UTN
Facultad Regional Córdoba

# Guía de estilo de Java TP6-IS-G2

# 1 Introducción

Este documento sirve como definición **completa** de los estándares de codificación del código fuente en el lenguaje de programación Java™. Un archivo fuente de Java se considera *de estilo TP6-IS-G2* si cumple con las reglas aquí descritas.

Al igual que otras guías de estilo de programación, los temas tratados no solo abarcan cuestiones estéticas de formato, sino también otros tipos de convenciones o estándares de codificación. Sin embargo, este documento se centra principalmente en las **reglas estrictas** que seguimos universalmente y evita dar *consejos* que no sean claramente aplicables (ya sea por humanos o por herramientas).

## 1.1 Notas terminológicas

En este documento, a menos que se aclare lo contrario:

1. El término *clase* se utiliza de forma inclusiva para significar una clase "ordinaria", una clase de enumeración, una interfaz o un tipo de anotación ( @interface ).
2. El término *miembro* (de una clase) se utiliza de manera inclusiva para significar una clase, un campo, un método *o un constructor* anidado ; es decir, todos los contenidos de nivel superior de una clase excepto los inicializadores y los comentarios.
3. El término *comentario* siempre se refiere a comentarios *de implementación.*

Ocasionalmente aparecerán otras "notas terminológicas" a lo largo del documento.

## 1.2 Notas de guía

El código de ejemplo de este documento **no es normativo** . Es decir, es posible que no ilustran la *única* forma elegante de representar el código. Las opciones de formato que se elijan en los ejemplos no deben aplicarse como reglas.

# 2 Conceptos básicos de los archivos fuente

## 2.1 Nombre del archivo

El nombre del archivo de origen consta del nombre, sensible a mayúsculas y minúsculas, de la clase de nivel superior que contiene (de las cuales hay exactamente una ), más la .java extensión.

## 2.2 Codificación de archivos: UTF-8

Los archivos fuente están codificados en **UTF-8**.

## 2.3 Caracteres especiales

### 2.3.1 Caracteres de espacio en blanco

Aparte de la secuencia de terminación de línea, el **carácter de espacio horizontal ASCII** ( **0x20** ) es el único carácter de espacio en blanco que aparece en cualquier parte de un archivo de origen. Esto implica que:

1. Todos los demás caracteres de espacio en blanco en cadenas y literales de caracteres se escapan.
2. Los caracteres de tabulación **no** se utilizan para la sangría.

# 3 Estructura del archivo fuente

Un archivo fuente consta, **en orden** :

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

**Exactamente una línea en blanco** separa cada sección presente.

## 3.1 Información sobre licencia o derechos de autor, si está presente

Si la información de licencia o copyright pertenece a un archivo, pertenece aquí.

## 3.2 Declaración del paquete

La declaración del paquete **no tiene un ajuste de línea** . El límite de columnas (Sección 4.4, Límite de columnas: 120 ) no se aplica a las declaraciones del paquete.

## 3.3 Declaraciones de importación

### 3.3.1 Sin cambios de línea

Las declaraciones de importación **no se ajustan a las líneas** . El límite de columnas (Sección 4.4, Límite de columnas: 120 ) no se aplica a las declaraciones de importación.

### 3.3.2 Ordenamiento y espaciado

Las importaciones se ordenan de la siguiente manera:

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

Si hay importaciones estáticas y no estáticas, una sola línea en blanco separa los dos bloques. No hay otras líneas en blanco entre las declaraciones de importación.

Dentro de cada bloque, los nombres importados aparecen en orden de clasificación ASCII. ( **Nota:** esto no es lo mismo que que las *declaraciones* de importación aparezcan en orden de clasificación ASCII, ya que '.' se ordena antes que ';'.)

### 3.3.3 No hay importación estática para clases

La importación estática no se utiliza para clases anidadas estáticas. Se importan con importaciones normales.

## 3.4 Declaración de clase

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

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

### 3.4.2 Ordenación de los contenidos de las clases

El orden que elija para los miembros e inicializadores de su clase puede tener un gran efecto en la facilidad de aprendizaje. Sin embargo, no existe una única receta correcta para hacerlo; las distintas clases pueden ordenar sus contenidos de distintas maneras.

