

# **ESTÁNDAR DE CODIFICACIÓN**

**Equipo 2**

**9 de febrero de 2025**

**Realizó Mantenimiento:**

Canul Ordoñez, Josué Israel

Garcilazo Cuevas, Mónica

Leo Fernández, José Carlos

Pool Flores, Endrick Alfredo

Rodríguez Coral, Samuel David

# 1 CONTROL DE DOCUMENTACIÓN

## 1.1 Control de configuración

<b>Título:</b>	Estándar de conteo
<b>Referencia:</b>	N/A
<b>Autor:</b>	Cristian David Pan Zaldivar
<b>Fecha:</b>	9 de febrero de 2025

## 1.2 Histórico de versiones

<b>Versión</b>	<b>Fecha</b>	<b>Estado</b>	<b>Responsable</b>	<b>Nombre del archivo</b>
1.0.2	16 – 02 – 2025	A	Cristian Pan	Estándar_de_codificacion_1.0.2
2.0.0	02 – 04 – 2025	B	Arturo Quezada Daniel Rosado	Estándar_de_codificacion_2.0.0

Estado: (B)orador, (R)evisión, (A)probado

## 1.3 Histórico de cambios

<b>Versión</b>	<b>Fecha</b>	<b>Cambios</b>
1.0.1	09 – 02 – 2025	Se creó la estructura, se añadieron las descripciones
1.0.2	16 – 02 – 2025	Se añadió la descripción de cada uno de los apartados del documento.
2.0.0	02 – 04 – 2025	Se añadió un apartado para las inner-classes

## CONTENIDO

1	Control de documentación .....	2
1.1	Control de configuración .....	2
1.2	Histórico de versiones .....	2
1.3	Histórico de cambios .....	2
3	Propósito.....	5
4	Conceptos Básicos de los archivos fuente.....	5
4.1	Nombre del archivo.....	5
5	Estructura del archivo fuente.....	5
5.1	Declaraciones de importación .....	5
5.1.1	Importaciones con comodines .....	5
5.1.2	Ordenamiento y Espaciado .....	6
6	Declaración de clase .....	7
6.1	Exactamente una declaración de clase de nivel superior .....	7
6.1.1	Ordenación de los contenidos de las clases .....	7
7	Formato .....	8
7.1	Uso de Llaves Opcionales.....	8
7.1.1	Uso de llaves en funciones lambda .....	8
7.2	Bloques no vacíos: estilo K y R .....	9
7.3	Indentación de bloque .....	10
7.4	Una Declaración por Línea.....	11
7.5	Ajuste de Línea .....	11
7.6	Donde Romper .....	12
7.7	Espacios en Blanco.....	12
7.7.1	Espacios verticales .....	13
7.8	Inner Classes .....	13
8	Constructos específicos .....	14
8.1	Clases de enumeración.....	14
8.2	Declaraciones de variables.....	14
8.2.1	Una variable por declaración.....	14
8.2.2	Declaración de variables cuando sea necesario.....	15
8.2.3	Arrays .....	15
8.3	Anotaciones .....	16
8.4	Comentarios .....	16

8.4.1	Estilo de comentarios de bloque.....	16
8.4.2	Comentarios en línea .....	17
9	Nombrado.....	17
9.1	Nombre de paquetes .....	17
9.2	Nombre de clases.....	18
9.3	Nombres de métodos.....	18
9.4	Nombres de constantes .....	18
9.5	Nombre de parámetros, variables locales, atributos .....	18
10	Referencias .....	18

## **2 PROPÓSITO**

El siguiente estándar de codificación tiene como objetivo establecer directrices claras y consistentes para el desarrollo de software en Java, siguiendo ciertas prácticas del Google Style Guide for Java así como recomendaciones propuestas por Oracle. Su propósito es garantizar que el código sea legible y mantenible, facilitando su comprensión y modificación a lo largo del ciclo de vida del software.

Por ende, se debe de seguir los lineamientos establecidos durante el desarrollo del proyecto final de la materia de Mantenimiento de Software, así como será la base para la definición del estándar de conteo para aspectos de formato por considerar.

