Convenções para Codificação em Java

Fábio Nogueira de Lucena[1]

Instituto de Informática
Universidade Federal de Goiás

19 de Outubro de 2009

Resumo. Este texto apresenta convenções para codificação em Java. Não se trata de mais um conjunto de convenções. Ao contrário, aquelas aqui apresentadas são compatíveis, estão em conformidade, com aquelas definidas pela Sun™, apesar de pequenas diferenças, que tornaram a presente versão mais rigorosa. Algumas flexibilidades foram eliminas sem criar uma estrutura rígida e capaz de dificultar o trabalho do programador. Os ambientes integradas de desenvolvimento como o Eclipse(tm) e o NetBeans(tm) são ferramentas que podem ser empregadas para aplicar as convenções aqui presentes em código que não está em conformidade. Houve também uma preocupação com a forma de apresentação do padrão com o propósito de facilitar a compreensão de casos particulares.

Introdução

> *"Qualquer tolo pode escrever código que um computador pode compreender. Bons programadores escrevem código que seres humanos podem compreender".*
> Martin Fowler et al, *Refactoring: Improving the Design of Existing Code*, Addison-Wesley, 1999.

Vários benefícios podem ser atribuídos a um leiaute adequado das construções de uma linguagem de programação. Está além dos interesses deste documento defender estes benefícios. Ao contrário, é assumido que o leitor já reconhece a importância de padronização e, em particular, está interessado na linguagem Java. Você também pode analisar a citação abaixo:

Seguir as convenções aqui fornecidas não afeta velocidade de execução, não afeta o uso de memória e também não afeta outro aspecto visível durante a execução de um programa, mas influencia a manutenção. O leitor que desejar conhecer explicações para as convenções que aqui são fornecidas encontrará extensa discussão no capítulo 31 do livro *Code Complete: A Practical Handbook of Software Construction*, 2nd edition, Steve McConnell, Microsoft Press, 2004. É deste livro que os objetivos de leiaute abaixo são extraídos:

- Representar com precisão a estrutura lógica de código
- Representar com consistência a estrutura lógica de código
- Melhorar a legibilidade de código
- Facilitar modificações em código

Além de se basear nos fundamentos discutidos no livro citado no parágrafo anterior, as convenções aqui estabelecidas baseiam-se naquelas empregadas no artigo J*ava Code Conventions,* disponibilizado em http://java.sun.com/docs/codeconv/ pela Sun™.

[1] Professor adjunto do Instituto de Informática da UFG. E-mail: fabio@inf.ufg.br.

O objetivo é fornecer diretrizes para programadores durante o processo de construção de código em Java sem que "estilos pessoais" ou convenções em amplo uso sejam, respectivamente, ressaltados ou preteridos. Deve ser claro ao leitor a necessidade de padronizar convenções de codificação, caso contrário, estaremos discutindo fé e crenças em uma ou outra proposta. Religião não é, definitivamente, o assunto deste texto. Portanto, não se sinta ferido pelo conteúdo deste texto. Sua equipe pode apresentar uma proposta mais adequada ao seu contexto.

Convém ressaltar que existem diferenças entre o padrão apresentado neste documento e aquele definido pela Sun™. As convenções aqui fornecidas são mais rigorosas. Opções existentes naquelas convenções, em alguns casos, desapareceram. A forma de apresentação das convenções também foi alterada com o propósito de facilitar a compreensão. O resultado final, contudo, é compatível com aquelas convenções definidas pela Sun™.

NOTA. Usuários do ambiente integrado de desenvolvimento Eclipse são "privilegiados", pois o comando CTRL-SHIFT-F (Windows) ou ESC-CTRL-F (Linux) reformata o código em edição conforme o padrão de codificação da Sun™.

Nomes de arquivos

O programador em Java faz uso de vários arquivos. Há dois tipos principais cujos sufixos bem como as regras para nomeá-los são fornecidos abaixo.

Sufixos `.java` e `.class`

Quando se desenvolve software Java é feito uso de dois sufixos para arquivos: `.java` e `.class`. O primeiro deles designa arquivo contendo código escrito em Java, enquanto o segundo é empregado para arquivos contendo código escrito em *bytecodes*. Não é permitido empregar código em Java com sufixo diferente de `.java`.

Tipo	Sufixo
Java	`.java`
bytecodes	`.class`

Convém ressaltar que código em Java não é executável pela Máquina Virtual Java (MVJ), ao contrário de código em *bytecodes*. Ou seja, a classe `UmSimplesExemplo`, contida no arquivo **UmSimplesExemplo.java**, após compilada dá origem ao arquivo **UmSimplesExemplo.class** contendo, em vez de código em Java, código funcionalmente correspondente, mas em *bytecodes*, passíveis de serem executados pela MVJ.

Nomes de arquivos

Em Java, a classe `NomeDaClasse` declarada `public` necessariamente está em um arquivo cujo nome é **NomeDaClasse.java**. Não há exceção para esta regra.

O arquivo que contém a classe `NomeClasse` deve ser chamado de **NomeClasse.java**. Mesmo que a classe não seja declarada `public`. Isto sugere que um arquivo deve conter no máximo uma única classe, conforme é discutido posteriormente. Também sugere que se uma classe possui *inner classes*, então é o nome da classe mais externa que dá origem ao nome do arquivo.

```
/* Arquivo: Teste.java
 * Autor: Zé
 * Copyright (c) 2004
 */

public class Teste {
}
```

A classe `public` está contida em um arquivo cujo nome é definido pelo nome da classe. A regra é a mesma até quando a classe não é `public`, conforme ilustrado abaixo. Casos podem ocorrer onde a existência de mais de uma classe em um único arquivo faz-se conveniente. Embora a

regra seja clara e dê preferências para uma única classe por arquivo, admite-se mais de uma classe quando a situação em questão for acompanhada de um bom motivo.

```
/* Arquivo: TestePackage.java
 * Autor: Zé
 * Copyright (c) 2004
 */

class TestePackage {
}
```

Também há o caso de classes aninhadas (*nested classes*). Classes que são declaradas como membros de outras classes assim como métodos e atributos. Um subconjunto destas classes é denominado de classes internas (*inner classes*). Estas últimas tem a vida de suas instâncias determinadas por instâncias das classes nas quais estão contidas, além de possuir acesso direto às propriedades das classes que as contêm. Este assunto é extenso e está além dos interesses deste documento uma discussão sobre Java. Para as nossas convenções o exemplo abaixo está correto. O nome do arquivo é definido pelo nome da classe mais externa.

```
/* Arquivo: Externa.java
 * Autor: Zé
 * Copyright (c) 2004
 */

class Externa {

    public classe Interna {
    }
}
```