Lo importante es que cada clase utilice ***un*** **orden lógico** , que su responsable podría explicar si se le pide. Por ejemplo, los nuevos métodos no se añaden habitualmente al final de la clase, ya que eso daría lugar a un orden "cronológico por fecha de incorporación", que no es un orden lógico.

#### 3.4.2.1 Sobrecargas: nunca dividir

Los métodos de una clase que comparten el mismo nombre aparecen en un único grupo contiguo sin otros miembros en el medio. Lo mismo se aplica a varios constructores (que siempre tienen el mismo nombre). Esta regla se aplica incluso cuando los modificadores como `static` o `private` difieren entre los métodos.

# 4 Formato

**Nota sobre terminología:** *una construcción en forma de bloque* se refiere al cuerpo de una clase, método o constructor.

# 4.1 Brackets

## 4.1.1 Uso de llaves opcionales

Las llaves se utilizan con las declaraciones **`if`** , **`else`** , y **`for`** , incluso cuando el cuerpo está vacío o contiene sólo una declaración. **`do while`**

Otras llaves opcionales, como las de una expresión lambda, siguen siendo opcionales.

## 4.1.2 Bloques no vacíos: estilo K y R

Las llaves siguen el estilo de Kernighan y Ritchie (" corchetes egipcios ") para bloques *no vacíos* y construcciones similares a bloques:

- No se permite ningún salto de línea antes de la llave de apertura, excepto como se detalla a continuación.
- 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 declaración o el cuerpo de un método, constructor o clase *con nombre . Por ejemplo, no* hay salto de línea después de la llave si va seguida de **`else`** o una coma.

Excepción: En los lugares donde estas reglas permiten una única sentencia que termina con un punto y coma ( **`;`** ), puede aparecer un bloque de sentencias, y la llave de apertura de este bloque está precedida por un salto de línea. Los bloques como estos se introducen normalmente para limitar el alcance de las variables locales, por ejemplo, dentro de sentencias switch.

En la Sección 4.8.1, Clases de enumeración, se ofrecen algunas excepciones para las clases de enumeración.

## 4.1.3 Bloques vacíos: pueden ser concisos

Un bloque vacío o una construcción similar a un bloque puede tener el estilo K & R (como se describe en la Sección 4.1.2 ). Alternativamente, puede cerrarse inmediatamente después de abrirse, sin caracteres ni saltos de línea entre ellos ( `{}` ), **a menos que** sea parte de una *declaración de varios bloques* (una que contenga directamente varios bloques: o ). **`if`**/**`else`** **`try`**/**`catch`**/**`finally`**

Ejemplos:

```
// Esto es aceptable void doNothing () {}

 // Esto es igualmente aceptable void doNothingElse () { }

// Esto no es aceptable: No hay bloques vacíos concisos en una
declaración de  doSomething (); } catch ( Exception e ) {}
```

## 4.2 Sangría de bloque: +2 espacios

Cada vez que se abre un nuevo bloque o una construcción similar a un bloque, la sangría aumenta en dos espacios. Cuando el bloque termina, la sangría vuelve al nivel de sangría anterior. El nivel de sangría se aplica tanto al código como a los comentarios en todo el bloque. (Vea el ejemplo en la Sección 4.1.2, Bloques no vacíos: estilo K y R ).

## 4.3 Una declaración por línea

Cada declaración va seguida de un salto de línea.

## 4.4 Límite de columnas: 120

El código Java tiene un límite de columnas de 120 caracteres. Un "carácter" significa cualquier punto de código Unicode. Salvo lo indicado a continuación, cualquier línea que exceda este límite debe ajustarse a su tamaño, como se explica en la Sección 4.5, Ajuste de línea .

Cada punto de código Unicode cuenta como un carácter, incluso si su ancho de visualización es mayor o menor. Por ejemplo, si utiliza caracteres de ancho completo , puede optar por ajustar la línea antes de donde lo requiere estrictamente esta regla.

**Excepciones:**

1. Líneas en las que no es posible respetar el límite de columnas (por ejemplo, una URL larga en Javadoc o una referencia de método JSNI larga).
2. **`package`** y **`import`** declaraciones (ver Secciones 3.2 Declaración del paquete y 3.3 Declaraciones de importación ).