## **3 CONCEPTOS BÁSICOS DE LOS ARCHIVOS FUENTE**

### **3.1 Nombre del archivo**

El nombre del archivo de origen debe coincidir exactamente, respetando mayúsculas y minúsculas, con el nombre de la clase de nivel superior que contiene (de la cual solo puede haber una por archivo), seguido de la extensión .java. Además, el nombre del archivo debe seguir la convención PascalCase y evitar el uso de caracteres especiales, espacios en blanco o acentos.

## **4 ESTRUCTURA DEL ARCHIVO FUENTE**

El orden del contenido de un archivo fuente debe de ser el siguiente, además de que deberán de estar separados exactamente por una línea en blanco.

1. Información de licencia o derechos de autor, si está presente
2. Declaración del paquete
3. Declaraciones de importación
4. Exactamente una clase de nivel superior

### **4.1 Declaraciones de importación**

#### **4.1.1 Importaciones con comodines**

No se deben utilizar importaciones de comodines (\*) en ninguna circunstancia, ya sean estáticas o de otro tipo. En su lugar, se deben importar explícitamente las clases necesarias.

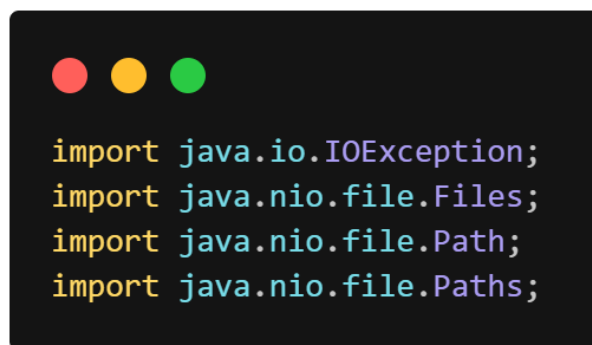
**Ejemplo incorrecto:**

A terminal window with a dark background and three colored window control buttons (red, yellow, green) at the top left. It displays the Java code `import java.util.*;` in a monospaced font with syntax highlighting: `import` is orange, `java.util.*;` is blue, and the semicolon is green.

```
import java.util.*;
```

*Imagen 1. Ejemplo de import con comodín*

**Ejemplo correcto:**

A terminal window with a dark background and three colored window control buttons (red, yellow, green) at the top left. It displays four lines of Java code, each on a new line: `import java.io.IOException;`, `import java.nio.file.Files;`, `import java.nio.file.Path;`, and `import java.nio.file.Paths;`. The code is syntax-highlighted: `import` is orange, the package and class names are blue, and the semicolons are green.

```
import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;
```

*Imagen 2. Ejemplo de import explícito*

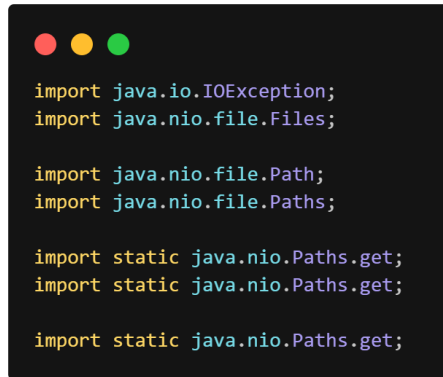
#### **4.1.2 Ordenamiento y Espaciado**

Las importaciones deben seguir el siguiente orden, dejando únicamente un salto de línea entre ellas:

1. Todas las importaciones deben estar en un solo bloque.
2. Todas las importaciones no estáticas deben agruparse en un solo bloque.

Además, no debe haber saltos de línea entre las importaciones de un mismo bloque.

