Dará gusto leer tu código si sigue las convenciones de codificación Java. Como anexo presentamos el documento de dichas convenciones en versión original y su traducción a castellano.

Quizás te preguntes, que siendo tu código tu obra, tu creación... ¿acaso no puedes programarlo como a ti me apetezca?

Las convenciones no fuerzan a dar un nombre u otro a tus clases, métodos, etc. Tan sólo indican qué condiciones estilísticas debes respetar (por ejemplo que los nombres de las clases comiencen en mayúscula y los de las variables de instancia en minúscula).

Ahora que estás esforzándote por escribir el código es muy posible que entiendas perfectamente cada línea al primer vistazo, sin embargo, si no respetas las convenciones, dentro de un tiempo incluso a ti mismo/a te será difícil mantenerlo, por no decir a otras personas que tengan que trabajar sobre él, como suele ocurrir en las empresas de desarrollo de software.

En las asignaturas de Ingeniería del Software, queremos sentar las bases y aportar los conocimientos necesarios para vuestro desarrollo como profesionales del software, por eso queremos fomentar, no sólo el que seáis capaces de diseñar y programar un sistema software, sino también de hacerlo de forma correcta, eficiente y fácil de entender y mantener, con lo que tendremos muy en cuenta que se sigan las reglas de estilo, al menos en el apartado 9 de la versión castellana (convenciones de nombres), 3 en la versión inglesa (naming conventions).

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

Java Programming Style Guidelines

Version 6.1, March 2008
Geotechnical Software Services

This document is available at http://geosoft.no/development/javastyle.html

---

Table of Content

---

1 Introduction

This document lists Java coding recommendations common in the Java development community.

The recommendations are based on established standards collected from a number of sources, individual experience, local requirements/needs, as well as suggestions given in [1], [2], [3], [4] and [5].

There are several reasons for introducing a new guideline rather than just referring to the ones above. Main reason is that these guides are far too general in their scope and that more specific rules (especially naming rules) need to be established. Also, the present guide has an annotated form that makes it easier to use during project code reviews than most other existing guidelines. In addition, programming recommendations generally tend to mix style issues with language technical issues in a somewhat confusing manner. The present document does not contain any Java technical recommendations at all, but focuses mainly on programming style.

While a given development environment (IDE) can improve the readability of code by access visibility, color coding, automatic formatting and so on, the programmer should never *rely* on such features. Source code should always be considered *larger*
than the IDE it is developed within and should be written in a way that maximize its readability independent of any IDE.

1.1 Layout of the Recommendations.

The recommendations are grouped by topic and each recommendation is numbered to make it easier to refer to during reviews.

Layout for the recommendations is as follows:

n. Guideline short description

`Example if applicable`
Motivation, background and additional information.

The motivation section is important. Coding standards and guidelines tend to start "religious wars", and it is important to state the background for the recommendation.

1.2 Recommendation Importance

In the guideline sections the terms *must*, *should* and *can* have special meaning. A *must* requirement must be followed, a *should* is a strong recommendation, and a *can* is a general guideline.

1.3 Automatic Style Checking

Many tools provide automatic code style checking. One of the most popular and feature rich one is *Checkstyle* by Oliver Burn.

Checkstyle is configured through an XML file of *style rules* which is applied to the source code. It is most useful if it is integrated in the build process or the development environment. There are Checkstyle plugins for all the popular IDEs available.

To use Checkstyle with the GeoSoft style rules below, use this configuration file: geosoft_checks.xml.

2 General Recommendations

1. Any violation to the guide is allowed if it enhances readability.
The main goal of the recommendation is to improve readability and thereby the understanding and the maintainability and general quality of the code. It is impossible to cover all the specific cases in a general guide and the programmer should be flexible.

3 Naming Conventions

3.1 General Naming Conventions

2. Names representing packages should be in all lower case.
`mypackage, com.company.application.ui`
Package naming convention used by Sun for the Java core packages. The initial package name representing the domain name must be in lower case.

3. Names representing types must be nouns and written in mixed case starting with upper case.
`Line, AudioSystem`
Common practice in the Java development community and also the type naming convention used by Sun for the Java core packages.

