**Guía de estilo de Kotlin**

Este documento es la definición completa de los estándares de codificación de Android creados por Google para código fuente en el lenguaje de programación Kotlin. Un archivo de origen de Kotlin solo se describe como apropiado para el estilo de Android de Google si cumple con las reglas que se indican en este documento.

Al igual que en las otras guías de estilo de programación, los temas incluidos no solo abarcan problemas estéticos de formato, sino también otros tipos de convenciones o estándares de codificación. Sin embargo, en este documento nos enfocamos principalmente en las reglas estrictas que se siguen a nivel universal y evitamos dar consejos que no se puedan aplicar claramente (ya sea mediante personas o herramientas).

**Archivos de origen**

Todos los archivos de origen se pueden codificar como UTF-8.

**Nombre**

Si un archivo de origen contiene solamente una clase de nivel superior, el nombre del archivo debe reflejar el nombre que distingue entre mayúsculas y minúsculas, además de la extensión .kt. De lo contrario, si un archivo de origen contiene varias declaraciones de nivel superior, elige un nombre que describa el contenido del archivo, aplica PascalCase y adjunta la extensión .kt.

// MyClass.kt  
class MyClass { }

// Bar.kt  
class Bar { }  
fun Runnable.toBar(): Bar = // …

// Map.kt  
fun <T, O> Set<T>.map(func: (T) -> O): List<O> = // …  
fun <T, O> List<T>.map(func: (T) -> O): List<O> = // …

**Caracteres especiales**

***Caracteres de espacio en blanco***

Además de la secuencia del terminador de línea, el carácter de espacio horizontal ASCII (0x20) es el único carácter de espacio en blanco que se muestra en cualquier lugar de un archivo de origen. Esto implica lo siguiente:

* Todos los demás caracteres de espacio en blanco de la string y los literales de caracteres tienen formato de escape.
* Los caracteres de tabulación *no* se usan para la sangría.

***Secuencias de escape especiales***

En el caso de los caracteres con una secuencia de escape especial (\b, \n, \r, \t, \', \", \\ y \$), se utiliza esa secuencia en lugar del carácter de escape de Unicode correspondiente (p. ej., \u000a).

***Caracteres ajenos a ASCII***

En el caso de los caracteres ajenos a ASCII, se utiliza el carácter de Unicode real (p. ej., ∞) o el carácter de escape de Unicode equivalente (p. ej., \u221e). La elección solo depende de qué opción hace que el código sea más fácil de leer y entender. Los caracteres de escape de Unicode no se aconsejan para caracteres imprimibles en ninguna ubicación y mucho menos fuera de comentarios y literales de string.

| Ejemplo | Debate |
| --- | --- |
| val unitAbbrev = "μs" | Óptimo: Perfectamente claro incluso sin escribir un comentario. |
| val unitAbbrev = "\u03bcs" // μs | Ineficiente: no hay motivo para usar un carácter de escape con un carácter imprimible. |
| val unitAbbrev = "\u03bcs" | Ineficiente: el lector no tiene idea de qué es esto. |
| return "\ufeff" + content | Bueno: Usa caracteres de escape para caracteres no imprimibles y escribe un comentario si es necesario. |

**Estructura**

Un archivo .kt consta de lo siguiente (en orden):

* Derechos de autor o encabezado de licencia (opcional)
* Anotaciones a nivel de archivo
* Declaración del paquete
* Declaraciones de importación
* Declaraciones de nivel superior

Exactamente una línea en blanco separa cada una de estas secciones.

Derechos de autor/licencia

Si el archivo incluye un encabezado de derechos de autor o licencia, debes ponerlo inmediatamente arriba del comentario de varias líneas.

/\*  
 \* Copyright 2017 Google, Inc.  
 \*  
 \* ...  
 \*/

No uses un comentario de estilo de una sola línea o de [estilo KDoc](https://kotlinlang.org/docs/reference/kotlin-doc.html).

/\*\*  
 \* Copyright 2017 Google, Inc.  
 \*  
 \* ...  
 \*/

// Copyright 2017 Google, Inc.  
//  
// ...

**Anotaciones a nivel de archivo**

Las anotaciones con el [objetivo del sitio de uso](https://kotlinlang.org/docs/reference/annotations.html#annotation-use-site-targets) del "archivo" se colocan entre cualquier comentario de encabezado y la declaración del paquete.

**Declaración del paquete**

La declaración del paquete no está sujeta a ningún límite de columnas y nunca se une con líneas.

