**[1.](#h.gjdgxs)****[Introducción](#h.gjdgxs)**

**[2.](#h.30j0zll)****[Idioma](#h.30j0zll)**

[2.1 Código](#h.1fob9te)

[2.2 Documentación](#h.3znysh7)

[2.3 Why Yungay?](#h.2et92p0)

**[3.](#h.tyjcwt)****[Configuración de entorno](#h.tyjcwt)**

[3.1](#h.3dy6vkm) *[Encoding](#h.3dy6vkm)* [y caracteres especiales](#h.3dy6vkm)

[3.2](#h.1t3h5sf) *[Packages](#h.1t3h5sf)*

[3.3 Nuevas clases y](#h.4d34og8) *[templates](#h.4d34og8)*

**[4.](#h.2s8eyo1)****[Variables](#h.2s8eyo1)**

[4.1 Caracteres permitidos](#h.17dp8vu)

[4.2 Prefijos para objetos de Java](#h.3rdcrjn)

[4.3 Prefijos para variables primitivas](#h.26in1rg)

[4.4 Prefijos para objetos propietarios](#h.lnxbz9)

[4.5 Prefijos para interfaces](#h.35nkun2)

[4.6 Prefijos para a](#h.1ksv4uv)*[rrays](#h.1ksv4uv)*

[4.7 Variables locales](#h.2jxsxqh)

4.8 Variables miembros de la clase

4.9 Variables clase base

[4.10 Variables estáticas](#h.z337ya)

[4.11 Variables en](#h.3j2qqm3) *[loops](#h.3j2qqm3)* [e](#h.3j2qqm3) *[iterations](#h.3j2qqm3)*

**[5.](#h.1y810tw)****[Buenas prácticas de programación](#h.1y810tw)**

[5.1 Variables](#h.4i7ojhp)

[5.2 Constantes](#h.2xcytpi)

[5.3 Paréntesis](#h.1ci93xb)

[5.4 Retorno de Valores](#h.3whwml4)

[5.5 Operador ternario '? :'](#h.2bn6wsx)

[5.6 Clases como parámetros de entrada](#h.qsh70q)

**[6.](#h.1pxezwc)****[Indentación](#h.1pxezwc)**

[6.1 Llaves](#h.49x2ik5)

[6.2 Espacios](#h.2p2csry)

[6.3 Declaraciones extensas](#h.3o7alnk)

**[7.](#h.23ckvvd)****[Logs](#h.23ckvvd)**

[7.1 Log de Métodos](#h.ihv636)

[7.2 Log de Métodos sobrecargados](#h.32hioqz)

[7.3 Log de Clases y Métodos](#h.1hmsyys)

[7.4 Log de Excepciones](#h.41mghml)

[7.5 Log de](#h.2grqrue) *[Threads](#h.2grqrue)*

[7.6 Log de Input y Output](#h.vx1227)

[7.7 Log de caracteres no visibles](#h.3fwokq0)

**[8.](#h.4f1mdlm)****[Comentarios](#h.4f1mdlm)**

[8.1 Caracteres permitidos](#h.2u6wntf)

[8.2 Pendientes y Código de prueba](#h.3tbugp1)

# Introducción

El presente estándar de programación surge en base a la necesidad de resolver diversos inconvenientes detectados durante el desarrollo de los proyectos:

* Asegurar que el código este bien escrito y contenga menos errores.
* Mejorar la comprensión del código, incluso si fue desarrollado por otra persona.
* Facilitar el mantenimiento del código, ya que será mantenido por diversos programadores.

Según las estadísticas, el 80% del tiempo invertido en la programación es destinado a su corrección y mantenimiento durante la fase de desarrollo y posterior entrega al cliente. Al estandarizar el estilo de programación se logra agilizar el mantenimiento y se asegura la entrega de un producto más estable y robusto.

# Idioma

## Código

Todo el código fuente de un proyecto debe ser desarrollado en ingles:

* Constantes, variables, métodos, clases, *packages.*
* *Keys*, atributos, parámetros.
* *Resources* (directorios del proyecto, imágenes, sonidos, properties, datos binarios, etc.).

## Documentación

Toda documentación del código fuente debe ser realizada en Javadoc:

* La documentación del código fuente entregado al cliente se redacta en castellano.
* La documentación del framework MyCore™ para celulares se redacta en ingles.