4. Variable names must be in mixed case starting with lower case.
`line, audioSystem`
Common practice in the Java development community and also the naming convention for variables used by Sun for the Java core packages. Makes variables easy to distinguish from types, and effectively resolves potential naming collision as in the declaration `Line line;`

5. Names representing constants (final variables) must be all uppercase using underscore to separate words.
`MAX_ITERATIONS, COLOR_RED`
Common practice in the Java development community and also the naming convention used by Sun for the Java core packages. In general, the use of such constants should be minimized. In many cases implementing the value as a method is

a better choice:

```
int getMaxIterations() // NOT: MAX_ITERATIONS = 25
{
  return 25;
}
```

This form is both easier to read, and it ensures a uniform interface towards class values.

6. Names representing methods must be verbs and written in mixed case starting with lower case.

```
getName(), computeTotalWidth()
```

Common practice in the Java development community and also the naming convention used by Sun for the Java core packages. This is identical to variable names, but methods in Java are already distinguishable from variables by their specific form.

7. Abbreviations and acronyms should not be uppercase when used as name.

```
exportHtmlSource(); // NOT: exportHTMLSource();
openDvdPlayer();    // NOT: openDVDPlayer();
```

Using all uppercase for the base name will give conflicts with the naming conventions given above. A variable of this type whould have to be named dVD, hTML etc. which obviously is not very readable. Another problem is illustrated in the examples above; When the name is connected to another, the readability is seriously reduced; The word following the acronym does not stand out as it should.

8. Private class variables should have underscore suffix.

```
class Person
{
  private String name_;
  ...
}
```

Apart from its name and its type, the *scope* of a variable is its most important feature. Indicating class scope by using underscore makes it easy to distinguish class variables from local scratch variables. This is important because class variables are considered to have higher significance than method variables, and should be treated with special care by the programmer.

A side effect of the underscore naming convention is that it nicely resolves the problem of finding reasonable variable names for setter methods:

```
void setName(String name)
{
  name_ = name;
}
```

An issue is whether the underscore should be added as a prefix or as a suffix. Both practices are commonly used, but the latter is recommended because it seem to best preserve the readability of the name.

It should be noted that scope identification in variables have been a controversial issue for quite some time. It seems, though, that this practice now is gaining acceptance and that it is becoming more and more common as a convention in the professional development community.

9. Generic variables should have the same name as their type.

```
void setTopic(Topic topic) // NOT: void setTopic(Topic value)
                           // NOT: void setTopic(Topic aTopic)
                           // NOT: void setTopic(Topic t)

void connect(Database database) // NOT: void connect(Database db)
                                // NOT: void connect(Database oracleDB)
```

Reduce complexity by reducing the number of terms and names used. Also makes it easy to deduce the type given a variable name only.

If for some reason this convention doesn't seem to *fit* it is a strong indication that the type name is badly chosen.

Non-generic variables have a *role*. These variables can often be named by combining role and type:

```
Point  startingPoint, centerPoint;
Name   loginName;
```

10. All names should be written in English.

English is the preferred language for international development.

11. Variables with a large scope should have long names, variables with a small scope can have short names [1].

Scratch variables used for temporary storage or indices are best kept short. A programmer reading such variables should be able to assume that its value is not used outside a few lines of code. Common scratch variables for integers are *i*, *j*, *k*, *m*, *n* and for characters *c* and *d*.

12. The name of the object is implicit, and should be avoided in a method name.

```
line.getLength();   // NOT: line.getLineLength();
```

The latter might seem natural in the class declaration, but proves superfluous in use, as shown in the example.

3.2 Specific Naming Conventions

13. The terms *get/set* must be used where an attribute is accessed directly.

```
employee.getName();
employee.setName(name);

matrix.getElement(2, 4);
matrix.setElement(2, 4, value);
```

Common practice in the Java community and the convention used by Sun for the Java core packages.

14. *is* prefix should be used for boolean variables and methods.

```
isSet, isVisible, isFinished, isFound, isOpen
```

This is the naming convention for boolean methods and variables used by Sun for the Java core packages.

Using the *is* prefix solves a common problem of choosing bad boolean names like *status* or *flag*. *isStatus* or *isFlag* simply doesn't fit, and the programmer is forced to chose more meaningful names.

