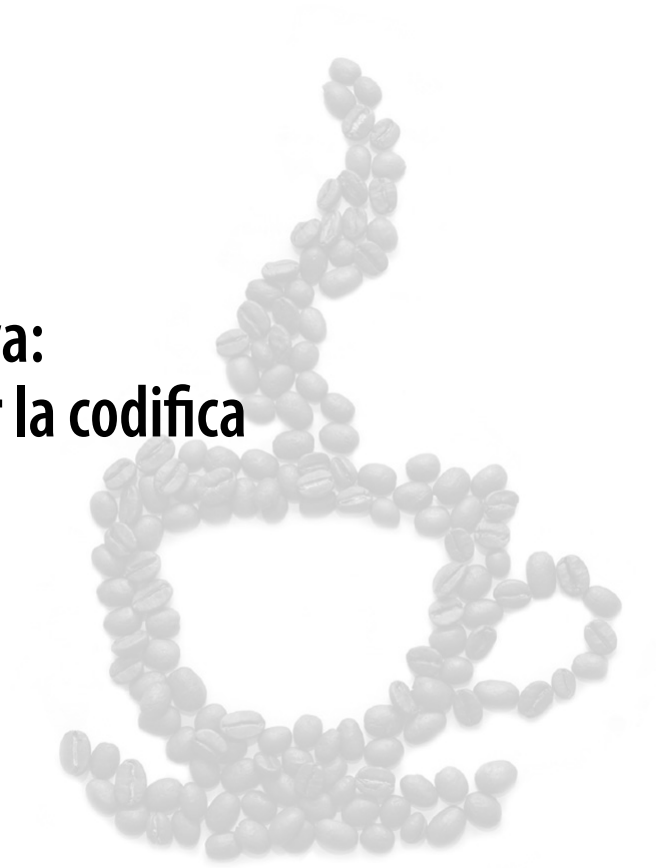
E

Linguaggio Java: linee guida per la codifica



Introduzione

Questa guida allo stile di codifica è una versione semplificata di un'altra che è stata usata con buon successo sia nella pratica industriale sia in corsi universitari.

Una guida dello stile di codifica è un insieme di requisiti obbligatori in materia di disposizione e di impostazione del codice. Uno stile uniforme agevola la lettura del codice scritto dai vostri docenti e dai vostri compagni di corso: avrete modo di apprezzarlo soprattutto quando farete un progetto di gruppo. Chi dovrà valutare i vostri programmi potrà, poi, identificare più rapidamente i punti chiave dei vostri elaborati.

Una guida dello stile vi rende più produttivi come programmatori perché *riduce le scelte inutili*: se non dovete prendere continuamente decisioni su questioni banali, potete dedicare le vostre energie alla soluzione dei problemi veri.

In queste linee guida, svariati costrutti sintattici sono espressamente banditi. Questo non vuol dire che i programmatori che se ne servono siano malvagi o incompetenti: significa, però, che questi costrutti non sono essenziali e si possono esprimere altrettanto bene o persino meglio con altri costrutti del linguaggio.

Se avete già esperienza di programmazione, in Java o in un altro linguaggio, forse vi sentirete a disagio nel rinunciare a certe abitudini alle quali siete affezionati. Tuttavia, mettere da parte le preferenze personali nelle questioni di minore importanza e accettare un compromesso a vantaggio del vostro gruppo è un segno di professionalità.

Queste linee guida sono necessariamente piuttosto noiose; fanno inoltre riferimento a funzionalità che potreste non aver ancora studiato. Ecco i punti più salienti:

- Le tabulazioni sono impostate ogni tre spazi.
- I nomi di metodi e di variabili sono in caratteri minuscoli, eventualmente con qualche carattere maiuscolo in posizioni intermedie.
- I nomi delle classi iniziano con una lettera maiuscola.
- I nomi delle costanti sono tutte in maiuscolo, eventualmente con qualche carattere di sottolineatura all'interno.
- Dopo le parole riservate e intorno agli operatori binari c'è un carattere di spaziatura.
- Le parentesi graffe devono allinearsi orizzontalmente o verticalmente.
- Non si possono utilizzare numeri magici.
- Occorre scrivere un commento per ogni metodo, eccetto il metodo `main` e i metodi sovrascritti.
- Non si possono usare più di 30 righe di codice per ciascun metodo.
- Non sono consentiti enunciati `continue` né `break`.
- Tutte le variabili che non siano `final` devono essere private.

