ESTÁNDARES DE PROGRAMACIÓN EN

JAVA EE

Estándares de Programación en Java EE Versión 1.2

Elaborado por

Ángel Augusto Velásquez Núñez

Contenido

Introducción

Objetivos

Este documento tiene por finalidad que establecer los estándares de programación y nomenclatura de objetos de programación en Java a utilizar en las sesiones de clase y los proyectos de desarrollo de software.

Convenciones de nomenclatura

Convenciones de nomenclatura generales

1. **Los nombres de paquetes debería estar totalmente en minúsculas**. Esto se basa en la convención especificada por Sun para la nomenclatura de paquetes.

```
mypackage, com.company.application.ui
```

El nombre inicial de dominio del paquete debe estar totalmente en minúsculas.

2. **Los nombres que representan tipos deben ser sustantivos y deben escribirse con mayúsculas y minúsculas iniciando con mayúscula.**

```
Line, AudioSystem
```

Esta es una práctica común en la comunidad de desarrollo en Java e incluso la convención de nomenclatura de tipos que utiliza Sun en los paquetes predefinidos de Java.

3. **Los nombres de variables deben utilizar mayúsculas y minúsculas iniciando con minúscula.**

```
line, audioSystem
```

Es una práctica común en la comunidad de desarrollo Java y también la convención de nomenclatura de nombres de variables utilizada por Sun para los paquetes predefinidos de Java. Esta hace que las variables sean fáciles de distinguir de los tipos y resuelve de

manera efectiva las posibles colisiones de nombre como en la declararación `Line line;`

4. **Los nombres que representan constantes (variables finales) deben estar totalmente en mayúsculas utilizando subrayados (guión bajo) para separar las palabras.**

```
MAX_ITERATIONS, COLOR_RED
```

Es una práctica común en la comunidad de desarrollo en Java y también la convención de nomenclatura de nombres de utilizada por Sun para los paquetes predefinidos de Java. En general, debería minimizarse el uso de dichas constantes. En muchos casos es una mejor opción implementar el valor como un método.

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


}
```

Esta forma es más fácil de leer y asegura una interface uniforme hacia los valores de la clase.

5. **Los nombres que representan métodos deben ser verbos y escribirse con mayúsculas y minúsculas iniciando con minúscula.**

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

Es una práctica común en la comunidad de desarrollo en Java y también la convención de nomenclatura de nombres de utilizada por Sun para los paquetes predefinidos de Java. Es idéntico al caso de los nombres de variables, pero los métodos en Java ya se distinguen de las variables por su forma particular.

6. **Las abreviaturas y acrónimos no deberían estar con mayúsculas cuando se usan como nombre.**

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

Utilizar todo en mayúsculas para el nombre base entraría en conflicto con las convenciones anteriores. Una variable de este tipo debería nombrarse dVD, hTML, etc., lo cual obviamente no es legible. Otro problema se ilustra en el ejemplo anterior. Cuando el nombre se conecta a otro, se reduce seriamente la legibilidad. La palabra a continuación del acrónimo no queda como debería.

7. **Variables privadas de clase**. Las variables privadas de clase deberían tener como prefijo un guión bajo (_).

```
class Person

{

  private String _name;

  ...



}
```

Aparte de su nombre y tipo, el *ámbito* de una variable es su característica más importante. Indicar el ámbito de la clase utilizando guión bajo hace fácil distinguir las variables de clase de las variables descartables. Es importante dado que las variables de clase se considera que tengan un significado más alto que las variables de métodos, y deberían ser tratadas con especial cuidado por el programador.

Un efecto lateral de la convención de nombres con guión bajo es que resuelve el problema de encontrar nombres de variables para los métodos *set*.

