UNIP UNIVERSIDADE PAULISTA Instituto de Ciências Exatas *Campus Brasília*	Curso: Ciência da Computação / Sistema de Informação Material : Padrão de codificação – Linguagem Java Prof(a): Msc Wanderley Gonçalves Freitas

Padrão de Codificação Java

Brasília
Agosto 2013

Wanderley.unip@gmail.comr

www.wg.pro.com.br

Sumário

1. Introdução

Este documento tem como objetivo definir um padrão de codificação para aplicações que utilizam a linguagem Java. Sendo as atividades de manutenção responsáveis pelo maior esforço do desenvolvimento de software, quanto mais fácil for o entendimento do código da aplicação, mais produtiva será a equipe de desenvolvimento. Freqüentemente as pessoas que escrevem o código não são as mesmas que o mantém e, quando são, geram uma dependência com o código desenvolvido que dificilmente é dissolvida.

Um padrão de codificação visa minimizar esses problemas, pois estabelece regras definindo como o código deve ser escrito para favorecer a impessoalidade do artefato sem modificar a funcionalidade da aplicação.

Qualquer código a ser implementado deverá seguir as regras deste padrão, destacadas com marcadores do tipo:**(1), (2),(3)**; desta forma, é possível fazer referência a uma determinada regra pelo seu número.

Por exemplo, a regra **2.1.1**, localizada na **seção 2.1** deste documento, refere-se a seguinte regra "Não se define mais de uma classe por arquivo".

2. Arquivos

2.1. *Nomenclatura*

(1) Não se define mais de uma classe por arquivo, exceto **para *Inner Classes***. As ***Inner Classe*** são classes declaradas dentro do escopo de outra classe. Esta regra afirma que só pode existir uma classe por arquivo, no entanto, esta classe pode conter ***Inner Classes***.

(2) O nome de um arquivo é o nome de sua classe pública. Não é recomendável haver um arquivo que não contém nem classe e nem estruturas públicas.

2.2. *Documentação*

Há três formas de comentário em Java:

- O estilo Javadoc2[1] inicia o comentário com **"/**"** e termina com **"*/"** e é usado para gerar documentação do código em **páginas *HTML*** .
- Outro estilo de documentação em Java foi herdado da **linguagem C** e, por isto, é freqüentemente chamado de "**estilo C**"; iniciado com **"/*"** e finalizado com **"*/"** é usado para

[1] Não faz parte do escopo deste documento apresentar o padrão Javadoc, para isto, é recomendada a leitura do artigo http://java.sun.com/j2se/javadoc/reference/docs/index.html disponível no site da Sun, entretanto, no Apêndice A deste documento há uma breve introdução ao padrão Javadoc, mostrando a utilidade das suas principais construções.

comentar código que não está sendo usado mas que o programador prefere manter comentado caso mude de idéia.

- Por último, os comentários em linha que iniciam com **"//"** e tem o efeito de comentar o restante da linha após a sua presença. Este último é usado em métodos para explicar regras de negócio e a lógica de execução dos métodos.

(1) Cada arquivo começa com um bloco de comentários contendo as seguintes informações: ***título do projeto***, ***nome da classe*** ou ***interface*** e informação de *copyright* relevante ao conteúdo do arquivo.

(2) Se um arquivo possuir mais de um tipo (**classe ou interface**), é inserida uma lista com uma pequena descrição de cada tipo que compõe o arquivo. É importante destacar uma explicação que justifique a declaração de mais de um tipo por arquivo, pois Java só permite um tipo público por arquivo, dificultando a busca dos tipos não públicos.

Abaixo, segue o modelo de documentação de arquivo.

```
/* Projeto: <Título do projeto>
*
* Tipo1: <Nome da classe ou interface pública>
* Tipo2: <Nome da classe ou interface>, descrição: <descrição da classe ou
interface>
* ...
* TipoN: <Nome da classe ou interface>, descrição: <descrição da classe ou
interface>
*
* Informação de Copyright opcional
*/
```

2.3. *Declaração*

(1) A classe ou interface pública é a primeira a ser declarada em um arquivo

(2) A tabela a seguir apresenta a ordem das declarações em um arquivo. Quanto mais acima a declaração na tabela, mais à frente ela irá aparecer no arquivo do código fonte.