File sorgenti

Ogni programma Java è un insieme di uno o più file sorgenti, che vanno compilati per ottenere il programma eseguibile. Organizzate il materiale di ciascun file nel modo seguente:

- enunciato `package`, se serve
- enunciati `import`
- un commento che spieghi lo scopo del file
- una classe `public`
- altre classi, se servono

Il commento che illustra lo scopo del file deve essere nel formato che viene riconosciuto dal programma `javadoc`. Iniziate con `/**` e usate i marcatori `@author` e `@version`, come in questo esempio:

```
/**
   COPYRIGHT (c) 2010 Harry Morgan. Tutti i diritti sono riservati.
   Classi per manipolare oggetti grafici.
   Soluzione dell'esercizio n. 3.
   @author Harry Morgan
   @version 1.01, 15 febbraio 2010
*/
```

Classi

Ogni classe deve essere preceduta da un commento che ne spiega lo scopo. Elencate dapprima tutte le caratteristiche pubbliche, poi tutte quelle private. All'interno di ciascuna sezione, pubblica e privata, usate questo ordine:

1. variabili di esemplare
2. variabili statiche

3. costruttori
4. metodi di esemplare
3. metodi statici
6. classi interne

Lasciate una riga vuota dopo ciascun metodo.

Tutte le variabili che non sono `final` devono essere private, anche se le variabili di esemplare di una classe interna privata possono essere pubbliche. Metodi e variabili `final` possono essere pubblici o privati, secondo il caso.

Tutte le caratteristiche devono essere contrassegnate come `public` o `private`: non utilizzate l'attributo `protected` né la visibilità predefinita (vale a dire, la visibilità di pacchetto).

Tutte le volte che è possibile, evitate le variabili statiche (eccetto quelle `final`). Nei rari casi in cui vi servissero variabili statiche, vi è consentita una sola variabile statica per classe.

Metodi

Ciascun metodo (tranne il metodo `main`) inizia con un commento in formato `javadoc`:

```
/**
   Converte date del calendario attuale in date del calendario giuliano.
   Nota: questo algoritmo è tratto da Press et al; Numerical Recipes in C,
   2nd ed., Cambridge University Press, 1992.
   @param day giorno della data da convertire
   @param month mese della data da convertire
   @param year anno della data da convertire
   @return il numero del giorno del calendario giuliano che inizia
           a mezzogiorno della data di calendario indicata.
*/
public static int getJulianDayNumber(int day, int month, int year)
{
   ...
}
```

I nomi dei parametri devono essere esplicativi, in particolar modo se si tratta di numeri interi o di valori booleani:

```
public Employee remove(int d, double s) // che significato hanno?

public Employee remove(int department, double severancePay) // più chiaro
```

Ciascun metodo deve svilupparsi, al massimo, su 30 righe di codice. L'intestazione del metodo, i commenti, le righe vuote e le righe che contengono soltanto parentesi graffe non sono comprese in questo conteggio. Questa regola vi costringe a spezzare elaborazioni complesse in più metodi.

Variabili e costanti

Non definite tutte le variabili all'inizio di un blocco:

```
{
   double xold; // non fatelo
   double xnew;
   boolean done;
   ...
}
```

Definite ciascuna variabile appena prima che venga usata per la prima volta:

```
{
   ...
   double xold = Integer.parseInt(input);
   boolean done = false;
   while (!done)
   {
      double xnew = (xold + a / xold) / 2;
      ...
   }
   ...
}
```

Non definite due variabili sulla stessa riga:

```
int dimes = 0, nickels = 0;  // non fatelo
```

Usate, invece, due definizioni separate:

```
int dimes = 0; // così va bene
int nickels = 0;
```

In Java, le costanti devono essere definite con la parola riservata `final`. Se la costante viene utilizzata da più metodi, dichiaratela come `static final`; è buona norma definire private le variabili `static final` se nessun'altra classe è interessata a loro.

