Estándar de codificación

para

Java

Estándar de codificación

Equipo 2
Jose Luis Carvajal Pacheco
Seth Noé Díaz Díaz
Midguet Arturo García Torres

1. Estructura de archivo fuente

1.1. Nombre del archivo

Los nombres de los archivos fuente deben consistir en las siguientes partes: **nombre representativo** + **extensión ".java".** El nombre representativo generalmente es el nombre de la misma clase, cuya inicial debe ser una mayúscula. Para nombres de archivo de tipo **FXML**, el nombre debe contener al final del **nombre representativo** la palabra "Vista".

```
Scanner.java

ScannerVista.fxml
```

Ejemplo 1.

1.2. 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 fuente de java.

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

Ejemplo 2.

1.3. 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, como, por ejemplo, el uso de otras clases que no se planeaban usar. **No** se debe hacer uso de **importaciones estáticas a excepción de clases de prueba**. Debe seguir a la sentencia de declaración 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. – **Forma correcta**

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

Ejemplo 4.– **Forma incorrecta**

1.4. Clases e interfaces

Las **clases** deben declararse en **archivos individuales** con un **nombre de archivo representativo**, además el **nombre de archivo representativo** 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. Si se hace uso de JavaFX con FXML, el nombre de la clase controladora debe contener al final del nombre representativo la palabra “Controller”.

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, en el orden «**public**», «***protected***», «***default***» (sin modificador de acceso), «***private***».

2.4.1. Constructores y métodos

Aquellos constructores y métodos que estén sobrecargados deben aparecer secuencialmente, sin ningún otro código intermedio. Los métodos con anotaciones «@FXML» deben aparecer respetando el orden «public», «*protected*», «*default*» (sin modificador de acceso), «*private*».

2.4.2. Variables

Las variables deben **declararse en el ámbito más pequeño posible**. Esto asegura que las variables sean válidas en todo momento. Y deben inicializarse donde se declaran, a veces es imposible inicializar una variable con un valor válido donde se le declara. En esos **casos** debería dejarse **sin inicializar**. Además, todas las variables declaradas se deben utilizar. No debe existir variables sin utilizar. Las variables de JavaFX con anotación «@FXML» deben estar después de las variables de clase y de instancia.

```
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           private boolean status;
11
12
```

```
13              public Tetris() {
14              }
15
16              public Tetris(String Title) {
17              }
18
19              private void resetGame() {
20                      status = true;
21              }
22
23              private void spawnPiece() {
24              }
25
26      }
```

Ejemplo 5.

1.5. Derechos de autor

La licencia del software **debería estar de manera externa a los archivos de código fuente o generados por el compilador.** Además, la licencia debería estar plasmado en un archivo de texto plano, que debe ubicarse junto al código fuente. Por ejemplo: Apache License 2.0, MIT, BSD License 2, GPLv2, GPLv3, o propio.

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

Ejemplo 6.

2.5.2. Propietario

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

```
/*
*  Copyright (C) {year} Estándar 12, UV
*  All rights reserved
*/
```

Ejemplo 7.

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:
*
*  Estándar 12, UV
*/
```

Ejemplo 8.

2. Formato

2.1. Una sentencia por línea

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

2.2. Sangría de bloques

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

NOTA: En caso de que el **IDE** o utilizado no permita tabular, el espacio determinado para este estándar es de 10 espacios cada vez que se abra un nuevo bloque de código.
NOTA EXTRA: La tabulación de un **IDE** a otro puede variar.

```
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() { … }

```

```
14                          public void clear() {
15                                  for(int i = 0; i < ROW_COUNT; i++){
16                                          for(int j = 0; j < COL_COUNT; j++){
17                                                  tiles[i][j] = null;
18                                          }
19                                  }
20                          }
21
22                  }
```

Ejemplo 9.– Tabulación por parte del **IDE**

```
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                                                  tiles[i][j] = null;
18                                          }
19                                  }
20                          }
21
22                  }
```

Ejemplo 10.– Tabulación de 10 espacios.

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 y en este caso debe haber un espacio en blanco.

```
1                   if( !isGameOver() ) {
2                           try {
3                              file = new File(“dir.txt”);
4                              …
4                           } catch (Exception e) {
5                              e.printStackTrace();
```

```
6                   } finally {
7                       if(file != null) {
8                           file.close();
9                       }
10                  }
11              }
```

Ejemplo 11.

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**, con un espacio en blanco 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               // Acceptable
2               void doNothing() { }
3
4               // Acceptable
5               void doNothingToo() {
6                }
7
8               // Acceptable
9               try {
10                      makeASound();
11              } catch(ArithmeticException e) {
12              }
13
14              // This is not acceptable
15              try {
16                      makeASound();
17              } catch(ArithmeticException e) { }
18
```

Ejemplo 12.

