# Las directrices de programación Java

Copyright © 1999 Scott Ambler, Ambysoft, Inc.

Las Instrucciones de codificación Java se proporcionan bajo licencia de Scott Ambler, Ambysoft Inc. , [www.ambysoft.com](http://www.ambysoft.com/#_blank) . Se han formateado para la inclusión en el Rational Unified Process.

Contenido

[Las directrices de programación Java 1](#_Toc437251429)

[1 Introducción  2](#_Toc437251430)

[2 Estándares de Codificación  3](#_Toc437251431)

[2.1 Convenciones de nombres  3](#_Toc437251432)

[2.2 Convenciones de la documentación  4](#_Toc437251433)

[2.2.1 Tipos de Java comentarios  5](#_Toc437251434)

[3 Normas para las funciones miembro  7](#_Toc437251435)

[3.1 Nomenclatura de metodos  7](#_Toc437251436)

[3.1.1 Descriptor de nombres funciones miembro  8](#_Toc437251437)

[3.2 Constructores de nombres  9](#_Toc437251438)

[3.3 Función Miembro visibilidad Public, Private, etc.  9](#_Toc437251439)

[3.4.2 Documentación interna  10](#_Toc437251440)

[3.5 Técnicas para escribir código limpio  12](#_Toc437251441)

[3.5.1 Documento El Código  12](#_Toc437251442)

[3.5.4 Seguir el 30-segunda regla  12](#_Toc437251443)

[3.5.5 Escribir corto, solo líneas de comando  12](#_Toc437251444)

[3.5.6 Especificar el orden de las operaciones  13](#_Toc437251445)

[4 Normas para los campos y propiedades  13](#_Toc437251446)

[4.1 Campos de nomenclatura  13](#_Toc437251447)

[4.1.1 Componentes de nombre (widgets, btn,txt,lbl)  13](#_Toc437251448)

[4.1.2 Nombrar constantes  14](#_Toc437251449)

[4.2 Visibilidad de campo  14](#_Toc437251450)

[4.3 Documentación de un campo  15](#_Toc437251451)

[4.4.2 Descriptores de nombres  16](#_Toc437251452)

[4.4.3 Descriptores técnicas avanzadas  16](#_Toc437251453)

[4.5 Visibilidad de los descriptores  17](#_Toc437251454)

[4.6 Inicializar siempre campos estáticos  17](#_Toc437251455)

[5 Normas para las variables locales  17](#_Toc437251456)

[5.1 Nombrar las variables locales  17](#_Toc437251457)

[5.1.1 Secuencias de nombres  18](#_Toc437251458)

[5.2 Declarar variables locales y documentación  18](#_Toc437251459)

[7 Normas para la realización de las clases, interfaces, paquetes, 18](#_Toc437251460)

[7.1 Normas para clases  18](#_Toc437251461)

[7.1.1 Clases de nombres  19](#_Toc437251462)

[7.1.2 Documentación de la clase  19](#_Toc437251463)

[7.1.3 Las declaraciones de clase  20](#_Toc437251464)

[7.3 Normas de paquetes  21](#_Toc437251465)

[7.3.1 Paquetes de nombres  21](#_Toc437251466)

[7.4 Las normas de recopilación unidades  21](#_Toc437251467)

[9 Varios estándares y problemas  21](#_Toc437251468)

[9.2 Importar las clases  22](#_Toc437251469)

[9.3 Optimizando código Java  22](#_Toc437251470)

[11.2.1 Comentario Java tipos  23](#_Toc437251471)

[11.2.2 ¿Qué documento  24](#_Toc437251472)

[12 Referencias  25](#_Toc437251473)

# 1 Introducción

Este documento describe una colección de normas, convenciones, y directrices para escribir código Java sólido. Se basan en sonido, ingeniería de software principios probados que conducen al código que es fácil de entender, para mantener y mejorar. Por otra parte, siguiendo los estándares de codificación la productividad como desarrollador Java debería aumentar notablemente. La experiencia demuestra que si se toma el tiempo para escribir código de alta calidad desde el principio, le resultará mucho más fácil modificar durante el proceso de desarrollo. Por último, después de un conjunto común de normas de codificación conduce a una mayor coherencia, hacer equipos de desarrolladores considerablemente más productivas.

# 2 Estándares de Codificación

Estándares de Codificación de Java son importantes porque permiten una mayor coherencia dentro del código y el código de tus compañeros. Una mayor coherencia lleva al código que es más fácil de entender, lo que significa que es más fácil de desarrollar y de mantener. Esto reduce el costo general de las aplicaciones que crea.

No hay que olvidar que el código Java se existir un largo tiempo; mucho después de que se haya trasladado a otros proyectos. Un objetivo importante durante el desarrollo es para asegurarse de que usted puede pasar su trabajo a otro desarrollador, o a otro equipo de desarrolladores para que puedan mantener y mejorar su trabajo sin tener que invertir un esfuerzo razonable para comprender el código. Código que es difícil de entender se corre el riesgo de que se va a desechar y volver a escribir.

## 2.1 Convenciones de nombres

Vamos a debatir convenciones de nombres en las normas, por lo que establecer la etapa con algunos conceptos básicos:

1. **Utiliza el Inglés descriptores que describen con precisión el valor de la variable, campo y clase obrera;** por ejemplo, utilizar nombres como **nombre** , **grandTotal** , o **al cliente corporativo** . Pese a que los nombres como **x1 , y1** , o fn son fáciles de escribir porque son cortos, éstos no proporcionan ninguna indicación de lo que representan y provocar que el código es difícil de entender, mantener y mejorar.
2. **Utilizar la terminología aplicable al dominio.** Si los usuarios referirse a sus clientes como clientes y, a continuación, utilice el término "cliente de la clase y no **Cliente".** Muchos desarrolladores hacen el error de crear términos genéricos para los conceptos en términos muy bien ya existen en la industria o el dominio.
3. **Utilizar mayúsculas y minúsculas para hacer nombres legibles.** utilizar letras minúsculas en general, pero mayúsculas la primera letra de los nombres de clase y nombres de interfaz, así como la primera letra de cualquier no-palabra. [[KAN97](file:///E:\C:Archivos%20de%20programaRationalRationalUnifiedProcessmanualsjavajava.htm#KAN97%23KAN97) ]
4. **Utilizar abreviaturas con moderación, pero si lo hace, a continuación, utilizarlos de manera inteligente.** Esto significa que usted debe mantener una lista de las formas cortas (siglas), debe elegir ellos sabiamente, y debe utilizar sistemáticamente. Por ejemplo, si desea utilizar un formulario para la palabra "número", a continuación, elija uno de **nbr** , no , o bloq num , documento que haya seleccionado (no importa cuál), y utilizar sólo que uno.
5. **Evitar nombres largos (< 15 caracteres es una buena idea).** A pesar de que el nombre de la clase **PhysicalOrVirtualProductOrService** podría parecer un buen nombre de clase en el tiempo, este nombre es demasiado largo y usted debería considerar cambiar el nombre por algo más corto, tal vez algo así como **ofrecer** .
6. **Evitar nombres que son similares o sólo se diferencian en el caso.** Por ejemplo, los nombres de variables **persistentObject e** **persistentObjects** no se deben utilizar conjuntamente, ni **anSqlDatabase** y **anSqlDatabase** .
7. **Evitar ni al final.** con nombres ni al final suelen estar reservados para el sistema y no puede ser utilizada para cualquier nombres creados por el usuario con excepción de preprocesador. Más importante aún, se pone molesto y difícil de escribir por lo que se debe tratar de evitar su uso siempre que sea posible.