Declaração de um arquivo	**Nota**
Comentários do arquivo	Ver subseção 2.2
Declaração do pacote	Ver subseção 3.2
Declaração de classe ou interface	Ver subseções 4.2 e 5.2
Declaração de *Inner classes*	Ver subseções 4.2
Declaração de enumerações	Ver subseção **Erro! Fonte de referência não encontrada.**

3. Pacote

3.1. *Nomenclatura*

(1) Os identificadores - as palavras que compõem o nome de um pacote – são separados por pontos e não há restrição quanto ao seu número; são iniciados com letras **minúsculas** e não devem conter caracteres especiais, como ***underscores***, ou caracteres específicos de uma língua.

Os pacotes são nomeados de acordo com os conceitos que agrupam.

(2) Para uma aplicação de gestão empresarial, por exemplo, poderiam ser definidos os pacotes ***gestao.financeiro.contasreceber*** e ***gestao.contabilidade*** para agrupar, respectivamente, as classes do subsistema de contas a receber do sistema financeiro e as classes da contabilidade.

(3) Recomenda-se a definição de pacote com o nome da empresa seguido do nome da aplicação ou sua abreviação. Por exemplo:

organização.aplicação.subsistema1.camada1

3.2. *Declaração*

(1) Após o comentário de início do arquivo, discutido na seção 2.2, é declarada a sentença ***package***, após isto, a lista de classes importadas é relacionada através da cláusula ***import***.

(2) As classes dos pacotes do núcleo (***core***) de Java, pacotes tipo " **java**.<**nome do pacote**>", são listadas primeiro, após estas, são listadas as classes das extensões de Java, pacotes tipo "**javax.<nome do pacote>",** e, por último, as classes específicas do sistema e outras APIs.

4. Classes

4.1. *Nomenclatura*

(1) Os nomes das classes não são abreviados, excetuando-se os casos em que as abreviações são mais conhecidas que o seu nome completo, pois a economia de palavras na elaboração de um nome não compensa a perda de expressividade associada.

(2) Não se usam artigos ou preposições para conectar substantivos e adjetivos, nem caracteres específicos de uma língua como é o caso do "**ç**" e os acentos da língua portuguesa.

(3) A primeira letra de cada palavra que compõe o nome de uma classe é **maiúscula**. O nome de uma classe representa um objeto da mesma e não o seu conjunto de objetos.

(4) Em conseqüência da regra anterior, o nome da classe é sempre no singular, exceto nos casos em que o próprio objeto represente uma coleção de outros objetos. Nestes casos, usa-se a palavra que represente a coleção no singular e o nome dos objetos no plural como, por exemplo, **RepositorioAlunos**.

(5) Toda classe que define uma exceção contém a palavra **Exception** no final de seu nome.

(6) O nome de uma classe não é mais abrangente nem mais específico que o conceito que representa, pois, no primeiro caso, o Desenvolvedor pode querer usar uma funcionalidade apesar de não ser oferecida pela classe; no outro caso, o Desenvolvedor poderia não encontrar a classe por não representar o conceito mais genérico que procura.

Seguem alguns exemplos de nomes de classes:

- **Aluno**
- **AlunoGraduacao**
- **ProvaInvalidaException** // uma exceção

4.2. *Declaração*

(1) Após a declaração do nome da classe e seus supertipos, são declaradas as constantes, variáveis de classe[2] *(usam-se os termos "**variáveis de classe**" e "**métodos de classe**" com o mesmo significado de "**variáveis estáticas**" e "**métodos estáticos**", respectivamente)*, propriedades de classe, variáveis de instância, propriedades de instância, Construtores estáticos de classe, construtores, finalizador, métodos de classe e métodos de instância, nesta seqüência.

(2) Os métodos são agrupados por funcionalidade e não pela forma de acesso ou sua condição de estático ou de instância.

(3) A tabela seguinte ilustra a ordem de declarações dos elementos de uma classe. Quanto mais acima a declaração do elemento na tabela, mais à frente ela irá aparecer no arquivo do código fonte.

Declaração da Classe	Nota
Comentários da classe (/** ...*/)	-
Sentença class	-
Comentários de implementação da classe	Caso necessário
Constantes	-
Variáveis de classe	Na seguinte ordem:

[2] Usa-se o termo variável e métodos de classe com o mesmo significado de variável e métodos estático, respectivamente.

