Estándar de codificación

para

Java

Estándar de codificación

VERSION 0.1 PROPUESTA

Equipo 3
Jose Luis Padilla Cruz
Angel Adrian Camal Garcia
Raul Orlando Sayago Martinez
PROCESOS DE CONSTRUCCIÓN DE SOFTWARE | UNIVERSIDAD VERACRUZANA

Tabla de contenido

1. Introducción

1.1. Justificación

El siguiente documento sirve como la definición del **estándar de codificación** para el Equipo 3 de la experiencia educativa de **Procesos de Construcción de Software** de la Universidad Veracruzana, en la Facultad de Estadística e Informática. Para que el código fuente sea considerado perteneciente al Equipo 3 de Procesos de Construcción de Software, esta misma debe seguir los estándares de codificación expuestos en este **documento** en las secciones posteriores.
Este documento está disponible para todas las personas, cuyo objetivo sea el de utilizar este estándar de codificación o que realicen alguna revisión o modificación a cualquiera de los códigos fuente, que empleen este estándar de codificación.
El presente estándar está **basado** en el lenguaje de programación **Java**.

Ángel Adrián Camal García – José Luis Padilla Cruz – Raúl Orlando Sayago Martínez

2. Estructura de archivo fuente

2.1. Nombre del archivo

Los nombres de los archivos fuente deben consistir en las siguientes partes: **nombre representativo** + **extensión ".java".**

```
Scanner.java
```

Ejemplo 1.0

2.2. Declaraciones de paquete

Los paquetes son una colección de clases relacionadas, cuya función es ayudar a **organizar** las **clases** en una estructura de carpetas. Es importante que estén al inicio, en la **primera línea** de código del archivo java.

```
1    package com.org.gui;
2    …
3    …
```

Ejemplo 2.0

2.3. Importación de bibliotecas

Para importar bibliotecas propias o externas, se debe de tener identificados únicamente las bibliotecas necesarias, que se requieran o se vayan a ocupar. **No** se hace uso de **importaciones comodines**, que pueden generar diversos problemas. **No** se debe hacer uso de **importaciones estáticas**. Debe seguir a la sentencia 2.2 Declaraciones de paquete.

```
1    package com.org.gui;
2
3    import java.utils.List;
4    import java.utils.Map;
5    import java.utils.HashMap;
```

Ejemplo 3.0 – **Forma correcta**

```
1    package com.org.gui;
2
3    import java.utils.*;
4    import javax.swing.*;
5    import java.awt.*;
```

Ejemplo 3.1 – **Forma incorrecta**

2.4. Declaraciones de las clases e interfaces

Las **clases** deben declararse en **archivos individuales** con un nombre de archivo representativo, además el **nombre de archivo** debe **coincidir** con el **nombre de la clase.** Las **clases privadas** secundarias pueden declararse, como clases internas y residir en el archivo de la clase a la que pertenecen.
Además, el contenido de las clases se debe organizar de la siguiente manera:

- Documentación de la Clase o Interfaz.
- Sentencia ***class*** o ***interface.***
- Variables de clase (estáticas), en el orden ***public, protected, default*** (sin modificador de acceso), ***private***.
- Variables de instancia, en el orden **public**, ***protected***, ***default*** (sin modificador de acceso), ***private.***
- Constructores.
- Métodos.

2.4.1. Constructores y métodos

Aquellos constructores y métodos que estén **sobrecargados** deben aparecer **secuencialmente**, sin ningún otro código intermedio.

2.4.2. Variables

Las variables deberían **inicializarse donde se declaran** y pueden **declararse en el ámbito más pequeño posible**. Esto asegura que las variables son válidas en todo momento. A veces es imposible inicializar una variable con un valor válido donde se le declara. En esos casos debería dejarse sin inicializar en vez de darle un valor sin sentido.

2.4.3. Sobrecarga de constructores y métodos

***

```
1       package com.org.gui;
2
3       import java.utils.List;
4       import java.utils.Map;
5       import java.utils.HashMap;
6
7       public class Tetris extends JFrame {
8                       private static final long serialVersionUID = -4722429764792514382L;
9                       private boolean isGameOver = false;
10
11                      public Tetris() {
12                      }
13
14                      public Tetris(String Title) {
15                      }
16
17                      private void resetGame() {
18                      }
19
```

```
20                  private void spawnPiece() {
21                  }
22
23      }
24
```

Ejemplo 4.0

2.5. Derechos de autor

Todo el archivo fuente de Java **debería contener la notificación de derechos de autor** al **inicio** del archivo fuente que más se **adapte**. por ejemplo: Apache License 2.0, MIT, BSD License 2, GPLv2, GPLv3, entre otros.