2.3. Espaciado entre signos de apertura y cierre

Estos espacios permiten una mejora legibilidad del código para poder identificar las secciones de código relacionadas.

3.3.1. Paréntesis

Al usar paréntesis y dentro de estos tengan elementos se debe de considerar el espaciado en las siguientes situaciones para este estándar:

- Tras cada coma en un listado de argumentos
- Separar un operador binario de sus operadores
- Separar expresiones incluidas en la sentencia «**for**»
- Al realizar una conversión de objetos.
-

```
1               // Acceptable
2               while(true) { }
3
4               // Acceptable
5               user.talk(“english”, “spanish”)
6
7               // Acceptable
8               sum = b + c;
9               substraction = b – c;
10              total = ( sum + 2 ) / ( substraction + 3 );
11
```

```
12          // Acceptable
13          for(int i = 0; i < 10; i+++) { }
14
15          // Acceptable
16          Stage stage = ( (Stage) ( (Node) event.getSource() ).getScene().getWindow() );
```

Ejemplo 13.

2.4. Comentarios

Los comentarios en el código fuente de Java, resultan importantes para poder llevar a cabo el mantenimiento a largo plazo de este. El objetivo de estos comentarios es mencionar la razón o motivo del porque de esa parte del código si así lo cree necesario e incluso se puede hacer uso de comentarios para aquellas situaciones donde se requiera arreglar un error posteriormente. Para ello en este estándar se requiere lo siguiente:

- **Idioma:** Todos los comentarios pueden **escribirse** únicamente en **un idioma**, **español e inglés**. Esto 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 14.

```
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      //                      // This method has errors! Fix it!
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 15.

2.5. 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 fibonacci(int position) {
8                       double goldenRatio = ( 1 + Math.sqrt(5) ) / 2;
9                       return (int) Math.round( Math.pow( goldenRatio, N) / Math.sqrt(5) );
10              }
11
```

Ejemplo 16.

2.6. Declaración de variables

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

```
1               // No acceptable
2               int base, height;
3
3               // Acceptable
4               int base;
5               int height;
```

Ejemplo 17.

2.7. 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 Level {
2             EASY, MEDIUM, HARD, IMPOSSIBLE
3     }
4
5     public enum Level {
6             EASY,
7             MEDIUM,
8             HARD,
9             IMPOSSIBLE
10    }
11
12    public enum Level {
13            EASY, MEDIUM,
14            HARD, IMPOSSIBLE
15    }
16
17    public enum Sector {
18            PRIMARY("Primario"),
19            SECONDARY("Secundario"),
20            TERTIARY("Terciario");
21
22            private String sector;
23
24            Sector(String sector){
25              this.sector = sector;
26            }
27
28            public String getSector() {
29              return sector;
30            }
31
32
33    }
```

Ejemplo 18.

2.8. Arreglo

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

```
1       new int[] arrayExample = {0,1,2,3}
2
3       new int[] arrayExample2 = {
4               0,
5               1,
6               2,
```

```
7                   3
8           }
9
10          new int[] arrayExample3 = {
11                  0,1,
12                  2,3
13          }
14
15          new int[] arrayExample4 = {
16                  0,1,2,3
17          }
```

Ejemplo 19.

3.9. Sentencia «switch»

Las sentencias «switch» se deben evitar para tratar de disminuir la complejidad de un código, en caso de ser necesario esta sentencia debe 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. Este comentario especial no es obligatorio en el último grupo de instrucciones del bloque de conmutadores.

```
1               switch(int option) {
2                       case 1:
3                       case 2:
4                               saveData();
5                               // fall-through
6                       case 3:
7                               eraseData();
8                       break;
9                       case 4:
10                              shareData();
11                      default:
12              }
```

Ejemplo 20.

3.10. Funciones anónimas o lambda

Las funciones lambda o funciones anónimas nos permiten usar menos código para hacer las mismas operaciones. La sintaxis de una expresión lambda es la siguiente
«(parámetro) -> expresión»
«(parámetros) -> expresión»
«(parámetros) -> { sentencias; }

