Convenciones de Código para el lenguaje de programación

JAVA™

Revisado 20 Abril de 1999
por Scott Hommel
Sun Microsystems Inc.

Traducido al castellano 10 Mayo del 2001
por Alberto Molpeceres
http://www.javahispano.com

Conveciones de código para el lenguaje de programación Java™

Revisado, 20 de Abril de 1999
Traducido al castellano, 10 de Mayo del 2001

Convenciones de código para el lenguaje de programación Java™

1 - Introducción

1.1 Por qué convenciones de código

Las convenciones de código son importantes para los programadores por un gran número de razones:

- El 80% del coste del código de un programa va a su mantenimiento.
- Casi ningún software lo mantiene toda su vida el auto original.
- Las convenciones de código mejoran la lectura del software, permitiendo entender código nuevo mucho más rapidamente y más a fondo.
- Si distribuyes tu código fuente como un producto, necesitas asegurarte de que esta bien hecho y presentado como cualquier otro producto.

  Para que funcionen las convenciones, cada persona que escribe software debe seguir la convención. Todos.

1.2 Agradecimientos

Este documento refleja los estandares de codificación del lenguaje Java presentados en *Java Language Specification* , de Sun Microsystems, Inc. Los mayores contribuidores son Peter King, Patrick Naughton, Mike DeMoney, Jonni Kanerva, Kathy Walrath, y Scott Hommel.

Este documento es mantenido por Scott Hommel. Enviar los comentarios a shommel@eng.sun.com

1.3 Sobre la traducción

Este documento ha sido traducido al español por Alberto Molpeceres, para el sitio web javaHispano (www.javaHispano.com), y se encuentra ligado al objetivo de dicha web para fomentar el uso y conocimiento del lenguaje Java dentro del mundo hispano-hablante.

Se ha intentado hacer una traducción lo más literal posible, y esta es la única parte del documento que no pertenece a la versión original.

Se pueden enviar los comentarios sobre la traducción a la dirección: al@javahispano.com

2 - Nombres de ficheros

Esta sección lista las extensiones más comunes usadas y los nombres de ficheros.

2.1 Extensiones de los ficheros

El software Java usa las siguientes extensiones para los ficheros:

Tipo de fichero	Extensión
Fuente Java	`.java`
Bytecode de Java	`.class`

2.2 Nombres de ficheros comúnes

Los nombres de ficheros más utilizados incluyen:

Nombre de fichero	Uso
`GNUmakefile`	El nombre preferido para ficheros "make". Usamos `gnumake` para construir nuestro software.
`README`	El nombre preferido para el fichero que resume los contenidos de un directorio particular.

3 - Organización de los ficheros

Un fichero consiste de secciones que deben estar separadas por líneas en blanco y comentarios opcionales que identifican cada sección.

Los ficheros de más de 2000 líneas son incómodos y deben ser evitados.

Para ver un ejemplo de un programa de Java debidamente formateado, ver "Ejemplo de fichero fuente Java" en la página 26.

3.1 Ficheros fuente Java

Cada fichero fuente Java contiene una única clase o interface pública. Cuando algunas clases o interfaces privadas estan asociadas a una clase pública, pueden ponerse en el mismo fichero que la clase pública. La clase o interfaz pública debe ser la primera clase o interface del fichero.

Los ficheros fuentes Java tienen la siguiente ordenación:

- Comentarios de comienzo (ver "Comentarios de comienzo" en la página 7)
- Sentencias package e import
- Declaraciones de clases e interfaces (ver "Declaraciones de clases e interfaces" en la página 7)

3.1.1 Comentarios de comienzo

Todos los ficheros fuente deben comenzar con un comentario en el que se lista el nombre de la clase, información de la versión, fecha, y copyright:

```
/*
 * Nombre de la clase
 *
 * Informacion de la version
 *
 * Fecha
 *
 * Copyright
 */
```

3.1.2 Sentencias package e import

La primera línea no-comentario de los ficheros fuente Java es la sentencia `package`. Después de esta, pueden seguir varias sentencias `import`. Por ejemplo:

```
package java.awt;

import java.awt.peer.CanvasPeer;
```

Nota: El primer componente de el nombre de un paquete único se escribe siempre en minúsculas con caracteres ASCII y debe ser uno de los nombres de dominio de último nivel, actualmente com, edu, gov, mil, net, org, o uno de los códigos ingleses de dos letras que especifican el pais como se define en el ISO Standard 3166, 1981.