2.5.1. Fuente abierta (Open Source)

Las licencias de fuente abierta (Open Source) permiten el **acceso** a su **código** de programación, lo que facilita **modificaciones** por parte de otros **programadores ajenos** a los creadores originales del software en cuestión.

```
1   /*
2   *  Copyright (C) {year} Equipo 3, LLC
3   *  http://equipo3.pcs.mx.com
4   *  equipo3.pcs@pcs.com
5   *
6   *  Licensed under the Apache License, Version 2.0 (the “License”);
7   *  you may not use this file except in compliance with the License.
8   *  You may obtain a copy of the License at
9   *
10  *  http://www.apache.org/licenses/LICENSE-2.0
11  *
12  *  Unless required by applicable law or agreed to in writing, software
13  *  distributed under the License is distributed on an “AS IS” BASIS,
14  *  WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
15  *  See the License for the specific language governing permissions and
16  *  limitations under the License.
17  */
```

Ejemplo 5.0

2.5.2. Propietario

Si el proyecto **no** está destinado a ser de **fuente abierta** (Open Source) **debe** tener las características del siguiente ejemplo:

```
1   /*
2   *  Copyright (C) {year} Equipo 3, LLC
3   *  http://equipo3.pcs.mx.com
4   *  equipo3.pcs@pcs.com
5   *  All rights reserved
6   */
```

Ejemplo 5.1

2.5.3. Terceras personas

Código escrito por **terceras personas** debe ser **formateado** de la siguiente manera

```
/*
*  Copyright (C) {year} {terceros}
*  {urlTerceros}
*  All rights reserved
*  Developed for {terceros} by:
*
*  Equipo 3, LLC
*  http://equipo3.pcs.mx.com
*  equipo3.pcs@pcs.com
*/
```

Ejemplo 5.2

3. Formato

3.1. Una sentencia por línea

Cada **declaración** es **seguida** por un **salto de línea**

3.2. Sangría de bloques

Cada vez que se abre **un nuevo bloque** o construcción similar a un bloque, la **sangría** debe **aumentar (una tabulación).** Cuando finaliza el bloque, la sangría **vuelve** al nivel de **sangría anterior**. El siguiente es un ejemplo de como se debe realizar.

```
package com.org.panel;

import java.awt.Color;
import java.awt.Dimension;
import javax.swing.JPanel;

public class Tetris extends JPanel {
        private static final int COL_COUNT = 10;
        private static final int ROW_COUNT = 10;
        private TileType[][] tiles;

        public Tetris() { … }

        public void clear() {
                for(int i = 0; i < ROW_COUNT; i++){
                        for(int j = 0; j < COL_COUNT; j++){
                                tiles[i][j] = null;
                        }
                }
        }

}
```

Ejemplo 6.0

3.2.1. Bloques de código: Estilo K & R

Las llaves siguen el estilo Kernighan y Ritchie (“Llaves egipcias”) para **bloques no vacíos** y construcciones similares a bloques:

- **Sin salto** de línea **antes** de la **llave de apertura.** Se **agrega** un **espacio en blanco** antes de las llaves de apertura «{».
- **Salto** de línea **después** de la **llave de apertura**.
- **Salto** de línea **antes** de la **llave de cierre**.
- **Salto** de línea **después** de la **llave de cierre**, **solo si** esa llave **termina una instrucción** o termina el cuerpo de un método, constructor o clase con nombre. Por ejemplo, no hay salto de línea después de la llave si es seguido por otra cosa o una coma.

```
1           if(!isGameOver()){
2                 try {
3                    archivo = new File(“dir.txt”);
4                    …
4                 } catch (Exception e) {
5                    e.printStackTrace();
6                 } finally {
7                    if(archivo != null) {
8                       archivo.close();
9                    }
10                }
11          }
```

Ejemplo 7.0

3.2.2. Bloques de código vacíos

Un **bloque vacío** o una construcción similar a un bloque **puede estar** en **estilo K&R** (como se describe en el apartado 3.2.1). Alternativamente, **puede cerrarse inmediatamente después de abrirse**, sin caracteres o saltos de línea entre ({}), **a menos** que sea **parte** de una **declaración de bloques múltiples** (una que contenga directamente bloques múltiples: if / else o try / catch / finally).

```
1       // Aceptable
2       void hacerNada(){}
3
4       // También es aceptable
5       void hacerNadaDeNuevo(){
6        }
7
8       // No es aceptable
9       try{
10            determinarCosto();
11      } catch(ArithmeticException e) {}
```

Ejemplo 8.0

3.3.Line-wrapping

Se puede hacer su uso en comentarios ya sea de JavaDocs.

