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

Revisión y corrección, marzo de 2007
por Lucas Vieites
http://www.codexion.com

Convenciones de código para el lenguaje de programación Java ™

Revisado, 20 de Abril de 1999

Traducido al castellano, 10 de Mayo del 2001

Revisión y corrección, marzo de 2007

Índice de contenido

1 Introducción

1.1 Por qué convenciones de código

Las convenciones de código son importantes para los programadores por muchas 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 rápidamente y más a fondo.
- Si distribuye su código fuente como un producto, necesita asegurarse de que está 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 estándares 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. Envíe sus 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 de fomentar el uso y conocimiento del lenguaje Java dentro del mundo hispanoparlante. 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.

Este documento ha sido revisado y corregido en marzo de 2007 por Lucas Vieites (www.codexion.com) .

2 Nombres de archivo

Esta sección enumera las extensiones y los nombres de archivo más usados .

2.1 Extensiones de los archivos

El software Java usa las siguientes extensiones para sus archivos:

Tipo de archivo	Extensión
Código java	.java
Bytecode de java	.class

2.2 Nombres de archivo comunes

Entre los nombres de archivo más utilizados se encuentran:

Nombre de archivo	Uso
GNUmakefile	El nombre preferido para un archivo «`make`». Usamos `gnumake` para construir nuestro software.
README	El nombre preferido para el archivo que resume los contenidos de un directorio en concreto.

3 Organización de los archivos

Un archivo está formado por secciones que deben estar separadas por líneas en blanco y comentarios opcionales que identifican cada sección. Se deberá evitar la creación de archivos de más de 2000 líneas puesto que son incómodos de manejar. Para ver un ejemplo de un programa de Java debidamente formateado, vea « 11.1 Ejemplo de archivo fuente Java».

3.1 Archivos de código fuente Java

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

Los archivos fuente Java tienen la siguiente ordenación:

- Comentarios de inicio
- Sentencias «`package`» e «`import`»
- Declaraciones de clases e interfaces

3.1.1 Comentarios de inicio

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

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

3.1.2 Sentencias package e import

La primera línea que no es un comentario de los archivos 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 del 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 de dos letras que especifican el país como se define en el estándar ISO 3166, 1981.

3.1.3 Declaraciones de clases e interfaces

La siguiente tabla describe las partes de la declaración de una clase o interfaz, en el orden en el que deberían aparecer. Vea « 11.1 Ejemplo de archivo fuente Java».

	Partes de la declaración de una clase o interfaz	**Notas**
1	Comentario de documentación de la clase o interfaz (`/**...*/`)	Vea « 5.2 Comentarios de documentación» para más información sobre lo que debe aparecer en este comentario.
2	Sentencia `class` o `interface`	
3	Comentario de implementación de la clase o interfaz si fuera necesario (`/*...*/`)	Este comentario debe contener cualquier información aplicable a toda la clase o interfaz que no sea apropiada para estar en los comentarios de documentación de la clase o interfaz.
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 después 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 ámbito 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 o tabuladores) no se especifica. Los tabuladores deben ser exactamente cada 8 espacios (no 4).

4.1 Longitud de la línea

Evite las líneas de más de 80 caracteres ya que no son interpretadas correctamente por muchas terminales y herramientas.

Nota: Los ejemplos destinados a uso en la documentación deben tener una longitud inferior, generalmente no más de 70 caracteres.

4.2 Rotura de líneas

Cuando una expresión no entre en una línea debe separarla de acuerdo con los siguientes 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 línea con el comienzo de la expresión al mismo nivel de la línea anterior.
- Si las reglas anteriores provocan que su código parezca confuso o que éste se acumule en el margen derecho, indente con 8 espacios.

Ejemplos de como romper la llamada a un método:

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

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

Veamos dos ejemplos de rotura de líneas en expresiones aritméticas. Se prefiere el primero ya que el salto de línea 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
```

A continuación dos ejemplos de indentación de declaraciones de métodos. El primero es el caso

convencional. El segundo conduciría la segunda y la tercera línea demasiado hacia la izquierda con la indentación convencional, así 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) {
    ...
}
```

La rotura de líneas para sentencias `if` deberá seguir generalmente la regla de los 8 espacios, ya que la indentación convencional (4 espacios) hace difícil ver el cuerpo. Por ejemplo:

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

//MEJORUSAR ESTA INDENTACIÓN
if ((condicion1 && condicion2)
        || (condicion3 && condicion4)
        ||!(condicion5 && condicion6)) {
    hacerAlgo();
}

//O 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 archivos HTML con la herramienta `javadoc`.