**Ejemplo incorrecto:**



```
import java.io.IOException;
import java.nio.file.Files;

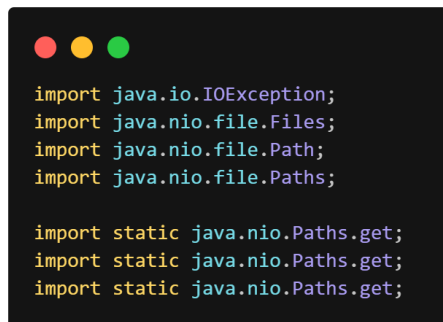
import java.nio.file.Path;
import java.nio.file.Paths;

import static java.nio.Paths.get;
import static java.nio.Paths.get;

import static java.nio.Paths.get;
```

*Imagen 3. Ejemplo de orden de imports incorrecto*

**Ejemplo correcto:**



```
import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;

import static java.nio.Paths.get;
import static java.nio.Paths.get;
import static java.nio.Paths.get;
```

*Imagen 4. Ejemplo correcto de orden y espacio de imports*

## 5 DECLARACIÓN DE CLASE

### 5.1 Exactamente una declaración de clase de nivel superior

Cada clase de nivel superior reside en un único archivo fuente propio.

#### 5.1.1 Ordenación de los contenidos de las clases

El contenido dentro de una clase debe seguir un orden lógico y consistente para mejorar la legibilidad y el mantenimiento del código. Por ejemplo, los nuevos métodos no deben añadirse habitualmente al final de la clase, ya que eso representaría un orden cronológico.


La única excepción a esta recomendación es la agrupación de métodos y constructores sobrecargados.

## 6 FORMATO

### 6.1 Uso de Llaves Opcionales

Las llaves {} deben utilizarse en todas las declaraciones de control de flujo (if/else, for, do, while), incluso cuando el cuerpo de la estructura contenga solo una instrucción o esté vacío.


**Ejemplo incorrecto:**



```
if (javaFiles.isEmpty()) throw new JavaFilesNotFoundException();
```

*Imagen 5. Ejemplo incorrecto de uso de llaves*

**Ejemplo correcto:}**




```
if (javaFiles.isEmpty()) {  
    throw new JavaFilesNotFoundException();  
}
```

*Imagen 6. Declaración de bloques adecuada*

#### 6.1.1 Uso de llaves en funciones lambda

En las funciones lambda, las llaves no son necesarias cuando solo contienen una única instrucción; sin embargo, si hay más de una instrucción, deben incluirse obligatoriamente.

**Ejemplo incorrecto:**



```
Consumer<String> imprimirConFormato = mensaje ->  
    System.out.println("Mensaje recibido:");  
    System.out.println(mensaje);
```

*Imagen 7. Declaración de función lambda incorrecta*



## Ejemplo correcto

```
Consumer<String> imprimirConFormato = mensaje -> {  
    System.out.println("Mensaje recibido:");  
    System.out.println(mensaje);  
};  
  
Consumer<String> imprimirConFormato = mensaje -> System.out.println(mensaje);
```

*Imagen 8. Declaración de función lambda correcta*

## 6.2 Bloques no vacíos: estilo K y R

Los bloques de código no vacíos deben seguir el estilo Kernighan & Ritchie (K&R):

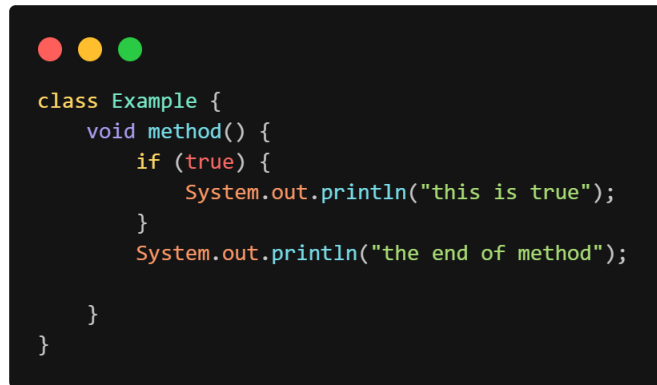
1. La llave de apertura { va en la misma línea que la declaración como (if, for, while, class, etc.), es decir, sin un salto de línea previo
2. Salto de línea después de la llave de apertura {.
3. Salto de línea antes de la llave de cierre }.
4. Salto de línea después de la llave de cierre }, exceptuando cuando la llave es seguida de una sentencia else o una declaración en la misma estructura.