**Declaraciones de importación**

Las declaraciones de importación para clases, funciones y propiedades se agrupan en una sola lista y se clasifican según ASCII.

Las importaciones de comodines (de cualquier tipo) no están permitidas.

Del mismo modo que la declaración de paquetes, las declaraciones de importación no están sujetas a un límite de columnas y nunca se unen con líneas.

**Declaraciones de nivel superior**

En un archivo de .kt, se pueden declarar uno o más tipos, funciones, propiedades o alias de tipo en el nivel superior.

El contenido de un archivo se debe enfocar en un solo tema. Dos ejemplos serían un tipo público o un conjunto de funciones de extensión que realizan la misma operación en varios tipos de receptores. Las declaraciones no relacionadas se deben separar en sus propios archivos y las declaraciones públicas dentro de un solo archivo se deben reducir al mínimo.

No se aplica ninguna restricción explícita en el número ni en el orden del contenido de un archivo.

Los archivos de origen se suelen leer de arriba abajo. El orden debería reflejar que las declaraciones de más arriba permiten la comprensión de las de más abajo. En los archivos, se puede ordenar el contenido de distintas maneras. Del mismo modo, un archivo puede incluir 100 propiedades, 10 funciones y una sola clase.

Lo importante es que cada archivo utilice algún orden lógico que el compilador pueda explicar si es necesario. Por ejemplo, las funciones nuevas no suelen agregarse al final del archivo, ya que se generaría un orden "cronológico por fecha en la que se agregó", lo que no es un orden lógico.

**Orden de miembro de clase**

El orden de los miembros dentro de una clase sigue las mismas reglas que las declaraciones de nivel superior.

**Formato**

***Llaves***

Las llaves no son necesarias para ramas de when ni para cuerpos de declaraciones de if que no tienen más de una rama de else y que caben en una sola línea.

if (string.isEmpty()) return  
  
val result =  
    if (string.isEmpty()) DEFAULT\_VALUE else string  
  
when (value) {  
    0 -> return  
    // …  
}

En cambio, las llaves son necesarias para cualquier if, for, rama de when, do o declaración de while, incluso cuando el cuerpo está vacío o contiene una sola declaración.

if (string.isEmpty())  
    return  // WRONG!  
  
if (string.isEmpty()) {  
    return  // Okay  
}

**Bloques no vacíos**

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
* 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 una función, un constructor o una clase *con nombre*; por ejemplo, *no* hay un salto de línea después de la llave si le sigue else o una coma)

return Runnable {  
    while (condition()) {  
        foo()  
    }  
}  
  
return object : MyClass() {  
    override fun foo() {  
        if (condition()) {  
            try {  
                something()  
            } catch (e: ProblemException) {  
                recover()  
            }  
        } else if (otherCondition()) {  
            somethingElse()  
        } else {  
            lastThing()  
        }  
    }  
}

A continuación, se indican algunas excepciones de [clases de enumeración](https://developer.android.com/kotlin/style-guide?hl=es-419#enum-classes).

Bloques vacíos

Un bloque vacío o una construcción similar a un bloque debe tener el estilo K&R.

try {  
    doSomething()  
} catch (e: Exception) {} // WRONG!

try {  
    doSomething()  
} catch (e: Exception) {  
} // Okay

Expresiones

Un condicional de if/else que se usa como expresión puede omitir las llaves *solo* si toda la expresión cabe en una sola línea.

val value = if (string.isEmpty()) 0 else 1  // Okay

val value = if (string.isEmpty())  // WRONG!  
    0  
else  
    1

val value = if (string.isEmpty()) { // Okay  
    0  
} else {  
    1  
}

**Sangría**

Cada vez que se abre un bloque o una construcción similar a un bloque nuevo, la sangría aumenta en cuatro espacios. Cuando termina el bloque, la sangría vuelve al nivel anterior. El nivel de sangría se aplica al código y los comentarios en todo el bloque.

**Una declaración por línea**

A cada declaración le sigue un salto de línea. No se usa el punto y coma.

**Ajuste de línea**

El código tiene un límite de columna de 100 caracteres. Excepto en los casos indicados a continuación, cualquier línea que exceda este límite se debe ajustar, como se indica más abajo.