```
void setName (String name)

  {

    _name = name;



  }
```

8. **Las variables genéricas deberían tener el mismo nombre que su tipo.**

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

1. `NOT: void setTopic (Topic aTopic)`
2. `NOT: void setTopic (Topic x)`

```
void connect (Database database)
```

1. `NOT: void connect (Database db)`
2. `NOT: void connect (Database oracleDB)`

Esto reduce la complejidad al reducir el número de términos y nombres utilizados. También hace fácil deducir el tipo contando solo con el nombre de la variable. Si por alguna razón esta

convención no parece *encajar*, esto es un fuerte indicador de que el nombre del tipo está mal elegido. Las variables no genéricas tienen un *rol*. Estas variables a menudo se pueden nombrar combinando el rol y el tipo.

```
Point startingPoint, centerPoint;



Name loginName;
```

9. **Todos los nombres deberían escribirse en inglés**.

```
fileName; // NOT: filNavn
```

El inglés es el lenguaje preferido para el desarrollo internacional.

10. **Los nombres de variables con un ámbito amplio deberían tener nombres largos, las variables con ámbito pequeño deberían tener nombres pequeños.** Las variables descartables utilizadas para el almacenamiento temporal o índices es mejor que sean cortas. Un programador que lee dichas variables debería poder asumir que su valor no se utiliza más allá de unas pocas líneas de código. Las variables descartables comunes para los enteros son *i*, *j*, *k*, *m*, *n* y para los caracteres *c* y *d*.

10. **El nombre del objeto está implícito y debería evitarse en un nombre de método.**

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

El uso de nombre del objeto podría parecer natural, en la declaración de la clase, pero se vuelve superfluo al utilizarlo, como se muestra en el ejemplo.

Convenciones de nomenclatura específicas

1. **Los términos *get/set* se deben utilizar cuando un atributo se accede directamente.**

```
employee.getName(); matrix.getElement
(2, 4); employee.setName (name);
matrix.setElement (2, 4, value);
```

Esta es la convención de nomenclatura para los métodos accesores utilzada por Sun en los paquetes predefinidos de Java. Cuando se escribe Java Beans esta convención en realidad es obligatoria para los métodos.

2. **El prefijo *is* debería usarse para las variables y métodos booleanos.**

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

Esta es la convención de nomenclatura para las variables y métodos booleanos utilizada por Sun en los paquetes predefinidos de Java. Cuando se escribe Java Beans esta convención en realidad es obligatoria para los métodos.

El uso del prefijo *is* resuelve el problema de elegir mal los nombres booleanos como *status* o *flag*. *isStatus* o *isFlag* simplemente no encajan y el programador se ve forzado a elegir otro nombre con mayor sentido. Hay unas cuantas alternativas que encajan mejor que el prefijo *is* en algunas situaciones. Estos son los prefijos *has*, *can*, y *should*.

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

3. **El término *compute* se puede utilizar en métodos donde algo se calcula.**

```
valueSet.computeAverage();
```

```
matrix.computeInverse();
```

De al lector la impresión inmediata que es una posible operación de consumo de tiempo, y que si se utiliza de manera repetida, él debería considerar guardar el resultado. Un consistente uso del término mejora la legibilidad.

4. **El término *find* se puede utilizar en los métodos donde se está buscando algo.**

```
vertex.findNearestVertex();
```

```
matrix.findMinElement();
```

De al lector la impresión inmediata que este es un simple método de búsqueda con un mínimo de cálculo implicado. El uso consistente del término incrementa la legibilidad.

5. **El término *initialize* se puede utilizar donde se establece un concepto u objeto.**

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

El *initialize* del inglés americano debería utilizarse en vez del *initialise* del inglés británico. Debe evitarse la abreviación *init*.

6. **Las variables JFC (Java Swing) deberían tener como sufijo el tipo de elemento.**

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

```
printerDialog
```

Esto mejora la legibilidad dado que el nombre da al usuario la impresión inmediata del tipo de variable y por consiguiente los recursos disponibles del objeto.

7. **La forma plural debería utilizarse en nombres que representen una colección de objetos.**

```
Collection points; // of Point int[]
values;
```

Esto mejora la legibilidad, dado que el nombre da al usuario la idea inmediata del tipo de variable y las operaciones que se pueden realizar sobre el objeto.

8. **El prefijo *n* debería utilizarse para variables que representen un número de objetos.**