3.1.3 Declaraciones de clases e interfaces

La siguiente tabla describe las partes de la declaración de una clase o interface, en el orden en que deberian aparecer. Ver "Ejemplo de fichero fuente Java" en la página 26 para un ejemplo que incluye comentarios.

	Partes de la declaración de una clase o interface	**Notas**
1	Comentario de documentación de la clase o interface (`/**...*/`)	Ver "Comentarios de documentación" en la página 14 para más información sobre lo que debe aparecer en este comentario.
2	Sentencia `class` o `interface`	
3	Comentario de implementación de la clase o interface si fuera necesario (`/*...*/`)	Este comentario debe contener cualquier información aplicable a toda la clase o interface que no era apropiada para estar en los comentarios de documentación de la clase o interface.
4	Variables de clase (`static`)	Primero las variables de clase `public`, después las `protected`, después las de nivel de paquete (sin modificador de acceso) , y despúes las `private`.
5	Variables de instancia	Primero las `public`, después las `protected`, después las de nivel de paquete (sin modificador de acceso), y después las `private`.
6	Constructores	
7	Métodos	Estos métodos se deben agrupar por funcionalidad más que por visión o accesibilidad. Por ejemplo, un método de clase privado puede estar entre dos métodos públicos de instancia. El objetivo es hacer el código mas legible y comprensible.

4 - Indentación

Se deben emplear cuatro espacios como unidad de indentación. La construcción exacta de la indentación (espacios en blanco contra tabuladores) no se especifica. Los tabuladores deben ser exactamente cada 8 espacios (no 4).

4.1 Longitud de la línea

Evitar las líneas de más de 80 caracteres, ya que no son manejadas bien por muchas terminales y herramientas.

Nota: Ejemplos para uso en la documentación deben tener una longitud inferior, generalmente no más de 70 caracteres.

4.2 Rompiendo líneas

Cuando una expresión no entre en una línea, romperla de acuerdo con estos principios:

- Romper después de una coma.
- Romper antes de un operador.
- Preferir roturas de alto nivel (más a la derecha que el "padre") que de bajo nivel (más a la izquierda que el "padre").
- Alinear la nueva linea con el comienzo de la expresion al mismo nivel de la linea anterior.
- Si las reglas anteriores llevan a código confuso o a código que se aglutina en el margen derecho, indentar justo 8 espacios en su lugar.

Ejemplos de como romper la llamada a un método:

```
unMetodo(expresionLarga1, expresionLarga2, expresionLarga3,
         expresionLarga4, expresionLarga5);

var = unMetodo1(expresionLarga1,
                unMetodo2(expresionLarga2,
                          expresionLarga3));
```

Ahora dos ejemplos de roptura de lineas en expresiones aritméticas. Se prefiere el primero, ya que el salto de linea ocurre fuera de la expresión que encierra los paréntesis.

```
nombreLargo1 = nombreLargo2 * (nombreLargo3 + nombreLargo4

               - nombreLargo5) + 4 * nombreLargo6; // PREFERIDA


nombreLargo1 = nombreLargo2 * (nombreLargo3 + nombreLargo4

                             - nombreLargo) + 4 * nombreLargo6;
// EVITAR
```

Ahora dos ejemplos de indentación de declaraciones de métodos. El primero es el caso convencional. El segundo conduciría la segunda y la tercera linea demasiado hacia la izquierda con la indentación convencional, asi que en su lugar se usan 8 espacios de indentación.