	▪ públicas (public), ▪ protegidas(protected), ▪ Sem modificadores(pacote) ▪ Privadas(private).
Variáveis de instância	Mesma ordem das variáveis de classe
Construtores	Mesma ordem das variáveis de classe
Métodos	Os métodos devem ser agrupados por funcionalidade e não por forma de acesso.

4.3. *Documentação*

(1) A linha imediatamente anterior a definição da classe deve conter um comentário descrevendo seu propósito.

Falta colocar exemplo

(2) Podem ser acrescentados lembretes sobre possíveis melhoramentos e defeitos existentes na classe. Esses textos devem estar dentro da ***// TODO***[3]. .

5. Interfaces

5.1. *Nomenclatura*

(1) O nome de uma interface é um adjetivo ou substantivo, e segue as mesmas regras para nomenclatura de classes definidas na subseção 4.1.

Abaixo, segue exemplos de possíveis nomes para interfaces:

- UIExecutavel
- UIEntradaDados

5.2. *Declaração*

(1) As declarações de interface seguem a ordem apresentada no modelo abaixo.

Declaração da Classe	**Nota**
Comentários da interface	-
Sentença interface	-
Propriedades da classe	-
Métodos	Os métodos devem ser agrupados por funcionalidade e não por forma de acesso.

[3] IDE – Eclipse 3...

5.3. *Documentação*

A documentação de interfaces é idêntica à descrita para variáveis na subseção XX.

6. Variáveis de classes, de instância e constantes

6.1. *Nomenclatura*

(1) Os nomes das variáveis de classe e instância atendem aos requisitos abaixo:
- A primeira letra das palavras, exceto da primeira palavra, é minúscula;
- Os nomes não são abreviados, exceto nos casos que a sua abreviação seja mais sugestiva que o nome completo, ou no caso de variáveis que armazenam componentes visuais
- Não se mistura palavras de mais de uma língua
- Não se utiliza nenhum caracter especial nem específico de uma língua

(2) Os nomes das constantes são compostos de palavras não abreviadas com todas as letras maiúsculas utilizando *underscores* como separadores.

Veja abaixo nomes de constantes e variáveis de classes e instância:

- VALOR_MAXIMO // constante
- autor // variável de instância
- livro // variável de instância
- contador // variável de classe

6.2. *Documentação*

(1) A documentação de uma variável ou constante é feita em linha (veja seção ...) exceto nos casos em em que não caiba em uma linha de código

(2) Há uma justificativa caso alguma variável ou constante não seja declarada privada

6.3. *Declaração*

(1) Há Apenas uma variável ou constante é declarada por linha de código.

7. Métodos

7.1. *Nomenclatura*

(1) Os nomes dos métodos atendem aos requisitos abaixo:

- A primeira letra das palavras é minúscula.
- Os nomes não são abreviados, exceto nos casos que a sua abreviação seja tão ou mais sugestiva que o nome completo.
- Não se misturam palavras de mais de uma língua.
- Não se utiliza nenhum caractere especial nem específico de uma língua.
- A primeira palavra deve ser um verbo no **infinitivo** representando a utilidade do método, exceto quando a função retornar um **boolean**, que deve ser representada com o verbo no **presente**.
- Devem-se utilizar verbos nos nomes dos métodos que executam alguma operação no objeto, por exemplo, c**adastrarFuncionario**().
- É redundante repetir o nome da classe no nome do método. Assim, deve usar Livro.CalcularDesconto() ao invés de Livro.CalcularDescontoLivro().
- Deve-se evitar utilizar nomes que são sujeitos a interpretação pessoal, como AnaliseIsto().

Apesar da aplicação deste padrão, geralmente, resultar em nomes maiores, necessitando de digitação extra, o efeito da sua conformidade é um código mais fácil de compreender, pois o propósito do método já é esclarecido no seu nome.

(2) Métodos de acesso a variáveis iniciam com ***get*** ou ***set*** finalizam com o nome da variável tendo a primeira letra de cada palavra maiúscula.