3.4.Comentarios

Los comentarios en el código fuente de Java, resultan importantes para poder llevar a cabo el mantenimiento a largo plazo de este. Para ello en este estándar se requiere lo siguiente:

- **Idioma:** Todos los comentarios pueden **escribirse** únicamente en **dos idiomas**, **español e inglés**. Dependerá del **acuerdo** que se lleve a cabo entre los **integrantes** de algún **proyecto** que deseen utilizar este estándar de codificación.

- **Uso de doble paréntesis:** Todos los comentarios que **no pertenezcan a JavaDoc** deben hacer uso de //.

- **Sangría de los comentarios:** Los comentarios deben tener la **sangría relativa** a su posición en el código ya que ayuda a evitar que los comentarios quiebren la estructura lógica del programa.

```
1               package com.org.panel;
2
3               import java.awt.Color;
4               import java.awt.Dimension;
5               import javax.swing.JPanel;
6
7               public class Tetris extends JPanel {
8                       private static final int COL_COUNT = 10;
9                       private static final int ROW_COUNT = 10;
10                      private TileType[][] tiles;
11
12                      public Tetris() { … }
13
14                      public void clear() {
15                              for(int i = 0; i < ROW_COUNT; i++) {
16                                      for(int j = 0; j < COL_COUNT; j++){
17                                              // Asignar NULL
18                                              tiles[i][j] = null;
19                                      }
20                              }
21                      }
22              }
```

Ejemplo 9.0

```
1               package com.org.panel;
2
3               import java.awt.Color;
4               import java.awt.Dimension;
5               import javax.swing.JPanel;
6
7               public class Tetris extends JPanel {
8                       private static final int COL_COUNT = 10;
9                       private static final int ROW_COUNT = 10;
10                      private TileType[][] tiles;
11
12                      public Tetris() { … }
13
14      //              // El método causa algunos errores…
15      //              public void clear(){
16      //                      for(int i = 0; i < ROW_COUNT; i++){
17      //                              for(int j = 0; j < COL_COUNT; j++){
18      //                                      tiles[i][j] = null;
19      //                              }
20      //                      }
21      //              }
22              }
```

Ejemplo 9.1

3.5. JavaDocs

Con JavaDoc han de usarse **etiquetas de HTML** o ciertas **palabras reservadas** precedidas por el carácter "**@**". Estas etiquetas se escriben al **principio** de cada **clase**, **miembro** o **método**, dependiendo de qué objeto se desee describir, mediante un comentario iniciado con "/**" y acabado con "*/".

Tag	Descripción	Uso	Versión
@author	Nombre del desarrollador.	nombre_autor	1.0
@version	Versión del método o clase.	Versión	1.0
@param	Definición de un parámetro de un método, es requerido para todos los parámetros del método.	nombre_parametro descripción	1.0
@return	Informa de lo que devuelve el método, no se puede usar en constructores o métodos "void".	Descripción	1.0
@throws	Excepción lanzada por el método, posee un sinónimo de nombre @exception	nombre_clase descripción	1.2
@see	Asocia con otro método o clase.	referencia (#método(); clase#método(); paquete.clase; paquete.clase#método()).	1.0
@since	Especifica la versión del producto	indicativo numerico	1.2
@serial	Describe el significado del campo y sus valores aceptables. Otras formas válidas son @serialField y @serialData	campo_descripcion	1.2
@deprecated	Indica que el método o clase es antigua y que no se recomienda su uso porque posiblemente desaparecerá en versiones posteriores.	Descripción	1.0

Tabla JavaDocs 1.0

```
1    /***
2    * Retorna un número entero, que corresponde a la posición del número de la serie de fibonacci.
3    * El parámetro N debe ser un número entero.
4    * @param N     la posición de la serie de fibonacci
5    * @return      el valor de la posición de la serie de fibonacci
6    */
7    public static int fib(int N) {
8              double goldenRatio = (1 + Math.sqrt(5)) / 2;
9              return (int)Math.round(Math.pow(goldenRatio, N)/ Math.sqrt(5));
10   }
11
```

Ejemplo 10.0

3.6.Declaración de variables

Cada declaración de variable se **declara** únicamente en una **sola variable**. Declaraciones tales como **int** a,b, solo se usan en los encabezados de ciclos for.

