Publicado en jpereza (http://jpereza.nom.es)

Creado 10/01/2008 - 17:39 Por jpereza

Java

Convenciones de Código Java

Traducción del artículo original de Sun Microsystems, Inc. Code Conventions for the Java™ Programming Language [1].

Por qué tener convenciones de código

Las convenciones de código son importantes para los programadores por numerosas razones:

- El 80% del tiempo de vida de un programa se dedica al mantenimiento del mismo.
- En pocas ocasiones, el programa es mantenido durante toda su vida útil por su autor original.
- Las convenciones de código aumentan la legibilidad de los programas, permitiendo a los desarrolladores comprender nuevo código rápida y perfectamente.
- Si se distribuye el código fuente como un producto, se necesita asegurar que está tan bien empaquetado y limpio como cualquier otro producto que se cree.

Reconocimientos

Este documento refleja los estándares de codificación del lenguaje Java presentados en la Especificación del Lenguaje Java [2], de Sun Microsystems, Inc. Sus principales contribuciones son de Peter King, Patrick Naughton, Mike DeMoney, Jonni Kanerva, Kathy Walrath y Scott Hommel.

Cualquier comentario sobre el documento original (en inglés) debería ser enviado a Sun Microsystems, Inc. a través del siguiente formulario de contacto [3].
Cualquier comentario sobre esta traducción debería ser enviado a Jesús Pérez Alcaide [4].

Nombres de fichero

Esta sección lista los nombres de fichero y sufijos usados comúnmente.

Sufijos de fichero

Java utiliza los siguientes sufijos de fichero:

Tipo de fichero	Sufijo
Código fuente Java	.java
Código compilado Java	.class

Nombres de ficheros comunes

Nombre de fichero	Uso
LEEME	Nombre preferido para el fichero que resume el contenido de un directorio en particular.

Organización de ficheros

Un fichero consiste en secciones que deberían estar separadas por líneas en blanco y un comentario opcional identificando cada sección.

Los ficheros de más de 2000 líneas son demasiado largos y deberían evitarse.

Para ver un ejemplo de un programa Java correctamente formado, ir al Ejemplo de código fuente Java [5].

Ficheros de código fuente Java

Cada fichero de código fuente Java contiene una única clase o interfaz público. Cuando una clase pública tiene clases privadas e interfaces asociados, se pueden poner en el mismo fichero de código fuente que la clase pública. La clase pública debería ser la primera clase o interfaz en el fichero.

Los ficheros de código fuente Java tienen la siguiente ordenación:

- Comentarios iniciales
- Sentencias `package` e `import`
- Declaraciones de clase e interfaz

Comentarios iniciales

Todos los ficheros de código fuente deberían comenzar con un comentario que muestre el nombre de la clase, información sobre la versión, la fecha y el copyright.

```
/*
 * Nombre de la clase
 *
 * Información sobre la versión
 *
 * Fecha
 *
 * Copyright
 */
```

Sentencias `package` e `import`

La primera línea que no sea un comentario de todos los ficheros de código fuente Java es una

sentencia `package`. Después, puede haber sentencias `import`. Por ejemplo:

```
package es.nom.jpereza;
import java.util.List;
```

Nota: El primer componente de un nombre de paquete único estaría siempre escrito en letras ASCII minúsculas y sería uno de los nombres de dominio de nivel superior (actualmente com, edu, gov, mil, net, org ó uno de los códigos de país de dos letras, como se especifica en el estándar ISO 3166 [6]).

Declaraciones de clase e interfaz

La siguiente tabla describe las partes de una declaración de clase o interfaz, en el orden que deben aparecer. En Ejemplo de código fuente Java [5] hay un ejemplo que incluye comentarios.