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

//INDENTACION DE 8 ESPACIOS PARA EVITAR GRANDES INDENTACIONES
private static synchronized metodoDeNombreMuyLargo(int unArg,
        Object otroArg, String todaviaOtroArg,
        Object unOtroMas) {
    ...
}
```

Saltar de lineas por sentencias `if` debera seguir generalmente la regla de los 8 espacios, ya que la indentacion convencional (4 espacios) hace difícil ver el cuerpo. Por ejemplo:

```
//NO USAR ESTA INDENTACION
if ((condicion1 && condicion2)
    || (condicion3 && condicion4)
    ||!(condicion5 && condicion6)) { //MALOS SALTOS
    hacerAlgo();                    //HACEN ESTA LINEA FACIL
DE OLVIDAR
}

//USE THIS INDENTATION INSTEAD
if ((condicion1 && condicion2)
        || (condicion3 && condicion4)
        ||!(condicion5 && condicion6)) {
    hacerAlgo();
}

//O USAR ESTA
if ((condicion1 && condicion2) || (condicion3 && condicion4)
        ||!(condicion5 && condicion6)) {
    hacerAlgo();
```

```
}
```

Hay tres formas aceptables de formatear expresiones ternarias:

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


alpha = (unaLargaExpresionBooleana) ? beta
                                    : gamma;


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

5 - Comentarios

Los programas Java pueden tener dos tipos de comentarios: comentarios de implementación y comentarios de documentación. Los comentarios de implementación son aquellos que también se encuentran en C++, delimitados por /*...*/, y //. Los comentarios de documentación (conocidos como "doc comments") existen sólo en Java, y se limitan por /**...*/. Los comentarios de documentación se pueden exportar a ficheros HTML con la herramienta javadoc.

Los comentarios de implementación son para comentar nuestro código o para comentarios acerca de una implementación particular. Los comentarios de documentación son para describir la especificación del código, libre de una perspectiva de implementación, y para ser leidos por desarrolladores que pueden no tener el código fuente a mano.

Se deben usar los comentarios para dar descripciones de código y facilitar información adicional que no es legible en el código mismo. Los comentarios deben contener sólo información que es relevante para la lectura y entendimiento del programa. Por ejemplo, información sobre como se construye el paquete correspondiente o en que directorio reside no debe ser incluida como comentario.

Son apropiadas las discusiones sobre decisiones de diseño no triviales o no obvias, pero evitar duplicar información que esta presente (de forma clara) en el código ya que es fácil que los comentarios redundantes se queden desfasados. En general, evitar cualquier comentario que pueda quedar desfasado a medida que el código evoluciona.

Nota: La frecuencia de comentarios a veces refleja una pobre calidad del código. Cuando se sienta obligado a escribir un comentario considere reescribir el código para hacerlo más claro.

Los comentarios no deben encerrarse en grandes cuadrados dibujados con asteriscos u otros caracteres.
Los comentarios nunca deben incluir caracteres especiales como backspace.

5.1 Formatos de los comentarios de implementación

Los programas pueden tener cuatro estilos de comentarios de implementación: de bloque, de una línea, de remolque, y de fin de línea

5.1.1 Comentarios de bloque

Los comentarios de bloque se usan para dar descripciones de ficheros, métodos, estructuras de datos y algoritmos. Los comentarios de bloque se podrán usar al comienzo de cada ficehro o antes de cada método. También se pueden usar en otro lugares, tales como el interior de los métodos. Los comentarios de bloque en el interior de una función o método deben ser indentados al mismo nivel que el código que describen.

Un cometario de bloque debe ir precedido por una línea en blanco que lo separe del resto del código.