Setter methods for boolean variables must have *set* prefix as in:

```
void setFound(boolean isFound);
```

There are a few alternatives to the *is* prefix that fits better in some situations. These are *has*, *can* and *should* prefixes:

```
boolean hasLicense();
boolean canEvaluate();
boolean shouldAbort = false;
```

15. The term *compute* can be used in methods where something is computed.

```
valueSet.computeAverage();
matrix.computeInverse()
```

Give the reader the immediate clue that this is a potential time consuming operation, and if used repeatedly, he might consider caching the result. Consistent use of the term enhances readability.

16. The term *find* can be used in methods where something is looked up.

```
vertex.findNearestVertex();
matrix.findSmallestElement();
node.findShortestPath(Node destinationNode);
```

Give the reader the immediate clue that this is a simple look up method with a minimum of computations involved. Consistent use of the term enhances readability.

17. The term *initialize* can be used where an object or a concept is established.

```
printer.initializeFontSet();
```

The American *initialize*should be preferred over the English *initialise*. Abbreviation *init* must be avoided.

18. JFC (Java Swing) variables should be suffixed by the element type.

```
widthScale, nameTextField, leftScrollbar, mainPanel, fileToggle, minLabel, printerDialog
```

Enhances readability since the name gives the user an immediate clue of the type of the variable and thereby the available resources of the object.

19. Plural form should be used on names representing a collection of objects.

```
Collection<Point>  points;
int[]              values;
```

Enhances readability since the name gives the user an immediate clue of the type of the variable and the operations that can be performed on its elements.

20. *n* prefix should be used for variables representing a number of objects.

```
nPoints, nLines
```

The notation is taken from mathematics where it is an established convention for indicating a number of objects.

Note that Sun use *num* prefix in the core Java packages for such variables. This is probably meant as an abbreviation of *number of*, but as it looks more like *number* it makes the variable name strange and misleading. If "number of" is the preferred phrase, *numberOf* prefix can be used instead of just *n*. *num* prefix must not be used.

21. *No* suffix should be used for variables representing an entity number.

```
tableNo, employeeNo
```

The notation is taken from mathematics where it is an established convention for indicating an entity number.

An elegant alternative is to prefix such variables with an *i*: `iTable, iEmployee`. This effectively makes them *named* iterators.

22. Iterator variables should be called *i*, *j*, *k* etc.

```
for (Iterator i = points.iterator(); i.hasNext(); ) {
  :
}

for (int i = 0; i < nTables; i++) {
  :
}
```

The notation is taken from mathematics where it is an established convention for indicating iterators.

Variables named *j*, *k* etc. should be used for nested loops only.

23. Complement names must be used for complement entities [1].

```
get/set, add/remove, create/destroy, start/stop, insert/delete,
increment/decrement, old/new, begin/end, first/last, up/down, min/max,
next/previous, old/new, open/close, show/hide, suspend/resume, etc.
```

Reduce complexity by symmetry.

24. Abbreviations in names should be avoided.

```
computeAverage();               // NOT: compAvg();
ActionEvent event;              // NOT: ActionEvent e;
catch (Exception exception) {   // NOT: catch (Exception e) {
```

There are two types of words to consider. First are the common words listed in a language dictionary. These must never be abbreviated. Never write:

`cmd`	instead of	`command`
`comp`	instead of	`compute`
`cp`	instead of	`copy`
`e`	instead of	`exception`
`init`	instead of	`initialize`
`pt`	instead of	`point`

etc.

Then there are domain specific phrases that are more naturally known through their acronym or abbreviations. These phrases should be kept abbreviated. Never write:

`HypertextMarkupLanguage`	instead of	`html`
`CentralProcessingUnit`	instead of	`cpu`
`PriceEarningRatio`	instead of	`pe`

etc.

25. Negated boolean variable names must be avoided.

```
bool isError; // NOT: isNoError
bool isFound; // NOT: isNotFound
```

The problem arise when the logical not operator is used and double negative arises. It is not immediately apparent what `!isNotError` means.

26. Associated constants (final variables) should be prefixed by a common type name.

```
final int  COLOR_RED   = 1;
final int  COLOR_GREEN = 2;
final int  COLOR_BLUE  = 3;
```

This indicates that the constants belong together, and what concept the constants represents.