## 2.2 Convenciones de la documentación

Asimismo, se consultará convenciones de la documentación, de modo que veamos algunos de los conceptos básicos en primer lugar:

1. **Comentarios que añadir a la claridad del código.** La razón por la que se documente el código es para que sea más comprensible para usted, sus compañeros de trabajo, y a cualquier otro desarrollador que viene después.
2. **Si su programa no es pena documentar, probablemente no es pena correr [** [NAG95 ]](file:///E:\C:Archivos%20de%20programaRationalRationalUnifiedProcessmanualsjavajava.htm#NAG95%23NAG95)
3. **Evitar decoración; es decir, que no use banner de como comentarios.** En los años 1960 y 1970, los programadores COBOL tiene la costumbre de dibujar cajas, normalmente con asteriscos, en torno a sus comentarios internos. Seguro que sí, pero les dio una salida para sus obras artísticas insta a, pero francamente, fue una gran pérdida de tiempo que poco valor añadido al producto final. Desea escribir código, no bastante código. Además, debido a que muchas de las fuentes utilizadas para mostrar e imprimir el código son proporcionales, y muchos de ellos no están, no se puede de todas maneras su correctamente las cajas.
4. **Mantener comentarios simple.** Algunos de los mejores comentarios son simples, de notas. No tiene que escribir un libro, sólo tiene que proporcionar información suficiente para que otras personas puedan entender el código.
5. **Escribir la documentación antes de escribir el código.** La mejor manera de código del documento consiste en escribir los comentarios antes de escribir el código. Esto le da la oportunidad de pensar acerca de cómo el código funcionará antes de escribir, y se asegurará de que la documentación se escribe. Por otro lado, se deben al menos documento el código a la hora de escribir. Puesto que la documentación permite que el código sea más fácil de comprender, es capaz de tomar ventaja de este hecho, mientras que se están desarrollando. Si se va a invertir el tiempo escribiendo documentación, al menos deberías tener algo de él. [[AMBAR98](file:///E:\C:Archivos%20de%20programaRationalRationalUnifiedProcessmanualsjavajava.htm#AMB98%23AMB98) ]
6. **Documento ¿por qué se está haciendo algo, no solo lo que, por ejemplo,** el código del ejemplo 1 que aparece a continuación muestra que un 5% de descuento se le está dando a las órdenes de 1.000 pesos o más. ¿Por qué se realiza? ¿Hay una regla de negocios que dice que las grandes órdenes obtener un descuento? ¿Hay un tiempo limitado de grandes pedidos especiales o si se trata de un programa permanente? Fue el programador original sólo ser generoso? Usted no sabe a menos que esté documentado en algún lado, ya sea en el código fuente o en un documento externo.

**Ejemplo 1:**

**Si ( ventaTotal >= 1000,00 )**

**{**

**Ventatotal= ventaTotal \* 0.95 ;**

**}**

### 2.2.1 Tipos de Java comentarios

Java tiene tres tipos de comentarios:

* Comentarios de la documentación que se inician con / \*\* y terminar con \* /
* Comentarios al estilo que comienzan con / \* y terminan con \* /
* Comentarios de una sola línea que comience con // y vaya hasta el final de la línea de código fuente

La siguiente tabla es un resumen de una *sugerencia de uso* para cada tipo de comentario, así como varios ejemplos.

|  |  |  |
| --- | --- | --- |
| **Tipo de comentario** | **Uso** | **Ejemplo** |
| **Documentación** | Documentación de uso comentarios inmediatamente ante las declaraciones de interfaces, clases, funciones miembro, y los campos en el documento. Comentarios de documentación *javadoc son procesadas por* , véase a continuación, para crear documentación externa para una clase. | / \*\*  Proveedor: un proveedor es cualquier persona u organización que le compramos productos y servicios para: @  autor S. W. Ambler  \* / |
| **Estilo C** | C-comentarios al estilo al documento de líneas de código que ya no son aplicables, pero que desea mantener sólo en caso de que tu los usuarios cambiar sus mentes, o porque desea desactivarlo temporalmente mientras se realiza la depuración. | / \*  Este código fue comentada por B. Gustafsson el 4 de junio de 1999 ya que fue sustituido por el código anterior. Eliminar después de dos años, en caso de que todavía no es aplicable.  . . (El código fuente ) \* / |
| **Una sola línea** | Utilice comentarios de una sola línea dentro de las funciones miembro para documento lógica de negocio, las secciones de código, y las declaraciones de las variables temporales. | // Aplicar un 5% de descuento a todos  // Venta más de $1000 por  generosidad // |

Lo importante es que la organización debe establecer un estándar en cuanto a la forma de estilo C comentarios y comentarios de una sola línea van a ser utilizados, y, a continuación, a seguir la norma uniforme. Utilizar un tipo de documento y de lógica del negocio con la otra al documento de código antiguo. Utilice comentarios de una sola línea para la lógica de negocios debido a que se puede poner la documentación sobre la misma línea que el código (esto se conoce como la réplica). C-comentarios al estilo de las viejas para documentar código ya que le permite comentar varias líneas a la vez. Porque C-estilo es muy similar al comentarios de documentación, a fin de evitar confusiones no se utilizan en otros lugares.

**Ten cuidado con fin comentarios -** [MCO93 ] [](file:///E:\C:Archivos%20de%20programaRationalRationalUnifiedProcessmanualsjavajava.htm#MCO93%23MCO93) enérgicamente contra el uso de comentarios en línea, también conocido como fin comentarios o final de la línea. McConnell señala que los comentarios tienen que estar alineados a la derecha del código, de manera que no interfieran con la estructura visual del código. En consecuencia, tienden a ser difíciles de formato y "si se utilizan muchos de ellos, se necesita tiempo para alinearlos. Tal vez no se gasta aprender más sobre el proyecto de código; que se dedica exclusivamente a la tediosa tarea de presionar la barra espaciadora o la tecla tab." también señala que usa comentarios también son difíciles de mantener porque cuando el código en la línea crece, se golpea el fin de comentario y si se están alineando, lo que tienes que hacer lo mismo para el resto de ellos.

#### 2.2.2 UNA descripción rápida de javadoc

Incluido en el Kit de desarrollo de Java (JDK) es un programa llamado *javadoc* que procesa archivos de código Java y produce documentación externa, en forma de archivos HTML, por sus programas en Java. *Javadoc* es compatible con un número limitado de etiquetas; palabras reservadas que marca el comienzo de una sección de documentación. Consulte el JDK *documentación javadoc* para más detalles.

|  |  |  |
| --- | --- | --- |
| **Etiqueta** | **Utilizado para** | **Fin** |
| @Nombre del autor | Las clases, las interfaces | Indica el autor(s) de un fragmento de código. Una etiqueta por autor debería ser utilizado. |
| @Deprecated | Clases, funciones miembro | Indica que la API de la clase se ha desestimado y, por lo tanto, no debe utilizarse más. |
| @Nombre de excepción descripción | Las funciones miembro | Describe las excepciones de las que una función miembro lanzamientos. Usted debe usar una etiqueta por excepción y dar el nombre de clase completo para la excepción. |
| @Param nombre descripción | Las funciones miembro | Se utiliza para describir un parámetro que se pasa a una función miembro, incluida su tipo o clase y su uso. Utilice una etiqueta por parámetro. |
| @Return descripción | Las funciones miembro | Describe el valor de retorno, si los hubiere, de una función miembro. Usted debe indicar el tipo o clase y el potencial uso(s) del valor de retorno. |
| @Desde | Clases, funciones miembro | Indica cuánto tiempo ha existido el tema; es decir, desde JDK 1.1 . |
| @Véase ClassName | Las clases, interfaces, funciones miembro, campos | Genera un vínculo de hipertexto en la documentación de la clase especificada. Puede, y probablemente debería utilizar un nombre de clase completo. |
| @Véase ClassName#miembro functionname | Las clases, interfaces, funciones miembro, campos | Genera un vínculo de hipertexto en la documentación de la función miembro especificado. Puede, y probablemente debería utilizar un nombre de clase completo. |
| @Versión texto | Las clases, las interfaces | Indica la información de la versión de un fragmento de código determinado. |

La manera en que se documente el código tiene un enorme impacto, tanto en su propia productividad y sobre la productividad de todos los que más tarde mantiene y mejora. Por documentar el código en las primeras fases del proceso de desarrollo, ser más productivos porque te obliga a pensar a través de su lógica antes de comprometerse a código. Además, cuando usted vuelva código que escribió días o semanas antes, se puede determinar fácilmente lo que estabas pensando cuando lo has escrito porque ya está documentado para usted.