Seguem abaixo alguns exemplos de nomes de métodos:

- adicionarAluno(Aluno aluno)
- removerAluno(Aluno aluno)
- existeAluno(int codigoAluno)
- getNumero() // método de acesso

7.2. *Documentação*

(1) A linha imediatamente anterior a definição do método deve ter um comentário contendo informações suficientes para seu entendimento e uso adequado, todos os parâmetros necessários para chamar o método , sua cláusula de retorno e as exceções levantadas

(2) Caso a decisão de visibilidade (*public*, *protected* e *private*) do método possa ser questionada, documenta-se a razão pela qual foi tomada esta decisão.

(3) Se necessário, são declaradas ao final do comentário referências a outras classes e métodos.

Segue um exemplo de documentação de método.

```
/**
*...Este método calculará o imposto devido bruto do cliente ...
*
*...Caso a decisão de visibilidade possa ser questionada a
* justificativa será feita neste trecho
*
* @param taxa Valor da taxa que será usada para calcular o imposto
*
* @return Valor do imposto bruto que será cobrado do cliente
*
* @exception Exceção1 Descrição da Exceção1
* @exception Exceção2 Descrição da Exceção2
*
* @see Tributo
* @since Data da criação do método opcional
*/
protected float calculaImposto(float taxa) throws ExcecaoTaxaInvalida
```

(4) Caso necessário, outros itens são acrescentados ao cabeçalho acima, como, por exemplo pré-condições e pós-condições, histórico de alterações do método, questões de concorrência, limitações e erros detectados no método.

(5) Não é aconselhável o uso de comentários dentro do corpo dos métodos. Ao invés disto, é sugerido produzir métodos simples, com comentários acima de sua assinatura.

(6) Caso o método possa ser melhorado ou corrigido, embora esteja funcionando corretamente, deve-se comentar isso informando a possível melhoria .

- `//CORRIGIR Comentário`

7.3. *Declaração*

(1) A declaração de métodos que retornam **Arrays** é feita da seguinte forma:

- double[] metodo()

e não da forma abaixo:

- double metodo() []

(2) Todas as declarações de variáveis locais são realizadas no início do código do método.

8. Variáveis locais e parâmetros

8.1. *Nomenclatura*

(1) A nomenclatura de variáveis locais é a mesma usada para variáveis de instância e classes (***Seção >>>>***), entretanto, por força de conveniência, essa nomenclatura é relaxada para os abaixos:

- Contadores – Usam-se letras do alfabeto para nomear contadores. A letra **i** é a primeira a ser usada; sendo necessário o uso de outros contadores, declara-se a letra **j** e assim sucessivamente até a última letra do alfabeto.
- Tipos definidos por Bibliotecas escritas em outras línguas – Usa-se o nome do tipo em letras minúsculas e uma descrição para o conceito que o objeto representa.
- Exceções – Como o uso de exceções é muito comum em Java, usa-se a letra **e** para declarar exceções genéricas.

Abaixo, seguem exemplos de nomes de variáveis locais:

- streamDadosRetornoBancario; // uma Stream
- i; // variável de laço

8.2. *Documentação*

(1) A documentação das variáveis locais é feita da mesma forma que a de variáveis de instância e de classe (veja secao XXXX):

8.3. *Declaração*

(1) Para facilitar a documentação e organização do método, apenas uma declaração de variável local é feita por linha de código.

9. Comandos

Nesta seção serão apresentadas as referências utilizadas para elaboração deste documento.

9.1. *return*

(1) Uma sentença return com valor de retorno não utiliza parênteses, a menos que a sentença fique mais clara.

9.2. *while, do-while, for*

(1) Abaixo, são apresentados os estilos de formatação recomendados:

- ```
  while (condição) {
      comandos;
  }
  ```
- ```
  do {
      comandos;
  } while (condição);
  ```
- ```
  for (inicialização; condição; atualização) {
      comandos;
  }
  ```

(2) As sentenças while e for vazias têm as seguintes formas:

- while (condição) {}
- for (*inicialização; condição; atualização*) {}
  ```

9.3. *if-else*

(1) O comando if-else é usado com as chaves – "{}" – para evitar ambigüidade no escopo do comando.

(2) As sentenças podem ter o **"{"** iniciando logo após o seu final, na mesma linha ou no início da outra linha, conforme indicado na regra Erro! Fonte de referência não encontrada. **da subseção 9.2.(verificar xxx)**

(3) Abaixo, são apresentados os estilos de formatação recomendados:

- ```
  if (condição) {
    comandos;
  }
  ```

- ```
  if (condição) {
    comandos;
  }
  else {
    comandos;
  }
  ```

- ```
  if (condição1){
    comandos;

  } else if (condição2) {
   comandos;
   }
     else {
          comandos;
     }
  ```

- ```
  if (condição1) {
    comandos;
  }
  else {
          if (condição2) {
             comandos;
          }
          else {
             comandos;
          }
  }
  ```

9.4. *switch*

(1) Abaixo, são apresentados os estilos de formatação recomendados:

```
switch (variável) {
 case ABC:
        comandos;
         break;
  case DEF:
        comandos;
        break;
   case XYZ:
        comandos;
        break;
   default:
        comandos;
        break;
}
```

O último comando de toda sentença case é o **break**.
Toda sentença switch possui uma cláusula **default**.

(2) As senteças devem ter o "{" iniciando logo após o seu final

9.5. *try-catch-finally*

(1) Abaixo, são apresentados os estilos de formatação recomendados:

```
try {
  comandos;
}
catch (ExceptionClass ex) {
   comandos;
}
```

```
try{
   comandos;
}
catch (ExceptionClass ex) {
   comandos;
}
finally {
   comandos;
}
```

(2) As sentenças devem ter o "{" iniciando logo após o seu final.

9.6. *Documentação*

(1) As declarações de blocos são comentadas no estilo em linha após o caracter de fechamento de bloco "**}**".

Abaixo, são apresentados exemplos de uso do comentário de blocos:

- ```
  if (condição) {
      ...
  } // fim do if (condição)
  ```

- ```
  switch (variável) {
      ...
  } // fim do switch (variável)
  ```

- ```
  for (exp1; exp2; exp3) {
      ...
  } // fim do for(exp1; exp2; exp3)
  ```

10. Espaçamento e Indentação

(1) Quatro (4) espaços em branco são usados como unidade de indentação[4].

10.1. *Tamanho da linha*

(1) As linhas têm menos de 80 caracteres para facilitar impressão e visualização do código.

10.2. *Quebra de Linha*

(1) Quando uma expressão não cabe em uma linha, as seguintes regras são utilizadas:

- A linha é quebrada depois de uma vírgula.
- A linha é quebrada antes de um operador.
- A nova linha é alinhada com o começo da expressão do mesmo nível da linha anterior.

Seguem alguns exemplos de chamadas de métodos em que é necessário utilizar mais de uma linha:

[4] Geralmente, isso pode ser automatizado pela IDE
  ```

- **this.someMethod**(longExpression1, longExpression2, longExpression3,
  longExpression4, longExpression5);

- var = **this.someMethod1**(longExpression1,
  someMethod2(longExpression2,
  longExpression3));

Seguem alguns exemplos de expressões aritméticas onde é necessário utilizar mais de uma linha:

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

(2) A quebra de linha na condição do comando if**,** na declaração de métodos e na declaração de classes, utiliza uma indentação de **8** espaços, conforme ilustrado nos exemplos abaixo.

//Não use está indentação:

```
if((condition1 && condition2)
 || (condition3 && condition4)
 || !(condition5 && condition6)) {
        DoSomethingAboutIt();
}
```

//use está :

```
if((condition1 && condition2)
      || (condition3 && condition4)
      || !(condition5 && condition6)){
            DoSomethingAboutIt();
}
```

//ou então:

```
if ((condition1 && condition2) || (condition3 && condition4)
        || !(condition5 && condition6)) {
              DoSomethingAboutIt();
}
```

// Ao invés de usar:

```
private static synchronized int getQuantidadeEstoque(Produto
                                                produto,DateateData){
```

//deve-se usar:

```
private static synchronized int getQuantidadeEstoque(
            Produto produto, Date ateData ) { ...
```

10.3. *Linhas em Brancos*

(1) Tem-se duas (**2**) linhas em branco nas seguintes:

- · Entre os imports e os comentários da classe.
- · Entre a declaração da classe e o bloco das variáveis de instância e estáticas.

