitaţi supraîncărcarea metodelor pe tipul de argument. (Supraîncărcarea bazată pe numărul de argumente (aritate) este OK, cum este în cazul când avem o versiune cu un argument şi o alta cu două). Dacă este nevoie să specializaţi comportamentul potrivit clasei unui argument, consideraţi în loc de asta alegerea unui tip general pentru tipul nominal al argumentului (adesea Object) şi folosirea expresiilor condiţionale pentru verificarea tipului de instanţă cu `instanceof`. Alternativele includ tehnici cum sunt dubla dispecerizare, sau adesea cea mai bună, reformularea metodelor (şi/sau reformularea argumentelor metodelor) pentru a elimina dependenţa de tipul exact al argumentului.

22. Folosiţi metoda `equals` în locul operatorului == la compararea obiectelor. În particular, nu folosiţi == pentru a compara String-uri.

23. Declaraţi o variabilă locală doar în locul din sursă unde ştiţi care va fi valoarea ei iniţială.

24. Declaraţi şi iniţializaţi o variabilă locală nouă în loc să refolosiţi (reasignaţi) una existentă şi a cărei valoare se întâmplă să nu mai fie folosită în acel punct din program.

25. Asignaţi null la orice variabilă referinţă care nu mai este folosită. (Aceasta se referă în special la elementele de tablouri.) Motivul: permite colectarea automată a memoriei nefolosite.

26. Evitaţi asignările ("=") în interiorul condiţiilor din `if` şi `while`.

27. Documentaţi cazurile în care valoarea returnată de o metodă este ignorată.

28. Asiguraţi că există undeva un `catch` pentru toate excepţiile neverificate care pot fi tratate. Motivul: Java vă permite să nu vă deranjaţi cu declararea unora dintre excepţiile comune şi uşor de tratat, cum este `java.util.NoSuchElementException`. Declaraţi-le şi interceptaţi-le în orice caz.

29. Încuibăriţi specificări de tip (cast) în expresiile condiţionale. Spre exemplu:

```
C cx = null;
if (x instanceof C) cx = (C)x;
else actiuneEvaziva();
```