```
1       lista.forEach( (valorN) -> System.out.println(valorN) );
2
3       lista.forEach( Integer valorN, Integer valorZ ) -> System.out.println(valor N + valorZ) );
4
5       lista.forEach( Integer valorN ) -> {
6               if(valorN.equals(“”) {
7                       System.out.println(“Vacio”);
8               } else {
9                       System.out.println(“Con elementos”);
10              }
11      }
```

Ejemplo 21.

3.11. Sentencia «try-catch»

Para que las sentencias «try-catch» deban ser consideradas pertenecientes a este estándar deben cumplir las siguientes reglas:

1. **Se debe limpiar los recursos en el bloque «finally» y no en el bloque «try»:** esto es para evitar que el recurso se quede abierto, ya que al lanzar una excepción puede que no llegue al final del bloque «try». Se permite el uso de la sentencia «try-with-resources», sin necesidad del bloque «finally», ya que ciertos recursos implementan la interfaz «AutoCloseable», los cuales se cierran automáticamente cuando el bloque del try se ejecuta o cuando se maneja una excepción.
2. **Se debe hacer uso de excepciones especificas:** debido a que se debe asegurar de proporcionar la mayor cantidad de información posible, ya que en un futuro se puede necesitar llamar al método y manejar la excepción.
3. **Se debe capturar las excepciones de lo más específico a lo más general**: ya que, si se hace de la forma contraria, las excepciones especificas nunca se ejecutarían.

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

Ejemplo 22.

3.12. Operador ternario

El operador ternario puede usarse únicamente para situaciones donde se requiera una condición y este condicional realice solo una y nada más que una acción para el valor verdadero y el valor falso. Antes de usar el operador ternario, se debe comentar una línea indicando que existe un operador ternario, esto debido a que este operador puede pasarse fácilmente desapercibido.

```
1               // T-O: Ternary Operator!
2               boolean statusAddOperation = ( courseDAO.addCourse() ) ? true : false;
3               …
```

Ejemplo 23.

3.13. El uso de conversiones o «casting»

Al hacer un «casting» o conversión de datos el dato que se esta transformando debe contener espacios para distinguir la conversión de una mejor manera.

```
1       // Acceptable
2       float valueCard = 5;
3       int roundedValued = ( (int) valueCard );
4
5       // Acceptable
6       float valueCard = 5;
7       int roundedValued = (int) valueCard;
8
9       // No acceptable
10      float valueCard = 5;
11      int roundedValued = (int)valueCard;
12
```

Ejemplo 24.

3. Nomenclatura

3.1. 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 25.

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 person {
2       }
3
4       // Forma correcta
5       public class Person {
6       }
```

Ejemplo 26.

Las clases de prueba se denominan a partir de el **nombre de la clase** que se está probando y la palabra “**Test**” .

```
1	public class MapTest {
2	}
```

Ejemplo 27.

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 <metodoBajoPruebaTest>**.

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

Ejemplo 28.

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**. En caso de ser una sola palabra está variable puede considerarse una constante o no, dependiendo del contexto. Una ayuda para determinar si es una constante, es si usa la palabra reservada «**final**»

```
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;
```

Ejemplo 29. Constantes

```
1	private int NRC;
2	private int IP;
3	private int ID;
```

Ejemplo 30. Posibles constantes

4.1.5. Nombres de campos no constantes

Los nombres de **variables** o **campos** deben estar escritos en **lowerCamelCase.** En caso de usar JavaFX e implementar nodos o elementos pertenecientes a este paquete, es necesario nombrarlos mediante el nombre que se le desee asignar a ese nodo y agregar al final el tipo de elemento que es.

```
1           private static BoardPanel board;
2           private sidePanel side;
3           private int score;
4           private static Clock logicTimer;
5
6           @FXML
7           private static TextField practitionerTextField;
8
9           @FXML
10          private static ObservableArrayList<?> practitionersArrayList;
12
```

Ejemplo 31.

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, a menos que el contexto de la aplicación sea entendible.

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

Ejemplo 32.

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.

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

Ejemplo 33.

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, también se usa en mapas)
- **S**, **U**, **V** etc. - usado para representar otros tipos.
-

```
1	public class Node<T> {
2		private T data;
3		private Node<T> next;
4
5		public T getData() {
6			return data;
7		}
8
9		public Nodo<T> getNext() {
10			return next;
11		}
12
13	}
```

Ejemplo 34.

4. Prácticas de programación

4.1. Comentario «@override»

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

4.2. Comentario «@FXML»

El uso de comentario o anotación «@FXML» debe en la parte superior de una variable o método. Esto es para facilitar la revisión.