	Parte de la declaración de Clase/Interfaz	**Notas**
1	Comentario de documentación de la clase/interfaz `(/** ... */)`	Ver Comentarios [7] para más información sobre el contenido de este comentario.
2	Sentencia `class` ó `interface`	
3	Comentario de la implementación de la clase/interfaz, si fuera necesario `(/* ... */)`	Este comentario debería contener cualquier información relativa a toda la clase o interfaz, que no sea apropiada para el comentario de documentación.
4	Variables de clase (estáticas) (`static`)	Primero las variables públicas (`public`), luego las protegidas (`protected`), después las de paquete (sin modificador de acceso) y por último las privadas (`private`).
5	Variables de instancia	Primero las variables públicas (`public`), luego las protegidas (`protected`), después las de paquete (sin modificador de acceso) y por último las privadas (`private`).
6	Constructores	
7	Métodos	Estos métodos deberían estar agrupados por funcionalidad en lugar de por ámbito o accesibilidad. Por ejemplo, un método estático privado puede estar entre dos métodos de instancia públicos. El objetivo es hacer la lectura y comprensión del código más fácil.

Tabulación

La unidad de tabulación deberían ser cuatro espacios. La forma exacta de la tabulación (espacios ó tabuladores) no se especifica.

Longitud de línea

Evitar las líneas de más de 80 caracteres, ya que algunas herramientas no las manejan bien.

Nota: Ejemplos para uso en documentación deberían tener una longitud de línea menor, generalmente no más de 70 caracteres.

Ruptura de líneas (Wrapping lines)

Cuando una expresión no cabe en una única línea, se debe romper de acuerdo a estos principios generales:

- Romper después de una coma.
- Romper antes de un operador.
- Preferir las rupturas de alto nivel a las de bajo nivel.
- Alinear la nueva línea con el principio de la expresión al mismo nivel que la línea anterior.
- Si las reglas anteriores llevan a un código confuso o demasiado pegado al margen derecho, entonces tabular sólo con 8 espacios.

Aquí hay algunos ejemplos de llamadas a métodos en varias líneas:

```
someMethod(longExpression1, longExpression2, longExpression3,
        longExpression4, longExpression5);

var = someMethod1(longExpression1,
                someMethod2(longExpression2,
                        longExpression3));
```

A continuación, dos ejemplos de cómo romper una expresión aritmética. El primero es más recomendable, puesto que la ruptura ocurre fuera de la expresión entre paréntesis, la cual es de mayor nivel.

```
longName1 = longName2 * (longName3 + longName4 - longName5)
           + 4 * longname6; // RECOMENDADA

longName1 = longName2 * (longName3 + longName4
                       - longName5) + 4 * longname6; // EVITAR
```

A continuación hay dos ejemplos de cómo tabular declaraciones de métodos. El primero es el caso convencional. El segundo dejaría la segunda y tercera líneas demasiado pegadas al margen derecho si se usara la tabulación convencional, por eso en cambio se tabula sólo con 8 espacios.

```
// TABULACION CONVENCIONAL
someMethod(int anArg, Object anotherArg, String yetAnotherArg,
           Object andStillAnother) {
    ...
}

// USAR 8 ESPACIOS PARA EVITAR PEGARSE AL MARGEN DERECHO
private static synchronized horkingLongMethodName(int anArg,
        Object anotherArg, String yetAnotherArg,
```

```
        Object andStillAnother) {
    ...
}
```

La ruptura de líneas para las sentencias `if` debería usar generalmente la regla de los 8 espacios, ya que la tabulación convencional (4 espacios) dificulta la lectura del cuerpo de la sentencia `if`. Por ejemplo:

```
// NO UTILIZAR ESTA TABULACION
if ((condition1 && condition2)
    || (condition3 && condition4)
    ||!(condition5 && condition6)) { // MALAS RUPTURAS
    doSomethingAboutIt();            // HACEN QUE ESTA LINEA SE PIERDA FACILMENTE
}

// USAR ESTA TABULACION EN SU LUGAR
if ((condition1 && condition2)
        || (condition3 && condition4)
        ||!(condition5 && condition6)) {
    doSomethingAboutIt();
}

// O USAR ESTA OTRA
if ((condition1 && condition2) || (condition3 && condition4)
        ||!(condition5 && condition6)) {
    doSomethingAboutIt();
}
```