# 3 Normas para las funciones miembro

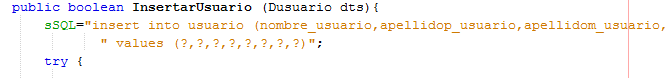
No debemos olvidar que el código que se escribe hoy puede estar en uso de aquí a muchos años y, lo más probable es que se mantengan y mejorada por alguien distinto de usted. Usted debe tratar de hacer que el código lo más "limpio" y comprensible como sea posible, porque estos factores hacen que sea más fácil de mantener y de mejorar.

## 3.1 Nomenclatura de métodos

Las funciones miembro debería ser llamado usando una descripción completa en Español, utilizando mayúsculas y minúsculas con la primera letra de cada palabra en mayúscula. También es práctica común que la primera palabra de un miembro nombre de la función de ser un fuerte, verbo activo y puede ser seguido de una segunda palabra compuesta.

**Ejemplos:**

**InsertarUsuario()**



Este convenio resultados en funciones miembro cuyo objetivo puede a menudo determinar con sólo mirar sus nombres. A pesar de que en este convenio resultados un poco más en escribir por el desarrollador porque a menudo resulta en nombres más largos, esto es más que compensado por el mayor comprensibilidad de su código.

### 3.1.1 Descriptor de nombres funciones miembro

Vamos a hablar sobre los descriptores de acceso, las funciones miembro que obtener y establecer los valores de los campos (los campos o propiedades) en mayor detalle en el capítulo siguiente. Las convenciones de nomenclatura para los descriptores de acceso, sin embargo, se resumen a continuación.

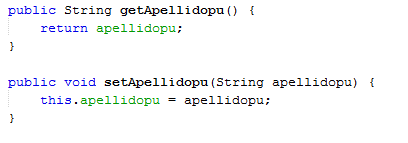
#### 3.1.1.1 Elementos get

Elementos son funciones miembro que devuelva el valor de un campo. Se debe anteponer la palabra "get" en el nombre del campo, a menos que se trate de un campo booleano y, a continuación, se le pone un prefijo "es" en el nombre del campo en lugar de "obtener".

**Ejemplos:**

**getNombre()**

**getApellidopu()**



Siguiendo esta convención de nomenclatura, se hacen evidente que una función miembro devuelve un campo de un objeto, y para el tipo boolean elementos que hacen evidente que devuelve true o false. Otra de las ventajas de esta norma es que sigue las convenciones de nomenclatura utilizada por los Frijoles Development Kit (BDK) para las funciones miembro get. [ [DES97 ]](file:///E:\C:Archivos%20de%20programaRationalRationalUnifiedProcessmanualsjavajava.htm#DES97%23DES97) La principal desventaja es que "se" es superflua y requiere escritura adicional.

**Otra convención de nombres de elementos, puede:**

Una alternativa viable, sobre la base de convenciones inglés correcto, es el de usar el prefijo "tiene" o "puede" en lugar de "es" para el tipo boolean atracciã³n. Por ejemplo, nombres como get **tieneDependents() y** **puedeImprimir()** hace un montón de sentido cuando se pone a leer el código. El problema con este enfoque es que el BDK no recoge en esta estrategia de nombres (todavía).

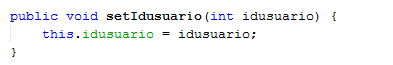
#### 3.1.1.2 Normas set

Normas, también conocido como los modificadores, son funciones miembro que modificar los valores de un campo. Se debe anteponer la palabra "set" en el nombre del campo, independientemente del tipo de campo.

**Ejemplos:**

**setNombre(String anombre)**

**setTipoUsuario(int aTipoUsuario)**



Siguiendo esta convención de nomenclatura, lo que, obviamente, una función miembro establece el valor de un campo de un objeto. Otra de las ventajas de esta norma es que sigue las convenciones de nomenclatura utilizada por el BDK para setter funciones miembro. [ [DES97 ]](file:///E:\C:Archivos%20de%20programaRationalRationalUnifiedProcessmanualsjavajava.htm#DES97%23DES97) La principal desventaja es que "set" es superflua y requiere escritura adicional.

## 3.2 Constructores de nombres

Los constructores son funciones miembro que lleve a cabo una inicialización cuando un objeto se crea por primera vez. Los constructores son siempre el mismo nombre que su clase. Por ejemplo, un constructor de la clase **VentaDtalle sería** **el DventataDetalle .** Tenga en cuenta que el mismo caso es utilizado.

**Ejemplos:**

****

Esta convención de nomenclatura se establece por Sun Microsystems y deben cumplirse estrictamente.

## 3.3 Función Miembro visibilidad Public, Private, etc.

De un buen diseño en el que se minimice el acoplamiento entre las clases, la regla general es que sea lo más restrictivo posible al establecer la visibilidad de una función miembro. Si una función miembro no tiene que ser público, entonces, a proteger, y si no tienen que ser protegidos, y luego hacer en privado.

|  |  |  |
| --- | --- | --- |
| **Visibilidad** | **Descripción** | **Uso correcto** |
| **Público** | Una función miembro pública puede ser invocada por cualquier otra función miembro en cualquier otro objeto o clase. | Cuando la función miembro debe ser accesible por objetos y clases fuera de la jerarquía de clases en las que la función miembro está definido. |
| **Protegido** | Un miembro protegido función puede ser invocada por cualquiera de los miembros de la clase en la que se ha definido o las subclases de la clase. | Cuando la función miembro proporciona comportamiento que es necesario internamente la jerarquía de clases pero no externamente. |
| **Privado** | Un miembro privado función sólo puede ser invocado por otras funciones miembro de la clase en la que se ha definido, pero no en las subclases. | Cuando la función miembro proporciona comportamiento que es específico de la clase. Las funciones miembro privado son a menudo el resultado de la refactorización, también conocida como la reorganización, el comportamiento de otras funciones miembro de la clase para encapsular un comportamiento específico. |
|  |  |  |

## 

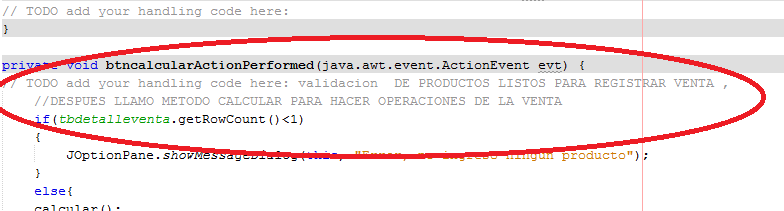
### 3.4.2 Documentación interna

Además de la documentación de las funciones miembro, usted también debe incluir comentarios dentro de sus funciones miembro para describir su trabajo. El objetivo es hacer que su función de miembro más fácil de entender, mantener y mejorar.