An alternative to this approach is to put the constants inside an interface effectively prefixing their names with the name of the interface:

```
  interface Color
  {
    final int RED   = 1;
    final int GREEN = 2;
    final int BLUE  = 3;
  }
```

27. Exception classes should be suffixed with *Exception*.

```
class AccessException extends Exception
{
  :
}
```

Exception classes are really not part of the main design of the program, and naming them like this makes them stand out relative to the other classes. This standard is followed by Sun in the basic Java library.

28. Default interface implementations can be prefixed by *Default*.

```
class DefaultTableCellRenderer
  implements TableCellRenderer
{
  :
}
```

It is not uncommon to create a simplistic class implementation of an interface providing default behaviour to the interface methods. The convention of prefixing these classes by *Default* has been adopted by Sun for the Java library.

29. Singleton classes should return their sole instance through method *getInstance*.

```
class UnitManager
{
  private final static UnitManager instance_ = new UnitManager();

  private UnitManager()
  {
    ...
  }

  public static UnitManager getInstance()  // NOT: get() or instance() or unitManager() etc.
  {
    return instance_;
  }
}
```

Common practice in the Java community though not consistently followed by Sun in the JDK. The above layout is the preferred pattern.

30. Classes that creates instances on behalf of others (*factories*) can do so through method *new[ClassName]*

```
class PointFactory
{
  public Point newPoint(...)
  {
    ...
  }
}
```

Indicates that the instance is created by *new* inside the factory method and that the construct is a controlled replacement of `new Point()`.

31. Functions (methods returning an object) should be named after what they return and procedures (*void* methods) after what they do.

Increase readability. Makes it clear what the unit should do and especially all the things it is *not* supposed to do. This again makes it easier to keep the code clean of side effects.

4 Files

32. Java source files should have the extension *.java*.

`Point.java`

Enforced by the Java tools.

33. Classes should be declared in individual files with the file name matching the class name. Secondary private classes can be declared as inner classes and reside in the file of the class they belong to.

Enforced by the Java tools.

34. File content must be kept within 80 columns.

80 columns is the common dimension for editors, terminal emulators, printers and debuggers, and files that are shared between several developers should keep within these constraints. It improves readability when unintentional line breaks are avoided when passing a file between programmers.

35. Special characters like TAB and page break must be avoided.

These characters are bound to cause problem for editors, printers, terminal emulators or debuggers when used in a multi-programmer, multi-platform environment.

36. The incompleteness of split lines must be made obvious [1].

```
totalSum = a + b + c +
           d + e;

method(param1, param2,
       param3);

setText ("Long line split" +
         "into two parts.");

for (int tableNo = 0; tableNo < nTables;
     tableNo += tableStep) {
  ...
}
```

Split lines occurs when a statement exceed the 80 column limit given above. It is difficult to give rigid rules for how lines should be split, but the examples above should give a general hint.

In general:

- Break after a comma.
- Break after an operator.
- Align the new line with the beginning of the expression on the previous line.

5 Statements

5.1 Package and Import Statements

37. The `package` statement must be the first statement of the file. All files should belong to a specific package.

The `package` statement location is enforced by the Java language. Letting all files belong to an actual (rather than the Java default) package enforces Java language object oriented programming techniques.

38. The `import` statements must follow the `package` statement. `import` statements should be sorted with the most fundamental packages first, and grouped with associated packages together and one blank line between groups.

```
import java.io.IOException;
import java.net.URL;

import java.rmi.RmiServer;
import java.rmi.server.Server;

import javax.swing.JPanel;
import javax.swing.event.ActionEvent;

import org.linux.apache.server.SoapServer;
```

The `import` statement location is enforced by the Java language. The sorting makes it simple to browse the list when there are many imports, and it makes it easy to determine the dependencies of the present package The grouping reduce complexity by collapsing related information into a common unit.

39. Imported classes should always be listed explicitly.

```
import java.util.List;      // NOT: import java.util.*;
import java.util.ArrayList;
import java.util.HashSet;
```

Importing classes explicitly gives an excellent documentation value for the class at hand and makes the class easier to comprehend and maintain.

Appropriate tools should be used in order to always keep the import list minimal and up to date.