Excepciones:

* Líneas en las que no se puede cumplir con el límite de columnas (por ejemplo, una URL larga en KDoc)
* Declaraciones package y import
* Líneas de comandos en un comentario que se pueden cortar y pegar en un shell

***Dónde realizar el ajuste de línea***

La regla general del ajuste de línea es que conviene realizarlo en un nivel sintáctico superior. También:

* Cuando se realiza un ajuste de línea en un operador o nombre de función infija, el ajuste va después del operador o nombre de función infija.
* Cuando se realiza un ajuste de línea en los siguientes símbolos "similares a un operador", el ajuste va antes del símbolo:
  + El separador de puntos (., ?.)
  + Los dos puntos de la referencia a un miembro (::)
* El nombre de un método o constructor permanece adjunto al paréntesis abierto (() que le sigue
* Una coma (,) permanece adjunta al token que la precede.
* Una flecha lambda (->) permanece adjunta a la lista de argumentos que la precede.

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 posible.

**Funciones**

Cuando la firma de una función no cabe en una sola línea, ajusta cada declaración de parámetro en su propia línea. Los parámetros definidos en este formato deben usar una sola sangría (+4). El paréntesis de cierre ()) y el tipo de datos que se muestra se colocan en su propia línea sin sangrías adicionales.

fun <T> Iterable<T>.joinToString(  
    separator: CharSequence = ", ",  
    prefix: CharSequence = "",  
    postfix: CharSequence = ""  
): String {  
    // …  
}

**Funciones de expresión**

Cuando una función solo contiene una expresión, se puede representar como una [función de expresión](https://kotlinlang.org/docs/reference/functions.html#single-expression-functions).

override fun toString(): String {  
    return "Hey"  
}

override fun toString(): String = "Hey"

**Propiedades**

Cuando un inicializador de propiedades no quepa en una sola línea, inserta el salto después del signo igual (=) y usa una sangría.

private val defaultCharset: Charset? =  
    EncodingRegistry.getInstance().getDefaultCharsetForPropertiesFiles(file)

Las propiedades que declaran un get o una función de set se deben colocar cada una en su propia línea con una sangría normal (+4). Si deseas formatearlas, utiliza las mismas reglas que para las funciones.

var directory: File? = null  
    set(value) {  
        // …  
    }

Las propiedades de solo lectura pueden usar una sintaxis más corta que quepa en una sola línea.

val defaultExtension: String get() = "kt"

**Espacio en blanco**

***Vertical***

Se mostrará una sola línea en blanco:

* *Entre* miembros consecutivos de una clase: propiedades, constructores, funciones, clases anidadas, etcétera.
  + Excepción: Es opcional el uso de una línea en blanco entre dos propiedades consecutivas (sin código entre ellas). Estas líneas en blanco se utilizan según sea necesario para crear grupos lógicos de propiedades y asociar propiedades con su propiedad de copia de seguridad (si tienen una).
  + Excepción: Las líneas en blanco entre constantes de enumeración se explican a continuación.
* Entre las declaraciones, *según sea necesario* para organizar el código en subsecciones lógicas.
* *De manera opcional*, antes de la primera declaración de una función, antes del primer miembro de una clase o después del último miembro de una clase (no se recomienda ni se desaconseja).
* Según sea necesario conforme a las otras secciones de este documento (como la sección [Estructura](https://developer.android.com/kotlin/style-guide?hl=es-419#structure)).

Se permiten varias líneas en blanco consecutivas, pero no se recomiendan ni se requieren.

***Horizontal***

Más allá de los requisitos de las reglas de estilo o el lenguaje, y excepto por los literales, los comentarios y KDoc, también se muestra un solo espacio de ASCII en los siguientes lugares:

* En las separaciones de cualquier palabra reservada, como if, for o catch, de un paréntesis de apertura (() que le sigue en esa línea.

// WRONG!  
for(i in 0..1) {  
}

// Okay  
for (i in 0..1) {  
}

* En las separaciones de cualquier palabra reservada, como else o catch, de una llave de cierre (}) que está antes en esa línea.

// WRONG!  
}else {  
}

// Okay  
} else {  
}

* Antes de cualquier llave de apertura ({).

// WRONG!  
if (list.isEmpty()){  
}

// Okay  
if (list.isEmpty()) {  
}

* En ambos lados de cualquier operador binario.

// WRONG!  
val two = 1+1

// Okay  
val two = 1 + 1

Lo mismo se aplica a los siguientes símbolos "similares a operadores":

* + la flecha en una expresión lambda (->)

// WRONG!  
ints.map { value->value.toString() }

// Okay  
ints.map { value -> value.toString() }

Pero no:

* + los dos puntos (::) de la referencia de un miembro

// WRONG!  
val toString = Any :: toString

// Okay  
val toString = Any::toString

* + el separador de punto (.)