Aquí se muestran tres maneras aceptables de escribir expresiones ternarias:

```
alpha = (aLongBooleanExpression) ? beta : gamma;

alpha = (aLongBooleanExpression) ? beta
                                 : gamma;

alpha = (aLongBooleanExpression)
        ? beta
        : gamma;
```

Comentarios

Los programas en Java pueden tener dos tipos de comentarios: comentarios de implementación y comentarios de documentación. Los comentarios de implementación son como los de C++, los cuales están delimitados por `/* ... */` y `//`. Los comentarios de documentación (conocidos como "comentarios Javadoc") son específicos de Java y están delimitados por `/** ... */`. Los comentarios de documentación se pueden extraer a ficheros HTML usando la herramienta `javadoc`.

Los comentarios de implementación están destinados a comentar el código o para comentarios sobre la implementación en particular. Los comentarios de documentación están destinados a describir la especificación del código, desde una perspectiva independiente de la implementación, para ser leídos por desarrolladores que pueden no tener necesariamente el código fuente a mano.

Los comentarios se deberían usar para dar una visión general del código y para proporcionar información adicional que no esté disponible fácilmente en el propio código. Los comentarios deberían contener solamente información que sea relevante para la lectura y comprensión del

programa. Por ejemplo, información sobre cómo se construye el paquete correspondiente o en qué directorio reside, no debería ser incluida como comentario.

Discusiones sobre decisiones de diseño que no sean obvias o triviales son apropiadas, pero evitar duplicar información que esté presente (y de forma clara) en el código. Es muy fácil que los comentarios redundantes expiren. En general, evitar cualquier comentario que sea probable que expire según evoluciona el código.

Nota: La frecuencia de los comentarios a veces refleja la pobre calidad del código. Cuando te sientas obligado a añadir un comentario, considera reescribir el código para hacerlo más claro.

Los comentarios no deberían estar encerrados en grandes cajas dibujadas con asteriscos u otros caracteres.
Los comentarios no deberían incluir nunca caracteres especiales como el avance de página (código ASCII 0x0C) o el carácter de retroceso (código ASCII 0x08).

Formatos de los comentarios de implementación

Los programas pueden tener cuatro estilos de comentarios de implementación: de bloque, de una sola línea, por detrás y de final de línea.

Comentarios de bloque

Los comentarios de bloque se usan para proporcionar descripciones de ficheros, métodos, estructuras de datos y algoritmos. Los comentarios de bloque pueden ser usados al principio de cada fichero y antes de cada método. También pueden ser usados en otros lugares, como en el interior de los métodos. Los comentarios de bloque dentro de una función o método deberían estar tabulados al mismo nivel que el código que describen.

Un comentario de bloque debería estar precedido por una línea en blanco para apartarlo del resto del código.

```
/*
 * Este es un comentario de bloque
 */
```

Los comentarios de bloque pueden comenzar por `/*-`, que es reconocido por la herramienta **`indent(1)`** como el principio de un comentario de bloque que no debería ser reformateado. Por ejemplo:

```
/*-
 * Este es un comentario de bloque con un formato especial
 * que quiero que indent(1) lo ignore.
 *
 *    uno
 *        dos
 *            tres
 */
```

Nota: Si no se usa **`indent(1)`**, no se tiene que usar `/*-` en el código o hacer cualquier otra concesión a la posibilidad que alguien use **`indent(1)`** en el código.

Ver también Comentarios de documentación [7].

Comentarios de una sola línea

Los comentarios cortos pueden aparecer en una sola línea tabulada al nivel del código que le sigue. Si un comentario no puede ser escrito en una sola línea, debería seguir el formato del comentario de bloque. Un comentario de una sola linea debería ser precedido por una línea en blanco. A continuación, un comentario de una sola línea en un código Java:

```
if (condition) {

    /* Tratar la condición. */
    ...
}
```

Comentarios por detrás

Los comentarios muy cortos pueden aparecer en la misma línea que el código que describen, pero deben estar suficientemente separados de las sentencias. Si en un trozo de código aparece más de un comentario corto, todos deberían tabularse al mismo nivel.

