PROJETO FRAMEWORK - CELEPAR

PADRÕES E CONVENÇÕES PARA CÓDIGO JAVA

Versão 1.4

Março – 2006

Sumário de Informações do Documento

Tipo do Documento: Definição

Título do Documento: Padrões e Convenções para Código Java

Estado do Documento: Elaborado

Responsáveis: Elisabeth Hoffmann, Vanderlei Ortêncio, Robson Valentin, Cristina Machado

Palavras-Chaves: armazenamento, padronização

Resumo: Documento que descreve os padrões para codificação JAVA

Número de páginas: 28

Software utilizado: OpenOffice

Versão	Data	Mudanças
1.0	16/09/04	Versão revisada após reunião com os seguintes participantes: Cleverson Budel, Cristina Filipak Machado, Fabio Sgoda, Elaine Kawa, Jean Marcelo, Jefferson Marçal, Jefferson Martins, Johnatan Lima, Luciana Maria Reis e Paulo Assis
1.1	25/01/05	Revisão Geral para aprovação feita por: Cristina Filipak, Fabio Sgoda, Cleverson Budel, Diego Pozzi, Marcio Hein, Jefferson Martins, Carlos Roland, Manoel Leal, Filipe Lautert, Luciano Mittmam, Jefferson Marçal, Danilo Akioshi, Artur Dittrich, Alexandre Yamauchi, Flavio Oliveira, P.Rosa e Robson Valentin.
1.3	15/07/05	Revisão do item DOCUMENTAÇÃO, conforme sugestões DIFAS - por: Cleverson Budel, Fabio Kolling
1.4	18/08/05	Correções de ortografia, estrutura e padronização - por: Cleverson Budel, Garten Nack, Manoel Flávio Leal e Elisabeth Hoffmann
1.5	13/03/06	Alteração no padrão dos comentários – por: Natasha Fortes – revisado por Elisabeth Hoffmann

SUMÁRIO

1 INTRODUÇÃO

Este documento tem como finalidade apresentar os padrões e convenções que deverão ser utilizadas na criação e/ou codificação de aplicativos Java.

2 ORGANIZAÇÃO DE ARQUIVOS

Um arquivo consiste de seções que devem ser separadas por linhas em branco e um comentário opcional identificando cada seção.

Arquivos maiores que 2.000 linhas são “inconvenientes” ou “deselegantes” e devem ser evitados.

2.1 Arquivos fonte Java

Cada arquivo fonte Java contém um classe pública ou uma interface. Quando classes privadas e interfaces são associadas a uma classe pública, você poderá colocá-las em um mesmo arquivo fonte. A classe pública ou a interface deverá ser a primeira no arquivo.

Arquivos fontes Java possuem a seguinte ordem:

- Comentários iniciais (opcional);
- Instruções (statements) de pacotes (package) e importações (import);
- Declaração de classes e interfaces.

```
Ex.
/*
 *  comentários iniciais (ver item 3 – Comentários de Documentação).
 *
 * Versão 1.0
 *
 * Data: 01/05/2004
 *
 * Copyright(c) 2004 Celepar - Companhia de Informática do Paraná.
 */

//pacote ao qual a classe pertence
```

```
package pesquisa.cep;

//importações de classes
import java.sql.Connection;
import java.util.Collection;
import gov.br.celepar.reuso.cep.Localidade;
import gov.br.celepar.reuso.cep.UF;

//declaração de classe
public class PesquisaCEP {
      ...
}
```

2.2 Comentários Iniciais (opcional)

Todos os arquivos fontes deverão começar com um comentário c-style (padrão C) que lista o nome da classe, versão e notas de direitos autorais.

```
/*
 * Classname.
 *
 * Version information
 *
 * Date
 *
 * Copyright notice
/*
```

2.3 Package e Import

A primeira linha 'sem-comentários' da maioria dos arquivos fontes Java é um *statement* de *package*. Depois disso, *statements* de *import* podem seguir. Por exemplo:

```
package java.awt;
import java.awt.peer. CanvasPeer;
```

2.4 Declaração de Classes e Interfaces

A tabela seguinte descreve as etapas (partes) da declaração de uma classe ou interface, na ordem em que deverão aparecer.