## Ejemplo incorrecto

```
class Example  
{  
    void metodo()  
    {  
        if (true)  
        {  
            System.out.println("Hola");  
        }  
        System.out.println("End of method");  
    }  
}
```

*Imagen 9. Ejemplo incorrecto de declaración de bloques*

**Ejemplo correcto:**



```

class Example {
    void method() {
        if (true) {
            System.out.println("this is true");
        }
        System.out.println("the end of method");
    }
}

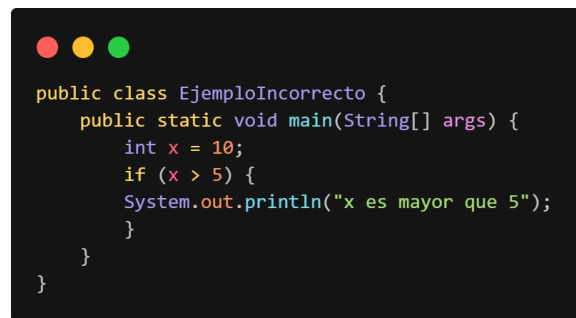
```

*Imagen 10. Declaración correcta de bloques de código*

### 6.3 Indentación de bloque

Cada vez que se abre un nuevo bloque o una estructura similar, la sangría debe aumentar en dos espacios. Cuando el bloque finaliza, la sangría debe regresar al nivel anterior.

#### Ejemplo incorrecto



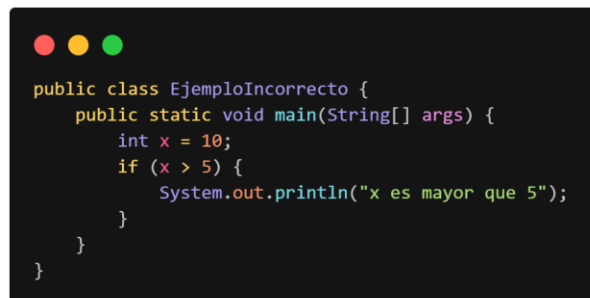
```

public class EjemploIncorrecto {
    public static void main(String[] args) {
        int x = 10;
        if (x > 5) {
            System.out.println("x es mayor que 5");
        }
    }
}

```

*Imagen 11. Indentación incorrecta*

#### Ejemplo correcto



```

public class EjemploIncorrecto {
    public static void main(String[] args) {
        int x = 10;
        if (x > 5) {
            System.out.println("x es mayor que 5");
        }
    }
}

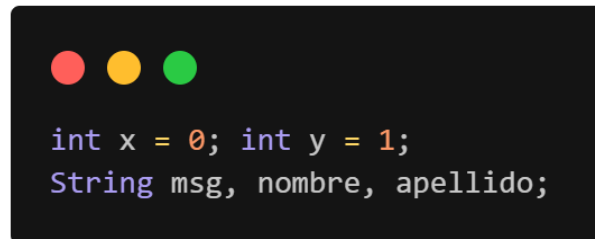
```

*Imagen 12. Indentación correcta*

## 6.4 Una Declaración por Línea

Cada declaración de variable con o sin inicialización, debe de realizarse en su propia línea para mejorar la claridad del código.

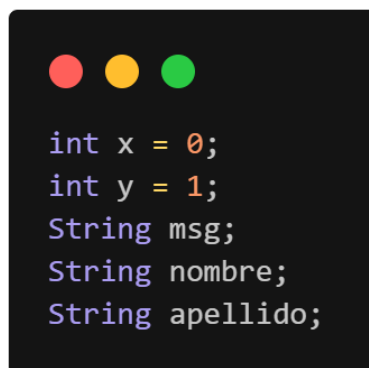
**Ejemplo incorrecto:**

A screenshot of a code editor with a dark background and three colored window control buttons (red, yellow, green) in the top left corner. The code is written in a light blue font and shows three lines of variable declarations: 'int x = 0; int y = 1;' on the first line and 'String msg, nombre, apellido;' on the second line. This is an incorrect way to declare multiple variables.

```
int x = 0; int y = 1;  
String msg, nombre, apellido;
```