(2) Usa-se uma (**1**) linha em branco nas seguintes situações:

- Entre o bloco de comentários do cabeçalho do arquivo e a cláusula package.
- Entre a cláusula package e a import.
- Entre declarações de métodos.
- Entre as variáveis locais de um método e a primeira sentença.
- Antes de um comentário.
- Entre blocos lógicos dentro do corpo de um método.
- Entre a última declaração de método da classe e o final da classe ("}").

10.4. *Espaços em Branco*

(1) Uma palavra reservada seguida por parênteses é separada destes por um espaço em branco:

- While (true) { // sem espaço entre o **while** e o **(true)**
  comandos;
  }

(2) Um espaço em branco aparece depois de cada vírgula em uma lista de parâmetros ou na inicialização dos elementos de um array:

- umObjeto = metodo(argumento1, argumento2);

(3) Todos os operadores binários, exceto o ".", são separados dos operandos por um espaço em branco, conforme os exemplo abaixo:

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

As expressões de um comando *for* são separadas por espaços em branco. Exemplo:

- for (exp1; exp2; exp3){
  comandos;
  }

(4) Um c*ast* é seguido por um espaço em branco, como em

- umaMoto = (Motocicleta) objeto.getVeiculo()

11. Apêndice A – Principais construções do Javadoc

Esta seção descreve as principais construções do padrão Javadoc. Abaixo, segue uma tabela com as principais construções, as situações de uso e suas utilidades.

Cláusula	Situação	Utilidade
@author nome	classes e interfaces	Indica o autor de uma classe ou interface. Caso a classe ou interface possua mais de um autor, deve ser criada uma entrada diferente para cada autor.
@deprecated	classes e métodos	Indica que o método, interface ou classe foi depreciado e não deve ser mais usado. Deve ser uma prática utilizar esta cláusula antes de retirar métodos da interface pública ou protegida das classes, evitando que outras classes façam referência para estes métodos.
@exception [nome] [descrição]	métodos	Descreve as exceções que um determinado método pode levantar.
@return [descrição]	métodos	Descreve o retorno do método e, se necessário, como deve ser usado.
@since	classes e métodos	Indica a quanto tempo o método ou classe existe.
@see nome	classes e interfaces	Cria um *hyperlink* para a classe ou interface referenciada como parâmetro. Se a estiver em um outro pacote, é necessário qualificar completamente o seu nome.
@see nome#metodo	métodos e variáveis de instância e de classe	Cria um *hyperlink* com um método, variável de instância ou de classe. Não é necessário se preocupar com a criação de âncoras HTML nos métodos, pois isto já é feito pelo Javadoc.
@version texto	classes e interfaces	Indica a versão de uma classe ou interface.

