|  |  |  |
| --- | --- | --- |
| Evento | Autor | Fecha |
| Creación del documento | Gilberto Mangones | 29/03/2014 |
| Revisión y aprobación | Jorge Luis Sarmiento Herrera  Nestor Andres Cabezas  Gilberto Mangones  Gustavo Suárez Pinto  Miguel Angel Dominguez  Diego Felipe Sanchez | 31/03/2014 |
| Actualización | Gilberto Mangones | 22/04/2014 |

[**ESTÁNDARES DE CODIFICACIÓN JAVA**](http://angelyanez.blogspot.com/2012/03/estandares-de-codificacion-java.html)

|  |  |
| --- | --- |
| Propósito | Guiar en la implementación de programas con los Estándares de Codificación Java. |
| Documentación JavaDoc | Se aconseja, como buena práctica de programación, incluir en la entrega de la aplicación la documentación de los ficheros fuente de todas las clases. Dicha documentación será generada por la herramienta "javadoc". La herramienta "javadoc" construirá la documentación a partir de los comentarios (incluidos en las clases) encerrados entre los caracteres "/\*\*" y "\*/" **@author**Nombre - Añade información sobre el autor o autores del código  **@version** InformacionVersion - Permite incluir información sobre la versión y         fecha del código.  **@param** NombreParametro Descripción - Inserta el parámetro especificado y su descripción en la sección "Parameters:" de la documentación del método en el que se incluya  **@return** Descripción - Inserta la descripción indicada en la sección "Returns:" de la documentación del método.  **@throws** NombreClase Descripción - Añade el bloque de comentario "Throws:" incluyendo el nombre y la descripción de la excepción especificada.  **@see** Referencia - Permite incluir en la documentación la sección de comentario "See also:", conteniendo la referencia indicada.  **@deprecated** Explicación - Esta etiqueta indica que la clase, interfaz, método o campo está obsoleto y que no debe utilizarse, y que dicho elemento posiblemente desaparecerá en futuras versiones. ".  **@since** Version - Se utiliza para especificar cuándo se ha añadido a la API la clase, interfaz, método o campo. Debería incluirse el número de versión u otro tipo de información. |
| Encabezado del Programa | Comience todos los programas con una cabecera descriptiva. |
| Formato de la cabecera Estándar Codificación Java | /\*  \* @(#)Discapacidad.java 1.7 \*   \* Copyright Sun Microsystems, Inc. All rights reserved.      \* SUN PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.  \*/  /\*\*  \* Esta clase implementa los métodos necesarios para poder utilizar los métodos necesarios en el manejo del registro de discapacidades.  \* @author EAIT  \* @version 1.7, 04/14/14   \* @since 1.4  \*/  Todo fichero fuente debe comenzar con un comentario que incluya el nombre de la clase, información sobre la versión del código, la fecha y el copyright.  El copyright indica la propiedad legal del código, el ámbito de distribución, el uso para el que fue desarrollado y su modificación.  Dentro de estos comentarios iniciales podrían incluirse adicionalmente comentarios sobre los cambios efectuados sobre dicho fichero (mejora, incidencia, error, etc.). |
| Sentencias de paquete | La primera línea no comentada de un fichero fuente debe ser la sentencia de paquete, que indica el paquete al que pertenece(n) la(s) clase(s) incluída(s) en el fichero fuente. Por ejemplo,  package javax.crypto; |
| Sentencias de importación | Tras la declaración del paquete se incluirán las sentencias de importación de los paquetes necesarios. Esta importación de paquetes obligatorios seguirá el siguiente orden:        1.      Paquetes del JDK de java.        2.      Paquetes de utilidades no pertenecientes al JDK de Java, de frameworks de               desarrollo o de proyectos opensource tales como apache, hibernate,               springframework, etc.        3.      Paquetes de la aplicación. |
