GUIA DE ESTILO DE PROGRAMAÇÃO EM JAVA

PROF. F. MÁRIO MARTINS

DI/UM

2008-2009

Introdução

Este documento tem por objectivo fundamental estabelecer um conjunto de normas, quase universais, sobre a melhor forma de, em programas JAVA, se representarem sintacticamente as entidades fundamentais da programação, tais como identificadores de elementos chave, como são as classes, as variáveis, os métodos, as constantes, etc., bem como questões topológicas, tais como regras de indentação, que na maioria das linguagens têm um papel apenas estético e de auxílio à legibilidade, mas que noutras (cf. Python e Haskell, por exemplo) têm valor semântico, pois a sua aplicação evita o uso de certos identificadores de blocos (cf. begin, end, {, e outros).

1.- Identificação dos elementos de programação.

1.1.- Identificação deve ter como objectivo um significado óbvio.

Devem ser usados nomes significativos para os identificadores de todos os elementos de programação da linguagem. Em JAVA, temos que identificar constantes, variáveis, classes, métodos e interfaces.

A língua a usar na identificação destes elementos é dependente do âmbito do projecto. Das duas únicas discutíveis neste contexto, português e inglês, é usual que academicamente se use o português mas que num contexto empresarial se use o inglês. Mas por vezes há "misturas" cf. `getX()`.

1.2.- As classes deverão ter um identificador iniciado por uma letra maiúscula seguida de minúsculas, cf. `Integer`, `Ponto`, `Turma`, `Aluno`, etc;

1.3.- As classes deverão ter um identificador que é um substantivo, ou nome singular (cf. `Comboio`, `Carro`, `Pessoa`, `Turma`, `Biblioteca`, etc.), mas não `Alunos`, `Pessoas`, `Livros`, etc., dado serem uma abstracção e definição de uma entidade singular do mundo real;

1.4.- Variáveis devem ser sempre escritas em minúsculas, quer sejam de tipo simples (cf. maior, menor, total), ou de tipo referenciado, isto é contenham apontadores para arrays e objectos), cf. `ponto1`, `lista`, `args`, etc. Porém, quando se pretender explicitar melhor o que a variável refere, pode usar-se concatenação de palavras, mas em que a 2ª, 3ª ou outra palavra, é iniciada por uma letra maiúscula, como em: `listaAlunos`, `listaNomes`, `tabPaisCapital`, etc.

1.5.- Constantes devem ser sempre escritas em maiúsculas, usando ou não o carácter _, e declaradas com o modificador **`final`**, cf. `final double MEU_PI, final int DIM = 200, final VEL_SOM = 340`, etc.;

1.6.- Os métodos públicos que permitem a consulta externa do estado interno de um objecto (e que, portanto fazem parte da API da classe) devem ser declarados genericamente como *`tipoRes`* `getNomeDeAtributo()`, (por exemplo `int getX()`, `String getNomeAluno()`, etc.), ainda que o mais importante seja que tal método seja disponibilizado na API para que tal atributo ou variável de instância possa ser consultada do exterior;

1.7.- Os métodos públicos que permitem a alteração externa do estado interno de um objecto (e que, portanto fazem parte da API da classe) devem ser identificados, genericamente, por `void setNomeDeAtributo(tipo novoValor)`, ainda que o mais importante seja que tal método seja disponibilizado na API ;

1.8.- Tal como nos exemplos anteriores, aumenta-se a legibilidade dos nomes dos métodos se nas letras interiores dos seus identificadores se usarem letras maiúsculas sempre que haja a necessidade de fazer a concatenação de abreviaturas de nomes (cf. `numDeAlunos()`, `totDePontos()`, `numNotasMaioresQue(double notaref)`, etc.);

2.- Topologia (“Layout”)

2.1.- Um nível de indentação será entre 2 e 4 espaços, cf. no exemplo

```
public int getContador() {
    return conta;
}
```

2.2.- Linhas em branco devem ser evitadas, excepto as que distinguem o final de um método do início de outro;

2.3.- Entre um qualquer operador binário e os seus operandos deve deixar-se um espaço (tal como para a atribuição), conforme em:

```
out.println(“O “ + nome + “tem “ + idade + “ anos.”);
val = Math.abs(numero);
total += valor;
if(x > y) y++;
```

2.4.- Nas declarações dos métodos e das classes, há dois estilos possíveis para a colocação do símbolo { de início de bloco de definições:

a) Colocado na mesma linha do início da declaração, cf. em:

```
public class Contador {
    // variáveis de instância
    private int x; // coordenada em X
    private int y; // coordenada em Y
    ……
    public int getX() {
        return x;
    }
}
```

b) Colocado sozinho no início da linha seguinte (gasta mais 1 linha !):

```
public class Contador
{
    // variáveis de instância
    private int x;
    private int y;
    //
    ……
    public int getX()
    {
        return x;
    }
}
```

2.5.- Porém, métodos simples e standard podem (e devem) ser monolinha, cf.

```
    public int getX() { return x; }
    public int setX(int val) { x = val; }
```