// WRONG  
it . toString()

// Okay  
it.toString()

* + el operador de rango (..)

// WRONG  
 for (i in 1 .. 4) print(i)

 // Okay  
 for (i in 1..4) print(i)

* Antes de los dos puntos (:), solo si se utiliza en una declaración de clase para especificar interfaces o una clase de base, o bien cuando se utiliza en una cláusula de where para [restricciones genéricas](https://kotlinlang.org/docs/reference/generics.html#generic-constraints).

// WRONG!  
class Foo: Runnable

// Okay  
class Foo : Runnable

// WRONG  
fun <T: Comparable> max(a: T, b: T)

// Okay  
fun <T : Comparable> max(a: T, b: T)

// WRONG  
fun <T> max(a: T, b: T) where T: Comparable<T>

// Okay  
fun <T> max(a: T, b: T) where T : Comparable<T>

* Después de coma (,) o dos puntos (:).

// WRONG!  
val oneAndTwo = listOf(1,2)

// Okay  
val oneAndTwo = listOf(1, 2)

// WRONG!  
class Foo :Runnable

// Okay  
class Foo : Runnable

* En ambos lados de una barra doble (//) que comienza un comentario de fin de línea. Aquí se permiten varios espacios, aunque no son obligatorios.

// WRONG!  
var debugging = false//disabled by default

// Okay  
var debugging = false // disabled by default

Esta regla nunca se interpreta como un requisito o una prohibición de los espacios adicionales al comienzo o al final de una línea, sino que solo se ocupa del espacio interior.

Construcciones específicas

Clases de enumeración

Una enumeración sin funciones ni documentación en sus limitaciones también se podría formatear como una sola línea.

enum class Answer { YES, NO, MAYBE }

Cuando las constantes en una enumeración se colocan en líneas por separado, no se necesita una línea en blanco entre ellas, excepto cuando definen un cuerpo.

enum class Answer {  
    YES,  
    NO,  
  
    MAYBE {  
        override fun toString() = """¯\\_(ツ)\_/¯"""  
    }  
}

Como las clases de enumeración son clases, se aplican las demás reglas de formato de clases.

**Anotaciones**

Las anotaciones de miembro o tipo se colocan en líneas por separado inmediatamente antes de la construcción con anotaciones.

@Retention(SOURCE)  
@Target(FUNCTION, PROPERTY\_SETTER, FIELD)  
annotation class Global

Las anotaciones sin argumentos se pueden colocar en una sola línea.

@JvmField @Volatile  
var disposable: Disposable? = null

Cuando solo hay una anotación sin argumentos, se debe colocar en la misma línea que la declaración.

@Volatile var disposable: Disposable? = null  
  
@Test fun selectAll() {  
    // …  
}

Solo se puede usar la sintaxis de @[...] con un objetivo explícito del sitio de uso y solo para combinar 2 o más anotaciones sin argumentos en una sola línea.

@field:[JvmStatic Volatile]  
var disposable: Disposable? = null

**Tipos implícitos de propiedad o de datos que se muestran**

Se puede omitir si el cuerpo de una función de expresión o un inicializador de propiedad es un valor escalar o el tipo de datos que se muestra se puede inferir claramente a partir del cuerpo.

override fun toString(): String = "Hey"  
// becomes  
override fun toString() = "Hey"

private val ICON: Icon = IconLoader.getIcon("/icons/kotlin.png")  
// becomes  
private val ICON = IconLoader.getIcon("/icons/kotlin.png")

Si escribes una biblioteca, retén la declaración de tipo explícito cuando sea parte de la API pública.

**Nombre**

Los identificadores solo usan letras y dígitos de ASCII y, en un reducido número de casos indicados a continuación, guiones bajos. Por lo tanto, cada nombre de identificador válido coincide con el \w+ de expresión regular.

Los prefijos o sufijos especiales, como los que se ven en los ejemplos name\_, mName, s\_name y kName, no se usan, salvo en el caso de las propiedades de copia de seguridad (consulta [Propiedades de copia de seguridad](https://developer.android.com/kotlin/style-guide?hl=es-419#backing-properties)).

**Nombres de los paquetes**

Los nombres de los paquetes siempre están en minúsculas, con palabras consecutivas concatenadas (sin guiones bajos).