5.2 Classes and Interfaces

40. Class and Interface declarations should be organized in the following manner:

1. **Class/Interface documentation.**
2. **`class` or `interface` statement.**
3. **Class (`static`) variables in the order `public`, `protected`, package (no access modifier), `private`.**
4. **Instance variables in the order `public`, `protected`, package (no access modifier), `private`.**
5. **Constructors.**
6. **Methods (no specific order).**

Reduce complexity by making the location of each class element predictable.

5.3 Methods

41. Method modifiers should be given in the following order:
<access> static abstract synchronized <unusual> final native
The *<access>* modifier (if present) must be the first modifier.

```
public static double square(double a);  // NOT: static public double square(double a);
```

<access> is one of *public*, *protected* or *private* while *<unusual>* includes *volatile* and *transient*. The most important lesson here is to keep the *access* modifier as the first modifier. Of the possible modifiers, this is by far the most important, and it must stand out in the method declaration. For the other modifiers, the order is less important, but it make sense to have a fixed convention.

5.4 Types

42. Type conversions must always be done explicitly. Never rely on implicit type conversion.

```
floatValue = (int) intValue; // NOT: floatValue = intValue;
```

By this, the programmer indicates that he is aware of the different types involved and that the mix is intentional.

43. Array specifiers must be attached to the type not the variable.

```
int[] a = new int[20];   // NOT: int a[] = new int[20]
```

The *arrayness* is a feature of the base type, not the variable. It is not known why Sun allows both forms.

5.5 Variables

44. Variables should be initialized where they are declared and they should be declared in the smallest scope possible.

This ensures that variables are valid at any time. Sometimes it is impossible to initialize a variable to a valid value where it is declared. In these cases it should be left uninitialized rather than initialized to some phony value.

45. Variables must never have dual meaning.

Enhances readability by ensuring all concepts are represented uniquely. Reduce chance of error by side effects.

46. Class variables should never be declared public.

The concept of Java information hiding and encapsulation is violated by public variables. Use private variables and access functions instead. One exception to this rule is when the class is essentially a data structure, with no behavior (equivalent to a C++ `struct`). In this case it is appropriate to make the class' instance variables public [2].

47. Arrays should be declared with their brackets next to the type.

```
double[] vertex;  // NOT: double vertex[];
int[]    count;   // NOT: int    count[];

public static void main(String[] arguments)

public double[] computeVertex()
```

The reason for is twofold. First, the *array-ness* is a feature of the class, not the variable. Second, when returning an array from a method, it is not possible to have the brackets with other than the type (as shown in the last example).

48. Variables should be kept alive for as short a time as possible.

Keeping the operations on a variable within a small scope, it is easier to control the effects and side effects of the variable.

5.6 Loops

49. Only loop control statements must be included in the *for()* construction.

```
sum = 0;                          // NOT: for (i = 0, sum = 0; i < 100; i++)
for (i = 0; i < 100; i++)                    sum += value[i];
  sum += value[i];
```

Increase maintainability and readability. Make a clear distinction of what *controls* and what is *contained* in the loop.

50. Loop variables should be initialized immediately before the loop.

```
isDone = false;           // NOT: bool isDone = false;
while (!isDone) {         //      :
  :                       //      while (!isDone) {
}                         //        :
                          //      }
```

51. The use of *do-while* loops can be avoided.

do-while loops are less readable than ordinary *while* loops and *for* loops since the conditional is at the bottom of the loop. The reader must scan the entire loop in order to understand the scope of the loop.

In addition, *do-while* loops are not needed. Any *do-while* loop can easily be rewritten into a *while* loop or a *for* loop. Reducing the number of constructs used enhance readbility.

52. The use of *break* and *continue* in loops should be avoided.

These statements should only be used if they prove to give higher readability than their structured counterparts.

5.7 Conditionals

53. Complex conditional expressions must be avoided. Introduce temporary boolean variables instead [1].

```
bool isFinished = (elementNo < 0) || (elementNo > maxElement);
bool isRepeatedEntry = elementNo == lastElement;
if (isFinished || isRepeatedEntry) {
  :
}

// NOT:
if ((elementNo < 0) || (elementNo > maxElement)||
     elementNo == lastElement) {
  :
}
```

By assigning boolean variables to expressions, the program gets automatic documentation. The construction will be easier to read, debug and maintain.