Un ejemplo de comentario por detrás en un código Java:

```
if (a == 2) {
    return true;            /* caso especial */
} else {
    return isPrime(a);      /* funciona solo con impares */
}
```

Comentarios de final de línea

El delimitador de comentario `//` puede comentar una línea completa o sólo una parte de una línea. No debería ser usado en múltiples líneas consecutivas para comentarios de texto. Sin embargo, puede ser utilizado en múltiples líneas consecutivas para comentar secciones de código. A continuación, ejemplos de los tres estilos:

```
if (foo > 1) {

    // Do a double-flip.
    ...
}
else {
    return false;          // Explicar por qué aquí.
}
//if (bar > 1) {
//
//    // Do a triple-flip.
//    ...
//}
//else {
//    return false;
//}
```

Comentarios de documentación

Nota: Ir a Ejemplo de código fuente Java [5] para ver ejemplos de los formatos de comentario descritos aquí.

Para obtener más detalles, ver "How to Write Doc Comments for Javadoc" [8] que incluye información sobre las etiquetas de comentarios Javadoc (`@return`, `@param`, `@see`).

Para obtener más detalles sobre los comentarios Javadoc y la herramienta `javadoc`, ver la página de javadoc [9].

Los comentarios Javadoc describen las clases Java, interfaces, constructores, métodos y campos. Cada comentario Javadoc se escribe dentro de los delimitadores `/** ... */`, con un comentario por clase, interfaz o miembro. Este comentario debería aparecer justo antes de la declaración:

```
/**
 * La clase Example proporciona ...
 */
public class Example { ...
```

Nótese que las clases e interfaces de alto nivel no tienen tabulación, mientras que sus miembros sí. La primera línea de un comentario Javadoc (`/**`) para clases e interfaces no está tabulada; las líneas siguientes del comentario tienen 1 espacio de tabulación (para alinear verticalmente los asteriscos). Los miembros, incluyendo los constructores, tienen 4 espacios para la primera línea y 5 espacios para las siguientes líneas.

Si se necesita dar información sobre una clase, interfaz, variable o método, que no sea apropiada para documentación, úsese un comentario de implementación de bloque o de una sola línea inmediatamente después de la declaración. Por ejemplo, los detalles sobre la implementación de una clase deberían ir en un comentario de implementación de bloque después de la sentencia `class`, no en el comentario de documentación de la clase.

Los comentarios Javadoc no deberían ser posicionados dentro de la definición de un constructor o método, porque Java asocia los comentarios de documentación con la primera declaración después del comentario.

Declaraciones

Número por línea

Se recomienda una declaración por línea ya que fomenta los comentarios. En otras palabras,

```
int level; // nivel de tabulación
int size;  // tamaño de la tabla
```

es preferido antes que

```
int level, size;
```

No poner tipos diferentes en la misma línea. Por ejemplo:

```
int foo, fooarray[]; // ¡MAL!
```

Nota: Los ejemplos anteriores usan un espacio entre el tipo y el identificador. Otra alternativa aceptable es usar tabuladores, ej:

```
int     level;          // nivel de tabulación
int     size;           // tamaño de la tabla
Object  currentEntry;   // elemento de la tabla seleccionado
```

Inicialización

Hay que intentar inicializar las variables locales donde se declaren. La única razón para no inicializar una variable donde se declara es si el valor inicial depende de algún cálculo que debe ocurrir primero.

Colocación

Poned las declaraciones sólo al principio de los bloques. (Un bloque es cualquier código rodeado por llaves "{" y "}".) No esperar a declarar las variables hasta su primer uso; puede confundir a un programador incauto y dificultar la portabilidad del código dentro del ámbito.

```
void myMethod() {
    int int1 = 0;         // principio del bloque de método

    if (condition) {
        int int2 = 0;     // principio del bloque de "if"
        ...
    }
}
```

La única excepción a esta regla es el índice de los bucles `for`, que en Java se puede declarar dentro de la sentencia `for`:

```
for (int i = 0; i < maxLoops; i++) { ... }
```