	Partes da declaração de classes e interface	***Observações***
1	Comentários de Documentação (/**..*/)	Veja "Comentários de documentação" para informações sobre como usar este comentário
2	Declaração de Classe ou Interface.	
3	Implementação da classe / interface, comentário (/* ... */), se necessário	Este comentário deverá conter qualquer informação sobre a classe ou interface, que não seja apropriado para documentação.
4	Variáveis Estáticas (static).	Primeiramente as variáveis públicas (public) da classe, então as protegidas (protected) e finalmente as privadas (private).
5	Instância de Variáveis	Primeiro a *public*, então a *protected* e finalmente a *private*;
6	Construtores	
7	Métodos	Os métodos devem ser agrupados pela funcionalidade de preferência pelo escopo ou acessibilidade. Por exemplo, um método privado de uma classe pode estar entre 2 duas instâncias de métodos públicos. O objetivo é tornar a leitura e o entendimento do código algo simples.

Nota: ver exemplo no item 9.

3. INDENTAÇÃO

Quatro espaços devem ser utilizados como unidade de indentação.

Nota: Recomenda-se utilizar a opção Format do Eclipse para facilitar a estruturação e organização apresentadas em todo o item 2.

3.1 Tamanho de Linha

Evite linhas maiores que 80 caracteres, geralmente não mais que 70 caracteres.

3.2 Wrapping Lines

Quando uma expressão não se ajusta somente em uma linha, quebre-a de acordo com estes princípios:

- Quebra depois de uma vírgula;
- Quebra antes de um operador;
- Alinhar a nova linha com o início de uma expressão no mesmo nível da linha anterior.

Aqui estão alguns exemplos de quebra para chamada de métodos:

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


var = someMethod1(longExpression1,
      someMethod2(longExpression,  longExpression));
```

Os dois exemplos a seguir são de quebra de expressões aritméticas. O primeiro é o preferido, desde que a quebra ocorra fora dos parênteses.

```
longName1 = longName2 * (longName3 + longName4 – longName5)
            + 4 * longName6; // recomendado

longName1 = longName2 * (longName3 + longName4
                        - longName5) + 4 * longName6; // evitar
```

Os dois exemplos a seguir são de indentação para declarações de métodos. O primeiro é o caso convencional. No segundo exemplo poderia ser deslocada a segunda e terceira linha para a direita, ao invés de indentar somente com 8 espaços.

```
// INDENTAÇÃO CONVENCIONAL
someMethod(int anArg, Object anotherArg, String yetAnotherArg,
```

```
                Object andStillAnother) {
    . . .
}


// INDENTAÇÃO COM 8 ESPAÇOS PARA EVITAR INDENTAÇÕES PROFUNDAS
private static synchronized horkingLongMethodName(int anArg,
        Object anotherArg, String, yetAnotherArg,
        Object andStillAnother) {
        . . .
}
```

Line wrapping para declarações **if** deveriam usar a regra de 8 espaços, desde que a convencional indentação (4 espaços) dificulte a visão do corpo. Por exemplo:

```
// NÃO USE ESTA INDENTAÇÃO
if ((condition1 && condition2)
     || (condition3 && condition4)
      || !(condition5 && condition6)) {   // bad wraps
   doSomethingAboutIt(); // torna esta linha fácil de perder-se
}

// USE ESTA INDENTAÇÃO
if ((condition1 && condition2)
            || (condition3 && condition4)
            || !(condition5 && condition6)) {
    doSomethingAboutIt();
}

// OU USE ESTA
if ((condition1 && condition2) || (condition3 && condition4)
            || !(condition5 && condition6)) {
    doSomethingAboutIt();
}
```

Aqui são apresentadas três maneiras para formatar operadores ternários:

```
alpha = (aLongBooleanExpression) ? beta : gamma;

alpha = (aLongBooleanExpression) ? beta
                                 : gamma;

alpha = (aLongBooleanExpression)
         ? beta
         : gamma;
```

4. COMENTÁRIOS

Os programas Java podem ter dois tipos de comentários: de implementação e de documentação. Comentários de implementação são aqueles encontrados em C++, delimitados por /* . . . */, e / /. Comentários de documentação (conhecido como “doc comments”) são somente Java, e são delimitados por /** . . . */. Comentários doc podem ser extraídos para arquivos HTML usando a ferramenta javadoc.

Comentários de implementação são significativos para comentar sobre uma implementação em particular. Comentários Doc são significativos para descrever a especificação de um código, de uma perspectiva livre de implementação para ser lida por desenvolvedores que, não necessariamente possuem o código fonte em mãos.

Comentários devem ser usados para dar uma visão geral do código e prover informação adicional para determinada parte do código, onde apenas lendo-o, não é possível o entendimento. Comentários devem conter somente informações relevantes para a leitura e entendimento do programa. Por exemplo, informações sobre como o determinado pacote é construído ou em qual diretório ele está armazenado, não devem ser incluídos como um comentário.

Nota: A freqüência de comentários algumas vezes reflete baixa qualidade do código. Quando você sente-se forçado a adicionar um comentário, considere reescrever o código para deixá-lo limpo.

Comentários não devem ser colocados em grandes caixas de desenho com asteriscos ou outros caracteres. Comentários nunca devem incluir caracteres especiais.

4.1 Formatos de implementação de comentários

Programas podem ter 4 estilos de comentários de implementação: bloco, linha simples, trailing e final de linha.

4.1.1 Comentários de bloco

Comentários de bloco são usados para prover descrições de arquivos, métodos, estrutura de dados e algoritmos. Comentários de bloco podem ser usados no início de cada arquivo e antes de cada método. Eles podem também ser usados em outros lugares como, dentro dos métodos. Comentários de bloco dentro de uma função ou método devem ser indentados no mesmo nível do código.

Um comentário de bloco deve ser precedido por uma linha em branco para defini-lo como "fora" do código.

```
/*
 * Aqui é um comentário de bloco.
 */
```

4.1.2 Comentário de linha simples

Comentários curtos podem aparecer em uma linha simples indentada no nível do código que ela segue. Se um comentário não puder ser escrito como uma linha simples, este deverá seguir o formato de um comentário de bloco. (ver item 3.1.1). Um comentário de linha simples deverá ser precedido de uma linha em branco. Aqui um exemplo de um comentário de linha simples dentro de um código Java.

```
If (condition) {

     /*  aqui o comentário para condição */
    . . .
}
```

4.1.3 Comentários Trailing

Comentários muito curtos podem aparecer na mesma linha de código, mas deverão estar longe

o suficiente para separá-los de outras instruções. Se mais de um comentário curto aparecer em um bloco de código, todos eles deverão usar a mesma indentação ou tabulação.

Aqui um exemplo de comentário trailing em um código Java:

```
If (a == 2)
    return TRUE;                        /* caso especial */
} else {
    return isPrime(a);                 /* trabalha somente para
impar */
}
```

4.1.4 Comentários de final de linha

O delimitador de comentário ‘ // ‘ pode ser utilizado para comentar toda uma linha de código ou parte dela, bem como excluir uma parte do código de sua seção. A seguir exemplos dos três estilos:

```
If (foo > 1) {
    // coloque o comentário aqui.
    . . .
}
else {
       return false;            // Explique a funcionalidade.
}

//if (bar > 1) {
//
//coloque o comentário aqui.
//         ...
//}
//else {
        return false;
//}
```

Nota: não comentar um código se ele não é usado.

4.2 Comentários de documentação

Comentários doc (comentários de documentação) descrevem classes Java, interfaces, construtores, métodos e atributos. Cada comentário doc é definido dentre os delimitadores ‘

/** . . . */ ‘, com um comentário por classe ou interface. Este comentário deverá aparecer antes das declarações da seguinte forma:

```
/**
 * Comentário.
 * @informações_especiais
 * /
```

Onde:

- Comentário indica descrições de abrangência da classe, interface ou método. Descreve funcionalidades e esclarecimentos de regras de negócios envolvidas na codificação.
- Informações Especiais são definidas pelo '@<nome_tag>'. Esses comentários são separados em blocos no documento gerado, para uma melhor estruturação.

  Essa estruturação pode se dar através das seguintes tags pré-definidas:

  @deprecated - adiciona um comentário de que a classe, método ou variável não deveria ser usada. O texto deve sugerir uma substituição.

  @since - descreve quando o elemento foi adicionado à especificação da API.

  @see - essa marca adiciona um *link* à seção "Veja também" da documentação.

  @author - autor do elemento.

  @version - número da versão atual.

  @param - descreve os parâmetros de um método acompanhado por uma descrição.

  @return - descreve o valor retornado por um método.

  @throws - indica as exceções que um dado método dispara.

  @serial - para documentar a serialização de objetos.

A primeira linha de um comentário doc (/**) para classes ou interfaces não é endentada, as linhas subseqüentes de comentários doc tem um espaço de indentação cada (verticalmente, alinhando os asteriscos).

Comentários doc não deverão ser inseridos dentro de um método ou de um bloco de definição de construtores.

4.2.1 Exemplos de Utilização

- Classes e Interfaces:

Documenta a funcionalidade da classe/interface, explicando seu objetivo e as funcionalidades abrangentes aos seus métodos e atributos.

Ex.

```
package gov.br.escola.dao;

import gov.pr.celepar.framework.RootDao;

/**
 * Classe DAO utilizada para referenciar o <br>
 * POJO gov.pr.escola.pojo.Pessoa.
 *@author kolling
 *@version 0.2
 *@since 15/07/2005
 *@see gov.pr.escola.pojo.Pessoa
 */

public class PessoaDao extends RootDao{
      ...
}
```

- Métodos e Atributos:

Documenta uma funcionalidade específica. Para um doc de qualidade, é importante observar que a primeira sentença do método deve conter uma descrição breve e objetiva terminando em ponto final.

Ex.

```
/**
 * O método efetua a listagem das pessoas pelo código <br>
 * da cidade informada como parâmetro.
 *@author kolling
 *@since 18/07/2005
 *@see gov.pr.escola.pojo.Pessoa
 *@see java.util.ArrayList
 *@param codCidade: Código da cidade
 *@return ArrayList<gov.pr.escola.pojo.Pessoa>: Lista de Pessoas
 *@throws ApplicationException caso código da cidade seja nulo
 *@throws Exception caso ocorra um erro inesperado
 */

public ArrayList<Pessoa> listarPessoasPorCidade(Integer codCidade)
throws ApplicationException, Exception{
      ....
}
```

Métodos que estão obsoletos e serão eliminados nas próximas versões do sistemas devem utilizar a diretiva:

```
/**
 * @deprecated
 * @see #metodoNovo
 */
@Deprecated public void metodoVelho(){
    ...
}
```

4.2.2 Versionamento de Classes/Interfaces (opcional)

Este controle é opcional pois esta informação pode ser obtida através da ferramenta de versionamento utilizada. Dependerá de cada projeto identificar a relevância desta informação no seu doc.

Em geral as versões das classes e interfaces seguem as seguintes regras:

- Começam sempre na versão 0.1;
- As mudanças de versão ocorrem quando são incluídos métodos de novos casos de uso.
- O incremento de versão: 0.1, 0.2, 0.3, ... , 0.9, 0.10, ... , 0.99, 0.100, ..., 0.N;
- A mudança de versão para N.0, ocorre quando é encerrado o módulo ou a etapa do projeto/sistema;
- As classes/interfaces de um novo módulo começam com versão 0.1;

5. DECLARAÇÕES

5.1 Números por Linha

Uma declaração por linha é recomendada desde que seja acompanhada de comentários. Em outras palavras,