As regras para nomeação de arquivos contendo classes são as mesmas para arquivos contendo interfaces. Ou seja, uma interface declarada `public` empresta seu nome ao nome do arquivo que a contém. Mesmo que não seja declarada pública, conforme o exemplo abaixo, o nome do arquivo deve ser definido pelo nome da única interface ali presente. À semelhança de arquivos contendo classes, tenha um bom motivo para depositar mais de uma interface ou uma combinação de interfaces e classes em um único arquivo.

```
/* Arquivo: CoresBasicas.java
 * Autor: Zé
 * Copyright (c) 2004
 */

interface CoresBasicas {

    int VERMELHO = 1;
    int VERDE = 2;
    int AZUL = 3;
}
```

Arquivo LEIAME.TXT

É comum o emprego do arquivo `README` para conter a apresentação de um diretório ou outras informações consideradas relevantes para determinado software. Empregue o nome `LEIAME.TXT` caso a documentação seja fornecida em português.

Organização de arquivos

Um arquivo consiste em seções que devem estar separadas por linhas em branco e um comentário opcional que identifique cada seção.

Arquivo com mais de 2000 linhas deve ser evitado. Definitivamente tenha um bom motivo para um arquivo deste tamanho. Conforme McConnell, *Code Complete,* Microsoft Press, 1993, rotinas com mais de 500 linhas de código tendem a conter mais erros. Alguns especulam que a partir de 500 linhas a quantidade de erros é proporcional ao tamanho. Também há evidências de que códigos compostos por rotinas relativamente pequenas, com menos de 150 linhas, são menos onerosas para serem corrigidas.

Um exemplo de programa em Java propriamente formatado encontra-se no final deste documento.

Arquivo contendo código em Java

Um arquivo fonte em Java deve conter uma única classe ou interface pública. Quando classes ou interfaces não forem públicas mas estiverem associadas a uma classe pública, então você pode agrupá-las em um `package` correspondente. Fornecê-las no mesmo arquivo, logo após o código da classe pública só deverá ser considerado em último caso. Por exemplo, no código abaixo, a interface `Interface` não é declarada pública, mas é fornecida em arquivo distindo da classe `ClasseImplementaInterface`, declarada pública.

```
/* Arquivo: ClasseImplementaInterface.java
 */
public class ClasseImplementaInterface implements Interface {
}
```

Abaixo segue a interface, em arquivo distinto daquele acima.

```
/* Arquivo: Interface.java
 * Uma simples interface
 */

interface Interface {
}
```

Arquivos de código fonte em Java seguem a seguinte ordenação: primeiro inicia-se com os comentários, depois segue a especificação do pacote (`package`) correspondente, se este existir, e, posteriormente, seguem as sentenças de importação (`import`). Estas são imediatamente seguidas por declarações de classes e interfaces. Conforme parágrafo anterior, a primeira é a classe ou interface pública do arquivo.

Organização de arquivo contendo código em Java
1. Comentário de arquivo
2. Sentença `package` (se for o caso)
3. Sentenças `import` (se for o caso)
4. Declaração de classe ou interface

Tenha um bom motivo para declarar mais de uma interface ou mais de uma classe, ou uma combinação de classes e interfaces, em um mesmo arquivo. Como regra, declare cada classe ou interface em seu próprio arquivo. Se há uma relação significativa entre um conjunto de classes e/ou interfaces, então considere a possibilidade de criar um `package` que as contenha.

Todo arquivo fonte deve possuir um comentário

Todo arquivo fonte deve ser iniciado por um comentário de arquivo. Este comentário deve conter: (a) o nome da classe, (b) informação acerca da versão do software, (c) data e (d) uma declaração sobre direitos autorais (*copyrigth*). Um exemplo típico é fornecido abaixo.

```
/* UmaClasse
 * Versão 1.02 beta
 * Data: 12/2/2035
 * Copyright (c) 2004 Pedro de Software
 */
```

Sentenças de pacote e importação

A primeira linha não comentada da maioria dos códigos em Java é uma sentença de pacote (`package`) seguida por sentenças de importação (`import`). Por exemplo:

```
package java.awt;

import java.awt.pear.CanvasPeer;
```

É bem provável que seu tão desejado jogo de cartas por computador esteja organizado em vários pacotes, um deles seria aquele correspondente à interface gráfica. Sem motivo para complicar você decide que o nome deste pacote deve ser `gui`. Neste pacote é natural a ocorrência de classes como `TelaJogo`, por exemplo. Assuma que esta classe faz uso da classe `Jogada`, contida no pacote `jogo.regras`. Neste cenário, teríamos o seguinte código resultante:

```
package jogo.gui;

import jogo.regras.Jogada;

public class TelaJogo {
}
```

No exemplo acima são empregadas linhas em branco para separar estas sentenças conforme orientações fornecidas adiante neste documento.

Declaração de classes e interfaces

Os itens seguintes descrevem, na ordem em que devem aparecer, as partes de uma declaração de classe ou interface:

1. Comentários de documentação (explanados adiante).
2. Declaração da classe (`class`) ou interface (`interface`).
3. Comentários gerais de implementação pertinentes à classe ou à interface.
4. Variáveis de classe (`static`). As variáveis de classe devem ser declaradas na seguinte ordem: `public`, `protected` e `private`. Ou seja, aquelas `protected` estarão sempre entre aquelas `public` e `private`. Aquelas `public` sempre irão preceder as `private`.
5. Variáveis de instância são declaradas após aquelas estáticas (`static`) e, à semelhança do caso anterior, primeiro aquelas `public`, depois aquelas `protected` e só então as `private`.
6. Construtores. Em Java, construtor possui o mesmo nome da classe e necessariamente não possui tipo de retorno. Portanto, não são nem se confundem com métodos.
7. Métodos. Devem estar agrupados conforme a funcionalidade em vez de escopo ou visibilidade. Ou seja, um método `private` pode ser fornecido entre dois métodos `public`. Métodos `get` e `set` de um determinado atributo não devem estar separados por outro método. Convém ressaltar que nem todo atributo é acompanhado destes dois métodos. Talvez apenas um destes métodos seja fornecido ou até mesmo nenhum deles.

```
/*
 * MinhaClasse.java
 * Versão 1.0
 * 30/02/200
 * Copyrigth © Eu da Silva
 */

package jogo.gui;

/**
 * Forneça alguma informação de valor sobre a classe!
 */
public class TelaPrincipal {
    private static int numInstances = 0;
    public boolean erro = false;

    protected TelaPrincipal() {
    }

    public void getNumInstances() {
        return numInstances;
    }
}
```

Formato de arquivos Java (encoding)

O padrão US-ASCII talvez seja o mais amplamente utilizado no momento. Contudo, para escrever código em Java cujos comentários são escritos em português, com a devida acentuação, este padrão de codificação é inadequado. Java também admite a criação de variáveis com identificadores como `açaí`, embora desaconselhados neste padrão.