Evitar declaraciones locales que oculten declaraciones de más alto nivel. Por ejemplo, no declare el mismo nombre de variable en un bloque interno:

```
int count;
...
myMethod() {
    if (condition) {
        int count = 0;     // EVITAR
        ...
    }
    ...
}
```

Declaraciones de clase e interfaz

Mientras se codifican clases e interfaces Java, se deberían seguir las siguientes reglas de formato:

- Ningún espacio entre el nombre del método y el paréntesis "(" que abre su lista de parámetros.
- La llave de apertura "{" aparece al final de la misma línea que la sentencia de declaración.
- La llave de cierre "}" comienza una línea nueva tabulada para coincidir con su sentencia de apertura correspondiente, excepto cuando es un bloque vacío que la llave de cierre "}" debería aparecer inmediatamente después de la de apertura "{".

- Los métodos están separados por una línea en blanco.

```
class Sample extends Object {
    int ivar1;
    int ivar2;

    Sample(int i, int j) {
        ivar1 = i;
        ivar2 = j;
    }

    int emptyMethod() {}

    ...
}
```

Sentencias

Sentencias simples

Cada línea debería contener una sentencia como mucho. Por ejemplo:

```
argv++;       // Correcto
argc--;       // Correcto
argv++; argc--;       // ¡EVITAR!
```

Sentencias compuestas

Las sentencias compuestas son sentencias que contienen una lista de sentencias encerradas entre llaves "{" y "}". Ver las siguientes secciones para encontrar ejemplos.

- Las sentencias internas deberían estar tabuladas un nivel más que la sentencia compuesta.
- La llave de apertura debería estar al final de la línea que comienza la sentencia compuesta; la llave de cierre debería estar en una nueva línea y estar tabulada al nivel del principio de la sentencia compuesta.
- Las llaves se usan en todas las sentencias compuestas, incluídas las sentencias únicas, cuando forman parte de una estructura de control, como una sentencia `if-else` o un bucle `for`. Esto hace más fácil introducir nuevas sentencias sin provocar errores accidentales al olvidarse añadir las llaves.

Sentencias `return`

Una sentencia `return` con un valor no debería usar paréntesis a menos que haga el valor devuelto más obvio de alguna manera. Por ejemplo:

```
return;

return myDisk.size();

return (size > 0 ? size : defaultSize);
```

Sentencias `if`, `if-else`, `if-else-if-else`

El tipo de sentencias `if-else` debería tener el siguiente formato:

```
if (condición) {
    sentencias;
}

if (condición) {
    sentencias;
} else {
    sentencias;
}

if (condición) {
    sentencias;
} else if (condición) {
    sentencias;
} else {
    sentencias;
}
```

Nota: Las sentencias `if` siempre llevan llaves {}. Evitar la siguiente forma, propensa a errores:

```
if (condición) // ¡EVITAR OMITIR LAS LLAVES! {}
    sentencia;
```

Sentencias `for`

Una sentencia `for` debería tener el siguiente formato:

```
for (inicialización; condición; actualización) {
    sentencias;
}
```

Una sentencia `for` vacía (aquella en la que todo el trabajo se hace en las cláusulas de inicialización, condición y actualización) debería tener el siguiente formato:

```
for (inicialización; condición; actualización);
```

Cuando se use el operador coma en la cláusula de inicialización o actualización, evitar la complejidad de utilizar más de tres variables. Si se necesita, usar sentencias separadas antes del bucle `for` (para la cláusula de inicialización) o al final del bucle (para la cláusula de actualización).

Sentencias `while`

Una sentencia `while` debería tener el siguiente formato:

```
while (condición) {
    sentencias;
}
```

Una sentencia `while` vacía debería tener el siguiente formato:

```
while (condición);
```

Sentencias `do-while`

Una sentencia `do-while` debería tener el siguiente formato:

```
do {
    sentencias;
} while (condición);
```

Sentencias `switch`

Una sentencia `switch` debería tener el siguiente formato:

```
switch (condición) {
case ABC:
    sentencias;
    /* continua con el siguiente */

case DEF:
    sentencias;
    break;

case XYZ:
    sentencias;
    break;

default:
    sentencias;
    break;
}
```

Cada vez que un caso continúa con el siguiente (no incluye una sentencia `break`), se añade un comentario donde iría la sentencia `break`. Esto se muestra en el ejemplo anterior con el comentario `/* continua con el siguiente */`.