```
nPoints, nLines
```

La notación es tomada de las matemáticas, donde es una convención para indicar un número de objetos. Note que Sun utiliza el prefijo *num* en los paquetes predefinidos de Java para tales variables. Esto probablemente debe significar una abreviación de *number of*, pero más parece *number* lo que hace ver a la variable extraña y engañosa. Si “number of” es la sentencia preferida, entonces debería usarse *numberOf* en vez de simplemente *n*. El prefijo *num* no debe usarse.

9. **El sufijo *No* debería utilizarse para variables que representen números de entidad.**

```
tableNo, employeeNo
```

La notación se toma en las matemáticas donde hay una convención establecida para indicar un número de entidad. Una alternativa elegante es colocar *i* como prefijo en dichas variables: *iTable*, *iEmployee*. Esto los hace de manera efectiva iteradores *nombrados*.

10. **Las variables Iterator deberían llamarse *i*, *j*, *k*, etc.**

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


}


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


}
```

Esta notación es tomada de las matemáticas donde hay una convención establecida para indicar a los iteradores. Las variables con nombre *j*, *k*, etc. deberían usarse solo para iteraciones anidadas.

11. **Los nombres complementarios deben usarse para las entidades complementarias**.

```
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
```

Esto reduce la complejidad en base a la simetría.

12. **Deben evitarse las abreviaciones en los nombres.**

```
computeAverage(); // NOT: compAvg(); ActionEvent event;
// NOT: ActionEvent e;
```

Hay dos tipos de palabras a considerar. Primero están los tipos de palabras comunes listadas en un diccionario de idiomas. Estas nunca deben ser abreviadas. Nunca escriba:

```
cmd          en vez de  command
comp         en vez de  compute
cp           en vez de  copy
e            en vez de  exception
init         en vez de  initialize
pt           en vez de  point
etc.
```

Por otro lado hay frases específicas difundidas que se conocen más por su acrónimo o abreviación. Estas frases deberían mantenerse abreviadas.

```
HypertextMarkupLanguage          en vez de  html
CentralProcessingUnit            en vez de  cpu
PriceEarningRatio                en vez de  pe
```

etc.

13. **Las variables booleanas negadas deben evitarse**.

```
boolean isError; // NOT: isNotError boolean
isFound; // NOT: isNotFound
```

El problema surge cuando el operador lógico *no* se usa y surge una doble negación. No es muy claro inicialmente el significado de !isNotError.

14. **Las constantes (variables finales) asociadas deben prefijarse con un nombre de tipo común.**

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

Esto indica que las constantes se relacionan y el concepto que las constantes representan. Una alternativa a esta propuesta es poner las constantes dentro de una interfaz y así prefijar sus nombres con el nombre de la interfaz.

```
interface Color
{
    final int RED           = 1;
```

```
            final int GREEN = 2;
    final int BLUE          = 3;
}
```

15. **Las clases Exception deberían tener como sufijo *Exception*.**

```
class AccessException

{

  :



}
```

Las clases Exception no son realmente parte del diseño principal del programa y nombrarlas así las hace ser independientes con respecto a otras clases. Este estándar es seguido por Sun en la biblioteca básica de Java.

16. **Las implementaciones por defecto de las interfaces deben tener como prefijo**

***Default*.**

```
class DefaultTableCellRenderer implements
TableCellRenderer

{

  :

}
```

No es poco común crear una clase de implementación simplista de una interfaz, la cual proporcione comportamiento por defecto para los métodos de la interfaz. La convención de poner el prefijo *Default* a estas clases ha sido adoptado por Sun para las bibliotecas de Java.

17. **Las funciones (métodos que retornan un objeto) deberían nombrarse después de lo que retornan y los procedimientos (métodos *void*) después de lo que hacen.**

Esto incrementa la legibilidad. Aclara qué es lo que la unidad debería hacer y todas las cosas que *no* se supone que hace. Esto nuevamente facilita el mantener el código limpio de efectos laterales.