Non utilizzate *numeri magici*! Un numero magico è una costante numerica che viene inserita nel codice, senza usare una definizione di costante. Qualsiasi numero, a eccezione di –1, 0, 1 e 2, è considerato magico:

```
if (p.getX() < 300) // non fatelo
```

Utilizzate, invece, variabili `final`:

```
final double WINDOW_WIDTH = 300;
. . .
if (p.getX() < WINDOW_WIDTH) // così va bene
```

Persino la più ragionevole costante cosmica cambierà un giorno o l'altro. Pensate che vi siano 365 giorni in un anno? I vostri clienti su Marte saranno molto infastiditi dal vostro sciocco pregiudizio. Definite una costante:

```
public static final int DAYS_PER_YEAR = 365;
```

così potrete produrre agevolmente una versione marziana del vostro programma senza dover stanare nel vostro codice tutti i numeri 365, 364, 366, 367, e così via.

Quando dichiarate variabili di tipo array, inserite le parentesi quadre dopo il nome del tipo, non dopo il nome della variabile.

```
int[] values; // così va bene
int values[]; // brutto retaggio del linguaggio C
```

Quando usate contenitori, non usate tipi "grezzi" ma tipi parametrici.

```
ArrayList<String> names = new ArrayList<>(); // così va bene
ArrayList names = new ArrayList(); // non va bene
```

Controllo di flusso

L'enunciato `if`

Usate le parentesi graffe per racchiudere i corpi degli enunciati di diramazione e di ciclo, anche se contengono un solo enunciato. Ad esempio, scrivete

```
if (x < 0)
{
   x++;
}
```

e non

```
if (x < 0)
   x++; // non va bene: usate le parentesi graffe
```

L'enunciato `for`

Utilizzate cicli `for` soltanto quando una variabile passa da un valore a un altro con un incremento/decremento costante:

```
for (int i = 0; i < a.length; i++)
{
   System.out.println(a[i]);
}
```

Oppure, ancor meglio, usate un ciclo `for` esteso:

```
for (int e : a)
{
   System.out.println(e);
}
```

Non ricorrete a cicli `for` per creare strani costrutti come questo:

```
for (a = a / 2; count < ITERATIONS; System.out.println(xnew))
   ... // non fatelo
```

Usate, invece, un ciclo `while`: in questo modo, la successione delle istruzioni è molto più chiara.

```
a = a / 2;
while (count < ITERATIONS) // così va bene
{
   ...
   System.out.println(xnew);
}
```

Controllo di flusso non lineare

Vi raccomandiamo di non utilizzare l'enunciato `switch`, perché è facile cadere accidentalmente in un `case` non desiderato. Utilizzate, invece, `if/else` multipli.

Evitate gli enunciati `break` o `continue`: utilizzate, piuttosto, un'ulteriore variabile booleana per controllare il flusso dell'esecuzione.

Eccezioni

Non contrassegnate un metodo con una specifica di eccezione troppo generale:

```
Widget readWidget(Reader in) throws Exception // pessimo
```

Invece, dichiarate specificamente le eccezioni a controllo obbligatorio che il vostro metodo potrebbe sollevare:

```
Widget readWidget(Reader in)
      throws IOException, MalformedWidgetException // ottimo
```

Non mettete a tacere le eccezioni:

```
try
{
   double price = in.readDouble();
}
catch (Exception e)
{ } // pessimo
```

I principianti spesso fanno questo errore per "far contento il compilatore": al contrario, se il metodo non è adatto per gestire l'eccezione, usate semplicemente una clausola `throws` e lasciate che l'eccezione sia gestita da chi invoca il metodo.