*Imagen 13. Declaración incorrecta de variables*

**Ejemplo correcto:**

A screenshot of a code editor with a dark background and three colored window control buttons (red, yellow, green) in the top left corner. The code is written in a light blue font and shows five lines of variable declarations, each on its own line: 'int x = 0;', 'int y = 1;', 'String msg;', 'String nombre;', and 'String apellido;'. This is the correct way to declare multiple variables.

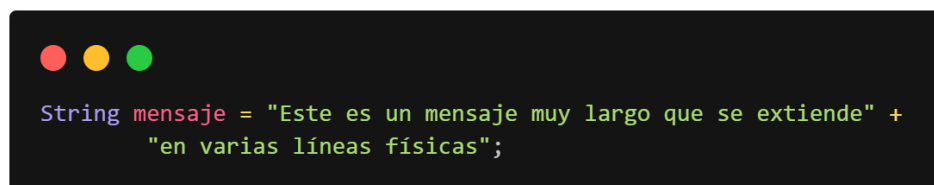
```
int x = 0;  
int y = 1;  
String msg;  
String nombre;  
String apellido;
```

*Imagen 14. Declaración correcta de variables*

## 6.5 Ajuste de Línea

Las líneas de código deben mantenerse dentro de un ancho razonable y, cuando sea necesario, dividirse en múltiples líneas siguiendo reglas de legibilidad.

**Ejemplo:**

A screenshot of a code editor with a dark background and three colored window control buttons (red, yellow, green) in the top left corner. The code is written in a light blue font and shows a single line of code that has been wrapped into two lines: 'String mensaje = "Este es un mensaje muy largo que se extiende" +' on the first line and '"en varias líneas físicas";' on the second line. This demonstrates how to handle long lines of code.

```
String mensaje = "Este es un mensaje muy largo que se extiende" +  
                "en varias líneas físicas";
```

*Imagen 15. Ejemplificación de ajuste de línea*

## 6.6 Donde Romper

La directiva principal del cambio de línea es: preferir cortar en un nivel sintáctica más alto.

Cuando se corta una línea en un operador que no es de asignación, el corte se realiza antes del símbolo. Esto se aplica a los siguientes operadores similares:

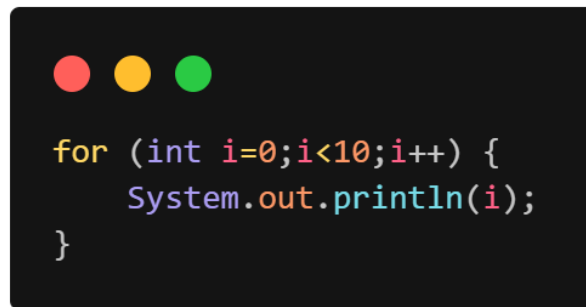
1. El separador de puntos (.)
2. Los dos puntos de referencia de un método (::)
3. Un ampersand en un tipo enlazada (<T extends Foo & Bar>)
4. Una tubería en un bloque de captura ( catch (IOException | BarException e)

## 6.7 Espacios en Blanco

El código debe incluir espacios en blanco para mejorar la legibilidad, por ejemplo:

- Antes y después de los operadores binarios.
- Después de las comas en listas de parámetros.
- Después de palabras clave de control (if, for, while).