12. Apêndice B – Exemplo de uma classe documenta em Java

Este exemplo tem como objetivo ilustrar algumas práticas recomendadas no padrão de codificação Java.

```
/*
* Projeto: CESAR, Vestibular Interativo
*
* Tipo: Usuario
*
* Marca Registrada (c) 1996-2000 CESAR
* Av. Professor Luis Freire, s/n, Cidade Universitária, Recife-PE
* CEP 50740-540, BRAZIL
* Todos os direitos reservados.
*
* Este software é confidencial e propriedade intelectual do
* CESAR – Centro de Estudos e Sistemas Avançados do Recife.
* Você não deve utilizar indevidamente este produto em desacordo com o
* contrato estabelecida com a empresa.
*/
package covest.usuarios;
import covest.util.*;
/**
* Usuario é um tipo que representa o comportamento de um usuário
* do sistema. Um usuário pode ser um aluno, uma escola ou um professor,
* todos envolvidos com o processo do vestibular interativo.
*
* Exemplo de Uso:
* Usuario usuarioJoao = new Aluno(“jbd”, “ugh99”, “Joao Barbosa Dueire”,
* “jbd@yol.com.br”,
* “Av. Prof. Luis Vanderlei, 449, Rio de Janeiro- RJ”);
* usuarioJoao.equals(usuario);
*
* @author Marcos Silveira
*
* @see covest.usuarios.Aluno
* @see covest.usuarios.Professor
* @see covest.usuarios.Escola
*/
public abstract class Usuario {

        private String login;
        private String senha;
        private String nome;
        private String email;
        private Endereco endereco;
```

```
/**
* Construtor da classe Usuario.
*
* @param login login do usuário no sistema
* @param senha senha do usuário no sistema
* @param lembrete lembrete para senha no sistema
* @param nome nome do usuário
* @param email endereço de email do usuário
* @param endereco Endereço para contato
*
*/
public Usuario(String login, String senha, String lembrete,
                String nome, String email, Endereco endereco) {
        this.login = login;
        this.senha = senha;
        this.lembrete = lembrete;
        this.nome = nome;
        this.email = email;
        this.endereco = endereco;
}

/**
* Seta o login do usuário
*
* @param Uma String contendo a senha do usuário
*
*/
public void setLogin(String login) {
        this.login = login;
}

/**
* Retorna o login do usuário
*
* @return Uma String contendo a senha do usuário
*
*/

public String getLogin() {
        return login;
}

/**
* Retorna a senha do usuário
*
* @return Uma String contendo a senha do usuário
*
*/
public String getSenha() {
        return senha;
```

```
}

/**
* Seta a senha do usuário
*
* @param Uma String contendo a senha do usuário
*
*/
public void setSenha(String senha) {
        this.senha = senha;
}

/**
* Retorna o nome do usuário
*
* @return Uma String contendo o nome do usuário
*
*/
public String getNome() {
        return nome;
}

/**
* Seta o nome do usuário
*
* @param Uma String contendo o nome do usuário
*
*/
public void setNome(String nome) {
        this.nome = nome;
}

/**
* Retorna o email do usuário
*
* @return Uma String contendo o endereço de email do usuário
*
*/
public String getEmail() {
        return email;
}

/**
* Seta o email do usuário
*
* @param Uma String contendo o endereço de email do usuário
*
*/
public void setEmail(String email) {
        this.email = email;
```

```
	}

	/**
	* Retorna o endereço de contato do usuário
	*
	* @return Uma String contendo o endereço de contato do usuário
	*
	*/
	public Endereco getEndereco() {
		return endereco;
	}

	/**
	* Seta o endereço de contato do usuário
	*
	* @param Uma String contendo o endereço de contato do usuário
	*
	*/
	public void setEndereco(Endereco endereco) {
		this.endereco = endereco;
	}

	/**
	* Compara se dois usuários são o mesmo objeto
	*
	* @param Uma String contendo o endereço de contato do usuário
	*
	* @return true se os dois objetos são o mesmo ou se os dois forem
	* nulos, caso contrário, retornará falso.
	*/
	public boolean equals(Usuario usuario) {
		return Funcao.equals(this,usuario);
	}
}
```

13. Apêndice C - Convenção de nomeação de código

13.1. – *Convenção de nomeação de variáveis*

Tipo	Prefixo
Array	arr
Boolean	bln
Byte	byt
Char	chr
Datetime	dtm
Decimal	dec
Double	dbl
Integer	int
Long	lng
Object	obj
Short	shr
Single	sng
String	str

13.2. *Componentes*

Os componente visuais oferecidos no pacote AWT (*Abstract Windowing Toolkit*), *Swing* e outros pacotes de terceiros são amplamente usados na elaboração de interfaces gráficas em Java. Visando padronizar a declaração destes componentes e manter a uniformidade do código, a tabela abaixo relaciona mnemônicos que representam tipos de componentes de interface gráfica freqüentemente usados na elaboração de formulários, caixas de diálogo e outros componentes visuais

Tipo	Prefixo
Button	btn
Checkbox	chk
ComboBox	cmb
Edit	edt
Frame	frm
Image	img
Label	lbl
Listbox	lst
Panel	pnl
PopupMenu	pop
ProgressBar	pgb
Radiobutton	rbt
RadioButton	rdb
Table	tbl
Timer	tmr

14. Referências

Scott W. Ambler. **The Elements of Java Style.** Disponível em: http://www.ambysoft.com/books/elementsJavaStyle.html />. Acesso em : 19/05/2005

_____;**Writing Robust Java Code** <http://www.ambysoft.com/downloads/javaCodingStandards.pdf/> Acesso em : 02/03/2005

SIERRA, kathy: BATES, Bert. **Certificação Sun para programadores e desenvolvedor Java 2.** Rio de Janeiro: Editora Altas Books. 2003.