Hay dos tipos de comentarios que se debe utilizar para documentar el funcionamiento interno del código C-style comentarios ( / \* y \* / ) y comentarios de una sola línea ( \* ). Como se mencionó anteriormente, usted debería considerar seriamente al elegir un estilo de comentarios para documentar la lógica de negocios del código y uno para comentar código innecesario. Se sugiere que utilice comentarios de una sola línea de su lógica empresarial, debido a que puede utilizar este estilo de comentarios tanto de las líneas de comentarios y de comentarios en línea que siga al final de una línea de código. C-comentarios estilo al documento líneas de código innecesario que resulta mucho más fácil de llevar a cabo varias líneas con un solo comentario. Además, debido a que el valor de C comentarios al estilo de aspecto mucho como comentarios de documentación, su uso puede ser muy confusa, que se aparta de la comprensibilidad del código. Por lo tanto, usar con moderación.

Internamente, usted siempre debe documentar las siguientes:

1. **Las estructuras de control. Describir** cada estructura de control, tales como instrucciones de comparación y bucles. Usted no debe tener que leer todo el código en una estructura de control para determinar qué hace; en su lugar, usted debe sólo hay que ver a un uno o dos comentario de línea inmediatamente anterior.
2. **Por eso, así como lo que el código hace.** siempre puede mirar una pieza de código y averiguar qué es lo que hace, pero para que el código no es evidente, que rara vez pueden determinar por qué se hizo así. Por ejemplo, usted puede mirar una línea de código y determinar fácilmente que un 5% de descuento se aplica al total de la orden. Eso es fácil. Lo que no es fácil es descubrir POR QUÉ ese descuento se está aplicando. Está claro que hay algún tipo de regla de negocio que dice a aplicar el descuento, por lo que la empresa debe ser, por lo menos a que se refiere el código para que otros desarrolladores puedan entender por qué el código hace lo que hace.
3. **Las variables locales, aunque se** tratará el tema en mayor detalle en el Capítulo 5 [E:\C:Archivos de programaRationalRationalUnifiedProcessmanualsjavajava.htm - 5 Standards for Local Variables%235 Standards for Local Variables](file:///E:\C:Archivos%20de%20programaRationalRationalUnifiedProcessmanualsjavajava.htm#5    Standards for Local Variables%235    Standards for Local Variables), cada variable local definida en una función miembro debe ser declarado en su propia línea de código y, por lo general, debe haber un comentario en línea describe su uso.
4. **Código difícil o complejo. Si usted encuentra que** usted no puede escribir, o no tienen el tiempo, entonces usted debe documentar cualquier código complejo en una función miembro. Una regla general es que si el código no es evidente, entonces usted necesidad de documento.
5. **El orden de procesamiento. Si** hay declaraciones en el código que se debe ejecutar en un orden determinado, entonces usted debe asegurarse de que este hecho se documenta [ [AMBAR98](file:///E:\C:Archivos%20de%20programaRationalRationalUnifiedProcessmanualsjavajava.htm#AMB98%23AMB98) ]. No hay nada peor que hacer una modificación simple en un fragmento de código sólo para descubrir que ya no funciona, luego pasar horas buscando un problema, sólo para encontrar que ha recibido las cosas fuera de orden.
6. **Documente sus llaves de cierre.** Cada tan a menudo, usted encontrará que usted tiene dentro de las estructuras de control las estructuras de control dentro de las estructuras de control. A pesar de que debe evitar la escritura de código como este, a veces se encuentra que es mejor escribir en esta manera. El problema es que se confunde en cuanto a que la llave final **} carácter** pertenece a, que estructura de control. La buena noticia es que algunos editores de código admite una función que, cuando se selecciona una llave de apertura, resaltar automáticamente el cierre correspondiente; la mala noticia es que no es compatible con todos los editores. He encontrado que marca el final de una línea llaves comentario, como **//fin si** , // , //fin de **interruptor de extremo** , permite que el código sea más fácil de entender.



## 3.5 Técnicas para escribir código limpio

Esta sección abarca varias técnicas que ayudan a separar los desarrolladores profesionales de programadores el hack. Estas técnicas son:

* Documento el código.
* El párrafo o guión el código.
* Utilice los espacios en blanco.
* Siga la 30-segunda regla.
* Escribir corto, de una sola línea de comandos.
* Especificar el orden de las operaciones.

### 3.5.1 Documento El Código

Recuerde: si el código no es pena documentar,, no vale la pena mantener. [ [NAG95 ]](file:///E:\C:Archivos%20de%20programaRationalRationalUnifiedProcessmanualsjavajava.htm#NAG95%23NAG95) Cuando se aplican los estándares de documentación y lineamientos propuestos en este documento, se puede mejorar en gran medida la calidad de su código.

### 3.5.4 Seguir el 30-segunda regla

Otros programadores deberían ser capaces de mirar a la función miembro y entender que es lo que lo hace, ¿por qué lo hace, y cómo lo hace en menos de 30 segundos. Si esto no es posible, entonces el código es demasiado difícil de mantener, por lo que deberían mejorarse. Treinta segundos; eso es todo. Una buena regla general es que si una función miembro es mucho más que una pantalla, probablemente es demasiado largo.

### 3.5.5 Escribir corto, solo líneas de comando

El código debe hacer una cosa por línea. En los días de las tarjetas perforadas, tiene sentido para tratar de obtener la mayor funcionalidad posible en una sola línea de código. Cada vez que se intenta hacer más de una cosa en una sola línea de código, se hace más difícil de entender. ¿Por qué hacer esto? Queremos hacer nuestro código más fácil de entender para que sea más fácil de mantener y mejorar. Al igual que una función miembro debe hacer una cosa y una cosa solamente, usted sólo debe hacer una cosa en una sola línea de código.

Además, usted debe escribir el código que sigue siendo visible en la pantalla [ [FRENTE96 ].](file:///E:\C:Archivos%20de%20programaRationalRationalUnifiedProcessmanualsjavajava.htm#VIS96%23VIS96) Usted no debería tener que desplazarse por la ventana de edición a la derecha para leer toda la línea de código, incluido el código que utiliza comentarios en línea.

### 3.5.6 Especificar el orden de las operaciones

Una manera fácil de mejorar la comprensibilidad del código es utilizar paréntesis, también llamado "entre paréntesis", para especificar el orden exacto de las operaciones en el código Java [ [NAG95](file:///E:\C:Archivos%20de%20programaRationalRationalUnifiedProcessmanualsjavajava.htm#NAG95%23NAG95) ] y [ [AMB98](file:///E:\C:Archivos%20de%20programaRationalRationalUnifiedProcessmanualsjavajava.htm#AMB98%23AMB98) ]. Si usted tiene que saber el orden de las operaciones de un lenguaje para entender el código fuente, entonces algo está muy mal. Esto es principalmente una cuestión de comparaciones lógicas donde usted Y y O varias otras comparaciones. Tenga en cuenta que si utiliza corto, de una sola línea de comandos, tal y como se había sugerido anteriormente, entonces esto no es convertirse en un problema.