Los comentarios de implementación son para comentar nuestro código o para comentarios acerca

de una implementación en concreto. Los comentarios de documentación son para describir la especificación del código, libre de una perspectiva de implementación, y para ser leídos 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, no se debe incluir como comentario información sobre cómo se construye el paquete correspondiente o en qué directorio reside.

Se podrán tratar discusiones sobre decisiones de diseño que no sean triviales u obvias, pero se debe evitar duplicar información que está presente (de forma clara) en el código ya que es fácil que los comentarios redundantes se queden obsoletos. En general, evite cualquier comentario que pueda quedar obsoleto a medida que el código evoluciona.

Nota: La frecuencia de comentarios a veces refleja la escasez de 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, finales, y de fin de línea

5.1.1 Comentarios de bloque

Los comentarios de bloque se usan para dar descripciones de archivos, métodos, estructuras de datos y algoritmos. Los comentarios de bloque se podrán usar al comienzo de cada archivo 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 comentario de bloque debe ir precedido por una línea en blanco que lo separe del resto del código.

```
/*
 * Aquí 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:

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

Nota: Si no se va a utilizar `indent(1)`, no se deberá usar `/*-` en el código o hacer ninguna otra concesión a la posibilidad de que alguien ejecute `indent(1)` sobre él. Véase también «Comentarios de documentación».

5.1.2 Comentarios de una línea

Pueden aparecer comentarios cortos de una única línea al nivel del código que sigue. Si un comentario no se puede escribir en una línea, debe seguir el formato de los comentarios de bloque (sección 5.1.1). Un comentario de una sola línea debe ir precedido de una línea en blanco. Veamos un ejemplo de comentario de una sola línea en código Java:

```
if (condicion) {
    /* Código de la condición. */
    ...
}
```

5.1.3 Comentarios finales

Pueden aparecer comentarios muy pequeños en la misma línea del código que comentan, pero lo suficientemente alejados de él. Si aparece más de un comentario corto en el mismo trozo de código, deben ser indentados con la misma profundidad. Por ejemplo:

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

5.1.4 Comentarios de fin de línea

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

```
if (foo > 1) {
    // Hacer algo.
    ...
} else {
    return false;  // Explicar aquí por que.
}
//if (bar > 1) {
//
//
//    Hacer algo.
//    ...
//}
//else {
//    return false;
//}
```

5.2 Comentarios de documentación

Nota: Vea « 11.1 Ejemplo de archivo fuente Java» para obtener ejemplos de los formatos de

comentarios descritos aquí. Para más detalles, vea «How to Write Doc Comments for Javadoc» que incluye información de las etiquetas de los comentarios de documentación (`@return`, `@param`, `@see`) en http://java.sun.com/products/jdk/javadoc/writingdoccomments.shtml.

visite el sitio web de `javadoc`: http://java.sun.com/products/jdk/javadoc/ para más detalles acerca de los comentarios de documentación y `javadoc.`

Los comentarios de documentación describen clases Java, interfaces, constructores, métodos y atributos. Cada comentario de documentación se encierra con los delimitadores de comentario `/**...*/`, 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 { ...
```

Fíjese en que las clases e interfaces de alto nivel no están indentadas, mientras que sus miembros sí lo están. La primera línea de un comentario de documentación (`/**`) para clases e interfaces no está indentada, las siguientes líneas tienen cada una un espacio de indentación (para alinear los asteriscos verticalmente). 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, interfaz, variable o método que no es apropiada para la documentación, utilice un comentario de implementación de bloque (sección 5.1.1) o de una línea (sección 5.1.2) para comentarlo inmediatamente *después* 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 ponga 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 indentación
int        size;           // tamaño de la tabla
Object     currentEntry;   // entrada de la tabla seleccionada actualmente
```

6.2 Inicialización

Intente 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 cálculos posteriores.

6.3 Colocación

Sitúe las declaraciones solo al principio de los bloques (un bloque es cualquier código encerrado por llaves «{» y «}»). No espere al primer uso para declararlas; puede confundir al programador incauto y limitar la portabilidad del código dentro de su ámbito.

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

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

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

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

Evite las declaraciones locales que oculten declaraciones de niveles superiores, por ejemplo, no declare el mismo nombre de variable en un bloque interno:

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

6.4 Declaraciones de clases e interfaces

Al programar 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 de declaración
- La llave de cierre «}» empieza una nueva línea indentada ajustada 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 máximo 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 }». Vea los ejemplos de las siguientes secciones.

- Las sentencias encerradas deben indentarse un nivel más que la sentencia compuesta.
- La llave de apertura se debe poner al final de la línea que comienza la sentencia compuesta; la llave de cierre debe empezar una nueva línea y ser indentada al mismo nivel que el principio 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 accidentalmente *bugs* por olvidar las llaves.

7.3 Sentencias de retorno

Una sentencia `return` con un valor no debe usar paréntesis a no ser que 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 «{}». Evite 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 cláusulas de inicialización, condición, y actualización) debe tener la siguiente forma:

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

Al usar el operador «coma» en la cláusula de inicialización o actualización de una sentencia `for` evite la complejidad de usar más de tres variables. Si es necesario utilice sentencias separadas antes de bucle `for` (para la cláusula de inicialización) o al final del bucle (para la cláusula de actualización).

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`), añada un comentario donde normalmente se encontraría la sentencia `break`. Esto se muestra en el ejemplo anterior con el comentario `/* este caso se propaga */`.

Cada sentencia `switch` debe incluir un caso «por defecto» (`default`). El `break` en el caso «por defecto» es redundante, pero previene 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 haya completado con éxito o no.