3. Líneas de comando en un comentario que se pueden copiar y pegar en un shell.
4. En las raras ocasiones en que se requieren identificadores muy largos, se permite que superen el límite de columnas.

## 4.5 Ajuste de línea

**Nota sobre terminología:** Cuando el código que de otro modo ocuparía legalmente una sola línea se divide en varias líneas, esta actividad se denomina *ajuste de línea*.

No existe una fórmula determinista y completa que muestre *exactamente* cómo realizar el ajuste de línea en cada situación. Muy a menudo, existen varias formas válidas de realizar el ajuste de línea en el mismo fragmento de código.

**Nota:** Si bien el motivo típico para el ajuste de línea es evitar desbordar el límite de columnas, incluso el código que de hecho encajaría dentro del límite de columnas *puede* ser ajustado a discreción del autor.

**Consejo:** extraer un método o una variable local puede resolver el problema sin necesidad de ajustar la línea.

**Nota:** El objetivo principal del ajuste de línea es tener un código claro, *no necesariamente* un código que quepa en la menor cantidad de líneas.

### 4.5.1 Identar las líneas de continuación al menos +4 espacios

Al ajustar una línea, cada línea después de la primera (cada *línea de continuación* ) tiene una sangría de al menos +4 desde la línea original.

Cuando hay varias líneas de continuación, la sangría se puede modificar más allá de +4 según se desee. En general, dos líneas de continuación utilizan el mismo nivel de sangría si y sólo si comienzan con elementos sintácticamente paralelos.

La Sección 4.6.3 sobre Alineación horizontal aborda la práctica desaconsejada de utilizar un número variable de espacios para alinear ciertos tokens con líneas anteriores.

## 4.6 Espacios en blanco

### 4.6.1 Espacios verticales

Siempre aparece una sola línea en blanco:

1. *Entre* miembros consecutivos o inicializadores de una clase: campos, constructores, métodos, clases anidadas, inicializadores estáticos e inicializadores de instancia. **Excepción:** una línea en blanco entre dos campos consecutivos (sin ningún otro código entre ellos) es opcional. Estas líneas en blanco se utilizan según sea necesario para crear *agrupaciones lógicas* de campos.

   **Excepción:** Las líneas en blanco entre constantes de enumeración se tratan en la Sección 4.8.1 .
2. Como lo requieren otras secciones de este documento (como la Sección 3, Estructura del archivo fuente y la Sección 3.3, Declaraciones de importación ).

También puede aparecer una sola línea en blanco en cualquier lugar que mejore la legibilidad, por ejemplo, entre declaraciones para organizar el código en subsecciones lógicas. No se recomienda ni se desaconseja dejar una línea en blanco antes del primer miembro o inicializador, o después del último miembro o inicializador de la clase.

Se permiten *varias líneas en blanco consecutivas, pero nunca son obligatorias (ni recomendadas).*

### 4.6.2 Espacios horizontales

Más allá de lo requerido por el lenguaje u otras reglas de estilo, y aparte de literales y comentarios, un único espacio ASCII también aparece en los siguientes lugares **solamente**.

1. Separar cualquier palabra reservada, como `if` , `for` o `catch` , de un paréntesis abierto ( `(` ) que la sigue en esa línea
2. Separar cualquier palabra reservada, como `else` o `catch` , de una llave de cierre ( `}` ) que la precede en esa línea
3. Antes de cualquier llave de apertura ( `{` ), con dos excepciones:
   `@SomeAnnotation({a, b})` (no se utilizan espacios)
   `String[][] x = {{"foo"}};` (no se requiere espacio entre `{{` , según el punto 4 a continuación)

4. Entre cualquier contenido y una barra doble ( `//` ) que inicia un comentario. Se permiten varios espacios.
5. Entre la barra doble ( `//` ) que inicia un comentario y el texto del comentario. Se permiten varios espacios.
6. Entre el tipo y la variable de una declaración: `List<String> list`

## 4.7 Agrupamiento de paréntesis: recomendado

Los paréntesis de agrupación opcionales se omiten solo cuando el autor y el revisor coinciden en que no hay ninguna posibilidad razonable de que el código se malinterprete sin ellos, ni habrían hecho que el código fuera más fácil de leer. *No* es razonable suponer que todos los lectores tienen memorizada toda la tabla de precedencia de operadores de Java.