// Okay  
package com.example.deepspace  
// WRONG!  
package com.example.deepSpace  
// WRONG!  
package com.example.deep\_space

**Nombres de tipos**

Los nombres de las clases se escriben en formato pascalCase y, por lo general, son sustantivos o frases nominales. Por ejemplo, Character o ImmutableList. Los nombres de la interfaz también pueden ser sustantivos o frases nominales (por ejemplo, List), aunque algunas veces pueden ser adjetivos o frases adjetivas (por ejemplo, Readable).

Los nombres de las clases de prueba comienzan con el nombre de la clase que están probando y terminan con Test. Por ejemplo, HashTest o HashIntegrationTest.

**Nombres de funciones**

Los nombres de las funciones están escritos en camelCase y suelen ser verbos o frases verbales. Por ejemplo, sendMessage o stop.

Los guiones bajos están permitidos en los nombres de las funciones de prueba a fin de separar componentes lógicos del nombre.

@Test fun pop\_emptyStack() {  
    // …  
}

Las funciones anotadas con @Composable que muestra Unit son PascalCased y llevan el nombre de sustantivos, como si fueran tipos.

@Composable  
fun NameTag(name: String) {  
    // …  
}

Los nombres de las funciones no deben contener espacios, ya que no todas las plataformas los admiten (en particular, no es totalmente compatible con Android).

// WRONG!  
fun `test every possible case`() {}  
// OK  
fun testEveryPossibleCase() {}

**Nombres de constantes**

Los nombres de constantes usan UPPER\_SNAKE\_CASE: solo letras mayúsculas, con palabras separadas por guiones bajos. Pero ¿qué *es* una constante exactamente?

Las constantes son propiedades de val sin función de get personalizada, cuyo contenido es profundamente inmutable y cuyas funciones no tienen efectos secundarios detectables. Se incluyen tipos inmutables y colecciones inmutables de tipos inmutables, además de escalares y strings, si están marcados como const. Si el estado observable de una instancia puede cambiar, no es una constante. La mera intención de nunca cambiar el objeto no es suficiente.

const val NUMBER = 5  
val NAMES = listOf("Alice", "Bob")  
val AGES = mapOf("Alice" to 35, "Bob" to 32)  
val COMMA\_JOINER = Joiner.on(',') // Joiner is immutable  
val EMPTY\_ARRAY = arrayOf()

Por lo general, estos nombres son sustantivos o frases nominales.

Los valores constantes solo se pueden definir dentro de un object o como una declaración de nivel superior. Los valores que cumplan el requisito de una constante y estén definidos dentro de un class deberán usar un nombre que no sea constante.

Las constantes que son valores escalares deben usar el [modificador](http://kotlinlang.org/docs/reference/properties.html#compile-time-constants) de const.

**Nombres no constantes**

Los nombres no constantes se escriben en camelCase. Se aplican a propiedades de instancias, propiedades locales y nombres de parámetros.

val variable = "var"  
val nonConstScalar = "non-const"  
val mutableCollection: MutableSet = HashSet()  
val mutableElements = listOf(mutableInstance)  
val mutableValues = mapOf("Alice" to mutableInstance, "Bob" to mutableInstance2)  
val logger = Logger.getLogger(MyClass::class.java.name)  
val nonEmptyArray = arrayOf("these", "can", "change")

Por lo general, estos nombres son sustantivos o frases nominales.

**Propiedades de copia de seguridad**

Cuando se necesita una [propiedad de copia de seguridad](https://kotlinlang.org/docs/reference/properties.html#backing-properties), su nombre debe coincidir exactamente con el de la propiedad real, excepto que tenga un guion bajo como prefijo.

private var \_table: Map? = null  
  
val table: Map  
    get() {  
        if (\_table == null) {  
            \_table = HashMap()  
        }  
        return \_table ?: throw AssertionError()  
    }

**Nombres de variables de tipo**

El nombre de cada variable puede tener uno de dos estilos:

* Una sola letra mayúscula, que puede estar seguida de un solo numeral (como E, T, X o T2)
* Un nombre con el formato utilizado para clases, seguido del T en letras mayúsculas (como RequestT o FooBarT)

**Mayúsculas y minúsculas**

En algunas ocasiones, hay más de una manera razonable de convertir una frase en inglés en formato camelCase (por ejemplo, cuando hay siglas o construcciones inusuales, como "IPv6" o "iOS"). Para mejorar la capacidad de predicción, usa el siguiente esquema.