```
/*
 * Aqui hay un comentario de bloque.
 */
```

Los comentarios de bloque pueden comenzar con `/*-`, que es reconocido por **indent**(1) como el comienzo de un comentario de bloque que no debe ser reformateado. Ejemplo:

```
/*-
```

```
 * Aqui tenemos un comentario de bloque con cierto
 * formato especial que quiero que ignore indent(1).
 *
 *    uno
 *        dos
 *            tres
 */
```

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

Ver también "Comentarios de documentación" en la página 14.

5.1.2 Comentarios de una linea

Pueden aparecer comentarios cortos de una única línea al nivel del código que siguen. Si un comentario no se puede escribir en una línea, debe seguir el formato de los comentarios de bloque. (ver sección 5.1.1). Un comentario de una sóla línea debe ir precedido de una línea en blanco. Aqui un ejemplo de comentario de una sola línea en código Java (ver también "Comentarios de documentación" en la página 14):

```
if (condicion) {

    /* Código de la condicion. */
    ...
}
```

5.1.3 Comentarios de remolque

Pueden aparecer comentarios muy pequeños en la misma línea que describen, pero deben ser movidos lo suficientemente lejos para separarlos de las sentencias. Si más de un comentario corto aparecen en el mismo trozo de código, deben ser indentados con la misma profundidad.

Aquí un ejemplo de comentario de remolque:

```
if (a == 2) {
    return TRUE;            /* caso especial */
} else {
    return isPrime(a);      /* caso gerenal */
}
```

5.1.4 Comentarios de fin de linea

El delimitador de comentario `//` puede convertir en comentario una línea completa o una parte de una linea. No debe ser usado para hacer comentarios de varias líneas consecutivas; sin embargo, puede usarse en líneas consecutivas para comentar secciones de código. Aquí teneis ejemplos de los tres estilos:

```
if (foo > 1) {

    // Hacer algo.
    ...
}
else {
    return false;          // Explicar aqui por que.
```

```
}
//if (bar > 1) {
//
//    // Hacer algo.
//    ...
//}
//else {
//    return false;
//}
```

5.2 Comentarios de documentación

Nota: Ver "Ejemplo de fichero fuente Java" en la página 26 para ejemplos de los formatos de comentarios descritos aquí.

Para más detalles, ver "How to Write Doc Comments for Javadoc" que incluye información de las etiquetas de los comentarios de documentación (@return, @param, @see):

http://java.sun.com/products/jdk/javadoc/writingdoccomments.shtml

Nota del Traductor: ¿Alguien se aníma a traducirlo?, háznoslo saber en coordinador@javahispano.com

Para más detalles acerca de los comentarios de documentación y javadoc, visitar el sitio web de javadoc:

http://java.sun.com/products/jdk/javadoc/

Los comentarios de documentación describen clases Java, interfaces, constructores, métodos y atributos. Cada comentario de documentación se encierra con los delimitadores de comentarios `/**...*/`, con un comentario por clase, interface o miembro (método o atributo). Este comentario debe aparecer justo antes de la declaración:

```
/**
 * La clase Ejemplo ofrece ...
 */
public class Ejemplo { ...
```

Darse cuenta de que las clases e interfaces de alto nivel son estan indentadas, mientras que sus miembros los están. La primera línea de un comentario de documentación (/**) para clases e interfaces no esta indentada, subsecuentes líneas tienen cada una un espacio de intentación (para alinear verticalmetne los asteriscos). Los miembros, incluidos los constructores, tienen cuatro espacios para la primera línea y 5 para las siguientes.

Si se necesita dar información sobre una clase, interface, variable o método que no es apropiada para la documentación, usar un comentario de implementación de bloque (ver sección 5.1.1) o de una línea (ver sección 5.1.2) para comentarlo inmediatamente *despues* de la declaración. Por ejemplo, detalles de implementación de una clase deben ir en un comentario de implementación de bloque *siguiendo* a la sentencia class, no en el comentario de documentación de la clase.

Los comentarios de documentación no deben colocarse en el interior de la definición de un método o constructor, ya que Java asocia los comentarios de documentación con la *primera declaración después* del comentario.

6 - Declaraciones

6.1 Cantidad por línea

Se recomienda una declaración por línea, ya que facilita los comentarios. En otras palabras, se prefiere

```
int nivel; // nivel de indentación
int tam;   // tamaño de la tabla
```

antes que

```
int level, size;
```

No poner diferentes tipos en la misma línea. Ejemplo:

```
int foo,  fooarray[]; //ERROR!
```

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

```
int      level;          // nivel de indentacion
int      size;            // tamaño de la tabla
Object   currentEntry;   // entrada de la tabla seleccionada
actualmente
```

6.2 Inicialización

Intentar inicializar las variables locales donde se declaran. La única razón para no inicializar una variable donde se declara es si el valor inicial depende de algunos cálculos que deben ocurrir.

6.3 Colocación

Poner las declaraciones solo al principio de los bloques (un bloque es cualquier código encerrado por llaves "{" y "}".) No esperar al primer uso para declararlas; puede confundir a programadores no preavisados y limitar la portabilidad del código dentro de su ámbito de visibilidad.

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

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

La excepción de la regla son los indices de bucles `for`, que en Java se pueden declarar en la sentencia `for`:

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

Evitar las declaraciones locales que ocultan declaraciones de niveles superiores. por ejemplo, no declarar la misma varible en un bloque interno:

```
int cuenta;
...
miMetodo() {
    if (condicion) {
        int cuenta = 0;     // EVITAR!
        ...
    }
```

```
    ...
}
```

6.4 Declaraciones de class e interfaces

Al codificar clases e interfaces de Java, se siguen las siguientes reglas de formato:

- Ningún espacio en blanco entre el nombre de un método y el paréntesis "(" que abre su lista de parámetros
- La llave de apertura "{" aparece al final de la misma línea de la sentencia declaracion
- La llave de cierre "}" empieza una nueva línea indentada para ajustarse a su sentencia de apertura correspondiente, excepto cuando no existen sentencias entre ambas, que debe aparecer inmediatamente después de la de apertura "{"

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

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

    int metodoVacio() {}

    ...
}
```

- Los métodos se separan con una línea en blanco

7 - Sentencias

7.1 Sentencias simples

Cada línea debe contener como mucho una sentencia. Ejemplo:

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

7.2 Sentencias compuestas

Las sentencias compuestas son sentencias que contienen listas de sentencias encerradas entre llaves "`{ sentencias }`". Ver la siguientes secciones para ejemplos.

- Las sentencias encerradas deben indentarse un nivel más que la sentencia compuesta.
- La llave de apertura se debe poner al final de la linea que comienza la sentencia compuesta; la llave de cierre debe empezar una nueva línea y ser indentada al mismo nivel que el pincipio de la sentencia compuesta.
- Las llaves se usan en todas las sentencias, incluso las simples, cuando forman parte de una estructura de control, como en las sentencias `if-else` o `for`. Esto hace más sencillo añadir sentencias sin incluir bugs accidentalmente por olvidar las llaves.

7.3 Sentencias de retorno

Una sentencia `return` con un valor no debe usar paréntesis a menos que hagan el valor de retorno más obvio de alguna manera. Ejemplo:

```
return;

return miDiscoDuro.size();

return (tamanyo ? tamanyo : tamanyoPorDefecto);
```

7.4 Sentencias if, if-else, if else-if else

La clase de sentencias `if-else` debe tener la siguiente forma:

```
if (condicion) {
    sentencias;
}

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

if (condicion) {
    sentencia;
} else if (condicion) {
    sentencia;
} else{
    sentencia;
}
```

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

```
if (condicion) //EVITAR! ESTO OMITE LAS LLAVES {}!
    sentencia;
```

7.5 Sentencias for

Una sentencia `for` debe tener la siguiente forma:

```
for (inicializacion; condicion; actualizacion) {
    sentencias;
}
```

Una sentencia `for` vacía (una en la que todo el trabajo se hace en las clausulas de inicialización, condicion, y actualizacion) debe tener la siguiente forma:

```
for (inicializacion; condicion; actualizacion);
```

Al usar el operador coma en la clausula de inicialización o actualización de una sentencia `for`, evitar la complejidad de usar más de tres variables. Si se necesita, usar sentencias separadas antes de bucle `for` (para la clausula de inicialización) o al final del bucle (para la clausula de actualizacion).