Archivos

Convenciones generales

1. **Los archivos fuente de Java deberían tener la extensión *.java*.**

```
Point.java
```

Esto es reforzado por las herramientas de Java.

2. **Las clases deberían declararse en archivos individuales con un nombre de archivo que concuerde con el nombre de la clase.** Las clases privadas secundarias pueden declararse como clases internas y residir en el archivo de la clase a la que pertenecen. Esto es apoyado por las herramientas de Java.

2. **El contenido del archivo debe mantenerse en 80 columnas.** 80 columnas es es la dimensión común para los editores, emuladores de terminal, impresoras y depuradores; los archivos que se comparten entre varios desarrolladores deberían alinearse a estas restricciones. Cuando el archivo pasa por varios programadores, se mejora la legibilidad cuando se evitan los cortes de línea sin intención.

2. **Debe evitarse caracteres especiales como TAB y salto de página.** Estas características tienden a causar problemas en los editores, impresoras, emuladores de terminales o depuradores cuando se usan en entornos multi-plataforma y/o multi-programador.

2. **Debe evidenciarse cuando una línea está incompleta por estar partida.**

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

```
function (param1,
param2, param3);
```

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

```
for (tableNo = 0; tableNo < maxTable;
tableNo += tableStep)
```

Particionar líneas ocurre cuando una sentencia ocupa más del límite de 80 columnas especificado arriba. Es difícil dar reglas rígidas de cómo deberían partirse las líneas. Pero los ejemplos pueden dar una pista general.

1. Corte después de una coma.
2. Corte después de un operador.
3. Alinear la nueva línea con el inicio de la expresión en la línea anterior.

Sentencias

Sentencias package e import

1. **La sentencia package debe ser la primera sentencia del archivo**. Todos los archivos pertenecen a un paquete específico. La ubicación de la sentencia package es apoyada por el lenguaje Java. El permitir que todos loa archivos pertenezcan a un paquete real (en vez del Java por defecto) refuerza las técnicas de programación orientadas a objeto de Java.

2. **La sentencia import debe seguir a las sentencia package**. La sentencias import deben ordenarse colocando el paquete fundamental primero, y agrupando los paquetes asociados, y colocando una línea en blanco de separación entre grupos.

```
import java.io.*; import
java.net.*;

import java.rmi.*



import java.rmi.server.*;

import javax.swing.*; import
javax.swing.event.*;



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

   La ubicación de la sentencia import es apoyada por el lenguaje Java. El ordenamiento facilita listar cuando hay muchos import, y hace fácil de determinar las dependencias de los paquetes presentes. El agrupamiento reduce la complejidad contrayendo la información relacionada en una unidad común.

Clases e interfaces

1. **Las declaraciones de clase deberían organizarse.** Esto debería hacerse de la siguiente manera:

1. Documentación de la Clase/Interface.
2. Sentencia class o interface.
3. Variables de clase (estáticas) en el orden public, protected, package (sin modificador de acceso), privadas.
4. Variables de instancia en el orden public, protected, package (sin modificador de acceso), private.
5. Constructores.
6. Métodos (sin orden específico).

   Reduzca la complejidad haciendo predecible la ubicación de cada elemento de clase.

Métodos

1. **Los modificadores de métodos deberían darse en el siguiente orden: <acceso> static abstract synchronized <inusual> final native.** El modificador <acceso> (si existe) debe ser el primer modificador.

   <acceso> es *public*, *protected*, o *private* mientras que <inusual> incluye *volatile* y *transient*. Lo más importante aquí es mantener el modificador *acceso* como el primer modificador. De entre los posibles modificadores, este es de lejos el más importante y debe estar al inicio en la declaración del método. Para otros modificadores, el orden es menos importante, pero es conveniente tener una convención fija.

Tipos

1. **Las conversiones de tipos deben hacerse siempre explícitas.** Nunca caiga en conversiones implícitas.

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

   El programador debe indicar que está conciente de los diferentes tipos implicados y que la mezcla es intencional.

Variables

1. **Las variables deberían inicializarse donde se declaran y deberían declararse en el ámbito más pequeño posible.** Esto asegura que las variables son válidas en todo momento. A veces es imposible inicializar una variable con un valor válido donde se le declara. En esos casos debería dejarse sin inicializar en vez de darle un valor sin sentido.

2. **Las variables nunca deben tener significado dual.** Esto mejora la legibilidad asegurando que todos los conceptos se representan de manera única. Reduce las posibilidades de errores producto de la dualidad.

3. **Las variables de clase nunca se deberían declarar public.** El concepto de ocultamiento de información y encapsulamiento de Java es violado por las variables públicas. Use las variables privadas y acceda en vez de ello a las funciones. Una excepción a esta regla es cuando la clase es básicamente una estructura de datos, sin comportamiento (equivalente al struct de C++). En este caso es apropiado hacer las variables de instancia de la “clase” públicas.

4. **Las variables relacionadas del mismo tipo se pueden declarar en una sentencia común.** Las variables no relacionadas no se deberían declarar en la misma sentencia.

   ```
   float x, y, z;