Ao fazer uso de um IDE ou editor para escrever código em Java, certifique-se de que o padrão **UTF-8** é empregado. Este deverá ser o argumento fornecido à opção `-encoding` para o compilador `javac`.

Alinhamento

Quatro espaços devem ser usados como unidade de alinhamento. Enquanto cada alinhamento significa mais 4 espaços à direita, cada tabulação corresponde a exatos 8 espaços.

Comprimento de linha

Evite linhas com mais de 80 caracteres. Há dois motivos: legibilidade e processamento. Legibilidade para quem for efetuar mudanças no código. Processamento para ferramentas e janelas de comandos que, muitas vezes, não apresentam bons resultados quando linhas ultrapassam 80 caracteres.

Quando um comentário incluir código, por exemplo, para ilustrar um uso típico de um método, o tamanho da linha deste código não deve ultrapassar 70 caracteres.

Quebra de linha

Embora uma linha deva conter menos de 80 caracteres, nem sempre a sentença em questão possui este número limitado. Nestes casos será necessário quebrar as linhas para que cada parte não ultrapasse os 80 caracteres permitidos.

A quebra de uma linha deve seguir as regras abaixo na ordem em que aparecem:

- Quebre depois de uma vírgula.
- Quebre antes de um operador
- Prefira quebra que envolva sentenças de mais alto nível a sentenças de mais baixo nível.
- Alinhe a nova linha com o inicio da expressão de mesmo nível na linha anterior.
- Se o código resultante da aplicação das regras acima for confuso ou se aproximar da margem direita, então use um alinhamento de 8 espaços.

Chamada de método

Ao realizar uma chamada a método que deve ser quebrada, a nova linha que se forma deve empregar um alinhamento de 8 espaços em relação ao nível imediato de mais alto nível, conforme ilustrado abaixo.

```
algumMetodo(longExpression1, longExpression2, longExpression3,
        longExpression4, longExpression5);

var = outroMetodo(longExpression1,
        segundoMetodoComChamadaQuebrada(longExpression2,
                longExpression3));
```

Declaração de método

A declaração de um método, à semelhança de uma chamada, deve observar o alinhamento de 8 espaços em relação ao início da declaração do método, conforme ilustrado abaixo.

```
// Alinhamento de 8 espaços para declaração de argumentos de método
public umMetodo(int umArg, Object outroArg, String umOutroArg,
        Object maisUmArg) {
    ...
}
```

Expressões aritméticas

Abaixo seguem dois exemplos de como quebrar expressões aritméticas. O primeiro é preferível ao segundo, pois a quebra está fora da sentença entre parênteses e portanto, de mas alto nível.

```
longName1 = longName2 * (longName3 + longName4 - longName5)
            + 4 * longname6;                                    // Correto


longName1 = longName2 * (longName3 + longName4
                         - longName5) + 4 * longname6;          // EVITE!
```

Sentenças if e operador condicional

Em sentenças `if` deve-se usar a regra de 8 espaços para o alinhamento da mesma forma como para a chamada e declaração de métodos, vistos anteriormente. Por exemplo:

```
if ((condição1 && condição2)
    || (condiçao3 && condiçao4)            // EVITE!
    || (condiçao5 && condiçao6)) {         // EVITE!
    aquiVemAlgo();
}

if ((condiçao1 && condiçao2)
        || (condiçao3 && condiçao4)        // Correto
        || (condiçao4 && condiçao5)) {     // 8 espaços
    aquiVemAlgo();
}

if (condiçao1 && condiçao2) || (condiçao3 && condiçao4)
        || (condição5 && condiçao6)  {    // Correto!
    aquiVemAlgo();
}
```

As três formas abaixo são aceitáveis para o operador condicional de Java:

```
Alfa = (umaExpressaoBooleanaLonga) ? beta : gama;

alfa = (umaExpressaoBooleanaLonga) ? Beta
                                   : gama;

alfa = (umaExpressaoBooleanaLonga)
       ? beta
       : gama;
```

Comentários

Programas em Java podem ter dois tipos de comentários: comentários de implementação e comentários de documentação. Comentários de implementação são aqueles encontrados em C++, que são delimitados por `/* ... */` e `//`. Comentários de documentação são exclusivos de Java e são delimitados por `/**...*/`. Comentários de documentação são utilizados para a geração automática de arquivos HTML através do emprego da ferramenta **javadoc.**

Comentários em Java	
Tipo	Forma
Comentário de documentação	`/** comentário de documentação */`
Comentário de implementação	`/* comentário de implementação */`
	`// comentário de implementação até fim de linha`

Comentários de implementação permitem comentar o código propriamente dito, por exemplo, a estratégia específica de implementação adotada. São comentários para serem lidos por quem irá realizar a manutenção neste código.

Comentários de documentação, por outro lado, devem ser empregados para descrever a especificação do código. Estes comentários são dirigidos àqueles que farão uso do código sem, necessariamente, ter acesso ao mesmo.

Comentários devem ser usados para fornecer uma visão geral do código e informações adicionais que não estão disponíveis diretamente no próprio código. Por exemplo, a sentença `lucroLiquido = FATORX * (lucroBruto - despesas)` não precisa de um comentário que informe que do valor em `lucroBruto` é retirado o valor de `despesas` e o resultado multiplicado por `FATORX` antes de ser depositado em `lucroLiquido`. Isto simplesmente não é um comentário, mas a interpretação da sentença em Java e, portanto, dispensável para um programador Java.

Comentários devem conter informações relevantes para a leitura e compreensão do código. Para a sentença comentada no parágrafo anterior, um comentário admissível segue abaixo.

```
/* FATORX deve ser aplicado conforme lei municipal 219 de 2030 */
lucroLiquido = FATORX * (lucroBruto - despesas);
```

Discussões acerca de decisões não triviais ou não óbvias de projeto do código em questão são apropriadas. Atente-se, contudo, para não duplicar informação que está claramente disponível no próprio código. Comentários que apresentam esta redundância facilmente ficam desatualizados. Se um comentário provavelmente ficará desatualizado à medida que o código evoluir, então resista à tentação de criá-lo.

Nota: a freqüência de comentários pode refletir uma baixa qualidade do código. Caso se sinta obrigado, com freqüência, a documentar determinado código, considere a possibilidade de reescrevê-lo de forma que os comentários se tornem desnecessários.