| Declaraciones de clases e interfaces | **Comentario de documentación de la clase/interfaz /\*\* ... \*/** - Permite describir la clase/interfaz desarrollada. Necesario para generar la documentación de la api mediante javadoc.  **Comentario de implementación de la clase/interfaz, si es necesario /\* ... \*/ -** Este comentario incluye cualquier información que no pueda incluirse en el comentario de documentación de la clase/interfaz.  **Variables de clase (estáticas) -** En primer lugar las variables de clase públicas (public), después las protegidas (protected), posteriormente las de nivel de paquete (sin modificador), y por último las privadas (private).  **Variables de instancia -** Primero las públicas (public), después las protegidas (protected), luego las de nivel de paquete (sin modificador), y finalmente las privadas (private).  **Métodos -** Deben agruparse por funcionalidad en lugar de agruparse por ámbito o accesibilidad. Por ejemplo, un método privado puede estar situado entre dos métodos públicos. El objetivo es desarrollar código fácil de leer y comprender. |
| Sangría | Como norma general se establecen 4 caracteres como unidad de sangría. Los entornos de desarrollo integrado (IDE) más populares, tales como Eclipse o NetBeans, incluyen facilidades para formatear código Java. |
| Longitud de línea | La longitud de línea no debe superar los 80 caracteres por motivos de visualización e impresión |
| División de líneas | Cuando una expresión ocupe más de una línea, esta se podrá romper o dividir en función de los siguientes criterios:           1.      Tras una coma.           2.      Antes de un operador.           3.      Se recomienda las rupturas de nivel superior a las de nivel inferior.           4.      Alinear la nueva línea con el inicio de la expresión al mismo nivel que la línea                  anterior.            5.      Si las reglas anteriores generan código poco comprensible, entonces estableceremos tabulaciones de 4 espacios.  Ejemplos:  unMetodo(expresionLarga1, expresionLarga 2, expresionLarga 3,          expresionLarga 4, expresionLarga 5);  if ((condicion1 && condicion2)         || (condicion3 && condicion4)                 ||!(condicion5 && condicion6)) {                 unMetodo(); } |
| Comentarios de implementación | Estos comentarios se utilizan para describir el código ("el cómo"), y en ellos se incluye información relacionada con la implementación, tales como descripción de la función de variables locales, fases lógicas de ejecución de un método, captura de excepciones, etc.Distinguimos tres tipos de comentarios de implementación:         1.       **Comentarios de bloque:** Permiten la descripción de ficheros, clases, bloques, estructuras de datos y algoritmos.   /\*  \* Esto es un comentario  \* de bloque  \*/           2.       **Comentarios de línea:** Son comentarios cortos localizados en una sola línea y tabulados al mismo nivel que el código que describen. Si ocupa más de una línea se utilizará un comentario de bloque. Deben estar precedidos por una línea en blanco.  /\* Esto es un comentario de línea \*/ // Esto es otro comentario de línea          3.       **Comentario a final de línea:** Comentario situado al final de una sentencia de código y en la misma línea.  int contador = 4 + 10;      // Inicialización del contador contador++;                    /\* Incrementamos el contador \*/ |
| Declaraciones | **Una declaración por línea:** Se recomienda el uso de una declaración por línea, promoviendo así el uso de comentarios. Ejemplo:  int idUnidad;          // Identificador de la unidad organizativa  String[] funciones; // Funciones de la unidad  **Inicialización:** Toda variable local tendrá que ser inicializada en el momento de su declaración, salvo que su valor inicial dependa de algún valor que tenga que ser calculado previamente.  Ejemplo:  int idUnidad          = 1;  String[] funciones = { "Administración", "Intervención"};  **Localización:**Las declaraciones deben situarse al principio de cada bloque principal en el que se utilicen, y nunca en el momento de su uso.  public void unMetodo() {  int contador = 0;  // inicio del método  ... }  La única excepción a esta regla son los índices de los bucles "for", ya que, en Java, pueden incluirse dentro de la propia sentencia "for".  for (int i=0; contador<10; i++) {  ... }  Se debe evitar el uso de declaraciones que oculten a otras declaraciones de ámbito superior.  int contador = 0;  // Inicio del método  public void unMetodo() {   if (condicion) {   int contador = 2;  // ¡¡ EVITAR !!   ...  }  ... }    **Declaración de clases / interfaces**: Durante el desarrollo de clases / interfaces se deben seguir las siguientes reglas de formateo:             No incluir ningún espacio entre el nombre del método y el paréntesis inicial del listado de parámetros.             El carácter inicio de bloque ("{") debe aparecer al final de la línea que contiene la sentencia de declaración.        1.  El carácter fin de bloque ("}") se sitúa en una nueva línea tabulada al mismo nivel que su correspondiente sentencia de inicio de bloque, excepto cuando la sentencia sea nula, en tal caso se situará detrás de "{".        2. Los métodos se separarán entre sí mediante una línea en blanco.  Ejemplo:  public class ClaseEjemplo extends Object {   int variable1;  int variable2;   public ClaseEjemplo() {   variable1 = 0;   variable2 = 1;  }  ... } |
| Sentencias | Cada línea debe contener como máximo una sentencia. Ejemplo,  int contador++; int variable--;               Las sentencias pertenecientes a un bloque de código estarán tabuladas un nivel más a la derecha con respecto a la sentencia que las contiene. El carácter inicio de bloque "{" debe situarse al final de la línea que inicia el bloque. El carácter final de bloque "}" debe situarse en una nueva línea tras la última línea del bloque y alineada con respecto al primer carácter de dicho bloque.               Todas la sentencias de un bloque deben encerrarse entre llaves "{ ... }", aunque el bloque conste de una única sentencia. Esta práctica permite añadir código sin cometer errores accidentalmente al olvidar añadir las llaves. Ejemplo,  if (condicion) {  variable++; }  La sentencia "try/catch" siempre debe tener el formato siguiente,  try {     sentencias; } catch (ClaseException e) {     sentencias; }             En el bloque "catch" siempre se imprimirá una traza de error indicando el tipo de excepción generada y posteriormente se elevará dicha excepción al código invocante, salvo que la lógica de ejecución de la aplicación no lo requiera.             Siempre se utilizará el bloque "finally" para liberar recursos y para imprimir trazas de monitorización de fin de ejecución.  try {     sentencias; } catch (ClaseException e) {     sentencias; } finally {     sentencias; } |
| Espacios en blanco | Las líneas y espacios en blanco mejoran la legibilidad del código permitiendo identificar las secciones de código relacionadas lógicamente. Se utilizarán espacios en blanco en los siguientes casos:      1.  Entre una palabra clave y un paréntesis. Esto permite que se distingan las llamadas a métodos de las palabras clave. Por ejemplo:  while (true) {     ... }        2.     Tras cada coma en un listado de argumentos. Por ejemplo:  objeto.unMetodo(a, b, c);        3.      Para separar un operador binario de sus operandos, excepto en el caso del operador ("."). Nunca se utilizarán espacios entre los operadores unarios ("++" o "--") y sus operandos. Por ejemplo:  a += b + c; a = (a + b) / (c + d); contador++;        4.  Para separar las expresiones incluidas en la sentencia "for". Por ejemplo: for (expresion1; expresion2; expresion3)        5. Al realizar el moldeo o "casting" de clases. Ejemplo:  Unidad unidad = (Unidad) objeto; |
| Clases e interfaces | Los nombres de clases deben ser sustantivos y deben tener la primera letra en mayúsculas. Si el nombre es compuesto, cada palabra componente deberá comenzar con mayúsculas.             Los nombres serán simples y descriptivos.             Debe evitarse el uso de acrónimos o abreviaturas.             Las interfaces se nombrarán siguiendo los mismos criterios que los indicados para las clases. Como norma general toda interfaz se nombrará con el prefijo "I" para diferenciarla de la clase que la implementa (que tendrá el mismo nombre sin el prefijo "I").  Ejemplo:  class Ciudadano class OrganigramaDAO class AgendaService class IAgendaService |
| Métodos | Los métodos deben ser verbos escritos en minúsculas. Cuando el método esté compuesto por varias palabras cada una de ellas tendrá la primera letra en mayúsculas.  Ejemplos:  public void insertaUnidad(Unidad unidad); public void eliminaAgenda(Agenda agenda); public void actualizaTramite(Tramite tramite) |