Per garantire la chiusura di risorse quando si verifica il lancio di un'eccezione, usate sempre un enunciato "try con risorse", come in questo esempio:

```
try (Scanner in = new Scanner(. . .);
     PrintWriter out = new PrintWriter(. . .))
{
   while (in.hasNextLine())
   {
      out.println(in.nextLine());
   }
}
```

Questioni lessicali

Convenzioni per l'assegnazione dei nomi

Le seguenti regole spiegano quando utilizzare lettere maiuscole e minuscole nei nomi degli identificatori:

- Tutti i nomi delle variabili e dei metodi sono scritti con lettere minuscole, eventualmente con qualche carattere maiuscolo in posizioni intermedie, come in `firstPlayer`.
- Tutti i nomi delle costanti sono scritti con lettere maiuscole, eventualmente con qualche carattere di sottolineatura all'interno, come in `CLOCK_RADIUS`.
- Tutti i nomi delle classi e delle interfacce iniziano con una lettera maiuscola, seguita da lettere minuscole, eventualmente con qualche lettera maiuscola all'interno, come in `BankTeller`.
- Tutti i nomi delle variabili che indicano un tipo generico sono, solitamente, singole lettere maiuscole.

I nomi devono essere ragionevolmente lunghi e descrittivi. Utilizzate `firstPlayer` e non `fp`. *Nn tglt l vcl* (non togliete le vocali!). Le variabili locali di uso comune possono essere brevi (`ch`, `i`) se sono soltanto banali segnaposto per un carattere in ingresso, per il contatore di un ciclo e così via. Ancora, non utilizzate `ctr`, `c`, `cntr`, `cnt`, `c2` per le variabili dei vostri metodi: tali variabili avranno certamente scopi specifici e si può dar loro un nome che li ricordi al lettore (per esempio `current`, `next`, `previous`, `result`, ...). Per i tipi generici, però, si è soliti utilizzare nomi costituiti da una singola lettera, come `T` o `E`.

Rientri e spazi

Impostate nell'editor gli arresti di tabulazione a un valore pari a tre spazi. Questo vuol dire che (molto probabilmente) dovrete modificare quelli predefiniti!

Utilizzate liberamente righe vuote per separare porzioni di un metodo che sono logicamente distinte.

Mettete uno spazio vuoto prima e dopo ogni operatore binario:

```
x1 = (-b - Math.sqrt(b * b - 4 * a * c)) / (2 * a);
      // ottimo

x1=(-b-Math.sqrt(b*b-4*a*c))/(2*a);
      // pessimo
```

Lasciate uno spazio vuoto dopo (e non prima) ogni virgola e ogni punto e virgola. Non lasciate uno spazio prima o dopo una parentesi tonda o quadra all'interno di un'espressione. Lasciate uno spazio attorno alle parentesi di un enunciato `if`, `while`, `for` o `catch`.

```
if (x == 0) { y = 0; }

f(a, b[i]);
```

Ciascuna riga deve stare su 80 colonne. Se dovete spezzare un enunciato, aggiungete due livelli di rientro (cioè sei spazi) per la continuazione:

```
a[n] = ..........................................................
      + ............................;
```

Se possibile, iniziate la riga rientrata con un operatore.

Parentesi graffe

Le parentesi graffe di apertura e di chiusura si devono allineare, orizzontalmente oppure verticalmente:

```
while (i < n) { System.out.println(a[i]); i++; }

while (i < n)
{
   System.out.println(a[i]);
   i++;
}
```

Alcuni programmatori non allineano le parentesi graffe in verticale, ma aprono la parentesi graffa dopo la parola chiave:

```
while (i < n) { // non fatelo
   System.out.println(a[i]);
   i++;
}
```

Facendo così si rende più difficile la verifica della corrispondenza tra le parentesi graffe aperte e chiuse.

Disposizione instabile

Ci sono programmatori che si compiacciono molto di allineare alcune colonne nel loro codice:

```
firstRecord = other.firstRecord;
lastRecord  = other.lastRecord;
cutoff      = other.cutoff;
```

L'aspetto è indubbiamente pulito, ma la disposizione non è stabile in caso di modifiche. Una nuova variabile che abbia un nome più lungo del numero di colonne prestabilite impone di spostare tutte le voci:

```
firstRecord = other.firstRecord;
lastRecord  = other.lastRecord;
cutoff      = other.cutoff;
marginalFudgeFactor = other.marginalFudgeFactor;
```

È proprio questo tipo di insidie che vi porta a decidere di usare invece un nome breve, come `mff`, per la nuova variabile. Usate una disposizione più semplice, che renda più facile apportare modifiche al vostro codice.