Comentários não devem ser fornecidos em caixas grandes desenhadas com asteriscos ou outros caracteres. Também não devem incluir caracteres especiais como *form-feed* e *backspace*.

Comentários de implementação

Programas podem ter quatro tipos de comentários de implementação: bloco, única linha, final de sentença e final de linha.

Comentários de bloco e aqueles de única linha devem ser precedidos de uma linha em branco.

Comentários em bloco

Comentários em bloco são usados para prover descrições que podem se estender por várias linhas. Um comentário em bloco deve ser necessariamente precedido por uma linha em branco.

```
/*
 * Segue aqui um comentário em bloco.
 */
```

Comentários em blocos podem iniciar com `/*-`, que é reconhecido por **indent** (uma ferramenta conhecida no ambiente UNIX) como o começo de um comentário em bloco cuja formatação deve ser mantida como fornecida. Exemplo:

```
/*-
 * Aqui é um comentário em bloco com algum tipo de
 * formatação especial que será mantida pelo indent
 *
 *    um
 *       dois
 *            três
 */
```

Nota: Se você não usa **indent**, então não tem que empregar `/*-` em seu código nem especular sobre a possibilidade de que alguém poderá utilizar esta ferramenta com o seu código.

Comentários de uma única linha

Pequenos comentários podem aparecer em uma única linha alinhados com o nível de código da sentença que o segue e ao qual está relacionada. Se um comentário não puder ser escrito em uma única linha, deve-se seguir um formato de bloco. Um comentário de um única linha deve ser precedido por uma linha em branco. Exemplo:

```
if (condição) {

    /* Trate a condição */
    ...
}
```

Comentário de final de sentença

São fornecidos na mesma linha mas após as sentenças aos quais fornecem informações. Estes devem estar relativamente afastados para que fique claro o fim da sentença e o início do comentário. Observemos os exemplos abaixo.

```
if (a == 10) {
    return true;           /* caso especial */
} else {
    return fazNovo(a);     /* faz uma nova chamada */
}
```

Comentários de final de linha

O comentário delimitado por `//` pode comentar toda uma linha ou somente parte da linha. Neste último caso quando for o último elemento da linha. Não empregue esta forma de comentário para comentários de múltiplas linhas consecutivas. A exceção é permitida quando se deseja excluir parte do código. As três formas são ilustradas abaixo.

```
if (variável > 1) {

    // faz alguma operação (Esta linha é somente para o comentário).
    ...
} else {
    return false; // não satisfez a condição (parte da linha comentada)
}


// if (x > 1) {

//    /* um comentário */
//     ...
// } else {
//     return false;
// }
```

Observações sobre comentários

- Jamais insira um comentário no interior de uma sentença, por exemplo, `x == /* igual */ 2`, é uma sentença sintaticamente correta em Java. Da perspectiva de comentários, contudo, apresenta duas situações a serem evitadas. Primeiro, não é necessário comentar o que é óbvio, ou melhor, uma interpretação do código. Supostamente todo programador deverá conhecer a linguagem que emprega. Segundo, comentário não deve ser inserido em uma sentença como ilustrado. Coloque o comentário após ou antes da sentença ao qual se refere. Se antes, então deve vir em um linha, precedida de uma linha em branco. Se após, então deve se tratar de um comentário de final de linha.
- Comentários devem ser precedidos por linha em branco como regra geral. Há uma exceção para indicar que a cláusula `break` não é fornecida. Veja sentença `switch`.

Comentários de documentação

Detalhes podem ser obtidos em *How to Write Doc Comments for the Javadoc™ Tool*, que inclui informações sobre *tags* para comentários de documentação como `@return`, `@param` e `@see`, entre outros. O texto acima se encontra disponível no endereço abaixo:

http://java.sun.com/j2se/javadoc/writingdoccomments/

Detalhes sobre a ferramenta **javadoc** encontram-se disponíveis em

http://java.sun.com/j2se/javadoc/

Comentários de documentação descrevem classes, interfaces, construtores, métodos e variáveis. Cada comentário de documentação é fornecido no interior dos delimitadores `/**...*/`. Deve existir um comentário deste para cada classe, interface, ou membro que se deseja comentar**.** Este comentário deve aparecer imediatamente antes da declaração correspondente.

```
/**
 * Comentário de documentação da classe Exemplo.
 */
public class Exemplo {
}
```

Note que no nível acima classes e interfaces não são alinhadas, enquanto seus membros são. A primeira linha do comentário de documentação (`/**`) para classes e interfaces não é alinhada; as linhas subseqüentes dos comentários de documentação cada uma tem um espaço de alinhamento (para alinhar verticalmente os asteriscos). Comentários de documentação de membros, incluindo construtores, estão tabulados com quatro espaços para a primeira linha e cinco espaços para as linhas seguintes**.**

Se você precisa dar uma informação sobre a classe, interface, variável, ou método que não é adequado para o comentário de documentação, use um comentário em bloco ou um comentário de única linha imediatamente depois da declaração. Por exemplos, detalhes sobre a implementação de uma classe deveriam estar dentro de um comentário de bloco que segue a declaração de classe, não no comentário de documentação da classe.

Comentário de documentação não deve ser colocado dentro de um método ou de um bloco de definição de construtor, porque Java associa comentários de documentação com a primeira declaração depois do comentário.

Declarações

Número e forma de declarações por linha

Empregue uma única declaração por linha. Em outras palavras,

```
int level;            // nível do alinhamento
int size;             // tamanho da mesa
```

É preferível à versão abaixo:

```
int level, size;     // EVITE!
```

Também devem ser evitados casos mais sutis, como aquele abaixo, onde o valor de uma variável é inclusive alterado.

```
System.out.println(++x + x + 2); // EVITE!
```

A versão adequada para o código acima é fornecida abaixo.

```
x++;
System.out.println(2 * x + 2);
```

Quando uma declaração se tratar de um *array* coloque os colchetes imediatamente após o tipo, conforme abaixo:

```
int[] inteiros;
int[] maisInteiros;

int v[];   // EVITE!
int [] u;  // EVITE!
```

Inicialização

Inicialize variáveis locais onde elas são declaradas. Exemplo:

```
int contador = 0;
```

Localização

Forneça as declarações de variáveis o mais próximas possíveis dos seus locais de emprego. Em geral, contudo, declarações são fornecidas no começo de blocos, pois as variáveis correspondentes são empregadas em boa parte do bloco. Um bloco é qualquer código fornecido entre `{` e `}`.

```
void myMethod () {
    int int1 = 0;          // começando do bloco myMethod

    if (condição) {
        int int2 = 0;      // começando o bloco if
        ...
    }

    float valor = (float)int1;
    valor = (2 * valor) + 3;
}
```