## 4.8 Constructores específicos

### 4.8.1 Clases de enumeración

Después de cada coma que sigue a una constante de enumeración, es opcional incluir un salto de línea. También se permiten líneas en blanco adicionales (normalmente, solo una). Dado que las clases de enumeración *son clases* , se aplican todas las demás reglas para formatear clases.

### 4.8.2 Declaraciones de variables

#### 4.8.2.1 Una variable por declaración

Cada declaración de variable (de campo o local) declara sólo una variable: declaraciones como `int a, b;` no se utilizan.

**Excepción:** Se aceptan declaraciones de múltiples variables en el encabezado de un `for` bucle.

#### 4.8.2.2 Declarado cuando sea necesario

Las variables locales **no** suelen declararse al comienzo del bloque que las contiene o de la construcción similar a un bloque. En cambio, las variables locales se declaran cerca del punto en el que se utilizan por primera vez (dentro de lo razonable), para minimizar su alcance. Las declaraciones de variables locales suelen tener inicializadores o se inicializan inmediatamente después de la declaración.

## 4.8.3 Anotaciones

#### 4.8.3.1 Anotaciones de uso de tipo

Las anotaciones de uso de tipo aparecen inmediatamente antes del tipo anotado. Una anotación es una anotación de uso de tipo si está meta-anotada con `@Target(ElementType.TYPE_USE)`.

#### 4.8.3.2 Anotaciones de clase

Las anotaciones que se aplican a una clase aparecen inmediatamente antes de la declaración y cada anotación se incluye en una línea propia (es decir, una anotación por línea). Estos saltos de línea no constituyen un ajuste de línea (Sección 4.5, Ajuste de línea ), por lo que el nivel de sangría no aumenta.

#### 4.8.3.3 Anotaciones de métodos y constructores

Las reglas para las anotaciones en las declaraciones de métodos y constructores son las mismas que en la sección anterior.

#### 4.8.3.4 Anotaciones de campo

Las anotaciones que se aplican a un campo también aparecen inmediatamente antes de la declaración, pero en este caso, se pueden enumerar *varias anotaciones (posiblemente parametrizadas) en la misma línea*.

#### 4.8.3.5 Anotaciones de parámetros y variables locales

No existen reglas específicas para formatear anotaciones en parámetros o variables locales (excepto, por supuesto, cuando la anotación es una anotación de uso de tipo).

## 4.8.4 Comentarios

En esta sección se abordan *los comentarios sobre la implementación* .

Cualquier salto de línea puede ir precedido de un espacio arbitrario seguido de un comentario de implementación. Este tipo de comentario hace que la línea no esté en blanco.

#### 4.8.4.1 Estilo de comentario en bloque

Los comentarios en bloque se indentan al mismo nivel que el código circundante. Pueden estar en `/* ... */` estilo o `// ...` en estilo. En el caso de comentarios de varias líneas `/* ... */` .

> **Consejo:** al escribir comentarios de varias líneas, utilice el `/* ... */` estilo si desea que los formateadores de código automáticos ajusten las líneas cuando sea necesario (estilo de párrafo). La mayoría de los formateadores no ajustan las líneas en `// ...` los bloques de comentarios con estilo.

## 4.8.5 Modificadores

Los modificadores de clase y miembro, cuando están presentes, aparecen en el orden recomendado por la Especificación del lenguaje Java:

```
public protected private abstract default static final
transient volatile synchronized native strictfp
```

## 4.8.6 Literales numéricos

`long` Los literales enteros con valor - utilizan un `L` sufijo en mayúscula, nunca en minúscula (para evitar confusiones con el dígito `1` ). Por ejemplo, `3000000000L` en lugar de `30000000001` .

# 5 Nombrado

## 5.1 Reglas comunes a todos los identificadores

Los identificadores utilizan únicamente letras y dígitos ASCII.