Todas las sentencias `switch` deberían incluir un caso por defecto. El `break` en el caso por defecto es redundante, pero previene un error de continuar con el siguiente si más adelante se incluye otro caso.

Sentencias `try-catch`

Una sentencia `try-catch` debería tener el siguiente formato:

```
try {
    sentencias;
} catch (ExceptionClass e) {
    sentencias;
}
```

Una sentencia `try-catch` puede venir seguida de una sentencia `finally`, la cual se ejecuta siempre independientemente de que el bloque `try` se haya completado correctamente o no.

```
try {
    sentencias;
} catch (ExceptionClass e) {
    sentencias;
} finally {
    sentencias;
}
```

Espacios en blanco

Líneas en blanco

Las líneas en blanco mejoran la legibilidad resaltando secciones de código que están relacionadas lógicamente.

En las siguientes circunstancias, siempre se deberían usar dos líneas en blanco:

- Entre secciones de un fichero fuente.
- Entre definiciones de clases e interfaces.

En las siguientes circunstancias, siempre se debería usar una línea en blanco:

- Entre métodos.
- Entre las variables locales de un método y su primera sentencia.
- Antes de un comentario de bloque o de una sola línea.
- Entre las secciones lógicas de un método, para mejorar la legibilidad.

Espacios en blanco

Los espacios en blanco deberían usarse en las siguientes circunstancias:

- Una palabra clave seguida por un paréntesis debería estar separado por un espacio. Por ejemplo:

  ```
  while (true) {
      ...
  }
  ```

  Nótese que entre el nombre de un método y sus paréntesis no debe haber espacios en blanco. Esto ayuda a distinguir entre palabras claves y llamadas a métodos.
- En las listas de argumentos, debería haber un espacio después de cada coma.
- Todos los operadores binarios, excepto el operador punto (`.`) deberían estar separados de sus operandos por espacios. Los operadores unarios (incremento `++`, decremento `--`, negativo `-`) nunca deberían estar separados de sus operandos. Por ejemplo:

  ```
  a += c + d;
  a = (a + b) / (c * d);

  while (d++ = s++) {
      n++;
  }
  print("el resultado es " + foo + "\n");
  ```

- Las expresiones de una sentencia `for` deberían estar separadas por espacios en blanco. Por ejemplo:

  ```
  for (expr1; expr2; expr3)
  ```

- Las conversiones de tipo (cast) deberían estar seguidas de un espacio en blanco. Por ejemplo:

  ```
  myMethod((byte) aNum, (Object) x);
  myMethod((int) (cp + 5), ((int) (i + 3))
  ```

```
+ 1);
```

Convenciones de nombrado

Las convenciones de nombrado hacen los programas más comprensibles haciéndolos más fáciles de leer. También pueden dar información acerca de la función del identificador (por ejemplo, si se trata de una constante, un paquete o una clase), que puede ayudar a entender el código.

Tipo de identificador	Reglas de nombrado	Ejemplos

Paquetes	El prefijo de un nombre de paquete único se escribe siempre en letras ASCII minúsculas y debería ser uno de los nombres de dominio de nivel superior (actualmente com, edu, gov, mil, net, org ó uno de los códigos de país de dos letras, como se especifica en el estándar ISO 3166 [6]).
	