Os índices de uma sentença `for`, por outro lado, devem ser declarados conforme abaixo:

```
for (int i = 0; i < linhas.length; i++) {

    /* Corpo do bloco do for segue aqui,
     * conforme abaixo.
     */
    x = x + x * i;
}
```

Evite declarações locais que possuem ocultar (eclipsar) declarações de níveis mais altos. Por exemplo, não declare uma variável com um nome já utilizado em um bloco mais externo dentro de um bloco mais interno:

```
int conta;
...
void meuMetodo() {
    if (condição) {
        int conta;        // EVITE!
        // Outras sentenças ...
    }
}
```

Declarações de classes e interfaces

As seguintes regras de formatação de classes e interfaces devem ser seguidas:

- Nenhum espaço deve existir entre um nome de método ou construtor e o parêntese `(` que determina o início da lista de argumentos.
- Abre chave `{` só aparece no término da mesma linha da sentença.
- Fecha chave `}` inicia uma nova linha alinhada no mesmo nível de sua sentença de abertura. Quando se tratar de uma sentença nula, o `}` deve aparecer imediatamente após o `{`.

Das regras acima observe que um `{` nunca é fornecido na mesma coluna do `}` correspondente.

```
abstract class Ponto extends Object {
    int x = 0;
    int y = 0;

    // sem espaço entre o nome do construtor e o parêntese
    Ponto(int x, int y) {
        this.x = x;
        this.y = y;
    }

    // Declaração vazia
    int desenhe() {
    }

    // Método abstrato
    abstract void translade(int dX, int dY);

    ...
}
```

Os métodos devem ser separados por uma linha em branco. Exemplo:

```
...
void umMetodo() {
    ...
}

void outroMetodo() {
    ...
}
```

Sentenças

Sentenças simples

Cada linha deve conter no máximo uma sentença. Exemplo:

```
argv++;         // Correto
argc++;         // Correto

argv++; argc--; // EVITE!
```

Sentenças compostas

Sentenças compostas são sentenças que contêm uma lista de sentenças incluídas entre chaves. As seções seguintes fornecem vários exemplos. Convenções a serem seguidas para sentenças compostas:

- As sentenças inclusas em uma sentença composta devem ter um nível a mais de alinhamento do que a sentença composta imediata.
- O abre chave `{` deve estar no final da linha que começa a sentença composta. O fecha chave `}` correspondente deve começar uma nova linha e deve estar alinhado de acordo com o inicio da sentença composta.
- Chaves devem envolver sentenças únicas, mesmo quando a linguagem permitir a ausência delas sem prejuízo para a semântica desejada.

Sentenças de retorno

Não se deve usar parênteses na sentença de retorno, exceto quando a legibilidade for favorecida por este recurso. Exemplo:

```
return;

return myDysk.size(); // Desaconselhado o emprego de parênteses aqui!

return (tamanhao ? tamanhao : defaulttamanhao);
```

Sentenças if, if-else, if else-if else

A sentença `if-else` apresenta as seguintes variações consideradas válidas. Qualquer outra deve ser evitada. Em todas elas se observa o emprego obrigatório das chaves.

```
if (condição) {
    sentenças;
}

if (condição) {
    sentenças;
} else {
    sentenças;
}

if (condição) {
    sentenças;
} else if (condição) {
    sentenças;
} else {
    sentenças;
}
```

Sentenças `if` devem usar chaves, tanto para identificar o corpo a ser executado caso a condição

seja verdadeira quanto para o corpo correspondente a uma avaliação falsa desta condição.

```
if (condição)       // EVITE!
    declaração;
```

Faça uso da versão abaixo em vez daquela anterior:

```
if (condição) {       // Correto
    declaração;
}
```

Também deve ser evitada a forma abaixo:

```
if (x > 2) {
    x = x + 1;
} else                // EVITE!
    x = x + 2;
```

Ao contrário de estruturas de iteração, que podem conter corpo nulo em alguns casos específicos, não é correto o emprego desta abordagem para as sentenças `if`.

```
if (x > 2) {          // EVITE!
} else {
    x = x + 2;
}
```

Em vez da versão anterior, deve ser empregada aquela abaixo, onde o resultado da avaliação foi invertido para evitar o corpo de sentenças nulo empregado.

```
if (x <= 2) {
    x = x + 2;
}
```

Sentenças for

Uma sentença `for` deve ter a seguinte forma. Observe o emprego de chaves, mesmo que uma única sentença seja declarada no corpo do `for`.

```
for (inicialização; condição; atualização) {
     sentenças;
}
```

Um `for` vazio (na qual todo o trabalho é terminado na inicialização, condição e uma clausula de atualização) deveria ter a seguinte forma:

```
for (inicialização; condição; atualização);
```

Ao usar o operador de vírgula na inicialização ou na cláusula de atualização de uma sentença `for`, evite a complexidade de usar mais de três variáveis. Se precisar, use sentenças separadas antes do `for` (para a cláusula de inicialização) ou no fim do corpo de iteração do `for` (para a cláusula de atualização).

```
int i = 0;
double d = 0.0;
for (int n = 10; n > 0; n--) {
    ...
}
```

Em vez de

```
for (int i = 0, double d = 0.0, int n = 10; n > 0; n--) // EVITE!
```

Sentenças while

O corpo de uma sentença `while` deve ser fornecido entre chaves, conforme ilustrado abaixo.

```
while (condição) {
    sentenças;
}
```

Mesmo que o corpo da sentença contenha uma única sentença, ou até mesmo nenhuma, este deverá ser fornecido entre chaves.

```
while (condição) {
}
```

Sentenças do-while

Uma sentença `do-while` deve ter a seguinte forma. Observe o emprego obrigatório das chaves.

```
do {
    sentenças;
} while (condição);
```

À semelhança do while, mesmo que uma única sentença faça parte do corpo a ser executado de forma iterativa, ou até mesmo nenhuma sentença, devem ser empregadas as chaves.

```
do {
} while (condição);
```

Sentenças switch

Uma sentença `swicth` deve ter o seguinte formato:

```
switch (condição) {
case ABC:
    sentenças;

    /* falls through */

case DEF:
    sentenças;
    break;

default:
    sentenças;
    break;
}
```

Toda cláusula `case` deve estar separada da seguinte ou da sentença `default` por uma única linha em branco. Todas estas devem estar no mesmo alinhamento do `switch`.