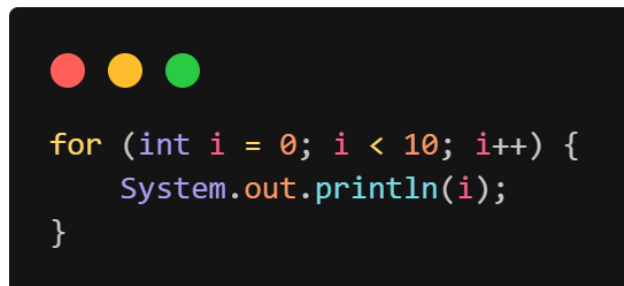
**Ejemplo incorrecto:**



```
for (int i=0;i<10;i++) {  
    System.out.println(i);  
}
```

*Imagen 16. Espaciado horizontal no recomendado entre elementos*

**Ejemplo correcto:**



```
for (int i = 0; i < 10; i++) {  
    System.out.println(i);  
}
```

*Imagen 17. Espaciado horizontal adecuado entre elementos*

### 6.7.1 Espacios verticales

Se deben utilizar espacios verticales estratégicamente para separar secciones de código relacionadas, como métodos dentro de una clase siempre que se adición mejore la legibilidad. Para ello se recomienda lo siguiente:

- Debe de haber una única línea en blanco entre elementos de una clase como campos, constructores, métodos, clases anidadas e inicializadores
- Se permite mejorar la línea en blanco entre campos consecutivos si no hay otro código entre ellos, lo que permite agruparlos lógicamente.

**Ejemplo:**

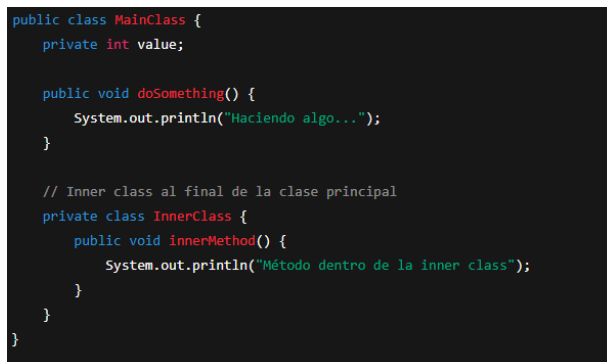
A screenshot of a code editor with a dark background and three colored window control buttons (red, yellow, green) at the top left. The code is a Java class named 'EjemploIncorrecto'. It contains a private integer field 'dato', a public void method 'method()' with a comment '//code', and a public void method 'otherMethod()' with a comment '//codio'. There is a single blank line between the field and the first method, and another single blank line between the two methods. The class is enclosed in curly braces.

```
public class EjemploIncorrecto {  
    private int dato;  
  
    public void method(){  
        //code  
    }  
  
    public void otherMethod(){  
        //codio  
    }  
}
```

*Imagen 18. Espaciado vertical recomendado*

### 6.8 Inner Classes

Las inner classes deben declararse al final de la clase principal para mantener la estructura organizada y facilitar la lectura del código. Esto permite que la lógica principal sea más accesible antes de los elementos internos auxiliares.

A screenshot of a code editor with a dark background. The code shows a Java class 'MainClass' with a private integer field 'value' and a public void method 'doSomething()' that prints 'Haciendo algo...'. At the end of the class, there is an inner class 'InnerClass' with a public void method 'innerMethod()' that prints 'Método dentro de la inner class'. The inner class is declared with the keyword 'private'. The code is well-formatted with consistent indentation.

```
public class MainClass {  
    private int value;  
  
    public void doSomething() {  
        System.out.println("Haciendo algo...");  
    }  
  
    // Inner class al final de la clase principal  
    private class InnerClass {  
        public void innerMethod() {  
            System.out.println("Método dentro de la inner class");  
        }  
    }  
}
```

*Imagen 19. Ejemplo de Inner Classes*

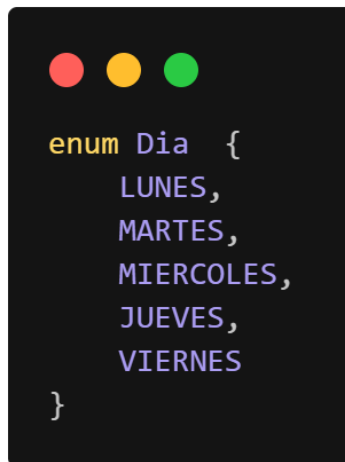
## 7 CONSTRUCTOS ESPECÍFICOS

### 7.1 Clases de enumeración

Las clases enum deben de seguir las mismas reglas que una clase en cuanto a nombramiento y a declaración de métodos. Así mismo deberán de respetar lo siguiente:

- Los valores deberán de ir en mayúsculas, con palabras separadas por guiones bajos.
- Los valores deberán de ir separados por como y con salto de línea
- Si es importante el valor numérico asociado a cada elemento del enum, se deberán de asignar valores explícitos.
- La clase de enumeración no puede declararse en una sola línea, debe de haber un espacio seguido del "{"

**Ejemplo:**

A screenshot of a code editor with a dark background and three colored window control buttons (red, yellow, green) at the top left. The code is written in a light blue font and shows a correct enum declaration for days of the week. The code is: 

```
enum Dia {  
    LUNES,  
    MARTES,  
    MIERCOLES,  
    JUEVES,  
    VIERNES  
}
```

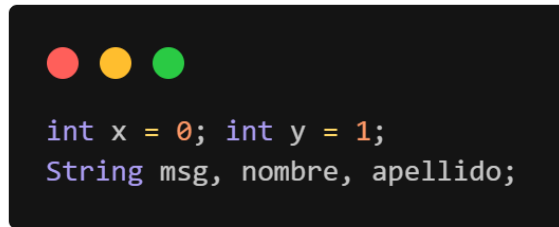
*Imagen 20. Ejemplo correcto de la declaración de un enum*

### 7.2 Declaraciones de variables

#### 7.2.1 Una variable por declaración

No se permite la declaración múltiple de variables, así como cada declaración de variable en funciones o variables locales, deberá de ser inicializada.

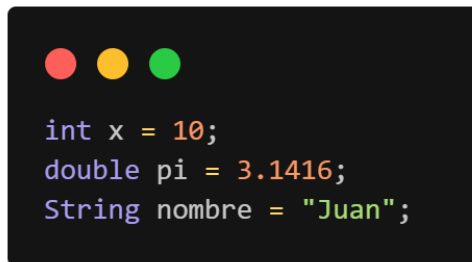
**Ejemplo incorrecto:**



```
int x = 0; int y = 1;  
String msg, nombre, apellido;
```

*Imagen 21. Declaración incorrecta de variables*

**Ejemplo correcto:**



```
int x = 10;  
double pi = 3.1416;  
String nombre = "Juan";
```

*Imagen 22. Declaración adecuada de variables*

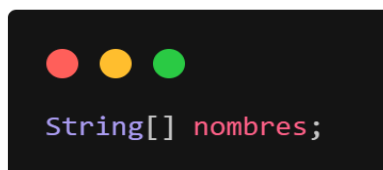
### **7.2.2 Declaración de variables cuando sea necesario**

Las variables locales no deberán de declararse al comienzo del bloque que las contiene o de la construcción similar a un bloque. En cambio, las variables locales deberán de ser declaradas cerca del punto en el que se utilizan por primera vez.

### **7.2.3 Arrays**

Los corchetes en la declaración de un array deberán de ir junto al tipo en vez del identificador de la variable. Así mismo, un arreglo no podrá definirse en una sola línea, se recomienda seguir el formato de Bloque no vacíos. Style K&R.


**Ejemplo:**



```
String[] nombres;
```

*Imagen 23. Ejemplo de ubicación de corchetes en declaración de Array*

**Ejemplo**




```
String[] examples = {  
    "Example One", "Example Two", "Example Three", "Example Four", "Example Five",  
    "Example Six", "Example Seven", "Example Eight", "Example Nine", "Example Ten"  
};
```

*Imagen 24. Ejemplo correcto de declaración de arreglo*

## 7.3 Anotaciones

Las anotaciones deben colocarse de tal manera que haya una sola declaración por línea, seguida de un salto de línea.