   float revenueJanuary, revenueFebrury, revenueMarch;
   ```

   El requerimiento común de tener declaraciones en líneas separadas no es útil en los ejemplos. Esto mejora la legibilidad al agrupar variables. Note que sin embargo que cuando sea posible, las variables deberían inicializarse donde se declaren (vea la regla 4.5.1), en caso que esta situación no ocurra.

5. **Los arreglos deberían declararse con corchetes junto al tipo**.

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


          public static void main (String[] arguments)

          public double[] computeVertex()
```

Esto tiene doble motivo. Primero, la característica de ser arreglo es de la clase, no de la variable. Segundo, cuando se retorna un arreglo en un método, no es posible tener los corchetes en otro lugar que no sea con el tipo (como se muestra en el ejemplo).

6. **Las variables deberían mantenerse con vida el menor tiempo posible**. Mantener las operaciones sobre una variable dentro de un ámbito pequeño, es más fácil de controlar efectos y efectos laterales de la variable.

Repeticiones

1. **Sólo las sentencias de control de flujo deben incluirse en la construcción del *for()***.

```
sum = 0;



for (i = 0; i < 100; i++) sum += value[i];
```

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

Esto incrementa la legibilidad y mantenibilidad. Hace una clara distinción que *controla* y que está *contenida* en la repetición.

2. **Las variables de repetición deberían inicializarse inmediatamente antes de la repetición**.

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

1. `}`

2. **Debería evitarse el uso de ciclos do…while**. Hay dos razones para esto. Primero es que la construcción es superflua. Cualquier sentencia que se puede escribir como un *do…while* se puede igualmente escribir como una repetición *while* o una repetición *for*. La complejidad se reduce si se utiliza la menor cantidad de construcciones. La otra razón es legibilidad. Un ciclo con la parte condicional el final es más difícil de leer que uno con el condicional arriba.

2. **Se debería evitar el uso de *break* y *continue* en las repeticiones.** Estas sentencias sólo se deberían usar si demuestran brindar más legibilidad que sus contrapartes estructuradas.

Condicionales

1. **Se debe evitar las expresiones condicionales.** Introduzca en su lugar variables booleanas temporales.

```
    if ((elementNo < 0) || (elementNo > maxElement)||
    elementNo == lastElement) {

  :




}
```

Debería reemplazarse por:

```
boolean isFinished = elementNo < 0 || elementNo > maxElement; boolean
isRepeatedEntry = elementNo == lastElement;

if (isFinished || isRepeatedEntry) {




  :

}
```

Al asignar variables booleanas a las expresiones, el programa obtiene documentación automáticamente. La construcción será más fácil de leer, depurar y mantener.

2. **El caso nominal debería colocarse en la parte *if* y la excepción en la parte *else* de una sentencia if**.

```
boolean isError = readFile (fileName);

if (!isError) {

  :

}

else {

  :

}
```

Asegúrese que las excepciones no restan claridad a la ruta normal de ejecución. Esto es importante tanto para la legibilidad como para el rendimiento.

3. **El condicional debería colocarse en una línea separada**.