3.7.Enums

Los “enums” son **clases especiales** de java que representan un **conjunto de variables constantes**. Se debe usar tipos de enumeración cada vez que necesite representar un conjunto fijo de constantes. “Eso incluye tipos de **enumeración naturales**, como los planetas de nuestro sistema solar y conjuntos de datos donde **conoce todos los valores posibles** en el momento de la compilación” (Oracle, s.f.). La clase enum, puede formatearse opcionalmente como si fuera una construcción similar a un bloque.

```
1      public enum Nivel {
2              FACIL, MEDIO, DIFICIL, IMPOSIBLE
3      }
4
4      public enum Nivel {
5            FACIL,
6            MEDIO,
7            DIFICIL,
8            IMPOSIBLE
9          }
10
11         new int[] Nivel {
12            FACIL, MEDIO,
13            DIFICIL, IMPOSIBLE
14         }
15
```

Ejemplo 23.0 – Ejemplos válidos

3.8.Arrays

Cualquier inicializador de matriz puede formatearse opcionalmente como si fuera una “construcción similar a un bloque”. Por ejemplo, los siguientes ejemplos son todos válidos:

```
1       new int[] array = {0,1,2,3}
2
3       new int[] array = {
4         0,
5         1,
6         2,
7         3
8       }
9
10      new int[] array = {
11        0,1,
12        2,3
13      }
14
15      new int[] array =
16      {0,1,2,3}
17
```

Ejemplo 11.0 – Ejemplos válidos

3.9.Sentencia switch

Las sentencias Switch deben presentar las siguientes características:

- **Sangría:** el contenido de los bloques switch, debe tener una **sangría (+1)** como cualquier otro bloque de código.
- **El caso default debe estar presente:** cada bloque switch, debe **contener** el caso **default**, incluso si este mismo no contiene código.
- **Comentario en continuación:** en el bloque switch, cada grupo de instrucciones **finaliza abruptamente** (break, continue, return) o se **marca** con **un comentario** para indicar que la ejecución continuará o podría continuar en el siguiente **grupo de instrucciones**. Cualquier comentario que **comunique la idea** de caída es suficiente (// fall-through). Este comentario especial no es obligatorio en el último grupo de instrucciones del bloque de conmutadores.

El ejemplo posterior es la forma correcta para este estándar:

```
1               switch(int n) {
2                     case 1:
3                     case 2:
4                         almacenarDatos();
5                         // fall-through
6                     case 3:
7                         romperCifrado();
8                         break;
9                     case 4:
10                        cambiarMetodo();
11                    default:
12              }
13
```

Ejemplo 12.0

4. Nombrado

4.1. Reglas por tipos de identificadores

4.1.1. Nombre de paquetes

Todos los nombres de los paquetes deben estar en **minúsculas** separadas por un punto.

```
1        package com.org.gui;
2        …
3        …
```

Ejemplo 13.0

4.1.2. Nombre de clases

Todos los nombres de clases deben empezar con letra mayúscula y seguir la notación **UpperCamelCase.**

```
1        // Forma incorecta
2        public class persona {
2        }
3
4        // Forma correcta
5        public class Persona {
6        }
```

Ejemplo 14.0

Las clases de prueba se denominan a partir de "**Prueba**" y el **nombre de la clase** que se está probando.

```
1        public class PruebaMapa {
2        }
```

Ejemplo 15.0

4.1.3. Nombre de métodos

Todos los métodos deben escribirse en la notación **lowerCamelCase.** Para nombrar los métodos de prueba, se debe seguir el **patrón <metodoBajoPrueba_estado>**.

```
1        public int drawImage(Graphics 2g) {…}
2
3        public class drawImage_Test() {…}
```

Ejemplo 16.0

4.1.4. Nombre de constantes

Los nombres constantes usan la notación **CONSTANT_CASE**: todas las letras mayúsculas, con cada **palabra separada** de la siguiente por un solo **guion bajo**.

```
1		private static final int TILE_SIZE = 12;
2		private static final int SHADE_WIDTH = 12;
3		private static final int TILE_COUNT = 5;
4		private static final int TILE_SIZE = 12;
5		private Tetris tetris;
```

Ejemplo 17.0

4.1.5. Nombres de campos no constantes

Los nombres de **variables** o **campos** deben estar escritos en **lowerCamelCase.**

```
1		private static BoardPanel board;
2		private sidePanel side;
3		private int score;
4		private static Clock logicTimer;
```