2.6.- Dentro de todos os outros blocos, o símbolo { é colocado na mesma linha da palavra reservada que define tal bloco, e as instruções devem estar indentadas 1 nível, como por exemplo em:

```
while(valor != -1) {
    total += valor; conta++;
}

if(valor >= 0) {
    totalpos += valor; pos++;
}
else {
    totalneg += valor; neg++;
}

for(int i = 0; i <= n; i++) {
  // instruções
}
```

2.7.- As variáveis de instância devem ser declaradas uma em cada linha, eventualmente usando comentários monolinha para clarificar o seu significado, cf. em:

```
    private int x; // coordenada em X
    private int y; // coordenada em Y
```

3.- Documentação

3.1.- Todas as classes devem possuir um comentário javadoc antes do seu cabeçalho, identificando o seu objectivo, identificando autor ou autores e indicando a sua versão, tal como em:

```
/**
 * Pontos descritos como duas coordenadas reais.
 * @author F. Mário Martins
 * @version 1.2005
 */
```

3.2.- Todos os métodos devem ser comentados usando o formato javadoc `/** … */`.

3.3.- Variáveis de instância e constantes devem ser declaradas uma em cada linha, podendo ser comentadas através de comentários monolinha (cf. `//` ……).

3.4.- Comentários no código não devem comentar o óbvio, pelo que apenas devem ser usados quando o código possa ser difícil de compreender.

4.- Uso da Linguagem - regras

4.1.- Numa classe, a ordem das definições (caso estejam presentes) deverá ser a seguinte: instrução de package, instruções de importação, comentário à classe, cabeçalho da classe, constantes (atributos `final`), variáveis de instância, construtores e métodos de instância.

4.2.- Se a classe possuir constantes, variáveis e métodos de classe, estes devem ser colocados imediatamente a seguir ao cabeçalho da classe.

4.3.- Todas as variáveis de instância devem ser declaradas `private` (encapsulamento).

4.4.- Usar sempre um modificador de acesso explícito, seja ele `public`, `private` ou `protected`. Nunca usar acesso por omissão (que é `package`). Métodos da API da classe são todos os que forem declarados `public`.

4.5.- Não importe bibliotecas inteiras, como em `import java.util.*;`. Importe as classes de que necessita de forma explícita, como em:

```
import java.util.Scanner;
import java.util.ArrayList;
```

4.6.- Inclua sempre um construtor, mesmo que o corpo seja vazio, como em:

```
public Ponto2D() { };
```

4.7.- Construtores devem inicializar todas as variáveis de instância.

4.8.- Crie sempre 3 tipos de construtores: construtor vazio, construtor a partir das partes e construtor de cópia (cria o objecto a partir de um objecto do mesmo tipo).

```
public Ponto2D() {x = 0; y = 0; };
public Ponto2D(int cx, int cy) {x = cx; y = cy; };
public Ponto2D(Ponto2D p) {x = p.getX(); y = p.getY(); };
```

4.9.- Defina sempre o método clone() criando uma cópia do receptor (ou seja this), usando o construtor de cópia dessa classe. Para a classe `Ponto2D` teríamos:

```
Ponto2D clone() { return new Ponto2D(this); }
```

Para uma qualquer outra classe de nome C teríamos:

```
C clone() { return new C(this); }
```

4.10.- Quando criar construtores de subclasses, garanta que a superclasse é sempre chamada, explicitamente escrevendo o código de chamada do superconstrutor, como em:

```
public Ponto3D extends Ponto2D {
   // variáveis de instância
   int z;  // cordenada em Z
   // construtores
 public Ponto3D() {
    super(); z = 0;
 };
```

5.- Uso da Linguagem - Colecções

5.1.- Sempre que pretender percorrer todos os elementos de uma colecção ou de um array sem que seja para os modificar (exº contagem, selecção, etc.), use o iterador for-each, como em:

```
for(int idade : idades) soma += idade;
for(String nome : nomes) listaNomes += nome + "\n";
```

5.2.- Se o algoritmo sobre a colecção for um algoritmo de procura, então crie um `Iterator` a partir da colecção, e para a procura use um normal ciclo `while()`em conjunto com o método `next()` de `Iterator` que lhe devolve automaticamente o próximo elemento a ser testado.

5.3.- Nunca se deve usar o método `clone()` pré-definido em todas as colecções. Este método clone() devolve uma colecção que partilha os objectos da inicial. Assim, para cada colecção deveremos reescrever o código de `clone()` criando um método que faça "deep clone", ou seja, crie uma nova colecção com cópias dos objectos da inicial.

5.4.- Se o algoritmo sobre a colecção for um algoritmo de procura e alteração de certos objectos de uma colecção, então deve criar-se um `Iterator` a partir da colecção, e depois usar um normal ciclo `while()`em conjunto com o método `next()` de `Iterator` que lhe devolve automaticamente o próximo elemento a ser testado e, eventualmente, modificado. A modificação produz-se directamente no objecto original da colecção, porque ele é partilhado no `Iterator`.

5.5.- Qualquer variável de instância de tipo referenciado (ou um seu objecto se for uma colecção) que deva ser passado para o exterior, em resultado de uma consulta, por exemplo, deve ser clonado antes de ser retornado pelo método respectivo. Se assim não for, no exterior alguém passa a ter o seu endereço e pode modificá-lo sem autorização.