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

Esto es para propósitos de depuración. Cuando se escribe en una sola línea no es visible cuando la evaluación es verdadera o falsa.

4. **Debe evitarse las sentencias ejecutables en los condicionales**.

```
            file = openFile (fileName); // NOT: if ((file = openFile (fileName)) !=
            null) {
if (file != null) {                     //      :
                :                       //     }
}
```

Los condicionales con sentencias ejecutables son difíciles de leer. Esto es particularmente cierto para los programadores en Java.

Miscelánea

1. **El uso de números mágicos en el código debería evitarse.** Los números diferentes de *0* y *1* se pueden considerar en vez de eso, declarados como constantes nombradas.

```
            private static final int TEAM_SIZE = 11;

            :



            Player[] players = new Player[TEAM_SIZE]; //
            NOT: Player[] players = new Player[11];
```

Si el número no tiene un significado obvio de por si, la legibilidad se mejora introduciendo en vez de eso una constante nombrada.

2. **Las constantes de punto flotante siempre se deberían escribir con punto decimal y con al menos un decimal.**

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

            double sum;

           :



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

Esto enfatiza la naturaleza diferente de los números enteros y de punto flotante. Matemáticamente los dos modelan conceptos diferentes y no compatibles. También, como en el ejemplo, esto enfatiza el tipo de variable asignada (sum) al punto punto en el código donde no podría ser evidente.

3. **Las constantes de punto flotante siempre se deberían escribir con un dígito antes del punto decimal.**

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

El número y sistema de expresión en Java es tomado de las convenciones matemáticas para la sintaxis en lo posible. Asimismo, 0.5 es más legible que .5; No hay manera que se pueda confundir con el entero 5.

4. **Las variables o métodos estáticos siempre se deben referir por el nombre de la clase y nunca por el nombre de una instancia.**

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

Esto enfatiza que el elemento que se referencia es estático e independiente de cualquier instancia particular. Por la misma razón, el nombre de clase debería incluirse cuando una variable o método se accede desde dentro de la misma clase.

Organización y comentarios

Organización

1. **La indentación básica debería ser 2.**

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

La indentación se utiliza para enfatizar la estructura lógica del código. La indentación de 1 es muy pequeña para alcanzarlo. La indentación mayor a 4 hace que el código muy profundo y difícil de leer e incrementa la probabilidad de que las líneas se partan. Eligiendo entre las indentaciones de 2, 3 y 4; 2 y 4 son las más comunes, y 2 se elige para reducir las posibilidades de partir las líneas de código. Note que las recomendaciones de Sun en este punto son 4.

2. **Organización de Bloques.** La organización de bloques debería ser como se ilustra en el ejemplo 1 abajo (recomendado) o el ejemplo 2, pero no como se muestra en el ejemplo 3. Los bloques de clase, interfaz y método deberían usar la organización de bloques del ejemplo 2.

El ejemplo 3 introduce un nivel de indentación extra que no enfatiza la estructura lógica del código tan claramente como el ejemplo 1 y 2.

3. **Declaración de clase e interfaz.** Las declaraciones de clases e interfaces deberían ser de la siguiente forma.

```
class SomeClass extends AnotherClass implements
SomeInterface, AnotherInterface

{

  ...



}
```

Esto sigue la regla general de bloques especificada arriba. Note que es común en la comunidad de desarrolladores de Java tener llaves abiertas al final de la línea de la palabra reservada class. Esto no se recomienda.

4. **Declaración de métodos.** Las declaraciones de métodos deberían tener la siguiente forma.

```
public void someMethod ()
throws SomeException

{

  ...

}
```

Vea los comentarios en las sentencias class arriba.