Toda vez que uma declaração `case` continua, ou seja, não inclui um `break`, adicione o comentário `/* falss through */` onde a sentença `break` normalmente estaria, conforme o exemplo acima. Observe que esta marca facilita a localização deste tipo de sentença. Como se trata de comentário de linha única, deve ser precedido por linha em branco. Como toda cláusula `case` de ser separada da seguinte por uma linha em branco, após este comentário deve seguir outra linha em branco.

Toda sentença `switch` deve incluir um caso `default`. Este caso deve ser o último da sentença `switch`. O `break` no caso `default` também deve ser empregado.

Sentenças try-catch

Uma sentença de `try-catch` deve ter o seguinte formato:

```
try {
    sentenças;
} catch (ExceptionClass e) {
    sentenças;
}
```

Uma sentença de `try-catch` também pode ser seguida por um `finally`, que executa independente da ocorrência de uma exceção e, mesmo que ocorra e seja tratada por um `catch`, o bloco `finally` é executado.

```
try  {
    sentenças;
} catch (ExceptionClass e) {
    sentenças;
} finally {
    // As sentenças abaixo serão executadas,
    // independente da ocorrência de exceções.
    sentenças;
}
```

Linhas e espaços em branco

Linhas em branco

Linhas em branco, quando empregadas de forma disciplinada, melhoram a legibilidade do código. Sempre deveriam ser usada uma linha em branco nas circunstâncias seguintes:

- Antes de um método.
- Antes de um comentário de bloco ou de um comentário de uma única linha.
- Entre seções lógicas distintas de um método para melhorar a legibilidade.

Espaços em branco

Deveriam ser usados espaços em branco nas seguintes circunstâncias:

- Entre uma palavra chave e um parêntese deve existir um espaço. Exemplo:

  ```
  while (x >= 2) {   // Correto
      ...
  }

  if(x >= 2) {       // EVITE!
      ...
  }
  ```

  Note que um espaço em branco não deve ser usado entre o nome de um método e o parêntese que se abre logo em seguida. Isto ajuda a distinguir palavras chaves de chamadas de método.
- Um espaço em branco deve seguir uma vírgula em listas de argumentos.
- Todos os operadores binários devem ser separados dos seus operandos através de um espaço à esquerda e outro à direita. Há uma exceção: o operador . empregado em chamadas de métodos. Ou seja, `System . out . println("Errado")`, embora sintaticamente correto em Java, deve ser evitado.
- Espaço em branco não deve separar os operadores unários de seus respectivos operandos. Exemplo:

```
a += c + d;
a = (a + b) / (c * d);

while (d++ == s++) {
    n++;
}

System.out.println("O tamanho é " + n + "\n");
```

- As expressões em um laço `for` devem estar separadas por espaços em branco. Exemplo:

```
for (expr1; expr2; expr3)
```

- Moldagens de tipos (*casting*) devem ser seguidas por um espaço em branco. Exemplos:

```
myMethod((byte) aNum, (Objeto) x);
myMethod((int) (cp + 5), ((int) (i + 3)) + 1);
```

Convenções de nomes

Seguir convenções para a criação de identificadores deixa o programa resultante mais compreensível, mais legível. Convenções de nome também podem oferecer informações sobre a função do identificador. Por exemplo, através da observação de um identificador, caso sejam seguidas algumas normas de construção de nomes, o leitor pode facilmente identificar se se trata de uma constante, pacote, ou classe.

Pacotes

O nome de um pacote (`package`) sempre é escrito em letras ASCII e todas minúsculas. Este identificador também deve ser um nome de domínio de alto nível como `edu`, `gov`, `mil`, `net`, `org`, `com` ou outro. Se nenhum destes domínios for de mais alto nível, então deve ser aplicado o código de duas letras correspondente ao país em questão. Por exemplo, aquele correspondente ao Brasil é `br`. Estes códigos de duas letras estão disponíveis no padrão ISO 3166, que pode ser obtido gratuitamente em http://www.iso.ch/.

Componentes subseqüentes, ou seja, pacotes no interior de um pacote de mais alto nível, cujo nome segue a regra apresentada no parágrafo anterior, seguem as convenções de nome da empresa em questão. Tais convenções podem especificar, por exemplo, que os nomes deverão ser utilizados conforme a hierarquia da empresa em divisões, departamentos, projetos, máquinas e até o nome de *login*. Por exemplo, o código criado pelo aluno Fulano, do Instituto de Informática da UFG poderia estar no pacote identificado por `br.ufg.inf.fulano`.

No Instituto de Informática da UFG, contudo, a regra diz claramente que o nome do autor não deve fazer parte de uma hierarquia de pacotes. O de nível mais externo deve ser o nome que identifica o projeto e, no interior deste projeto, os nomes dos pacotes devem sugerir a organização do projeto em questão. Por exemplo, as classes que interagem com o usuário (interface gráfica) de um projeto de um CAD podem estar definidas no pacote `cad.gui`. Outro exemplo, um dos sistemas desenvolvidos pelo Instituto de Informática é identificado por SISPG. Este sistema possui uma camada de persistência relativamente sofisticada, dividida em classes que cuidam da interação direta com SGBDs e classes que realizam o mapeamento objeto/relacional. Para este cenário é possível a existência de pacotes como `sispg.dados.mapeamento` e `sispg.dados.sgbd`.

Abordagens camelCasing e PascalCasing

É comum o emprego das abordagens conhecidas por camelCasing e PascalCasing na definição de identificadores.

Na abordagem camelCasing, conforme o próprio nome ilustra, as palavras que formam um identificador estão unidadas e a primeira letra da primeira delas, ou seja, a primeira letra do identificador, é fornecida em minúsculas, enquanto todas as demais primeiras letras das palavras que formam o identificador são fornecidas em maiúsculas. Todas as letras que não são iniciais de palavras do identificador são fornecidas necessariamente em letras minúsculas. Por exemplo, `umObjeto` e `aqueleOutroObjeto` empregam a abordagem camelCasing.

Na abordagem PascalCasing, todas as letras iniciais das palavras que formam o identificador são fornecidas em maiúsculas. Todas as demais letras do identificador são fornecidas em minúsculas. Por exemplo, `UmaClasse` e `GeraExtrata` são identificadores que seguem a abordagem PascalCasing.

Em Java, a abordagem PascalCasing é empregada para identificar classes e interfaces, enquanto camelCasing para métodos, variáveis e outras.

Classes

Nomes de classe devem ser substantivos. A primeira letra é necessariamente maiúscula. Caso o identificador da classe seja formado por mais de uma palavra, então cada palavra é iniciada por maiúscula. Todas as demais letras do identificador devem ser minúsculas. O nome deve ser simples e ao mesmo tempo sugestivo. Use palavras inteiras. Evite abreviações e acrônimos, exceto quando se tratar de caso bem conhecido como `URL` ou `HTML`, por exemplo. Ou seja, `Livro` e `UmaClasseComMaisDeUmNome` são nomes válidos de classes.