# 4 Normas para los campos y propiedades

El término *campo que se usa* aquí se refiere a un campo que el BDK llamadas

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| El término *campo que se usa* aquí se refiere a un campo que el BDK pide una propiedad [ [DES97 ].](file:///E:\C:Archivos%20de%20programaRationalRationalUnifiedProcessmanualsjavajava.htm#DES97%23DES97) Un campo es un elemento de datos que describe un objeto o una clase. Los campos pueden ser de tipo de datos, como una cadena o un flotador, o puede ser un objeto, como un cliente o una cuenta bancaria. 4.1 Campos de nomenclatura  Usted debe usar un descriptor Inglés completo de nombre a los campos, [ [NS96 ]](file:///E:\C:Archivos%20de%20programaRationalRationalUnifiedProcessmanualsjavajava.htm#GOS96%23GOS96) y [ [AMBAR98](file:///E:\C:Archivos%20de%20programaRationalRationalUnifiedProcessmanualsjavajava.htm#AMB98%23AMB98) ], lo que hace evidente lo que el campo representa. Los campos que son colecciones, como las matrices o vectores, debe asignar nombres que son plurales para indicar que representan varios valores.  **Ejemplos:**  **Nombre**  **Código postal** 4.1.1 Componentes de nombre (widgets, btn,txt,lbl)  Con los nombres de los componentes (widgets de interfaz), usted debe usar un descriptor Inglés completo postfixed por el tipo de accesorio (btn, lbl, txt). Esto hace que sea más fácil para que el usuario pueda identificar el propósito del componente, así como su tipo, lo que hace más fácil encontrar cada uno de los componentes de una lista. Muchos entornos de programación visual proporcionan listas de todos los componentes en un applet o aplicación y puede prestarse a confusión cuando todo es denominado button1 , button2 y así sucesivamente.  **Ejemplos:**    La principal ventaja es que este es un estándar de la industria común de código C++ para muchas personas ya que seguir. En el nombre de la variable, los desarrolladores pueden juzgar rápidamente su tipo y su modo de uso. Las principales desventajas son que la notación de prefijo se vuelve complicado cuando se tiene una gran cantidad de el mismo tipo de widgets y romper desde el descriptor Inglés completo convención de nomenclatura. 4.1.2 Nombrar constantes  En Java, las constantes de valores que no cambian y se aplican habitualmente en la estática *campos final* de las clases. La convención reconocida es utilizar palabras en inglés completo, todo en mayúsculas, con guiones bajos entre las palabras [ [NS96 ].](file:///E:\C:Archivos%20de%20programaRationalRationalUnifiedProcessmanualsjavajava.htm#GOS96%23GOS96)  **Ejemplos:**    **MINIMUM\_BALANCE**  **VALOR\_MÁX**  **DEFAULT\_START\_DATE**  La ventaja principal de este convenio es que ayuda a distinguir las constantes de las variables. Veremos más adelante en este documento en el que puede aumentar considerablemente la flexibilidad y el mantenimiento del código al no definir constantes; en su lugar se deben definir las funciones miembro get que devuelva el valor de las constantes. 4.2 Visibilidad de campo  Cuando los campos están declaradas como protegidas *,* existe la posibilidad de las funciones miembro de acceder directamente a las subclases, aumentando de forma eficaz la acoplamiento dentro de una jerarquía de clases. Esto hace que las clases más difíciles de mantener y mejorar, por lo tanto, debe ser evitado. Los campos nunca se debe acceder a ellos directamente; en su lugar las funciones miembro descriptor (ver más adelante) se debe utilizar.   |  |  |  | | --- | --- | --- | | **Visibilidad** | **Descripción** | **Uso correcto** | | Público | Un campo público se puede acceder por cualquier otra función miembro en cualquier otro objeto o clase. | No que los campos público. | | Protegido | Un campo protegido se puede acceder a cualquiera de los miembros de la clase en la que se ha declarado o cualquiera de los miembros las funciones definidas en las subclases de la clase. | Que los campos no protegidos. | | Privado | Un campo privado sólo se puede acceder mediante las funciones miembro de la clase en la que se ha declarado, pero no en las subclases. | Todos los campos deben ser privados y deberían ser accesibles por getter y setter funciones miembro (descriptores). |   Para los campos que no son constantes (no se guardan en almacenamiento permanente), usted debe marcar como *static o transient* [ [DES97 ].](file:///E:\C:Archivos%20de%20programaRationalRationalUnifiedProcessmanualsjavajava.htm#DES97%23DES97) Esto les permite ajustarse a las convenciones de la BDK. 4.3 Documentación de un campo  Todos los campos deben ser suficientemente documentado para que otros desarrolladores puedan entender. Para ser eficaz, necesita para documento:   1. **Su descripción.** Es necesario describir un campo para que la gente sepa cómo usarlo. 2. **Todas las invariantes.** invariantes de un campo son las condiciones que son siempre verdaderas. Por ejemplo, una invariante en el campo **dayOfMonth** podría ser que su valor está entre 1 y 31 (obviamente, usted puede conseguir mucho más complejos con esta condición invariable, la restricción del valor del campo en el mes y el año). Al documentar las restricciones sobre el valor de un campo, ayudar a definir reglas de negocio importante que hacer que sea más fácil comprender cómo funciona el código. 3. **Ejemplos de** campos que tienen reglas de negocio complejas asociadas con ellos, usted debe proporcionar varios valores de ejemplo para que sea más fácil de entender. Un ejemplo de ello es a menudo una buena imagen: que vale más que mil palabras. 4. **Problemas de concurrencia.** concurrencia es un concepto nuevo y complejo para muchos desarrolladores; de hecho, en el mejor de los casos, se trata de un antiguo y complejo tema concurrente para programadores experimentados. El resultado final es que si se utiliza la programación concurrente características de Java, entonces usted necesidad de documentar a fondo. 5. **Visibilidad decisiones.** Si ud. ha declarado el campo para que sea algo privado, pero, a continuación, debe documentar por qué lo ha hecho. Visibilidad de campo había sido tratado con anterioridad en la sección 4.2 , [visibilidad de Campo](file:///E:\C:Archivos%20de%20programaRationalRationalUnifiedProcessmanualsjavajava.htm#Field Visibility%23Field Visibility) y el uso de las funciones miembro descriptor es el encapsulamiento de apoyo cubiertos siguiente en la sección 4.4 [, utilizando las funciones miembro de descriptor](file:///E:\C:Archivos%20de%20programaRationalRationalUnifiedProcessmanualsjavajava.htm#Using Accessor Member Functions%23Using Accessor Member Functions) . La línea inferior es que es mejor que tenga una buena razón para no declarar una variable como privado.  4.4.2 Descriptores de nombres  Las funciones miembro get debe darse el nombre de "get" + nombre de campo, a menos que el campo representa un valor booleano (true o false), y, a continuación, el método get es el nombre de "es" + nombre de campo. Las funciones miembro Setter se debe dar el nombre de "set" + nombre de campo, independientemente del tipo de campo ( [ [NS96 ]](file:///E:\C:Archivos%20de%20programaRationalRationalUnifiedProcessmanualsjavajava.htm#GOS96%23GOS96) y [ [DES97 ] ).](file:///E:\C:Archivos%20de%20programaRationalRationalUnifiedProcessmanualsjavajava.htm#DES97%23DES97) Tenga en cuenta que el nombre del campo siempre en mayúsculas y minúsculas con la primera letra de todas las palabras mayúsculas. Esta convención de nomenclatura se emplea de manera sistemática dentro de la JDK y es necesario para las judías.  **Ejemplos:**   |  |  |  |  | | --- | --- | --- | --- | | **Campo** | **Tipo** | **Getter nombre** | **Nombre Setter** | | **Nombre** | **Cadena** | **GetPrimerNombre()** | **SetPrimerNombre()** | | **Dirección** | **Dirección**  **Objeto** | **GetDireccion().