**Ejemplo:**



```
@Override  
public void myMethod() {  
  
}
```

*Imagen 25. Declaración de correcta de anotaciones del lenguaje*


## 7.4 Comentarios

### 7.4.1 Estilo de comentarios de bloque

Se debe emplear `/* ... */` para comentarios extensos que expliquen algoritmos complejos o secciones de código. Para ello, se debe utilizar el formato que se presenta a continuación:

**Nota:** Cada salto de línea en un comentario de bloque debe comenzar con un asterisco.

**Ejemplo:**



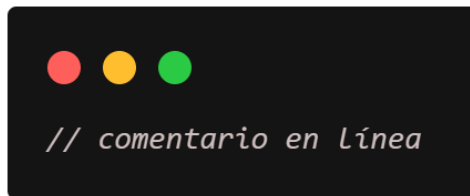
```
/**  
 * comentario de bloque  
 *  
 */
```



*Imagen 26. Declaración de comentarios por bloque*

### 7.4.2 Comentarios en línea

Usar “//” para comentarios que breves que solo requieren de una línea para su explicación. Se recomienda evitar su uso para comentarios obvios o que simplemente repitan lo que el código ya dice.



*Imagen 27. Declaración de comentarios en línea*

## 8 NOMBRADO

### 8.1 Nombre de paquetes

Los nombres de paquetes utilizan únicamente letras minúsculas y dígitos (sin guiones bajos). Las palabras consecutivas únicamente se concatenan entre sí.

**Ejemplo incorrecto:**



*Imagen 28. Declaración incorrecta de paquetes*

**Ejemplo correcto:**



*Imagen 29. Declaración correcta paquetes*

## 8.2 Nombre de clases

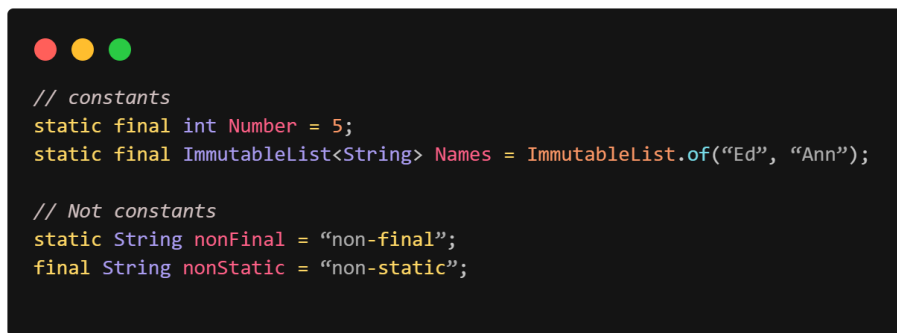
Las clases deben de escribirse en UpperCamelCase y deben de representar sustantivos o frases nominales, por ejemplo, Characeter o ImmutableList.

## 8.3 Nombres de métodos

Los nombres de métodos deben de escribirse en lowerCamelCase y deben de ser verbos o frases verbales, por ejemplo, sendMessage o stop.

## 8.4 Nombres de constantes

Los nombres de constantes utilizan UPPER\_SNAKE\_CASE. Para ello se considera como constante aquellos atributos estáticos y finales que serán inmutables.

A screenshot of a code editor with a dark background and three colored window control buttons (red, yellow, green) in the top left corner. The code is written in Java and demonstrates the correct naming and declaration of constants. It shows two sections: one for constants using 'static final' and one for non-constants using 'static' and 'final'.

```
// constants
static final int Number = 5;
static final ImmutableList<String> Names = ImmutableList.of("Ed", "Ann");

// Not constants
static String nonFinal = "non-final";
final String nonStatic = "non-static";
```

*Imagen 30. Ejemplificación de aspectos a considerar para declaración de constantes*

## 8.5 Nombre de parámetros, variables locales, atributos

Todos los nombres de parámetros, variables locales y atributos deberán de ser escritos en lowerCamalCase y deberán de ser descriptivos, representando la intención del código.

# 9 REFERENCIAS

<https://www.oracle.com/technetwork/java/codeconventions-150003.pdf>

<https://google.github.io/styleguide/javaguide.html>