7.6 Sentencias while

Una sentencia `while` debe tener la siguiente forma:

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

Una sentencia `while` vacía debe tener la siguiente forma:

```
while (condicion);
```

7.7 Sentencias do-while

Una sentencia `do-while` debe tener la siguiente forma:

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

7.8 Sentencias switch

Una sentencia `switch` debe tener la siguiente forma:

```
switch (condicion) {
case ABC:
    sentencias;
    /* este caso se propaga */

case DEF:
    sentencias;
    break;

case XYZ:
    sentencias;
    break;

default:
    sentencias;
    break;
}
```

Cada vez que un caso se propaga (no incluye la sentencia `break`), añadir un comentario donde la sentencia `break` se encontraría normalmente. Esto se muestra en el ejemplo anterior con el comentario `/* este caso se propaga */`.

Cada sentencia `switch` debe incluir un caso por defecto. El `break` en el caso por defecto es redundante, pero prevee que se propague por error si luego se añade otro caso.

7.9 Sentencias try-catch

Una sentencia `try-catch` debe tener la siguiente forma:

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

Una sentencia `try-catch` puede ir seguida de un `finally`, cuya ejecución se ejecutará independientemente de que el bloque `try` se halla completado con éxito o no.

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

8 - Espacios en blanco

8.1 Líneas en blanco

Las líneas en blanco mejoran la facilidad de lectura separando secciones de código que están lógicamente relacionadas.

Se deben usar siempre dos líneas en blanco en las siguientes circunstancias:

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

Se debe usar siempre una línea en blanco en las siguientes circunstancias:

- Entre métodos
- Entre las variables locales de un método y su primera sentencia
- Antes de un comentario de bloque (ver sección 5.1.1) o de un comentario de una línea (ver sección 5.1.2)
- Entre las distintas secciones lógicas de un método para facilitar la lectura.

8.2 Espacios en blanco

Se deben usar espacios en blanco en las siguientes circunstancias:

- Una palabra clave del lenguaje seguida por un paréntesis debe separarse por un espacio. Ejemplo:

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

  Notar que no se debe usar un espacio en blanco entre el nombre de un método y su paréntesis de apertura. Esto ayuda a distinguir palabras claves de llamadas a métodos.

- Debe aparecer un espacio en blanco después de cada coma en las listas de argumentos.
- Todos los operadores binarios excepto `.` se deben separar de sus operandos con espacios en blanco. Los espacios en blanco no deben separadar los operadores unarios, incremento ("++") y decremento ("--") de sus operandos. Ejemplo:

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

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

- Las expresiones en una sentencia `for` se deben separar con espacios en blanco. Ejemplo:

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

- Los "Cast"s deben ir seguidos de un espacio en blanco. Ejemplos:

```
miMetodo((byte) unNumero, (Object) x);
miMetodo((int) (cp + 5), ((int) (i + 3))
                                + 1);
```

9 - Convenciones de nombres

Las convenciones de nombres hacen los programas más entendibles haciendolos más fácil de leer. También pueden dar información sobre la función de un indentificador, por ejemplo, cuando es una constante, un paquete, o una clase, que puede ser útil para entender el código.

Tipos de identificadores	Reglas para nombras	Ejemplos
Paquetes	El prefijo del nombre de un paquete se escribe siempre con letras ASCII en minúsculas, y debe ser uno de los nombres de dominio de alto nivel, actualmente com, edu, gov, mil, net, org, o uno de los códigos ingleses de dos letras que identifícan cada país como se especifica en el ISO Standard 3166, 1981. Los subsecuentes componentes del nombre del paquete variarán de acuerdo a las convenciones de nombres internas de cada organización. Dichas convenciones pueden especificar que algunos nombres de los directorios correspondan a divisiones, departamentos, proyectos o máquinas.	com.sun.eng com.apple.quicktime.v2 edu.cmu.cs.bovik.cheese
Clases	Los nombres de las clases deben ser sustantivos, cuando son compuestos tendrán la primera letra de cada palabra que lo forma en mayúsculas. Intentar mantener los nombres de las clases simples y descriptivos. Usar palabras completas, evitar acrónimos y abreviaturas (a no ser que la abreviatura sea mucho más conocida que el nombre completo, como URL or HTML).	class Cliente; class ImagenAnimada;
Interfaces	Los nombres de las interfaces siguen la misma regla que las clases.	interface ObjetoPersistente; interface Almacen;
Métodos	Los métodos deben ser verbos, cuando son compuestos tendrán la primera letra en minúscula, y la primera letra de las	ejecutar(); ejecutarRapido();