| Variables | Las variables se escribirán siempre en minúsculas. Las variables compuestas tendrán la primera letra de cada palabra componente en mayúsculas.        Las variables nunca podrán comenzar con el carácter "\_" o "$". Los nombres de variables deben ser cortos y sus significados tienen que expresar con suficiente claridad la función que desempeñan en el código. Debe evitarse el uso de nombres de variables con un sólo carácter, excepto para variables temporales.  Ejemplos:  Unidad unidad; Agenda agenda; Tramite tramite; |
| Constantes | Todos los nombres de constantes tendrán que escribirse en mayúsculas. Cuando los nombres de constantes sean compuestos las palabras se separarán entre sí mediante el carácter de subrayado "\_".  Ejemplos:  int LONGITUD\_MAXIMA; int LONGITUD\_MINIMA;        Los valores constantes (literales) nunca aparecerán directamente en el código. Para designar dichos valores se utilizarán constantes escritas en mayúsculas y se declararán, según su ámbito de uso, o bien en una Clase de constantes creada para tal efecto, o bien en la clase donde sean utilizadas.  Ejemplos:  // Uso incorrecto codigoErrorUsuarioEncontrado = 1; ... switch (error) {   case codigoErrorUsuarioEncontrado:    ... }  // Uso correcto public final int CODIGOERROR\_USUARIOENCONTRADO = 1; ... switch (error) {   case CODIGOERROR\_USUARIOENCONTRADO:    ... } |
| Visibilidad de atributos de instancia y de clase | Los atributos de instancia y de clase serán siempre privados, excepto cuando tengan que ser visibles en subclases herederas, en tales casos serán declarados como protegidos.        El acceso a los atributos de una clase se realizará por medio de los métodos "get" y "set" correspondientes, incluso cuando el acceso a dichos atributos se realice en los métodos miembros de la clase.  Ejemplo:  public class Unidad {  private int id;  private String nombre;  ...   public void actualizaUnidad(Unidad unidad) {   this.setId(unidad.getId());   this.setNombre(unidad.getNombre());  }  ... } |
| Referencias a miembros de una clase | Evitar el uso de objetos para acceder a los miembros de una clase (atributos y métodos estáticos). Utilizaremos en su lugar el nombre de la clase. Por ejemplo  metodoUtilidad();    // Acceso desde la propia clase estática ClaseUtilidad.metodoUtilidad(); // Acceso común desde cualquier clase |
| Asignación sobre variables | Se deben evitar las asignaciones de un mismo valor sobre múltiples variables en una misma sentencia, ya que dichas sentencias suelen ser difíciles de leer.  int a = b = c = 2;  // Evitar        No utilizar el operador de asignación en aquellos lugares donde sea susceptible de confusión con el operador de igualdad. Por ejemplo:  // INCORRECTO if ((c = d++) == 0) { }  // CORRECTO c = d++; if (c == 0) { }        No utilizar asignaciones embebidas o anidadas. Ejemplo:  c = (c = 3) + 4 + d;  // Evitar  debería escribirse  c = 3; c = c + 4 + d; |
| Paréntesis | Es una buena práctica el uso de paréntesis en expresiones que incluyan distintos tipos de operadores para evitar problemas de precedencia de operadores. Aunque la precedencia de operadores nos pueda parecer clara, debemos asumir que otros programadores no tengan un conocimiento exhaustivo sobre las reglas de precedencia.  Ejemplo:  if (w == x && y == z)      // INCORRECTO if ((w == x) && (y == z)) // CORRECTO |
| Valores de retorno | Los valores de retorno tendrán que ser simples y comprensibles, de acuerdo al propósito y comportamiento del objeto en el que se utilicen.  // INCORRECTO public boolean esProgramador(Empleado emp) {  if (emp.getRol().equals(ROL\_PROGRAMADOR)) {   return true;  } else {   return false;  } }  // CORRECTO public boolean esProgramador(Empleado emp) {   boolean esUnProgramador = false;   if (emp.getRol().equals(ROL\_PROGRAMADOR)) {   esUnProgramador = true;  }   return esUnProgramador; } |
| Expresiones en el operador condicional ternario | Toda expresión compuesta, por uno o más operadores binarios, situada en la parte condicional del operador ternario deberá ir entre paréntesis.  Ejemplo: (x >= y) ? x : y; |