```
int level;          // qual o nível
int size;           // tamanho da tabela
```

é melhor que

```
int level, size;
```

Não coloque diferentes tipos de dados na mesma linha. Exemplo:

```
int foo, fooarray[];
```

Nota: Os exemplos acima usam um espaço entre o tipo e o identificador. Outra alternativa

aceitável é o uso de *tabs* (tabulações), exemplo:

```
int             level;                  // qual o nível
int             size;                   // tamanho da tabela
Object          currentEntry;           // tabela de entrada de dados
```

5.2 Inicialização

Tente inicializar variáveis locais onde elas são declaradas. A única razão, para não inicializar uma variável onde ela é declarada, é se o valor inicial depende de algum outro processamento primeiro.

5.3 Placement (localização)

Coloque declarações somente no início dos blocos. (Um bloco é definido pelo código que está entre as chaves '{' e '}'). Não espere a variável ser usada a primeira vez para declará-la ou inicializá-la isto pode confundir um programador descuidado.

```
void myMethod() {
    int int1 = 0;             // No inicio do bloco do método.

    if (condition) {
       int int2 = 0;         // inicio do bloco " if "
       . . .
          }
    }
```

A única exceção a esta regra são os índices, como de *loops* **for**, que em Java podem ser declarados na própria instrução.

```
for (int i = 0; i < maxLoops; i++) { . . . }
```

Evite declarações locais que escondem declarações de níveis maiores. Por exemplo, não declare o mesmo nome de variável em um bloco interno:

```
int count;
. . .
myMethod() {
     if (condition) {
          int count;               // evitar!
          . . .
     }
```

```
        . . .
    }
```

5.4 Declarações de classes e interfaces

Quando estiver codificando classes e interfaces Java, as seguintes regras de formatação deverão ser seguidas:

- Sem espaços entre um método e o parênteses “ (“que inicia a lista de parâmetros;
- A chave “{“ aparece no final da mesma linha da declaração da instrução;
- A chave “}” aparece no final do bloco da instrução indentada de acordo com a instrução, exceto no caso em que instrução é nula, assim a chave fecha logo após a que foi aberta “{“.

```
Class Sample extends Object {
      int ivar1;
      int ivar2;

      Sample(int i, int j) {
            ivar1 = i;
            ivar2 = j;
      }
      int emptyMethod() {}
      . . .
}
```

- Métodos são separados por uma linha em branco.

6. STATEMENTS (INSTRUÇÕES)

6.1 Instruções simples

Cada linha deverá conter no máximo uma instrução. Exemplo:

```
argv++;                    // Correto
argc++;                    // Correto
argv++; argc --;           // Evitar
```

6.2 Instruções compostas

Instruções compostas possuem uma lista de subinstruções entre chaves. Veja os exemplos nas próximas sessões.

- Subinstruções deverão ser indentadas um ou mais níveis que as instruções compostas.
- A chave aberta "{" deverá estar no final da linha onde inicia-se a instrução composta; a chave fechada "}" deverá começar em uma linha indentada com o inicio da instrução composta.
- As chaves são usadas "ao redor" de todas as instruções, exceto para instruções simples, quando elas são partes de uma estrutura de controle, tais como uma instrução IF-ELSE or FOR.

6.3 Instrução Return

A instrução return com um valor não deverá usar parênteses a menos que este valor torne-se mais claro com o uso de parênteses.

```
return;

return myDisk.Size();

return (size ? size : defaultSize);
```

6.4 Instruções if, if-else, if else-if else

A instrução if-else deverá utilizar uma das seguintes formas:

```
if  (condition) {
     statements;
}

if (condition) {
    statements;
} else {
```

```
        statements;
    }

    if (condition) {
        statements;
    } else if (condition) {
        statements;
    } else {
        statements;
    }
```

Nota: Instruções if sempre utilizam chaves {}. Evite usar as seguintes formas, pois são consideradas como erros.