## 5.2 Reglas por tipo de identificador

### 5.2.1 Nombres de paquetes

Los nombres de los paquetes utilizan letras mayúsculas, minúsculas y dígitos (sin guiones bajos). Las palabras consecutivas simplemente se concatenan entre sí. Por ejemplo,

`com.example.deepspace` , not `com.example.deepSpace` o `com.example.deep_space` .

### 5.2.2 Nombres de clases

Los nombres de clases se escriben en UpperCamelCase .

Los nombres de clase suelen ser sustantivos o frases nominales. Por ejemplo, **`Character`** o **`ImmutableList`** . Los nombres de interfaz también pueden ser sustantivos o frases nominales (por ejemplo, **`List`** ), pero a veces pueden ser adjetivos o frases adjetivas (por ejemplo, **`Readable`** ).

No existen reglas específicas ni siquiera convenciones bien establecidas para nombrar los tipos de anotaciones.

Una clase *de prueba* tiene un nombre que termina con **`Tests`** , por ejemplo, **`IsTp6BackApplication`** . Si cubre una sola clase, su nombre es el nombre de esa clase más **`Tests`** , por ejemplo **`IsTp6BackApplicationTests`** .

### 5.2.3 Nombres de métodos

Los nombres de los métodos se escriben en lowerCamelCase .

Los nombres de los métodos suelen ser verbos o frases verbales. Por ejemplo, `sendMessage` o `stop` .

### 5.2.4 Nombres de constantes

Los nombres de las constantes utilizan `UPPER_SNAKE_CASE`, pero ¿qué *es* exactamente una constante?

Las constantes son campos finales estáticos cuyo contenido es profundamente inmutable y cuyos métodos no tienen efectos secundarios detectables. Algunos ejemplos incluyen primitivos, cadenas, clases de valores inmutables y cualquier cosa establecida en `null` . Si alguno de los estados observables de la instancia puede cambiar, no es una constante. Simplemente *intentar* nunca mutar el objeto no es suficiente.

Estos nombres suelen ser sustantivos o frases nominales.

### 5.2.5 Nombres de campos no constantes

Los nombres de campos no constantes (estáticos o no) se escriben en lowerCamelCase .

Estos nombres suelen ser sustantivos o frases nominales. Por ejemplo, `computedValues` o `index`.

### 5.2.6 Nombres de parámetros

Los nombres de los parámetros se escriben en lowerCamelCase .

Se deben evitar los nombres de parámetros de un solo carácter en los métodos públicos.

### 5.2.7 Nombres de variables locales

Los nombres de las variables locales se escriben en lowerCamelCase .

Incluso cuando son finales e inmutables, las variables locales no se consideran constantes y no se deben considerar como constantes.

### 5.2.8 Nombres de variables de tipo

Cada variable de tipo se nombra en uno de dos estilos:

Una sola letra mayúscula, seguida opcionalmente por un solo número (como `E` , `T` , `X` , `T2` )

Un nombre en el formato utilizado para las clases (ver Sección 5.2.2, Nombres de clases ), seguido de la letra mayúscula `T` (ejemplos: **`RequestT`** , **`FooBarT`** ).

# 6 Prácticas de programación

## 6.1 `@Override` : siempre usado

Un método se marca con la `@Override` anotación siempre que sea legal. Esto incluye un método de clase que reemplaza un método de superclase, un método de clase que implementa un método de interfaz y un método de interfaz que vuelve a especificar un método de superinterfaz.

## 6.2 Excepciones detectadas: no ignoradas

Salvo lo que se indica a continuación, rara vez es correcto no hacer nada en respuesta a una excepción detectada. (Las respuestas típicas son registrarla o, si se considera "imposible", volver a lanzarla como un **`AssertionError`**).

Cuando realmente es apropiado no realizar ninguna acción en un bloque catch, la razón por la que esto está justificado se explica en un comentario.

**Excepción:** en las pruebas, una excepción detectada puede ignorarse sin comentarios *si* su nombre es o comienza con `expected` . El siguiente es un modismo muy común para garantizar que el código en prueba *genere* una excepción del tipo esperado, por lo que no es necesario incluir un comentario aquí.

## 6.3 Miembros estáticos: calificados usando la clase

Cuando se debe calificar una referencia a un miembro de una clase estática, se califica con el nombre de esa clase, no con una referencia o expresión del tipo de esa clase.