Convenciones de código para Java

1 - Introducción

1.1 ¿Por qué tener 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 autor original.
- Las convenciones de código mejoran la lectura del software, permitiendo a los ingenieros entender nuevo código mucho más rápido y más a fondo.
- Si distribuyes tu código fuente como un producto, necesitas asegurarte de que está bien hecho y presentado como cualquier otro producto.

Para que funcionen las convenciones, TODOS los programadores deben seguirlas.

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. Enviar los comentarios a shommel@eng.sun.com

2 - Nombres de ficheros

Esta sección muestra las extensiones y nombres de ficheros más comúnmente usadas.

2.1 Extensiones de los ficheros

El software Java usa las siguientes extensiones para los ficheros:

Tipo de archivo	Extensión
Fuente Java	.java
Bytecode de Java	.class

2.2 Nombres comunes para ficheros

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 20.

3.1 Ficheros fuente Java

Cada fichero 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 fichero que la clase pública. La clase o interfaz pública debe ser la primera del fichero.

Los ficheros fuentes Java tienen la siguiente ordenación:

- Comentarios de comienzo (ver "Comentarios de comienzo" en la página 2).
- Sentencias Package e Import.
- Declaraciones de clases e interfaces (ver "Declaraciones de clases e interfaces" en la página 3).

3.1.1 Comentarios de comienzo

Todos los ficheros fuente deben comenzar con un comentario (al estilo lenguaje C) en el que se muestra el nombre de la clase, información de la versión, fecha, y copyright:

```
/*
 * Nombre de la clase
 *
 * Información de la version
 *
 * Fecha
```

```
 *
 * Copyright
 */
```

3.1.2 Sentencias Package e Import

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

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

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 que deberían aparecer. Ver "Ejemplo de fichero fuente Java" en la página 20 para un ejemplo que incluye comentarios.

	Partes de la declaración de una clase o interfaz	**Notas**
1	Comentario de documentación de la clase o interfaz (`/**...*/`)	Ver "Comentarios de documentación" en la página 8 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 era 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 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 cada 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 que de bajo nivel.
- Alinear la nueva línea con el comienzo de la expresión al mismo nivel de la línea 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 ruptura 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
```

Ahora dos ejemplos de indentación en 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 unArg, Object otroArg, String todaviaOtroArg,
      Object yOtroMas) {
...
}

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

La ruptura 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 INDENTACION
if ((condicion1 && condicion2)
      || (condicion3 && condicion4)
      ||!(condicion5 && condicion6)) { //MALOS SALTOS
      hacerAlgo(); //HACEN ESTA LINEA FACIL DE OLVIDAR
}

// USAR ESTA INDENTACION
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 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 propio código. Los comentarios deben contener sólo información que es relevante para la lectura y entendimiento del programa. Por ejemplo, información sobre cómo se construye el paquete correspondiente o en qué directorio reside, no debe ser incluida como comentario.

Son apropiadas las discusiones sobre decisiones de diseño no triviales o no obvias, pero evitan 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 rescribir 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 fichero y 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.

```
/*
 * 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 usa **indent**(1), no se tiene que usar `/*-` en el código o hacer cualquier otra concesión a la posibilidad de que alguien ejecute **indent**(1) sobre él.
Ver también "Comentarios de documentación" en la página 8.

5.1.2 Comentarios de una línea

Pueden aparecer comentarios cortos de una única línea indentados al nivel del código que siguen. Si un comentario no se puede escribir en una única línea, debe seguir el formato de los comentarios de bloque. (ver sección 5.1.1). Un comentario de una sola línea debe ir precedido de una línea en blanco. Aquí un ejemplo de comentario de una sola línea en código Java (ver también
"Comentarios de documentación" en la página 8):

```
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 aparece 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 esPrimo(a);      /* caso cuando a es impar */
}
```