```
if (condition)       // evite, omitir as chaves nestes casos.
   statements;
```

6.5 Instrução for

Uma instrução for deverá ter a seguinte forma:

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

Quando utilizar o operador “virgula” na cláusula de inicialização ou atualização de uma instrução “for”, evite usar mais de 3 variáveis. Se necessário, use instruções em separado antes do laço “for” (para inicialização da cláusula) ou no final do loop (para a cláusula de atualização).

6.6 Instrução while

Uma Instrução while deverá ter a seguinte forma:

```
while (condition) {
     statements;
```

```
}
```

6.7 Instrução do-while

Uma instrução do-while deverá ter a seguinte forma:

```
do {
        statements;
} while (condition);
```

6.8 Instrução switch

Uma instrução switch deverá ter a seguinte forma:

```
switch (condition) {
case ABC:
        statements;
        /* “falsa passagem” */
case DEF:
        statements;
        break;

case XYZ:
        statements;
        break;

default:
        statements;
        break;
}
```

Cada instrução *switch* deverá incluir um caso *default*. O uso do *break* em uma instrução default é redundante, mas não previne os erros de “passagens falsas” se outra instrução *case* for adicionada.

6.9 Instruções try-catch

Uma instrução *try-catch* deverá ter o seguinte formato:

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

Uma instrução *try-catch* pode também ser seguida da instrução *finally*.

```
 try {
        statements;
} catch (ExceptionClass e) {
        statements;
} finally {
        statements;
}
```

7. ESPAÇOS EM BRANCO

7.1 Linhas em branco

Linhas em branco facilitam a leitura do código, uma vez que se definem seções de código relacionadas.

Duas linhas em branco deverão ser sempre usadas para as seguintes circunstâncias:

- Entre seções de um arquivo fonte.
- Entre a as declarações de classes e interfaces.

Uma linha em branco deverá sempre ser usada nas seguintes circunstâncias:

- Entre métodos.
- Entre variáveis locais em um método e sua primeira declaração.
- Antes de um bloco ou de um comentário simples.

- Entre seções lógicas dentro de um método para facilitar a leitura.

7.2 Espaço em branco

Espaços em branco deverão ser usados nas seguintes circunstâncias:

- Uma palavra chave seguida de parênteses deverá ser separada por um espaço. Exemplo:

```
while (true) {
      . . .
}
```

Note que um espaço em branco não deve ser usado entre o nome de um método e o parênteses que será aberto. Isto ajuda a distinguir “as chaves” da chamada de um método.

- Um espaço em branco deverá aparecer depois de vírgulas em listas de argumentos.
- Todos os operadores binários exceto “ . “ deverão ser separados de seus operandos por espaços. Espaços em branco nunca deverão aparecer entre operadores unários, como o operador “-“ ou o operador de incremento “++” e decremento “—“. Exemplos:

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

while (d++ = s++) {
      n++;
}
prints(“size is ” + foo + “\n”);
```

- As expressões dentro de uma instrução “for” deverão ser separadas por espaços em branco. Exemplo:

```
For (expr1; expr2; expr3)
```

- Operadores de *cast* deverão ser seguidos por um espaço em branco. Exemplo:

```
myMethod((byte) aNum, (Object) x);
myMethod((int) (cp + 5), ((int) (I + 3)) + 1);
```

8. PRÁTICAS DE PROGRAMAÇÃO

8.1 Referenciando classes e métodos

Evite o uso de um objeto para acessar uma variável estática ou uma classe. Use o nome da classe instanciada. Por Exemplo:

```
classMethod();                    // ok
Aclass.classMethod();             // ok
anObject.classMethod()            // evitar
```

8.2 Constantes

Constantes numéricas não deverão ser codificadas diretamente, exceto para os casos, -1, 0 e 1, que podem aparecer em um laço “for” como um contador.