	siguientes palabras que lo forma en mayúscula.	cogerFondo();

Variables	Excepto las constantes, todas las instancias y variables de clase o método empezarán con minúscula. Las palabras internas que lo forman (si son compuestas) empiezan con su primera letra en mayúsculas. Los nombres de variables no deben empezar con los caracteres subguión "_" o signo del dolar "$", aunque ambos estan permitidos por el lenguaje.
	
	Los nombres de las variables deben ser cortos pero con significado. La elección del nombre de una variable debe ser un mnemónico, designado para indicar a un observador casual su función. Los nombres de variables de un solo caracter se deben evitar, excepto para variables índices temporales. Nombres comúnes para variables temporales son `i`, `j`, `k`, `m`, y `n` para enteros; `c`, `d`, y `e` para caracteres.
		`int i;`
		`char c;`
		`float miAnchura;`

Constantes	Los nombres de las variables declaradas como constantes deben ir totalmente en mayúsculas separando las palabras con un subguión ("_"). (Las constantes ANSI se deben evitar, para facilitar su depuración.)	static final int ANCHURA_MINIMA = 4;
		
		static final int ANCHURA_MAXIMA = 999;
		
		static final int COGER_LA_CPU = 1;

10 - Hábitos de programación

10.1 Proporcionando acceso a variables de instancia y de clase

No hacer ninguna variable de instancia o clase pública sin una buena razón. A menudo las variables de instancia no necesitan ser asignadas/consultadas explícitamente, a menudo esto sucede como efecto lateral de llamadas a métodos.

Un ejemplo apropiado de una variable de instancia pública es el caso en que la clase es esencialemente una estructura de datos, sin comportamiento. En otras palabras, si usarías la palabra `struct` en lugar de una clase (si Java soportara `struct`), entonces es adecuado hacer las variables de instancia públicas.

10.2 Referencias a variables y métodos de clase

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

```
metodoDeClase();             //OK
UnaClase.metodoDeClase();    //OK
unObjeto.metodoDeClase();    //EVITAR!
```

10.3 Constantes

Las constantes numéricas (literales) no se deben codificar directamente, excepto -1, 0, y 1, que pueden aparecer en un bucle `for` como contadores.

10.4 Asignaciones de variables

Evitar asignar el mismo valor a varias varibles en la misma sentencia. Es difícil de leer. Ejemplo:

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

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

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

se debe escribir:

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

No usar asignación embebidas como un intento de mejorar el rendimient en tiempo de ejecución. Ese es el trabajo del compilador. Ejemplo:

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

se debe escribir:

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

10.5 Hábitos varios

10.5.1 Paréntesis

En general es una buena idea usar paréntesis en expresiones que implican distintos operadores para evitar problemas con el orden de precedencia de los operadores. Incluso si parece claro el orden de precedencia de los operadores, podría no ser así para otros, no se debe asumir que otros programadores conozcan el orden de precedencia.

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

10.5.2 Valores de retorno

Intentar hacer que la estructura del programa se ajuste a su intención. Ejemplo:

```
if (expresionBooleana) {
    return true;
} else {
    return false;
}
```

en su lugar se debe escribir

```
return expressionBooleana;
```

Similarmente,

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

se debe escribir:

```
return (condicion ? x : y);
```

10.5.3 Expresiones antes de `?' en el operador condicional

Si una expresión contiene un operador binario antes de `?` en el operador ternario `?:` , se debe colocar entre paréntesis. Ejemplo:

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

10.5.4 Comentarios especiales

Usar `XXX` en un comentario para indicar que algo tiene algún error pero funciona. Usar `FIXME` para indicar que algo tiene algún error y no funciona.