5. **If-else.** Las sentencias de clase if-else deberían tener la siguiente forma.

```
if (condition) {

   statements;
}

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

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

```
    statements;
  }
```

Esto sigue en parte la regla general de bloques. Sin embargo, debería discutirse si la cláusula *else* debería estar en la misma línea que el cierre de corchetes de la cláusula *if* o *else* previa.

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

Esto es equivalente a la recomendación de Sun. El estilo elegido se considera mejor en el sentido que cada parte de la sentencia if-else se escribe en líneas separadas del archivo. Esto haría más fácil manipular la sentencia, por ejemplo al mover las cláusulas *else*.

6. **Sentencia for.** La sentencia for debería tener la siguiente forma.

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



}
```

Esto concuerda con la regla general de bloques.

7. **Sentencia for vacía.** Una sentencia for vacía debería tener la siguiente forma.

```
for (initialization; condition; update)

  ;
```

Esto enfatiza el hecho de que la sentencia for está vacía y hace obvio para el lector que es intencional.

8. **Sentencia while.** La sentencia while debería tener la siguiente forma.

```
while (condition) {
statements;



}
```

Esto sigue la regla general de bloques.

9. **Sentencia do-while.** La sentencia do-while debería tener la siguiente forma.

```
  do { statements;

} while (condition);
```

Esto sigue la regla general de bloques.

10. **Sentencia switch.** La sentencia switch debería tener la siguiente forma.

```
  switch (condition) { case
  ABC :

    statements;



    // Fallthrough case DEF :

    statements;

    break; case XYZ :

    statements;

    break; default :

    statements;

    break;

}
```

Esto difiere de la recomendación de Sun tanto para indentación como para espaciado. En particular, cada palabra reservada *case* se indenta relativa a la sentencia switch como un todo. Esto hace consistente toda la sentencia switch. Note igualmente el espacio extra antes del carácter. El comentario explícito *// Continuar ejecutando* debería incluirse cuando hay una sentencia case sin un cláusula *break.* El dejar la sentencia *break* fuera es un error común y debe quedar claro que es intencional cuando no está ahí.

11. **Sentencia try-catch**. Una sentencia try-catch debería tener la siguiente forma.

```
  try { statements;

}

  catch (Exception exception) {
  statements;



}

  try { statements;

}

  catch (Exception exception) {
  statements;

}
```

```
    finally { statements;

  }
```

Esto forma parte de la regla general de bloques. Esta forma difiere de la recomendación de Sun en la misma forma que la sentencia *if-else* descrita arriba.

12. **Sentencias if-else, for o while simples.** Las sentencias if-else, for o while deberían escribirse sin llaves.

```
if (condition)
statement;



while (condition)
statement;



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

Es una recomendación común (incluyendo la recomendación de Sun) que las llaves deberían siempre usarse en todos los casos. Sin embargo, las llaves son en general una construcción del lenguaje que agrupa varias sentencias. Las llaves son por definición superfluas en una sentencia simple. Un argumento contra esta sintaxis es que el código

> va a cortarse si se adiciona una sentencia sin agregar las llaves. En general sin embargo, el código nunca debería escribirse para acomodar los cambios que podrían surgir.

Espacios en blanco

1. **Reglas generales.** Debería seguirse las siguientes reglas:

1. Los operadores deberían rodearse por un carácter de espacio.
2. Las palabras reservadas de Java deberían estar seguidas por un espacio en blanco.
3. Las comas deberían estar seguidas por un espacio en blanco.
4. Los dos puntos deberían estar rodeados por espacio en blanco.
5. Los puntos y coma en las sentencias *for* deberían estar seguidos de un espacio en blanco.

```
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++){
```

Hace los componentes individuales de las sentencias consistentes y mejora la legibilidad. Es difícil dar una lista completa del uso sugerido del espacio en blanco en el código en Java. Los ejemplos arriba sin embargo deberían dar una idea general de las intenciones.

2. **Nombres de Funciones.** Los nombres de funciones deberían estar seguidos de un espacio en blanco cuando son seguidos de otro nombre.