8.3 Atribuições

Evite atribuir para muitas variáveis um mesmo valor em uma única instrução. Isto dificulta a leitura. Exemplo:

```
fooBar.fChar = barFoo.lchar = ‘c’;          // evitar
```

Não use operador de atribuição em lugares onde podem ser confundidos facilmente com o operador de igualdade. Exemplo:

```
if (c++ = d++) {                    // evite
            . . .
}
```

Deverá ser escrito como

```
if ((c++ = d++) != 0) {
        . . .
 }
```

8.4 Práticas gerais

8.4.1 Parênteses

O uso de parênteses é geralmente considerado uma boa prática, principalmente em expressões que utilizam vários operadores com o objetivo de evitar problemas de precedência. Sempre que a precedência não ficar clara, a melhor solução é inserir parênteses na expressão.

```
if (a == b && c == d)                // evite

if ( (a == b) && (c == d))          // faça desta forma.
```

8.4.1 Retorno de valores

Tente deixar a estrutura de seu programa com os relacionamentos endentados. Exemplo:

```
if (boolenaExpression) {
     return true;
} else {
     return false;
}
```

em vez de expressar assim:

```
return booleanExpression;
```

Similar,

```
if (condition) {
     return x;
}
return y;
```

deverá ser expressa assim:

```
return (conditon ? x : y);
```

9. NOMENCLATURAS

Nomenclaturas fazem com que os programas se tornem de fácil leitura e entendimento. Elas também fornecem informação do funcionamento de um identificador – por exemplo, para uma constante, pacote ou classe – isto torna o código "elegante".

Tipo do Identificador	***Regras para nomenclatura***	***Exemplos***
Pacotes (Packages)	O prefixo do nome de um pacote é escrito com letras minúsculas e deverá representar o topo mais alto do nome de seu domínio como, com, edu, gov, mil, net, org . Em seguida o nome dos componentes do pacote podem variar de acordo com as seguintes convenções: nome do diretório onde os componentes serão usados, departamento, projeto, máquina ou nome de login.	gov.pr.[nome-do-cliente].[nome-do-pacote] gov.pr.celepar.[nome-do-pacote] Exemplos: gov.pr.celepar.reuso.cep gov.pr.detran.veiculo.licenciamento gov.pr.detran.veiculo.emplacamento
Classes	O nome de uma classe é sempre um substantivo, uma mistura de letras maiúsculas e minúsculas. A primeira letra é maiúscula e para cada outra palavra (interna). Tente manter o nome das classes simples e descritivo. Evite utilizar acrônimos e abreviações.	class Raster; class ImageSprite;
	Para nomenclaturas de classes que implementam patern utilizar a concatenação do nome da classe com o nome do pattern.	class MatriculaFacade; class AlunoDAO; class MunicipioForm; class DisciplinaAction; class UsuarioDTO;

Tipo do Identificador	*Regras para nomenclatura*	*Exemplos*
Interfaces	O mesmo padrão utilizado nas classes.	interface RasterDelegate; interface Storing;

Métodos	Métodos deverão ser verbos, uma mistura de letras minúsculas com maiúsculas. A primeira letra é minúscula e as demais, para cada nova palavra, maiúscula.	run();
		runFast();
		getBackground();

Variáveis	Variáveis começam com a primeira letra ou palavra em minúsculo, as demais palavras (internas) são em maiúsculo. Os nomes não podem começar com caracteres especiais. Nome das variáveis devem ser pequenos e de fácil entendimento. Variáveis cujo nome é de apenas um caractere devem ser evitados, com exceção de variáveis temporárias utilizadas em laços como i,j,k.	int i; char c; float myWidth;

Constantes	O nome das constantes deve ser declarado em letras maiúsculas, separadas em por *underscores* "_".	static final int MIN_WIDTH = 4;
		static final int MAX_WIDTH = 999;
		static final int GET_THE_CPU = 1;

10. REFERÊNCIAS

How to Write Doc Comments for the Javadoc Tool. Disponível em: http://java.sun.com/j2se/javadoc/writingdoccomments/. Acesso em 25 jan 2005.