Os padrões de implementação aqui definidos proporcionam um guia útil para manter a qualidade e a consistência do código em projetos de ciência da computação em Java, contribuindo para um desenvolvimento mais eficiente e colaborativo. É essencial que a equipe de desenvolvimento esteja ciente e adote esses padrões de forma consistente.

Esse documento contém um conjunto de regras e padrões que têm como objetivo principal garantir a integridade, legibilidade e padronização do código do projeto.

Padronização de código e implementação Java

Pacotes e Importações

Os padrões a serem seguidos para importações de pacotes são:

Importações Explícitas:

- Sempre importe classes de forma **explícita**, evitando o uso de importações generalizadas(*). Isso torna o código mais legível e fácil de entender.
    - Exemplo: utilize “ **import java.util.ArrayList;** ” frente a “ import java.util.*; ”
- Agrupe importações por pacote para facilitar a leitura. Comece importando as bibliotecas padrão, depois as bibliotecas de terceiros e, por fim, as classes internas.
- Remova importações que não estão sendo usadas. Isso mantém o código limpo e fácil de entender.
- Evite importações que possam conflitar com nomes existentes no seu código. Use nomes completos ou aliases (“ import com.exemplo.pacote.Classe as NomeExemplo ”) quando necessário.
- Organize as importações de forma clara, agrupando-as por pacotes e em ordem alfabética, de preferência.
- Exemplo:
    -
```
import java.util.ArrayList;
import java.util.List;

import com.exemplo.pacote.ExemploClasse;
```

Indentação, Formatação e Espaçamento

- Indentação e Formatação:
  - Use uma convenção consistente para a indentação (‘*tab’*). Formate o código de forma organizada e consistente para facilitar a leitura. Utilize chaves ‘{}’ mesmo para blocos de código de uma única linha.

- Uso de Espaços em Branco:
  - Use espaços em branco de forma consistente para melhorar a legibilidade. Deixe espaços em torno de operadores e após vírgulas.
- Sem espaço entre um método e o parênteses e o nome do método "(" início de lista de parâmetros;
- Após qualquer comentário JavaDoc adicione uma linha em branco;
- Não é necessário adicionar uma linha em branco após declaração de uma classe;

- Abertura da chaves "{" aparece no fim da mesma linha que foi declarado o código;
- Fechamento da chaves "}" começa uma linha alinhada no conjunto do método a qual foi criada, exceto quando há códigos em parte em branco(vazio) ou nulo }"devendo aparecer imediatamente depois de aberto com "{"
- Métodos são sempre separados por uma linha em branco.

- Utilizar o ‘*tab’* ao invés do espaçamento múltiplo.
- Utilizar espaço entre operadores e operandos.
- Utilizar espaço após vírgulas.

- Evitar a criação de linhas muito longas (mais de 80 caracteres).
  - Em caso de linhas longas: Deve-se quebrar após vírgulas e parênteses.
  - No caso de operadores lógico-aritméticos: Sempre quebrar antes do operador, evitando que se quebre a legibilidade de caracteres envolvidos na operação.
- Exemplo:
  -

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

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

    int emptyMethod() {}

    ...
}
```

Comentários - JavaDoc

Comentários

- Faça uso extensivo de comentários para explicar o código complexo, algoritmos, intenções e possíveis melhorias.
- Escreva comentários claros e concisos, mas evite comentários óbvios ou redundantes.
- Comentários simples, que utilizam uma linha, podem ser inseridos com “ // ”. Em geral, comentários específicos e resumidos.
- Evite comentários ao lado de uma linha de código, ou seja, dê preferência a comentários isolados em uma única linha.

JavaDoc

- **Todos** os arquivos, classes e métodos devem ser documentados conforme padronização exigida. O objetivo é obter um arquivo JavaDoc que aborde todo o código criado, facilitando muito a legibilidade do código e a interpretação do mesmo.
- As possíveis *tags*:

Tag	Significado
`@author`	Especifica o autor da classe ou do método em questão.
`@deprecated`	Identifica classes ou métodos obsoletos. É interessante informar nessa tag, quais métodos ou classes podem ser usadas como alternativa ao método obsoleto.
`@link`	Possibilita a definição de um link para um outro documento local ou remoto através de um URL.
`@param`	Mostra um parâmetro que será passado a um método.
`@return`	Mostra qual o tipo de retorno de um método.
`@see`	Possibilita a definição referências de classes ou métodos, que podem ser consultadas para melhor compreender idéia daquilo que está sendo comentada.
`@since`	Indica desde quando uma classe ou métodos foi adicionado na aplicação.
`@throws`	Indica os tipos de exceções que podem ser lançadas por um método.
`@version`	Informa a versão da classe.

- Exemplo descrição de classe:

```
/**  Classes  que  herda  a  classe  abstrata  Usuário  e  implementa  as
funcionalidades do Aluno
 * @author Aroldo Augusto Barbosa Simões - 4250
 * @since 09/11/2022  - 22:00
 * @version 1.2
 */

public class Aluno extends Usuario {
```

- Exemplo descrição de método:

```
 /** Método Aluno, construtor da classe Aluno
     * @author Aroldo Augusto Barbosa Simões - 4250
     * @param nome String - Nome do Aluno
     * @param matricula int - Numero da Matricula
     * @param senha String - Senha de acesso
     * @since 01/11/2022 - 20:17
     */

    public Aluno(String nome, String matricula, String senha) {
        super(nome, matricula, senha, TipoUsuario.ALUNO);
    }
```

Nomenclatura

- Convenções de **Nomenclatura**:
  - Utilize nomes significativos para variáveis, métodos, classes e pacotes. Use camelCase para nomes de variáveis e métodos (por exemplo, nomeVariavel), e PascalCase para nomes de classes (por exemplo, MinhaClasse).

Métodos

- Tamanho e Complexidade do Método:
  - Mantenha os métodos concisos e focados em uma tarefa específica. Evite métodos muito longos ou altamente complexos. Se um método estiver crescendo demais, considere dividi-lo em métodos menores.
- Sintaxe:
  - **{[A.Z][a.z]}()** → **xxxXxxx()**, onde "xxxXxxx" define o nome do método.
- Letras minúsculas exceto em novas palavras internas.
- Não utilizar hífen (-) ou subtraço (_).
- Utilizar verbos de preferência seguidos de substantivos.
- Bons exemplos:
  - comprar() / comprarProduto().
- Maus exemplos:
  - Compra() / vender_PRODUTO()

Classes

- Sintaxe:
  - **class {[A.Z][a.z]}** → **class XxxxXxxx**, onde "Xxxx" indica o nome da classe.
- Começar com letra maiúscula, seguida de letras minúsculas, exceto no início de novas palavras.
- Não utilizar hífen (-) ou subtraço (_).
- Evitar o uso de abreviações, deixando o nome mais descritivo possível.
- Bons exemplos:
  - class Produto / class ProdutoVendido.
- Maus exemplos:
  - class produto / class PRODUTO_vend

Variáveis

- Uso de Variáveis:
  - Minimize o escopo das variáveis, declarando-as o mais próximo possível de onde serão usadas. Evite variáveis globais, a menos que sejam estritamente necessárias.

- Sintaxe:
  - **tipo {[A.Z][a.z]} → tipo xxxXxxx**, onde “xxxXxxx” define o nome da variável.
- Letras minúsculas exceto em novas palavras internas.
- Não utilizar hífen (-) ou subtraço (_).
- Nomes curtos e com significado claro.
- Evitar nomes com uma letra, excepto se a variável tem um âmbito reduzido. Isto é o caso dos índices utilizados no contexto de iteradores - i, j, k, por exemplo.
- Bons exemplos:
  - int i / char sexo.
- Maus exemplos:
  - int numero / float saldo_Caixa.

Constantes

- Use a palavra-chave final para declarar constantes (por convenção, com letras maiúsculas e sublinhados para separar palavras, por exemplo, MINHA_CONSTANTE).
- Sintaxe:
  - **tipo {[A.Z]} → tipo XXX_XXX** onde “XXX_XXX” define o nome da constante.
- Letras maiúsculas e palavras separadas por subtraço (_) .
- Bom exemplo:
  - static final int ALTURA_MIN = 10

Pacotes

- Sintaxe:
  - **import {[a.z]} → import xxx.xxx.xxx**…, onde “xxx” define o nome do pacote.
- O prefixo do nome do pacote deve ser único sempre ser escrito em letras minúsculas.
- Deve ser um dos nomes de domínio de nível superior (edu, gov, mil, net, org, com).
- Componentes posteriores do nome do pacote variam de acordo com convenções de nomenclatura internas.
- Exemplo:
  - com.sun.eng / com.apple.quicktime.v2.

Composição de Classes

- Para a declaração de uma classe, as seguintes configurações devem ser aplicadas:

1. Documentação;
    a. Comentário sobre a classe;
2. Declaração;
3. Variáveis
    a. Devem ser declaradas seguindo a ordem Public, Protected e Private.
4. Instâncias;
5. Construtores;
6. Métodos;
    a. Agrupados por nível de funcionalidade.
    b. Getters e Setter em primeiro por convenção.

Exemplo