Ejemplo 18.0

4.1.6. Nombre de parámetros

Los nombres de los parámetros se escriben en **lowerCamelCase** y se debe **evitar** los nombres de parámetros de un **solo carácter** en métodos públicos.

```
1		private void drawTile(TileType type, int x, int y, Graphics g) { … }
2
```

Ejemplo 19.0

4.1.7. Nombre de variables locales

Los nombres de **variables** locales deben estar escritas en **lowerCamelCase.** Las variables locales, que tienen la palabra reservada final y son inmutables, no se consideran constantes y no deberían tener el estilo como una constante.

```
1		private void drawTile(TileType type, int x, int y, Graphics g) { … }
2
```

Ejemplo 20.0

4.1.8. Nombres de tipos genéricos

Las variables genéricas pueden seguir las siguientes convenciones de nombres según los criterios elegidos por los integrantes del equipo que elaboren cualquier proyecto usando esta convención:

- **E** - Elemento
- **K** - Llave (usado en mapas)
- **N** - Números

- **T** – Tipo (Representa un tipo, es decir, una clase)
- **V** - Valor (representa el valor, tambien se usa en mapas)
- **S**,**U**,**V** etc. - usado para representar otros tipos.

```
1       public class Nodo<T> {
2           private T dato;
3           private Nodo<T> siguiente;
4
5           public T getDato() {
6               return dato;
7           }
8
9           public Nodo<T> getSiguiente() {
10              return siguiente;
11          }
12
13      }
14
```

Ejemplo 21.0

5. Prácticas de programación

5.1.Comentario @override

El comentario o anotación “@override” en aquellos métodos que sean sobrescritos. Esta puede omitirse cuando el método principal ya es obsoleto: “@Deprecated”.

6. Patrones de diseño

"Los patrones de diseño pueden acelerar el proceso de desarrollo, proporcionando a prueba, los paradigmas de desarrollo probadas." Los patrones de diseño se han dividido en al **menos tres categorías**:

1. **Patrones de creación (C):**
   Los patrones de diseño creacional son los patrones de diseño que tienen que ver con los mecanismos de creación de objetos, tratando de crear objetos de una manera adecuada a la situación:
   - Abstract Factory
   - Builder
   - Factory o Factory Method
   - Prototype
   - Singleton
   - Object Pool

2. **Patrones estructurales (S):**
   Los patrones de diseño estructural son los patrones de diseño que facilitan el diseño mediante la identificación de una forma sencilla de darse cuenta de las relaciones entre las entidades:

   - Adapter
   - Composite
   - Decorator
   - Facade
   - Proxy
   - Bridge
   - Flyweight

3. **Patrones de comportamiento (B):**

   Los patrones de diseño de comportamiento son los patrones de diseño que identifican patrones comunes de comunicación entre los objetos y se dan cuenta de estos patrones. Al hacerlo, estos patrones de aumentar la flexibilidad en la realización de esta comunicación:
   - Command
   - Observer
   - Strategy
   - Chain of Responsability
   - Memento
   - Mediator
   - Template method
   - Iterator
   - Visitor
   - State
   - Interpreter

Sin embargo, solo se detallará solo uno de los patrones de diseño, el cual, es uno de los mas usados.

6.1.Singleton

Este patrón es ampliamente utilizado ya que hay multitud de ocasiones en las que se necesita que necesita **compartir alguna información** en la aplicación, tener un **único punto de acceso** a un recurso o cualquier situación en la que se necesite tener **un solo objeto de una clase**.

```
1       public class Bank {
2               private static Bank instance = new Bank();
3
4               public Bank(){}
5
6               public static Bank getInstance(){
7                       return instance;
8               }
9
```

Ejemplo 22.0

Para aplicaciones donde se hace uso de **varios hilos de ejecución**, el ejemplo anterior puede traer ciertos **inconvenientes**. Para **evitar** esos inconvenientes en nuestra aplicación o sistema con varios hilos de ejecución, debe seguir un **patrón** como en el siguiente ejemplo:

```
1       public class Banco {
2               private static Banco instancia = new Banco();
3
4               private Banco(){}
5
6               private synchronized static void crearInstancia(){
7                       if(instancia == null){
8                       instancia = new Banco();
9                       }
10              }
11
12              public static Bank getInstancia(){
13                      crearInstancia();
14                      return instancia;
15              }
16      }
```

Ejemplo 23.0