Interfaces

As mesmas regras que se aplicam às classes são aplicadas às interfaces. Ou seja, `CorBasica` e `IntegralDupla` são nomes adequados.

Métodos

- Os nomes de métodos devem ser verbos. Este verbo deve indicar a função do método.
- O identificador deve seguir a abordagem camelCasing, ou seja, a primeira letra deve ser minúscula. Caso mais de uma palavra faça parte do identificador do método, então as palavras seguintes devem ser iniciadas por letras maiúsculas, todas as demais são minúsculas.

```
execute();
executeRapido();
calculeLucroMinimo();
```

Variáveis

Variáveis de instância e de classe, quando formadas por mais de uma palavra, tem a primeira letra minúscula. Todas as demais palavras do nome devem ser iniciadas por letras maiúsculas.

Nomes de variáveis não devem iniciar com *underscore* `_` ou o caractere `$`, embora sejam elementos válidos em identificadores em Java.

O nome de uma variável deve lembrar ao leitor ocasional a intenção da variável. Os nomes de variáveis formados por um único caractere devem ser evitados, exceto quando se tratar de variáveis temporárias, locais a métodos.

Os nomes comuns para variáveis temporárias inteiras são `i`, `j`, `k`, `m` e `n`. Variáveis identificadas por `c`, `d`, e `e` são empregadas para o tipo `char`. Identificadores de variáveis devem seguir a abordagem camelCasing, conforme ilustrado abaixo.

```
int i;
char c;
float larguraMaxima;
```

Constantes

Nomes de variáveis declaradas como constantes devem ser formados exclusivamente por letras maiúsculas cujas palavras são separadas pelo caractere _ (*underscore*).

```
static final int JANEIRO = 1;
static final int FEVEREIRO = 2;
static final int ANO_BASE = 2004;
```

Práticas de programação

Métodos de conveniência para gerenciar associações

Métodos de conveniência deverão ser fornecidos para toda e qualquer classe que possua associações. As classes `Empregado` e `Empresa` que, nesta ordem, participam de uma associação n-para-um, terão métodos como `setEmpresa(Empresa)` para a classe `Empregado` e o método `addEmpregado(Empregado)` para a classe `Empresa`. A presente prática de programação sugere que estes métodos se comportem de tal forma que não seja necessário à classe cliente realizar manualmente a operação, conforme o parágrafo abaixo explica.

Código cliente ao enviar a mensagem `setEmpresa` para um objeto `Empregado` terá, antes que tal operação seja executada, remover a referência que eventualmente exista de uma instância de `Empresa` para o objeto `Empregado` em questão. Caso esta referência não seja removida teremos duas instâncias ou mais de `Empresa` referenciando o mesmo objeto `Empregado`, o que é um erro. Uma situação similar ocorre com o método `addEmpregado`.

Métodos de conveniência eliminam o esforço de programadores ao manipular associações fazendo com que as operações deixem o resultado final em um estado inconsistente. Observe que sem a presença deste código de conveniência poderíamos ter mais de uma Empresa contendo em seu "quadro de funcionários" um mesmo empregado, o que não é permitido pelo modelo.

Operadores lógicos (short-circuit)

Quando uma expressão lógica envolvendo os operadores `&&` e `||` é executada, o operando da direita só será avaliado caso a avaliação do operando da esquerda não seja suficiente para determinar o resultado da expressão. Por exemplo, `false && funcRetornaBoolean()` jamais fará com que a função `funcRetornaBoolean()` seja executada, pois a expressão não pode ser verdadeira. Ao contrário desse comportamento, `false & funcRetornaBoolean()` sempre irá fazer com que a função `funcRetornaBoolean()` seja executada, mesmo que o resultado da expressão não possa ser verdadeiro, como neste exemplo. Os mesmos comentários valem para o operador `||` e `|`. Não é recomendado o emprego da segunda forma, sem "curto-circuito". O objetivo é evitar efeitos colaterais. Noutras palavras, se algo deve ser executado, então faça com que esta execução seja clara aos olhos de quem irá ler o código que está sendo produzido. Também convém ressaltar a perda de desempenho que a segunda forma provoca.

Programação estruturada

Embora sentenças válidas em Java, tanto o `continue` quanto o `break` rotulados não são construções que, em geral, conduzem a código legível. Em conseqüência, é fortemente recomendado o não emprego destas construções com rótulo. De fato, considere como desaconselhado o emprego de rótulos em Java.

A palavra reservada `break` é empregada para interromper um laço ou em conjunto com a sentença `switch`. Estes empregos não são restritos. Em um laço, contudo, deverá existir no máximo um único emprego de `break`.

A palavra reservada `continue` apenas termina a iteração corrente do laço. Esta palavra reservada não é recomendada. Considere fortemente desaconselhado o emprego de `continue`.

Saída padrão

Toda e qualquer mensagem enviada para a saída padrão, por exemplo, através da sentença `System.out.println()` e suas variantes, deve possuir um bom motivo para existir. Em caso de dúvido, simplistemente remova esta sentença. Convém ressaltar que o recurso de *logging* deverá ser empregado para o registro de informações que possam ser úteis para auditoria, informações de erros, interrupções inesperadas do software e outras. Em conseqüência, parece não existir espaço, com o emprego de *logging*, para o emprego de `System.out` ou `System.err` e suas variantes.

Não empregue o depurador (*debugger*)

O principal instrumento de teste do desenvolvedor são os testes de unidade, ou seja, códigos que testam "pequenas" partes do código desenvolvido. O emprego de um depurador com o propósito similar é desaconselhado. Em cenários específicos, por outro lado, pode ser útil acompanhar o valor resultante da execução de uma sentença ou consultar o valor de determinada variável. Novamente, apenas em casos bem peculiares pode-se encontrar alguma motivação que justifique o uso do depurador. Em todos os demais casos, crie testes de unidade.

Nunca faça uso de System.exit(int)

O emprego de `System.exit(int)` é fortemente desaconselhado. Convém ressaltar que esta sentença termina a execução não só da aplicação corrente como também da máquina virtual Java em que esta aplicação executa tal sentença. O término da MVJ fará com que todas as aplicações ali executadas também sejam interrompidas.

Oferecer acesso a variáveis de instância e variáveis de classe

Não declare qualquer variável pública, seja de classe ou de instância, sem uma boa razão. Siga a lei do menor privilégio.