54. The nominal case should be put in the *if*-part and the exception in the *else*-part of an if statement [1].

```
boolean isOk = readFile(fileName);
if (isOk) {
  :
}
else {
  :
}
```

Makes sure that the exceptions does not obscure the normal path of execution. This is important for both the readability and performance.

55. The conditional should be put on a separate line.

```
if (isDone)       // NOT: if (isDone) doCleanup();
  doCleanup();
```

This is for debugging purposes. When writing on a single line, it is not apparent whether the test is really true or not.

56. Executable statements in conditionals must be avoided.

```
InputStream stream = File.open(fileName, "w");
if (stream != null) {
  :
}

// NOT:
if (File.open(fileName, "w") != null)) {
  :
}
```

Conditionals with executable statements are simply very difficult to read. This is especially true for programmers new to Java.

5.8 Miscellaneous

57. The use of magic numbers in the code should be avoided. Numbers other than *0* and *1*can be considered declared as named constants instead.

```
private static final int  TEAM_SIZE = 11;
:
Player[] players = new Player[TEAM_SIZE]; // NOT: Player[] players = new Player[11];
```

If the number does not have an obvious meaning by itself, the readability is enhanced by introducing a named constant instead.

58. Floating point constants should always be written with decimal point and at least one decimal.

```
double total = 0.0;    // NOT:  double total = 0;
double speed = 3.0e8;  // NOT:  double speed = 3e8;

double sum;
:
sum = (a + b) * 10.0;
```

This emphasize the different nature of integer and floating point numbers. Mathematically the two model completely different and non-compatible concepts.

Also, as in the last example above, it emphasize the type of the assigned variable (`sum`) at a point in the code where this might not be evident.

59. Floating point constants should always be written with a digit before the decimal point.

```
double total = 0.5;  // NOT:  double total = .5;
```

The number and expression system in Java is borrowed from mathematics and one should adhere to mathematical conventions for syntax wherever possible. Also, 0.5 is a lot more readable than .5; There is no way it can be mixed with the integer 5.

60. Static variables or methods must always be refered to through the class name and never through an instance variable.

```
Thread.sleep(1000);    // NOT: thread.sleep(1000);
```

This emphasize that the element references is static and independent of any particular instance. For the same reason the class name should also be included when a variable or method is accessed from within the same class.

6 Layout and Comments

6.1 Layout

61. Basic indentation should be 2.

```
for (i = 0; i < nElements; i++)
  a[i] = 0;
```

Indentation is used to emphasize the logical structure of the code. Indentation of 1 is to small to acheive this. Indentation larger than 4 makes deeply nested code difficult to read and increase the chance that the lines must be split. Choosing between indentation of 2, 3 and 4; 2 and 4 are the more common, and 2 chosen to reduce the chance of splitting code lines. Note that the Sun recommendation on this point is 4.

62. Block layout should be as illustrated in example 1 below (recommended) or example 2, and must not be as shown in example 3. Class, Interface and method blocks should use the block layout of example 2.

```
while (!done) {
  doSomething();
  done = moreToDo();
}
```

```
while (!done)
{
  doSomething();
  done = moreToDo();
}
```

```
while (!done)
  {
    doSomething();
    done = moreToDo();
  }
```

Example 3 introduce an extra indentation level which doesn't emphasize the logical structure of the code as clearly as example 1 and 2.

63. The *class* and *interface* declarations should have the following form:

```
class Rectangle extends Shape
  implements Cloneable, Serializable
{
  ...
}
```

This follows from the general block rule above. Note that it is common in the Java developer community to have the opening bracket at the end of the line of the class keyword. This is not recommended.

64. Method definitions should have the following form:

```
public void someMethod()
  throws SomeException
{
  ...
}
```

See comment on `class` statements above.

65. The *if-else* class of statements should have the following form:

```
if (condition) {
  statements;
}

if (condition) {
```

```
  statements;
}
else {
   statements;
}

if (condition) {
  statements;
}
else if (condition) {
  statements;
}
else {
  statements;
}
```

This follows partly from the general block rule above. However, it might be discussed if an `else` clause should be on the same line as the closing bracket of the previous `if` or `else` clause:

```
  if (condition) {
    statements;
  } else {
    statements;
  }
```

This is equivalent to the Sun recommendation. The chosen approach is considered better in the way that each part of the `if-else` statement is written on separate lines of the file. This should make it easier to manipulate the statement, for instance when moving `else` clauses around.