** | **SetDireccion()** | |  |  |  |  |  4.4.3 Descriptores técnicas avanzadas  Los descriptores se pueden utilizar para algo más que obtener y establecer los valores de campos de instancia. Esta sección explica cómo aumentar la flexibilidad de su código mediante descriptores de acceso para:   * Inicializar los valores de los campos * Acceder a valores constantes * Tener acceso a las colecciones * Tener acceso a varios campos a la vez   Es muy común utilizar inicialización diferida en los campos que son en realidad otros objetos almacenados en la base de datos. Por ejemplo, cuando se crea un nuevo elemento de inventario, no es necesario buscar cualquier elemento de inventario tipo de la base de datos que tiene como valor predeterminado. En su lugar, utilice inicialización diferida para definir este valor la primera vez que se tiene acceso a ella, de modo que sólo hay que leer el artículo de inventario tipo de objeto de la base de datos cuando y si es que lo necesita.  Este enfoque es ventajoso para los objetos que tienen los campos que no se tiene acceso. ¿Por qué hacer frente a la sobrecarga de recuperar algo de almacenamiento persistente si no lo va a utilizar?  Inicialización diferida cuando se utiliza en un getter función miembro, usted debe documentar el valor por defecto es lo que es, como hemos visto en el ejemplo anterior. Cuando usted hace esto, usted toma el misterio de cómo los campos son utilizados en su código, que mejora tanto su mantenimiento y su extensibilidad. 4.5 Visibilidad de los descriptores  Siempre tratan de hacer que los accesos protegidos, de manera que sólo las subclases pueden tener acceso a los campos. Sólo cuando un "fuera de clase" necesita tener acceso a un campo debe realizar el correspondiente método get o setter públicos. Tenga en cuenta que es común que la función miembro get público y el setter protegidos.  A veces, es necesario que acudas privadas para garantizar ciertas invariantes. Por ejemplo, una **clase de la orden** puede tener un campo que representa una colección de casos OrderItem, y un segundo campo denominado **orderTotal** , que es el total de toda la orden. La **orderTotal** campo es una comodidad que es la suma o todos los subtotales de los productos solicitados. El único miembro las funciones que debería actualizar el valor de **orderTotal** son aquellos que manipulan la colección de artículos del pedido. Suponiendo que las funciones miembro están implementados con el fin , debe hacer **setOrderTotal privado** , aunque **getOrderTotal() es** más que probable. 4.6 Inicializar siempre campos estáticos  Campos estáticos, también conocido como campos de clase, se debe dar valores válidos porque no se puede asumir que las instancias de una clase se cree antes un campo estático se accede. 5 Normas para las variables locales  Una variable local es un objeto o elemento de datos que se define dentro del ámbito de una cuadra, a menudo una función miembro. El alcance de una variable local es el bloque en el que se define. Los estándares de codificación de las variables locales se centran en:   * Convenciones de nomenclatura * Las declaraciones y convenciones de la documentación  5.1 Nombrar las variables locales  En general, las variables locales se nombran siguiendo el mismo convenciones utilizadas en los campos; en otras palabras, utilice los descriptores inglés completo con la primera letra de cualquier no-inicial palabra en mayúsculas.  Por razones de comodidad, sin embargo, esta convención de nomenclatura está relajado para varios tipos específicos de variable local:   * Arrays * Contadores de bucle * Objetos de Excepción  5.1.1 Secuencias de nombres  Cuando hay una sola entrada y/o flujo de salida que se abrió, el uso y, a continuación, se cierra en una función miembro, la convención común consiste en utilizar dentro y fuera de los nombres de estos arroyos, respectivamente [ [NS96 ].](file:///E:\C:Archivos%20de%20programaRationalRationalUnifiedProcessmanualsjavajava.htm#GOS96%23GOS96) Para un flujo de entrada y de salida, la implicación es usar el nombre **inOut.**  Una alternativa común a esta convención de nomenclatura, aunque en conflicto con las recomendaciones de Sun, es el de usar los nombres **inputStream** y **outputStream** , **iostream y** en lugar de entrada/salida , respectivamente. 5.2 Declarar variables locales y documentación  Hay varias convenciones relativas a la declaración y la documentación de las variables locales en Java. Estos convenios son:   1. **Declarar una variable local por línea de código, lo que es coherente** con una instrucción por línea de código, lo que permite documentar cada variable con un comentario en línea. 2. **Documento variables locales con un comentario en línea.** En Línea comentar es un estilo, en el que un solo comentario de línea, denotada por //, sigue inmediatamente un comando en la misma línea de código (esto se llama un fin comentario). Debe documentar lo que una variable local es usado, donde su uso es apropiado y por eso se utiliza. Esto hace que el código sea más fácil de entender. 3. **Utilice variables locales para una sola cosa.** Siempre que se utilice una variable local por más de un motivo, que disminuir de forma efectiva su cohesión y hacen que sea difícil de entender. También puede aumentar las posibilidades de introducir errores en el código de los efectos secundarios inesperados de los valores anteriores de local variable anteriormente en el código. Sí, reutilizando las variables locales es más eficiente porque necesita menos memoria que se asignen, pero reutilizando las variables locales reduce el mantenimiento del código, lo que hace que sea más frágil. Por lo general, esto no se vale el pequeño ahorro obtenido por no tener que asignar más memoria.  7 Normas para la realización de las clases, interfaces, paquetes,7.1 Normas para clases  Las normas que son importantes para las clases se basan en:   * Convenciones de nomenclatura * Convenciones de la documentación * Declaración convenios * La interfaz public y protected  7.1.1 Clases de nombres  Clase tipo interfaz  La convención estándar de Java utiliza un descriptor inglés abreviado, comenzando con las 3 primeras letras en minúscula (frm) y continua con el nombre de la entidad ala que hace referencia comenzando con mayúsculas y minúsculas para el resto del nombre. ( [ [NS96 ]](file:///E:\C:Archivos%20de%20programaRationalRationalUnifiedProcessmanualsjavajava.htm#GOS96%23GOS96) y [ [AMBAR98](file:///E:\C:Archivos%20de%20programaRationalRationalUnifiedProcessmanualsjavajava.htm#AMB98%23AMB98) ])  **Ejemplos:**    Clase tipo controlador  La convención estándar de Java utiliza un descriptor inglés abreviado, comenzando con la primera letra en mayúscula (L), por su paquete Logica y continúa con el nombre de la entidad ala que hace referencia comenzando con minúsculas para el resto del nombre. ( [ [NS96 ]](file:///E:\C:Archivos%20de%20programaRationalRationalUnifiedProcessmanualsjavajava.htm#GOS96%23GOS96) y [ [AMBAR98](file:///E:\C:Archivos%20de%20programaRationalRationalUnifiedProcessmanualsjavajava.htm#AMB98%23AMB98) ])    Clase tipo datos  La convención estándar de Java utiliza un descriptor inglés abreviado, comenzando con la primera letra en mayúscula (D), por su paquete Datos y continúa con el nombre de la entidad ala que hace referencia comenzando con minúsculas para el resto del nombre. ( [ [NS96 ]](file:///E:\C:Archivos%20de%20programaRationalRationalUnifiedProcessmanualsjavajava.htm#GOS96%23GOS96) y [ [AMBAR98](file:///E:\C:Archivos%20de%20programaRationalRationalUnifiedProcessmanualsjavajava.htm#AMB98%23AMB98) ])  **Ejemplos:**   7.1.2 Documentación de la clase  La siguiente información debe aparecer en los comentarios de documentación inmediatamente anterior a la definición de una clase:   1. **El propósito de la clase.