Muitas vezes variáveis de instância não precisam ser acompanhadas dos métodos `set` e `get`, cujos efeitos correspondentes seriam obtidos indiretamente através de chamadas de métodos.

Referência a variáveis e métodos de classe

Evite usar um objeto para ter acesso a uma variável ou método de classe. Variável ou método de classe é aquele cuja declaração faz uso da palavra reservada `static`. Em vez do emprego de um objeto, faça uso do nome da classe em questão. Por exemplo:

```
classMethod();                // Correto
UmaClasse.classMethod();      // Correto
UmObjeto.classMethod();       // EVITE!
```

Constantes

Literais numéricas (valores numéricos fornecidos no código fonte) não devem ser empregadas diretamente no código, com exceção de `-1`, `0`, e `1`, que podem aparecer na declaração de um laço `for` como valores de contadores. Nos demais casos, as literais deverão ser empregadas indiretamente através da declaração de constantes (variáveis declaradas como `final`).

Atribuições

Evite atribuir a variáveis um mesmo valor em uma única sentença, conforme abaixo.

```
fooBar.fChar = barFoo.lChar = 'c'; // EVITE!
```

Em vez da versão acima, reescreva-a como abaixo.

```
fooBar.fChar = 'c';
barFoo.lChar = 'c';
```

Não use o operador de atribuição em um lugar onde pode ser confundido facilmente com um operador de igualdade. Por exemplo, o emprego das variáveis `c` e `d` do tipo `boolean`, no trecho de código abaixo, deve ser evitado.

```
if (c = d) { // EVITE!
    ...
}
```

Deveria ser escrito como

```
c = d;
if  (d) {
    ...
}
```

Não use expressões longas ou complicadas em uma tentativa de melhorar o desempenho do programa. Isto é trabalho do compilador.
Exemplo:

```
d = (a = b + c) + r; // EVITE!
```

Deveria ser escrito como

```
a = b + c;
d = a + r;
```

Outras práticas

Parênteses

O emprego de parênteses em expressões que envolvem vários operadores deve ser obrigatório, em alguns casos, para se evitar confusões.

```
if (a == b && c == d)       // EVITE!
if ((a == b) && (c == d))   // Correto
```

Não são apenas as expressões lógicas que se beneficiam do emprego de parênteses.

```
x = 10+2 * x+y;         // EVITE!
x = 10 + (2 * x) + y;   // Correto
```

Como regra, assuma que expressões com dois ou mais operadores de precedências distintas devem fazer uso obrigatório dos parênteses. Mesmo expressões que empregam um único operador podem se beneficiar de parênteses, conforme os exemplos abaixo.

```
x = (3 * x) + 2;                    // Correto
x = 3 * x + 2;                      // EVITE!

r = receitas - despesas + juros;    // EVITE!
r = (receitas - despesas) + juros; // Correto
```

Retornando valores

A estrutura do seu programa deve ser o mais simples possível. Observe o exemplo abaixo.

```
if (expressãoLógica) {  // EVITE!
    return true;
} else {
    return false;
}
```

Deveria ser escrito como

```
return  expressãoLógica;
```

De forma similar,

```
if (condição) {  // EVITE!
    return x;
}
return y;
```

Deveria ser escrito como

```
return (condição ? x : y);
```

Operador condicional

Se uma expressão que contém um operador binário aparece antes de `?` (parte do operador ternário `?:`), então deverá ser fornecida entre parênteses. Exemplo:

```
(x >= 0) ? x : -x;
```

Comentários especiais

Use `XXX` em um comentário para sinalizar algo que está inadequado, mas funciona. Use `FIXME` para sinalizar algo que está errado e não funciona. Em ambos os casos, `XXX` e `FIXME` permitem identificar elementos de um programa em Java que precisam ser alterados.

Comentário	Descrição
`XXX`	Funciona, provavelmente de forma inadequada, e precisa ser corrigido.
`FIXME`	Não funciona, está errado, precisa ser corrigido.
`TODO`	A ser implementado.

Exemplos de código

O exemplo seguinte mostra como formatar um arquivo fonte de Java que contém uma única

```
/*
 * @ (#)Blah.java 1.82 99/03/18
 *
 * Copyright (c) 2003 Instituto de Informática (UFG)
 * Caixa postal 970
 * Todos os direitos reservados.
 *
 * Coloque aqui outras informações relevantes.
 */

package java.blah;

import java.blah.blahdy.BlahBlah;

/**
 * Descrição da classe.
 *
 * @version 1.82 18 Arruinam 1999
 * @author Firstname Lastname
 */
public class Blah extends NossaClasse {

    /* Algum comentário de implementação de classe */

    /** comentário de documentação de vardClasse1 */
    public static int vardClasse1;

    /**
     * comentário de documentação de vardClasse2 que deve
     * ser mais que um linha grande
     */
    public static Object vardClasse2;

    /** comentário de documentação de vardInstancia1 */
    public Object vardInstancia1;

    /**
     * ... comentário de documentação do construtor de Blah
     * /
    public Blah() {
        // ... a implementaçao vem aqui...
    }

    /**
     * ... comentairio de documentação do método ...
     * @param descrição de  someParam
     */
    public void doSomethingElse(Object someParam) {
        // ... implementaçao do método ...
    }
}
```

classe pública. Interfaces são formatadas de forma análoga.

Considerações finais

O código abaixo encontra-se rigorosamente correto da perspectiva léxica, sintática e semântica. Trata-se de um exemplo, ao final deste conjunto de convenções, de algo que não se deve fazer. Também ressalta que computadores e seres humanos precisam ser considerados de formas distintas.

```
public classe MeuPrimeiroPrograma{public static void main(String[]args){System
.out   .println("Meu primeiro programa");}}
```

O objetivo das convenções citadas é melhorar a compreensão do código, facilitar a revisão, mesmo quando isto ocorrer muito tempo depois de ser escrito e, possivelmente, por outra pessoa. Seguir convenções pode conduzir a estruturas de codificação bem mais nítidas e fáceis de serem compreendidas do que aquelas que seria produzidas sem que convenções sejam seguidas.

O leitor que se interessar pelo tema poderá aprofundar as discussões subjacentes a convenções de codificação em *Human Factors and Typography for More Readable Programs*, Baecker e Marcus, Addison-Wesley, 1990.

Agradecimentos

Este trabalho agrega contribuições de várias pessoas, sem as quais a qualidade seria definitivamente inferior. Versões iniciais deste documento contaram com a ativa participação de Ruither Junio Queiroz. Vários erros foram indicados por André Lovato, que também forneceu sugestões que foram acrescentadas às convenções.