```java
package br.ufv.caf.modelo;

/** Classes que herda a classe abstrata Usuário e implementa as funcionalidades do Aluno
 * @author Aroldo Augusto Barbosa Simões - 4250
 * @since 09/11/2022  - 22:00
 * @version 1.2
 */

public class Aluno extends Usuario {
    private int variavelExemplo;

    /** Método Aluno, construtor da classe Aluno
     * @author Aroldo Augusto Barbosa Simões - 4250
     * @param nome String - Nome do Aluno
     * @param matricula int - Numero da Matricula
     * @param senha String - Senha de acesso
     * @since 01/11/2022 - 20:17
     */

    public Aluno(String nome, String matricula, String senha) {
        super(nome, matricula, senha, TipoUsuario.ALUNO);
    }
```

```
    /** Método validaMatricula, tem a finalidade de verificar o formato
da matricula
     * @author Aroldo Augusto Barbosa Simões - 4250
     * @return boolean
     * @since 09/11/2022 - 22:00
     */

    @Override
    public boolean validaMatricula() {
        return this.getMatricula().matches("\\d{4}");
    }

    /** Método validaMatricula, tem a finalidade de verificar o formato
da senha
     * @author Aroldo Augusto Barbosa Simões - 4250
     * @return boolean
     * @since 09/11/2022 - 22:00
     */

    @Override
    public boolean validaSenha() {
        return this.getSenha().matches(".{4,}");
    }
}
```

Boas Práticas

- **Estruturação de arquivos deve estar de acordo com o padrão de projeto definido pela equipe!**

- Ciclo de Vida de Desenvolvimento:
    - Siga o ciclo de vida de desenvolvimento do projeto, para garantir entregas incrementais e melhor colaboração da equipe. Além disso, siga os padrões de projeto.
- Revise seu próprio código.
- Aplique padrões de projeto apropriados para melhorar a manutenibilidade e a escalabilidade do código.
- Minimize o acoplamento entre classes e módulos. Siga o princípio de "baixo acoplamento, alta coesão".
- Use estruturas de controle (if, switch, loops) de forma eficiente e lógica, tornando o código mais claro e fácil de entender.
- Faça o tratamento adequado de exceções usando blocos try-catch ou lançando exceções, dependendo do contexto. Não ignore exceções.
- Iteração sobre Coleções:
    - Use for-each para percorrer coleções sempre que possível, pois é mais conciso e legível.

Referências

- ChatGPT. Disponível em: <https://chat.openai.com>. Último acesso em 10 de outubro de 2023.
- CONVENÇÕES DE CÓDIGO EM JAVA. Devmedia, 2012. Disponível em: <https://www.devmedia.com.br/convencoes-de-codigo-java/23871#:~:text=As%20convenções%20de%20nomenclatura%20tem,útil%20na%20compreensão%20do%20código.>. Último acesso em: 10 de outubro de 2022.
- JAVADOC - IMPLEMENTANDO DOCUMENTAÇÃO ATRAVÉS DO NETBEANS. Disponível em: <https://www.devmedia.com.br/javadoc-implementando-documentacao-atraves-do-netbeans/2495>. Último acesso em: 10 de outubro de 2023.