** Los desarrolladores deben conocer el objetivo general de una clase para que ellos puedan determinar si cumple o no sus necesidades. El hábito de documentar cualquier cosas buenas para saber sobre una clase; por ejemplo, es parte de un patrón o hay alguna interesante limitaciones para utilizar [ [AMB98](file:///E:\C:Archivos%20de%20programaRationalRationalUnifiedProcessmanualsjavajava.htm#AMB98%23AMB98) ]? 2. **Errores conocidos** . Si hay cualquier problema pendiente con una clase, deberán ser documentados para que otros desarrolladores a conocer las deficiencias y las dificultades con la clase. Además, la razón de no corregir los fallos también debe documentarse. Tenga en cuenta que si un error es específico para una única función miembro, entonces debe ser directamente relacionados con la función miembro en su lugar. 3. **El desarrollo o mantenimiento de la clase historia.** Es una práctica común para incluir una historia tabla en la que se muestran las fechas, autores, así como resúmenes de los cambios realizados en una clase. Esto proporciona una visión de los programadores mantenimiento todas las modificaciones realizadas a una clase en el pasado y los documentos que ha hecho lo que a una clase. 4. **Documento aplicable las invariantes.** Un invariante es un conjunto de afirmaciones sobre una instancia o una clase que debe ser cierto en todos los "estable" veces, donde una época estable se define como el período antes de la función miembro se invoca en el objeto o clase e inmediatamente después de una función miembro se invoca [ [MEY88](file:///E:\C:Archivos%20de%20programaRationalRationalUnifiedProcessmanualsjavajava.htm#MEY88%23MEY88) ]. Al documentar las invariantes de una clase, se debe proporcionar información valiosa para otros desarrolladores acerca de cómo una clase puede utilizarse. 5. **La concurrencia estrategia** . Cualquier clase que implemente la interfaz **Runnable** debe tener su concurrencia estrategia describe en su totalidad. Programación concurrente es un tema complejo para muchos programadores, por lo tanto, necesidad de invertir el tiempo adicional para garantizar que las personas puedan comprender su trabajo. Es importante documentar su estrategia y concurrencia por qué escogieron esa estrategia sobre otros. Concurrencia Común estrategias [[READ97 ]](file:///E:\C:Archivos%20de%20programaRationalRationalUnifiedProcessmanualsjavajava.htm#LEA97%23LEA97) incluyen los siguientes:  * Sincronizar objetos * Al meter objetos * Objetos protegidos * Versioned objetos * Controladores de simultaneidad * Clientes  7.1.3 Las declaraciones de clase  Una manera de hacer que sus clases sean más fáciles de entender es la de declarar de una forma coherente. El enfoque común en Java es declarar una clase en el siguiente orden:   * Funciones miembro públicas * Campos públicos * Las funciones miembro protegido * Campos protegidos * Las funciones miembro privado * Campos privados   Puntos de constructores y **finalizar()** debe aparecer en primer lugar, probablemente debido a que estos son los primeros las funciones miembro que otro desarrollador de entender cómo utilizar la clase. Por otra parte, debido a que tienen un nivel de declarar todos los campos como privada, la orden de declaración realmente:  Constructores.  **Finalize()**  Funciones miembro públicas  Las funciones miembro protegido  Las funciones miembro privado  Campos privados  Dentro de cada grupo de funciones miembro, es común en un listado en orden alfabético. Muchos programadores eligen hacer una lista de las funciones miembro estáticas dentro de cada grupo en primer lugar, seguida de las funciones miembro de la instancia; y, a continuación, dentro de cada una de estas dos sub-grupos, haga una lista de las funciones miembro por orden alfabético. Ambos enfoques son válidos, ya que sólo se tienen que elegir uno y mantenerlo. 7.3 Normas de paquetes  Las normas que son importantes para los paquetes se basan en:   * Convenciones de nomenclatura * Convenciones de la documentación  7.3.1 Paquetes de nombres  Hay varias reglas de nombres asociados con los paquetes. En fin, estas reglas son:   1. **Los identificadores son separados por puntos.** Para hacer nombres de paquetes más legible, Sun sugiere que los identificadores de nombres de paquetes separados por períodos de tiempo. Por ejemplo, el nombre del paquete **java.awt** se compone de dos identificadores, java y **awt** . 2. **Los paquetes de distribución estándar de Java de Sun comenzar con el identificador "java".** Sun ha reservado este derecho para que los paquetes estándar de Java se denominan de forma coherente, con independencia del proveedor de su entorno de desarrollo Java. 3. **Paquete Local nombres comienzan con un identificador que no es todo en mayúsculas.** paquetes locales son utilizados internamente en su organización y no se distribuirá a otras organizaciones. Ejemplos de estos nombres de paquetes incluyen la persistencia **.mapping.relacionales y** **interfaz.las pantallas** . 4. **Paquete Global nombres comienzan con el nombre de dominio de Internet para su organización.** un paquete que se distribuirá a varias organizaciones debe incluir la organización de origen es el nombre de dominio, con el dominio de nivel superior tipo capitalizados. Por ejemplo, para distribuir los paquetes anteriores, sería nombrado **com.rational.www.persistence.mapping.relacionales y** **com.rational.www.interface.screens** .0  7.4 Las normas de recopilación unidades  Las normas y directrices para la compilación se basa en:   * Convenciones de nomenclatura * Convenios Documentación  9 Varios estándares y problemas  Este capítulo describe varios importantes normas y directrices que son lo suficientemente generales que necesitan su propio capítulo. 9.2 Importar las clases  La instrucción de importación permite el uso de caracteres comodín al indicar los nombres de las clases. Por ejemplo, la instrucción  **Import java.awt. \* ;**    Aporta en todas las clases del paquete **java.awt** . En realidad, eso no es completamente cierto. Lo que ocurre es que cada una de las clases se utiliza desde el paquete java.awt se pondrá en el código cuando se compiló; las clases que no se utilice no se. A pesar de que esto parece ser una buena característica, que reduce la legibilidad del código. Un mejor enfoque es calificar por completo el nombre de las clases que el código utiliza [ [LAF97](file:///E:\C:Archivos%20de%20programaRationalRationalUnifiedProcessmanualsjavajava.htm#LAF97%23LAF97) ]; [ [VIS96 ].](file:///E:\C:Archivos%20de%20programaRationalRationalUnifiedProcessmanualsjavajava.htm#VIS96%23VIS96) La mejor manera de importar clases se muestra en el ejemplo siguiente:  **Import java.awt.Color**  **Import java.awt.Button**  **Import java.awt.Container** 9.3 Optimizando código Java  La optimización de su código es una de las últimas cosas que los programadores deben pensar, no uno de los primeros. Dejar optimización para el final porque desea optimizar sólo el código que necesita. Muy a menudo un pequeño porcentaje de su código resultados en la gran mayoría del tiempo de procesamiento, y este es el código que usted debe optimizar. UN error clásico de programadores inexpertos es la de tratar de optimizar todos sus códigos, incluso el código que ya funciona lo suficientemente rápido.   1. **No pierdas tu tiempo optimización del código que a nadie le importa!