11 - Ejemplos de código

11.1 Ejemplo de fichero fuente Java

El siguiente ejemplo muestra como formatear un fichero fuente Java que contiene una sola clase pública. Los interfaces se formatean similarmente. Para más información, ver "Declaraciones de clases e interfaces" en la página 7 y "Comentarios de documentación" en la página 14.

```
/*
 * @(#)Bla.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.
 *
 * Más información y descripción del Copyright.
 *
 */


package java.bla;

import java.bla.blabla.BlaBla;

/**
 * La descripción de la clase viene aqui.
 *
 * @version      datos de la versión (numero y fecha)
 * @author       Nombre Apellido
 */
public class Bla extends OtraClase {
    /* Un comentario de implemantación de la clase viene aquí.
*/

    /** El comentario de documentación de claseVar1 */
    public static int claseVar1;

    /**
     * El comentario de documentación de classVar2
     * ocupa más de una línea
     */
    private static Object claseVar2;

    /** Comentario de documentación de instanciaVar1 */
    public Object instanciaVar1;

    /** Comentario de documentación de instanciaVar2 */
    protected int instanciaVar2;

    /** Comentario de documentación de instanciaVar3 */
    private Object[] instanciaVar3;

    /**
     * ...Comentario de documentación del constructor Bla...
     */
    public Bla() {
        // ...aquí viene la implementación...
    }
```

```
    /**
     * ...Comentario de documentación del método hacerAlgo...
     */
    public void hacerAlgo() {
        // ...aquí viene la implementación...
    }

    /**
     * ...Comentario de documentación de hacerOtraCosa...
     * @param unParametro descripción
     */
    public void hacerOtraCosa(Object unParametro) {
        // ...aquí viene la implementación...
    }
}
```

Convenciones del código Java: Copyright de Sun.

Puedes copiar, adaptar y redistribuir este documento para uso no comercial o para uso interno de un fin comercial. Sin embargo, no debes republicar este documento, ni publicar o distribuir una copia de este documento en otro caso de uso que el no comercial o el interno sin obtener anteriormente la aprobación expresa y por escrito de Sun.

Al copiar, adaptar o redistribuir este documento siguiendo las indicaciones anteriores, estas obligado a conceder créditos a Sun. Si reproduces o distribuyes el documento sin ninguna modificación sustancial en su contenido, usa la siguiente línea de creditos:

```
Copyright 1995-1999 Sun Microsystems, Inc. All rights reserved. Used by
permission.
```

Si modificas este documento de forma que altera su significado, por ejemplo, para seguir las convenciones propias de tu empresa, usa la siguiente línea de créditos:

```
Adapted with permission from JAVA CODE CONVENTIONS. Copyright 1995-1999 Sun
Microsysytems, Inc. All rights reserved.
```

En ambos caso, añade por favor un enlace de hipertexto o otra referencia al sitio web de las convenciones de código de Java: http://java.sun.com/docs/codeconv/

Convenciones del código Java: Copyright de la traducción de javaHispano.

Se puede y se debe aplicar a esta traducción las mismas condiciones de licencia que al original de Sun en lo referente a uso, modificación y redistribución.

Si distribuyes esta traducción (aunque sea con otro formato de estilo) estas obligado a dar creditos a javaHispano por la traducción indicandolo con la siguientes lineas:

```
Adapted with permission from JAVA CODE CONVENTIONS. Copyright 1995-1999 Sun
Microsysytems, Inc. All rights reserved.

Copyright 2001 www.javaHispano.com. Todos los derechos reservados.
Otorgados derechos de copia y redistribución para uso interno y no
comercial.
```

Así mismo, aunque no se requiere, se agradecerá que incluyas un link al sitio web de javaHispano: http://www.javaHispano.com