66. The *for* statement should have the following form:

```
for (initialization; condition; update) {
  statements;
}
```

This follows from the general block rule above.

67. An empty *for* statement should have the following form:

```
for (initialization; condition; update)
  ;
```

This emphasize the fact that the for statement is empty and it makes it obvious for the reader that this is intentional.

68. The *while* statement should have the following form:

```
while (condition) {
  statements;
}
```

This follows from the general block rule above.

69. The *do-while* statement should have the following form:

```
do {
  statements;
} while (condition);
```

This follows from the general block rule above.

70. The *switch* statement should have the following form:

```
switch (condition) {
  case ABC :
    statements;
    // Fallthrough

  case DEF :
    statements;
    break;
```

```
  case XYZ :
    statements;
    break;

  default :
    statements;
    break;
}
```

This differs slightly from the Sun recommendation both in indentation and spacing. In particular, each `case` keyword is indented relative to the switch statement as a whole. This makes the entire switch statement stand out. Note also the extra space before the *:* character. The explicit *Fallthrough* comment should be included whenever there is a case statement without a `break` statement. Leaving the `break` out is a common error, and it must be made clear that it is intentional when it is not there.

71. A *try-catch* statement should have the following form:

```
try {
  statements;
}
catch (Exception exception) {
  statements;
}

try {
  statements;
}
catch (Exception exception) {
  statements;
}
finally {
  statements;
}
```

This follows partly from the general block rule above. This form differs from the Sun recommendation in the same way as the `if-else` statement described above.

72. Single statement `if-else`, `for` or `while` statements can be written without brackets.

```
if (condition)
  statement;

while (condition)
  statement;

for (initialization; condition; update)
  statement;
```

It is a common recommendation (Sun Java recommendation included) that brackets should always be used in all these cases. However, brackets are in general a language construct that groups several statements. Brackets are per definition superfluous on a single statement. A common argument against this syntax is that the code will break *if* an additional statement is added without also adding the brackets. In general however, code should never be written to accommodate for changes that *might* arise.

6.2 White Space

73.
- Operators should be surrounded by a space character.
- Java reserved words should be followed by a white space.
- Commas should be followed by a white space.
- Colons should be surrounded by white space.
- Semicolons in for statements should be followed by a space character.

```
a = (b + c) * d; // NOT: a=(b+c)*d

while (true) {   // NOT: while(true){
  ...

doSomething(a, b, c, d);  // NOT: doSomething(a,b,c,d);
```

```
case 100 :  // NOT: case 100:

for (i = 0; i < 10; i++) {  // NOT: for(i=0;i<10;i++){
  ...
```

Makes the individual components of the statements stand out and enhances readability. It is difficult to give a complete list of the suggested use of whitespace in Java code. The examples above however should give a general idea of the intentions.

74. Method names can be followed by a white space when it is followed by another name.

```
doSomething (currentFile);
```

Makes the individual names stand out. Enhances readability. When no name follows, the space can be omitted (`doSomething()`) since there is no doubt about the name in this case.

An alternative to this approach is to require a space *after* the opening parenthesis. Those that adhere to this standard usually also leave a space before the closing parentheses: `doSomething( currentFile );`. This do make the individual names stand out as is the intention, but the space before the closing parenthesis is rather artificial, and without this space the statement looks rather asymmetrical (`doSomething( currentFile);`).

75. Logical units within a block should be separated by one blank line.

```
// Create a new identity matrix
Matrix4x4 matrix = new Matrix4x4();

// Precompute angles for efficiency
double cosAngle = Math.cos(angle);
double sinAngle = Math.sin(angle);

// Specify matrix as a rotation transformation
matrix.setElement(1, 1,  cosAngle);
matrix.setElement(1, 2,  sinAngle);
matrix.setElement(2, 1, -sinAngle);
matrix.setElement(2, 2,  cosAngle);

// Apply rotation
transformation.multiply(matrix);
```

Enhances readability by introducing white space between logical units. Each block is often introduced by a comment as indicated in the example above.

76. Methods should be separated by three blank lines.

By making the space larger than space within a method, the methods will stand out within the class.