**   ¿Qué debe buscar cuando optimizar el código? [KOE97 Como [](file:///E:\C:Archivos%20de%20programaRationalRationalUnifiedProcessmanualsjavajava.htm#KOE97%23KOE97) ], señala, los factores más importantes son los gastos fijos y el rendimiento en grandes aportaciones. La razón de esto es simple: fijo velocidad domina la ejecución de pequeñas aportaciones y el algoritmo domina para grandes aportaciones. Koenig la regla general es que un programa que funciona bien tanto en el caso de los pequeños y grandes aportaciones, probablemente funciona bien para las empresas medianas.  Los desarrolladores que han de escribir software que funciona en varias plataformas de hardware y/o sistemas operativos tienen que ser conscientes de idiosincrasia en las distintas plataformas. Las operaciones que parecen tener una determinada cantidad de tiempo, tales como la forma que la memoria y tampones son manejados, frecuentemente presentan importantes variaciones entre las plataformas. Es común encontrar que usted necesita para optimizar su código diferente, dependiendo de la plataforma.  Otra cuestión a tener en cuenta para optimizar código es las prioridades de los usuarios, ya que la gente sea sensible a los retrasos, según el contexto. Por ejemplo, los usuarios probablemente será más feliz con una pantalla en la que se llama inmediatamente y, a continuación, se tarda ocho segundos en cargar los datos en lugar de con una pantalla en la que se dibuja tras tomar cinco segundos para cargar los datos. En otras palabras, la mayoría de los usuarios están dispuestos a esperar un poco más de tiempo, siempre y cuando estén dadas información inmediata de conocimientos importantes para cuando la optimización de su código.   1. **No es necesario hacer que el código funcione más rápido para optimizar en los ojos de los usuarios.**   Aunque optimización puede significar la diferencia entre el éxito y el fracaso de su aplicación, no debemos olvidar que es mucho más importante para obtener su código para que funcione correctamente. No debemos olvidar que el lento software que funciona siempre es preferible a un software que no. 11.2.1 Comentario Java tipos  La siguiente tabla describe los tres tipos de Java comentarios y usos sugeridos por ellos.   |  |  |  | | --- | --- | --- | | **Tipo de comentario** | **Uso** | **Ejemplo** | | Documentación | Documentación de uso comentarios inmediatamente ante las declaraciones de interfaces, clases, funciones miembro, y los campos en el documento. Comentarios de documentación *javadoc son procesadas por* , véase a continuación, para crear documentación externa para una clase. | / \*\*  Cliente: un cliente es cualquier persona u organización que vendemos productos y servicios para: @  autor S. W. Ambler  \* / | | Estilo C | C-comentarios al estilo al documento de líneas de código que ya no son aplicables, pero que desea mantener sólo en caso de que tu los usuarios cambiar sus mentes, o porque desea desactivarlo temporalmente mientras se realiza la depuración. | / \*  Este código fue comentada por B. Gustafsson, 4 de junio de 1999 ya que fue sustituido por el código anterior. Eliminar después de dos años, en caso de que todavía no es aplicable.  . . (El código fuente ) \* / | | Una sola línea | Utilice comentarios de una sola línea dentro de las funciones miembro para documento lógica de negocio, las secciones de código, y las declaraciones de las variables temporales. | // Aplicar un 5% de descuento en todas las facturas  // más de $1000, definido por la  generosidad Sarek //  // campaña comenzó en febrero de 1995. |  11.2.2 ¿Qué documento  El siguiente gráfico resume lo que en el documento sobre cada porción de código Java que se escribe.   |  |  | | --- | --- | | **Tema** | **¿Qué documento** | | Argumentos y parámetros | El tipo del parámetro  Lo que debe ser utilizado para  Las restricciones o condiciones previas  Ejemplos | | Campos y  campos/propiedades | Su descripción  Documento todas las invariantes  Ejemplos  Problemas de concurrencia  Visibilidad decisiones | | Clases | El propósito de la clase  Errores conocidos  El desarrollo y el mantenimiento de la clase  Documento aplicable invariantes  La concurrencia estrategia | | Compilación unidades | Cada clase o interfaz definido en la clase, incluida una breve descripción  El nombre del archivo y/o información de identificación  Información sobre Derechos de Autor | | Getter función miembro | ¿Por qué documento inicialización diferida utilizada, si procede | | Las interfaces | El propósito  ¿Cómo se debe y no debe ser utilizado | | Las variables locales | Su uso o propósito | | Funciones Miembro: Documentación | Qué y por qué la función miembro hace lo que hace  Lo que una función miembro se debe pasar como parámetros  Lo que una función miembro  Errores conocidos  Las excepciones que una función miembro lanza  Visibilidad decisiones  ¿Cómo una función miembro cambia el objeto  Incluye una historia de los cambios en el código  Ejemplos de cómo invocar a la función miembro si procede  Condiciones previas y postcondition aplicables | | Funciones Miembro: comentarios internos | Estructuras de Control  Por eso, así como lo que hace el código  Las variables locales  Código difícil o complejo  El orden de procesamiento | | Paquete | La razón de ser del paquete  Las clases del paquete |  12 Referencias   |  |  | | --- | --- | | [ AMBAR98 ] | Ambler, S. W. (1998) . *Crear aplicaciones que funcionan: El manual paso a paso para desarrollar sistemas robustos con tecnología de objetos* . Nueva York: SIGS Books/Cambridge University Press. | | [ COA97 ] | Coad, P. y Mayfield, M. (1997). *Java Diseño: construcción de mejores aplicaciones y applets .* Upper Saddle River, NJ: Prentice Hall Inc. | | ] [ DES97 | DeSoto, A. (1997). *Con el Kit de desarrollo Granos 1,0 Febrero de 1997: UN Tutorial* . Sun Microsystems. | | [ NS96 ] | Gosling, J. , Alegría, B. , Steele, G. (1996). *La especificación de lenguaje Java* . Reading, MA: Addison Wesley Longman Inc. | | [ GRA97 ] | Grand, M. (1997). *Referencia del lenguaje Java* . Sebastopol, CA: O. Reilly & Associates, Inc. | | [ KAN97 ] | Kanerva, J. (1997). *La Java FAQ .* Reading, MA: Addison Wesley Longman Inc. | | [ KOE97 ] | Koenig, A. (1997). *La importancia--y los riesgos, de la medición de los resultados* . Nueva York: SIGS Publicaciones, Revista de Programación Orientada a Objetos, 1997 Enero, 9 (8), págs. 58-60. | | [ LAF97 ] | Laffra, C. (1997). *Java avanzado: expresiones idiomáticas, los escollos, estilos y consejos de programación* . Upper Saddle River, NJ: Prentice Hall Inc. | | [ READ97 ] | Read, D. (1997). *Programación concurrente en Java: Principios de Diseño y patrones* . Reading, MA: Addison Wesley Longman Inc. | | [ MCO93 ] | McConnell, S. (1993). *Código completo: UN Manual Práctico de Construcción del Software* . Redmond, WA: Microsoft Press. | | [ MEY88 ]. | Meyer, B. (1988). *Construcción de sistemas orientados a objetos* . Upper Saddle River, NJ: Prentice Hall Inc. | | [ NAG95 ] | Nagler, J. (1995). *Estilo de codificación y de Buenas Prácticas de Computación* . http://wizard.ucr.edu/~nagler/coding\_style.html | | [ SUN96 ] | Sun Microsystems (1996). *Javadoc - El API de Java Generador de Documentación* . Sun Microsystems. | | [ SUN97 ] | Sun Microsystems (1997). *100% Puro Java Cookbook para desarrolladores de Java: Las normas y consejos para maximizar la portabilidad de los programas Java de Sun* Microsystems. | | [ FRENTE96 ] | Visión 2000 CCS Equipo paquete y la aplicación (1996). *Estándares de Codificación de C, C++ y Java* . http://v2ma09.gsfc.nasa.gov/coding\_standards.html |   [Copyright © 1987 - 2001 Rational Software Corporation](file:///E:\C:Archivos%20de%20programaRationalRationalUnifiedProcesscopyritecopyrite.htm) |

|  |
| --- |
| Rational Unified Process |