	Los siguientes componentes del nombre del paquete varían de acuerdo a las propias convenciones de nombrado internas de las organizaciones. Dichas convenciones pueden especificar que ciertos componentes de nombre de directorio sean división, departamento, proyecto, máquina o nombres de usuario.
		`com.sun.eng`
		
		`com.apple.quicktime.v2`
		
		`edu.cmu.cs.bovik.cheese`
		
		`es.nom.jpereza`

Clases	Los nombres de clases deberían ser sustantivos, escritos en *CamelCase* [10] con la primera letra en mayúscula. Tratar de	`class Raster`
		
		`class ImageSprite`

Tipo de identificador	Reglas de nombrado	Ejemplos
	mantener los nombres de clases simples y descriptivos. Usar palabras completas, evitar acrónimos y abreviaturas (a menos que la abreviatura sea mucho más usada que la forma larga, como URL o HTML).	
Interfaces	Los nombres de interfaz deberían ser escritos como los nombres de clases.	`interface RasterDelegate` `interface Storing`

Métodos	Los nombres de métodos deberían ser verbos, escritos en *CamelCase* [10] con la primera letra en minúscula.	`run();`
		`runFast();`
		`getBackground();`

Variables	Todos los nombres de variable deberían estar escritos en *CamelCase* [10] con la primera letra en minúscula. No deberían comenzar con un caracter de subrayado (_) o un signo de dólar ($), aunque ambos están permitidos.
	
	Los nombres de variables deberían ser cortos aunque significativos. La elección de un nombre de variable debería ser mnemotécnica, esto es, pensada para indicar la intención de su uso a un posible observador ocasional. Se deberían evitar los nombres de
		`int     i;`
		`char    c;`
		`float   myWidth;`
		`String streetName;`

Tipo de identificador	Reglas de nombrado	Ejemplos
	variables de un solo caracter excepto para variables temporales "desechables". Algunos nombres comunes para variables temporales son `i`, `j`, `k`, `m` y `n` para números enteros; `c`, `d` y `e` para caracteres.	

Constantes	Los nombres de variables declaradas como constantes de clase deberían estar escritos todo en mayúsculas separando las palabras con un caracter de subrayado (_).	`static final int MIN_WIDTH = 4;`
		`static final int MAX_WIDTH = 999;`
		`static final String DEFAULT_PROTOCOL = "http";`

Prácticas de programación

Proporcionar acceso a variables de clase e instancia

No hacer pública ninguna variable de clase o instancia sin una buena razón. Con frecuencia, las variables de instancia no necesitan ser accedidas o modificadas explícitamente.

Un ejemplo apropiado de variables de instancia públicas es el caso en el que la clase es esencialmente una estructura de datos, sin comportamiento. En otras palabras, si se hubiése usado un `struct` en lugar de una clase (si Java soportara `struct`), entonces es apropiado hacer públicas las variables de instancia.

Referenciar variables y métodos de clase

Evitar usar una instancia de un objeto para acceder a un método o variable de clase (estática). Usar el nombre de la clase en su lugar. Por ejemplo:

```
classMethod();              // OK
AClass.classMethod();       // OK
anObject.classMethod();     // ¡EVITAR!
```

Constantes

Las constantes numéricas no deberían codificarse directamente, excepto -1, 0 y 1, que pueden

aparecer en un bucle `for` como contadores. Por ejemplo:

```
static final int MAX_SIZE = 25;

for (i = 0; i < MAX_SIZE; i++)
```

Asignaciones de variables

Evitar asignar a varias variables el mismo valor en una sola sentencia. Es dificil de leer. Ejemplo:

```
fooBar.fChar = barFoo.lchar = 'c'; // ¡EVITAR!
```

No usar el operador de asignación (=) en un lugar donde se pueda confundir fácilmente con el operador de igualdad (==). Ejemplo:

```
if (c++ = d++) {        // ¡EVITAR! (Java lo rechaza)
    ...
}
```

debería ser escrito así:

```
if ((c++ = d++) != 0) {
    ...
}
```

No usar asignaciones incrustadas en un intento de mejorar el rendimiento en tiempo de ejecución. Esto es trabajo del compilador. Ejemplo:

```
d = (a = b + c) + r;        // ¡EVITAR!
```

debería ser escrito así:

```
a = b + c;
d = a + r;
```

Prácticas varias

Paréntesis

Generalmente, es una buena idea usar paréntesis generosamente en expresiones que tienen operadores mezclados para evitar problemas de precedencia de operadores. Incluso si la precedencia de operador parece clara, puede no serlo para otros (no se debería suponer que otros programadores conocen tan bien la precedencia de operadores como tú).

```
if (a == b && c == d)     // ¡EVITAR!
if ((a == b) && (c == d)) // BIEN
```