```
doSomething (parameter);            // NOT: doSomething(parameter);
doSomething();                      // OK
```

Esto hace los nombres individuales consistentes y mejora la legibilidad. Cuando no lo sigue nombre alguno, el espacio puede omitirse dado que no hay duda sobre el nombre en este caso. Una alternativa a esta propuesta es requerir un espacio *después* de abrir el paréntesis. Los que se adhieren a este estándar usualmente dejan un espacio antes de cerrar el paréntesis. Esto hace los nombres individuales consistentes como es la intención, pero el espacio antes del cierre de paréntesis es algo artificial, y sin este espacio la sentencia luce asimétrica.

3. **Unidades lógicas.** Las unidades lógicas deberían separarse por una línea en blanco.

```
Matrix4x4 matrix = new Matrix4x4();

double cosAngle = Math.cos (angle); double
sinAngle = Math.sin (angle);



matrix.setElement (1, 1, cosAngle);
matrix.setElement (1, 2, sinAngle);
matrix.setElement (2, 1, -sinAngle);
matrix.setElement (2, 2, cosAngle);

multiply (matrix);
```

Esto mejora la legibilidad introduciendo un espacio en blanco entre las unidades lógicas de un bloque.

4. **Métodos.** Los métodos deberían separarse por 3-5 espacios en blanco. Haciendo el espacio mayor que el espacio dentro de un método, los métodos serán diferenciados dentro de la clase.

5. **Variables.** Las variables en las declaraciones deberían alinearse a la izquierda, mejorando así la legibilidad del código.

```
TextFile file;

int nPoints;



double x, y;
```

Las variables son más fáciles de distinguir de los tipos alineándolas.

6. **Sentencias.** Las sentencias deberían alinearse donde se mejore la legibilidad.

```
if                (a == lowValue)    compueSomething();
                  (a ==
else if           mediumValue)       computeSomethingElse();
else if           (a == highValue)   computeSomethingElseYet();
                                                    / constant1
value =           (potential        * oilDensity)     +
                                                    / constant2
                  (depth            * waterDensity)   +
                  (zCoordinateValue * gasDensity)     / constant3;
minPosition               = computeDistance (min,      x, y, z);

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


          switch (value) {
            case PHASE_OIL       : phaseString = "Oil"; break;
            case   PHASE_WATER : phaseString = "Water"; break;
               case PHASE_GAS ; : phaseString = "Gas"; break;
          }
```

Hay varios lugares en el código donde el espacio en blanco se puede incluir para mejorar la legibilidad incluso si esto viola las reglas comunes. Muchos de los casos tienen que ver con la alineación de código. Las reglas generales para la alineación de código son difíciles de dar, pero los ejemplos arriba deberían dar una idea general de la idea. En términos simples, cualquier construcción que mejore la legibilidad debería permitirse.

Comentarios

1. **Código confuso.** El código confuso no debería comentarse sino reescribirse. En general, el uso de los comentarios debería minimizarse haciendo el código auto documentado con elecciones apropiadas de nombres y estructuras lógicas explícitas.

2. **Idioma.** Todos los comentarios deberían escribirse en inglés. En un entorno internacional el inglés es el idioma preferido.

3. **Uso de //.** Use // para todos comentarios que no son de JavaDoc, incluyendo los comentarios multilínea.

1. `Comment spanning`
2. `more than one line`

Dado que los comentarios multi-nivel no es soportado, el uso de los comentarios // asegura que siempre sea posible comentar secciones enteras de un archivo usando /* */ para propósitos de depuración, etc.

4. **Indentación de comentarios.** Los comentarios deberían indentarse relativos a su posición en el código.

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

```
}                                   //      }
```

Esto ayuda a evitar que los comentarios quiebren la estructura lógica del programa.

5. **Colecciones.** La declaración de variables *collection* debería estar seguida de un comentario que establezca el tipo común de los elementos de la colección.

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

Sin el comentario extra puede ser difícil imaginar en qué consiste la colección y como tratar a los elementos de la colección. En métodos que toman como entrada variables colección, el tipo común de los elementos debería especificarse en el comentario asociado de JavaDoc.

6. **Clases y funciones.** Todas las clases públicas y las funciones públicas y protegidas dentro de las clases públicas deberían documentarse utilizando las convenciones de documentación de Java (javadoc). Esto hace más fácil mantener al día la documentación en línea del código.