```
try {
    sentencias;
```

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

8 Espacio 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 archivo 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 o de un comentario de una línea
- 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) {
      ...
  }
  ```

  Fíjese en 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 clave 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 separar los operadores unarios, incremento («`++`») y decremento («`--`») de sus operandos. Ejemplo:

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

  while (d++ == s++) {
      n++;
  }
  prints("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`» 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 nomenclatura

Las convenciones de nomenclatura hacen que el código sea más inteligible al hacerlo más fácil de leer. También pueden dar información sobre la función de un identificador, por ejemplo, cuando es una constante, un paquete o una clase, lo cual puede ser útil para entender el código.

Tipo de identificador	**Reglas de nomenclatura**	**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 identifican cada país como se especifica en el estándar ISO 3166, 1981.
	Los siguientes componentes del nombre del paquete variarán de acuerdo a las convenciones de nomenclatura 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. Intente mantener los nombres de las clases simples y descriptivos. Use palabras completas, evite acrónimos y abreviaturas (a no ser que la abreviatura sea mucho más conocida que el nombre completo, como `URL` o `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 siguientes palabras que lo forma en mayúscula.	`ejecutar();`
		`ejecutarRapido();`
		`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 *guión bajo* «_» o signo de dólar «$», aunque ambos están permitidos por el lenguaje.
	Los nombres de las variables deben ser cortos pero significativos. La elección del nombre de una variable debe ser un mnemónico, designado para indicar a un observador ocasional su función. Los nombres de variables de un solo carácter se deben evitar, excepto para variables índices temporales. Nombres comunes 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 guión bajo («_»). (Las constantes ANSI se deben evitar, para facilitar la 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 Proporcionar acceso a variables de instancia y de clase

No haga pública ninguna variable de instancia o clase sin una buena razón. A menudo las variables

de instancia no necesitan ser asignadas (*set*) o consultadas *(get)* explícitamente, esto suele suceder como efecto de una llamada a método.

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

10.2 Referencias a variables y métodos de clase

Evite usar un objeto para acceder a una variable o método de clase (`static`). Use 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

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

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

No use 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 use asignaciones embebidas como un intento de mejorar el rendimiento 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 involucran a 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

Intente hacer que la estructura del programa se ajuste a su objetivo. Ejemplo:

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

en su lugar se debe escribir

```
return expressionBooleana;
```

Del mismo modo,

```
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

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

11 Ejemplo de código

11.1 Ejemplo de archivo fuente Java

El siguiente ejemplo muestra el formato de archivo Java con una sola clase pública. Las interfaces

se formatean de modo similar. Vea « 6.4 Declaraciones de clases e interfaces» y « 5.2 Comentarios de documentación».

```
/*
 * @(#)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;

/**
 * Aquí va la descripción de la clase.
 *
 * @version    1.82 18 Mar 1999
 * @author     Firstname Lastname
 */
public class Blah extends SomeClass {
    /* Aquí puede ir un comentario acerca de la implementación de la clase. */

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

    /**
     * Comentario de la documentación de classVar2 que resulta tener más de
     * una línea de longitud.
     */
    private static Object classVar2;

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

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

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

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

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

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

Convenciones del código Java: Copyright de Sun.

Puede copiar, adaptar y redistribuir este documento para uso no comercial o para uso interno de un fin comercial. Sin embargo, no debe volver a publicar 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, está obligado a conceder el crédito a Sun. Si reproduce o distribuye el documento sin ninguna modificación sustancial en su contenido, use la siguiente línea de crédito:

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

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

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

En ambos casos añada un enlace de hipertexto u 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 créditos a javaHispano por la traducción indicándolo con la siguientes líneas:

```
Adapted with permission from JAVA CODE CONVENTIONS. Copyright 1995-1999 Sun Microsytems, 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.
```

Asimismo, aunque no se requiere, se agradecerá que incluya un enlace al sitio web de javaHispano: http://www.javaHispano.com