Devolver valores

Tratar de hacer que la estructura del programa coincida con su propósito. Ejemplo:

```
if (expresiónBoolean) {
    return true;
} else {
```

```
    return false;
}
```

debería ser escrito así:

```
return expresiónBoolean;
```

Igualmente,

```
if (condición) {
    return x;
}
return y;
```

debería ser escrito así:

```
return (condición ? x : y);
```

Expresiones antes de '?' en el operador condicional

Si una expresión con un operador binario aparece antes de `?` en el operador ternario `?:`, debería ser puesta entre paréntesis. Ejemplo:

```
(x >= 0) ? x : -x;
```

Comentarios especiales

Usar XXX en un comentario para indicar algo que funciona pero que no está del todo bien.
Usar FIXME (fix me, corrígeme) para indicar algo que no funciona del todo y debe corregirse.
Usar TODO (to do, hacer) para indicar algo que no está totalmente terminado.

Ejemplo de código fuente Java

El siguiente ejemplo muestra el formato de un fichero de código fuente Java que contiene una clase pública. Los interfaces tienen un formato similar.

```
/*
 * @(#)Blah.java        1.82 99/03/18
 *
 * Copyright (c) 1994-1999 Sun Microsystems, Inc.
 * 901 San Antonio Road, Palo Alto, California, 94303, U.S.A.
 * All rights reserved.
 *
 * This software is the confidential and proprietary information of Sun
 * Microsystems, Inc. ("Confidential Information").  You shall not
 * disclose such Confidential Information and shall use it only in
 * accordance with the terms of the license agreement you entered into
 * with Sun.
 */


package java.blah;

import java.blah.blahdy.BlahBlah;
```

```
/**
 * La descripción de la clase va aquí.
 *
 * @version     1.82 18 Mar 1999
 * @author      Nombre Apellido
 */
public class Blah extends SomeClass {
    /* Un comentario de implementación de clase puede ir aquí. */

    /** Comentario de documentación de classVar1 */
    public static int classVar1;

    /**
     * Comentario de documentación de classVar2
     * que ocupa más de una línea
     */
    private static Object classVar2;

    /** Comentario de documentación de instanceVar1 */
    public Object instanceVar1;

    /** Comentario de documentacion de instanceVar2 */
    protected int instanceVar2;

    /** Comentario de documentación de instanceVar3 */
    private Object[] instanceVar3;

    /**
     * ...comentario de documentación del constructor de Blah...
     */
    public Blah() {
        // ...la implementación va aquí...
    }

    /**
     * ...comentario de documentación del método doSomething...
     */
    public void doSomething() {
        // ...la implementación va aquí...
    }

    /**
     * ...comentario de documentación del método doSomethingElse...
     * @param someParam descripción del parámetro
     */
    public void doSomethingElse(Object someParam) {
        // ...la implementación va aquí...
    }
}
```

URL de Origen (recibido en 25/08/2009 - 22:33): http://jpereza.nom.es/convenciones-de-codigo-java

Enlaces:
[1] http://java.sun.com/docs/codeconv/
[2] http://java.sun.com/docs/books/jls/index.html
[3] http://java.sun.com/docs/forms/sendusmail.html
[4] http://jpereza.nom.es/contacto

[5] http://jpereza.nom.es/convenciones-de-codigo-java/ejemplo-de-codigo-fuente-java
[6] http://www.iso.org/iso/en/prods-services/iso3166ma/02iso-3166-code-lists/list-en1.html
[7] http://jpereza.nom.es/convenciones-de-codigo-java/comentarios
[8] http://java.sun.com/j2se/javadoc/writingdoccomments/
[9] http://java.sun.com/javadoc/
[10] http://es.wikipedia.org/wiki/CamelCase