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. Aquí tenéis ejemplos 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: Ver "Ejemplo de fichero fuente Java" en la página 20 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
```

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, interfaz o miembro (método o atributo). Este comentario debe aparecer justo antes de la declaración:

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

Nótese que las clases e interfaces de alto nivel no están indentadas, mientras que sus miembros los están. La primera línea de un comentario de documentación (/**) para clases e interfaces no esta indentada, sucesivas líneas tienen cada una un espacio de indentación (para alinear verticalmente 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, interfaz, 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 *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 nivel, tam;
```

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         nivel;           // nivel de indentación
int         tam;             // tamaño de la tabla
Object      entradaActual;   // entrada de la tabla seleccionada
```

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 previos.

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 y limitar la portabilidad del código dentro de su visibilidad.

```
void miMetodo() {
      int int1 = 0; // comienzo del bloque del método
      if (condicion) {
            int int2 = 0; // comienzo del bloque del "if"
            ...
      }
}
```

La excepción de 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++) { ... }
```

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

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

6.4 Declaraciones de clases 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 de declaración.
- 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 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 }`". 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 línea que comienza la sentencia compuesta; la llave de cierre debe empezar una nueva línea y ser indentada al mismo nivel que el comienzo 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 errores accidentales por olvidar las llaves.

7.3 Sentencias return

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 la inicialización, condición, y actualización) debe tener la siguiente forma:

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

Al usar el operador coma en la 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 inicialización) o al final del bucle (para la 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`), 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 prevé 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 reservada del lenguaje seguida por un paréntesis debe separarse por un espacio. Ejemplo:

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

  Notemos 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 reservadas de llamadas a métodos.

- Debe aparecer un espacio en blanco después de cada coma en las listas de argumentos.

- Todos los operadores binarios excepto el punto "." deben ir separados de sus operandos mediante espacios. Los operadores unarios como el "menos unario", el incremento ("++") y el decremento ("--") no deben ir nunca separados mediante espacios en blanco 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 - Convenios de nombrado

Los convenios de nombrado hacen los programas más entendibles facilitando su lectura. También pueden dar información sobre la función de un identificador, por ejemplo, cuando es una constante, un paquete, o una clase, que puede ser útil para entender el código.

Tipo de Identificador	**Reglas de nombrado**	**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 ISO Standard 3166, 1981.
	Los sucesivos 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, máquinas o nombres de usuarios.
		`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 o HTML).	`class Cliente;`
		
		`class ImagenAnimada;`

Interfaces	Los nombres de las interfaces siguen la misma regla que las clases.	`interface ClienteDelegado;` `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();`
		
		`obtenerFondo();`

Tipo de Identificador	Reglas de nombrado	Ejemplos

Variables	Excepto las variables, todas las instancias, clases y constantes de clase 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 del dólar "$", aunque ambos están 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 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 de clase y las constantes ANSI deben ir totalmente en mayúsculas separando las palabras con un guión bajo ("_"). (Las constantes ANSI se deben evitar, para facilitar su depuración.)	`static final int MIN = 1;`
		
		`static final int MAX = 9;`
		
		`static final int NUM = 5;`

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, 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 esencialmente una estructura de datos, sin comportamiento. En otras palabras, si se usase 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

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 deberían ser codificadas 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 variables en la misma sentencia, dificulta su lectura. Ejemplo:

```
fooBar.fChar = barFoo.Ichar = '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 no lo permite)
      ...
}
```

se debe escribir:

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

No usar asignaciones incrustadas como un intento de mejorar el rendimiento en tiempo de ejecución. Ése 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 única clase pública. Las interfaces se formatean similarmente. Para más información, ver "Declaraciones de clases e interfaces" en la página 10 y "Comentarios de documentación" en la página 8.

```
/*
 * @(#)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 aquí.
 *
 * @version datos de la versión (numero y fecha)
 * @author Nombre Apellido
 */
public class Bla extends OtraClase {
      /* Un comentario de implementación 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...
	}
}
```