Comienza con el formato en prosa del nombre:

1. Convierte la frase en ASCII simple y quita los apóstrofos. Por ejemplo, la frase en inglés "Müller’s algorithm" podría convertirse en "Muellers algorithm".
2. Divide este resultado en palabras. Para ello, separa según los espacios y los signos de puntuación restantes (por lo general, guiones). *Recomendación:* Si una palabra ya tiene una apariencia convencional en formato camelCase, y se utiliza comúnmente, divídela en las partes que la compongan (p. ej., "AdWords" se convierte en "ad words"). Ten en cuenta que una palabra como "iOS" en realidad no está en formato camelCase y es la excepción a cualquier convención, de manera que esta recomendación no se aplica.
3. Ahora, escribe en minúscula todas las letras (incluidas las siglas) y, luego, haz lo siguiente:
   * Escribe en mayúscula el primer carácter de cada palabra para que quede con el formato pascalCase.
   * Escribe en mayúscula el primer carácter de cada palabra, excepto la primera, para que quede con el formato camelCase.
4. Por último, une todas las palabras en un solo identificador.

Recuerda que el formato de las palabras originales casi nunca se tiene en cuenta.

| Formato en prosa | Correcto | Incorrecto |
| --- | --- | --- |
| "Solicitud Http XML" | XmlHttpRequest | XMLHTTPRequest |
| "ID de cliente nuevo" | newCustomerId | newCustomerID |
| "cronómetro interno" | innerStopwatch | innerStopWatch |
| "admite IPv6 en iOS" | supportsIpv6OnIos | supportsIPv6OnIOS |
| "importador de YouTube" | YouTubeImporter | YoutubeImporter\* |

(\* Aceptable, pero no recomendado)

Nota: En inglés, algunas palabras usan guion de manera ambigua. Por ejemplo, tanto "nonempty" como "non-empty" son opciones correctas, de manera que los nombres de métodos checkNonempty y checkNonEmpty también son correctos.

Documentación

Formato

El formato básico de los bloques de KDoc se puede ver en este ejemplo:

/\*\*  
 \* Multiple lines of KDoc text are written here,  
 \* wrapped normally…  
 \*/  
fun method(arg: String) {  
    // …  
}

... o en este ejemplo de una sola línea:

/\*\* An especially short bit of KDoc. \*/

El formato básico siempre se acepta. El formato de una sola línea se puede sustituir cuando todo el bloque de KDoc (incluidos los marcadores de comentarios) puede caber en una sola línea. Ten en cuenta que este formato se aplica solamente cuando no hay etiquetas de bloque, como @return.

**Párrafos**

Entre los párrafos y antes del grupo de etiquetas de bloque si está presente, se muestra una línea en blanco, es decir, una línea que solo incluye el asterisco inicial alineado (\*).

**Etiquetas de bloque**

Todas las "etiquetas de bloque" utilizadas se muestran en el orden @constructor, @receiver, @param, @property, @return, @throws, @see, y nunca con una descripción vacía. Cuando una etiqueta de bloque no cabe en una sola línea, las líneas de continuación tienen una sangría de cuatro espacios a partir de la posición de @.

**Fragmento de resumen**

Cada bloque de KDoc comienza con un fragmento de resumen breve. Este fragmento es muy importante, ya que es la única parte del texto que se muestra en algunos contextos, como los índices de clase y método.

Se trata de un fragmento, una frase nominal o verbal, no una oración completa. No comienza con "A `Foo` is a..." ni con "This method returns...", y tampoco tiene que formar una oración imperativa completa, como "Save the record.". Sin embargo, el fragmento está escrito en mayúsculas y lleva la puntuación de una oración completa.

**Uso**

Como mínimo, KDoc está presente para cada tipo de public y cada miembro de public o protected de ese tipo, con las excepciones que se indican a continuación.

Excepción: Funciones autoexplicativas

KDoc es opcional para funciones "simples y obvias" como getFoo y propiedades como foo en casos en los que realmente no hay nada que valga la pena decir sino "Muestra el foo".

No es apropiado citar esta excepción para justificar la omisión de información relevante que un lector típico podría necesitar saber. Por ejemplo, para una función llamada getCanonicalName o una propiedad llamada canonicalName, no omitas su documentación (con la justificación de que solo diría /\*\* Returns the canonical name. \*/) si es posible que un lector típico no sepa qué significa el término "nombre canónico".

Excepción: Anulaciones

KDoc no siempre está presente en un método que anula un método de supertipo