77. Variables in declarations can be left aligned.

```
TextFile  file;
int       nPoints;
double    x, y;
```

Enhances readability. The variables are easier to spot from the types by alignment.

78. Statements should be aligned wherever this enhances readability.

```
if      (a == lowValue)    compueSomething();
else if (a == mediumValue) computeSomethingElse();
else if (a == highValue)   computeSomethingElseYet();

value = (potential        * oilDensity)   / constant1 +
        (depth            * waterDensity) / constant2 +
        (zCoordinateValue * gasDensity)   / constant3;

minPosition     = computeDistance(min,     x, y, z);
averagePosition = computeDistance(average, x, y, z);

switch (phase) {
  case PHASE_OIL   : text = "Oil";   break;
  case PHASE_WATER : text = "Water"; break;
```

```
  case PHASE_GAS   : text = "Gas";   break;
}
```

There are a number of places in the code where white space can be included to enhance readability even if this violates common guidelines. Many of these cases have to do with code alignment. General guidelines on code alignment are difficult to give, but the examples above should give some general hints. In short, any construction that enhances readability should be allowed.

6.3 Comments

79. Tricky code should not be commented but rewritten [1].

In general, the use of comments should be minimized by making the code self-documenting by appropriate name choices and an explicit logical structure.

80. All comments should be written in English.

In an international environment English is the preferred language.

81. Javadoc comments should have the following form:

```
/**
 * Return lateral location of the specified position.
 * If the position is unset, NaN is returned.
 *
 * @param x    X coordinate of position.
 * @param y    Y coordinate of position.
 * @param zone Zone of position.
 * @return     Lateral location.
 * @throws IllegalArgumentException  If zone is <= 0.
 */
public double computeLocation(double x, double y, int zone)
  throws IllegalArgumentException
{
  ...
}
```

A readable form is important because this type of documentation is typically read more often *inside* the code than it is as processed text.

Note in particular:

- The opening /** on a separate line
- Subsequent * is aligned with the first one
- Space after each *
- Empty line between description and parameter section.
- Alignment of parameter descriptions.
- Punctuation behind each parameter description.
- No blank line bewteen the documentation block and the method/class.

Javadoc of class members can be specified on a single line as follows:

```
  /** Number of connections to this database */
  private int nConnections_;
```

82. There should be a space after the comment identifier.

```
// This is a comment    NOT: //This is a comment

/**                     NOT: /**
 * This is a javadoc           *This is a javadoc
 * comment                     *comment
 */                          */
```

Improves readability by making the text stand out.

83. Use // for all non-JavaDoc comments, including multi-line comments.

```
// Comment spanning
// more than one line.
```

Since multilevel Java commenting is not supported, using // comments ensure that it is always possible to comment out entire sections of a file using /* */ for debugging purposes etc.

84. Comments should be indented relative to their position in the code [1].

```
while (true) {       // NOT:  while (true) {
  // Do something              // Do something
  something();                   something();
}                               }
```

This is to avoid that the comments break the logical structure of the program.

85. The declaration of anonymous collection variables should be followed by a comment stating the common type of the elements of the collection.

```
private Vector  points_;    // of Point
private Set     shapes_;    // of Shape
```

Without the extra comment it can be hard to figure out what the collection consist of, and thereby how to treat the elements of the collection. In methods taking collection variables as input, the common type of the elements should be given in the associated JavaDoc comment.

Whenever possible one should of course qualify the collection with the type to make the comment superflous:

```
  private Vector<Point>  points_;
  private Set<Shape>     shapes_;
```

86. All public classes and public and protected functions within public classes should be documented using the Java documentation (javadoc) conventions.

This makes it easy to keep up-to-date online code documentation.

7 References

[1] Code Complete, Steve McConnel - Microsoft Press

[2] Java Code Conventions
http://java.sun.com/docs/codeconv/html/CodeConvTOC.doc.html

[3] Netscape's Software Coding Standards for Java
http://developer.netscape.com/docs/technote/java/codestyle.html

[4] C / C++ / Java Coding Standards from NASA
http://v2ma09.gsfc.nasa.gov/coding_standards.html

[5] Coding Standards for Java from AmbySoft
http://www.ambysoft.com/javaCodingStandards.html