## Why Yungay?

Existen diversas razones positivas para programar en ingles:

* La lectura del código es más ágil ya que los keywords del lenguaje están en ingles.
* Es más fácil compartir código con otros programadores extranjeros.
* Es más fácil que otros programadores ayuden en fórums, grupos de correo, etc.
* La jerga técnica en inglés es más precisa y expresiva, describiendo mejor un problema.
* La mejor información técnica se consigue en documentos, fórums y web sites en ingles.

# Configuración de entorno

## *Encoding* y caracteres especiales

Para evitar problemas al cambiar de plataforma o manipularlos con determinadas herramientas, el *encoding* de todos los archivos en el proyecto debe ser **UTF-8** y sin BOM (*Byte Order Mark*).

## *Packages*

Todos los *packages* deben ser definidos en minúscula, con caracteres ASCII entre las letras **a - z** y con el siguiente formato:

pe.tmd.*projectname.yourpackage*

## Nuevas clases y *templates*

Cuando se crea una nueva clase, debe auto-generarse al inicio de la misma un mensaje de *copyright* parecido al siguiente:

/\*\*  
 \*  
 \* @class NewClass.java  
 \* @date Apr 09, 2011 – 12:26 AM  
 \* @author [NOMBRE USUARIO]  
 \*  
 \* (C) XXXXX S.A.C. All Rights Reserved. http://www.xxxxx.com/  
 \* Any redistribution or reproduction of part or all of this source code in any   
 \* form, including binary form, is prohibited. Contact owner for permissions.  
 \*/

La presentación puede diferir según el tipo de clase creada pero el *copyright notice* debe coincidir.

La descripción del contenido o propósito de la clase debe escribirse antes del *tag* @class.

# Variables

Las variables deben usar un prefijo para indicar su tipo, salvo cuando se indique lo contrario. Los prefijos establecidos en este documento son una variante de la *[notación húngara](http://en.wikipedia.org/wiki/Hungarian_notation)*, la cual propone ventajas como:

* El uso múltiples variables con similar semántica en el mismo bloque de código, ej. bytWidth, intWidth, dblWidth,
* Interpretar rápidamente el tipo de variable por su propio nombre.
* Evidenciar la asignación errónea de valores entre variables de distintos tipos.

## Caracteres permitidos

Los nombres de las variables, constantes, objetos y recursos deben contener únicamente los siguientes caracteres ASCII permitidos en programación: **a - z**, **A - Z**, **0 - 9**, ***underscore***.

* No deben contener el símbolo dólar “$” ni iniciar o finalizar con *underscore*

private byte \_bytConexión; // ☹ NO se permite underscore y tilde

private String strCompañia$; // ☹ NO se permite letra ñ y signo $

## Prefijos para objetos de Java

Los objetos más comunes en Java deben ser instanciados con los siguientes prefijos:

|  |  |  |
| --- | --- | --- |
| Tipo | Prefijo | Ejemplo |
| String | str | strName |
| List | lst | lstClient |
| Object | obj | objExample |
| Map | map | mapValues |
| InputStream | is | isConnection |
| OutputStream | os | osConnection |
| ByteArrayInputStream | bais | baisInput |
| ByteArrayOutputStream | baos | baosOutput |
| EntityManager | em | emConnection |
| FileInputStream | fis | fisExampleFile |
| Vector | lst | lstClient |
| HashMap | hm | hmOrders |
| HashTable | ht | htOrders |

## Prefijos para variables primitivas

Las variables primitivas deben usar los prefijos establecidos para indicar su tipo:

|  |  |  |
| --- | --- | --- |
| Tipo | Prefijo | Ejemplo |
| boolean | bool | isMenuShow |
| integer | int | intProductQuantity |
| long | lng | lngProductQuantity |
| Float | flt | fltValue |
| double | dbl | dblProductDiscount |
| char | chr | chrValue |
| short | sht | shtValue |
| byte | byt | bytValue |

## Prefijos para objetos propietarios

Se usara el nombre completo en camelCase. En caso que sea un componente se sufijara el nombre del componente.

## Sufijos según paquetes

Para la creación de las clases en un paquete se deberá sufijar el nombre del paquete al nombre de la clase, a excepción del paquete Model.

|  |  |
| --- | --- |
| Paquete | Ejemplo |
| activity | CalendarActivity |
| screen | MainScreen |
| home | ChannelHome |
| servlet | ChannelServlet |
| controller | ChannelController |
| security | UserSecurity |
| template | ChannelTemplate |

## Prefijos para interfaces

Los nombres de las interfaces deben tener como sufijo “**IF**”.

Los nombres de las clases que implementan una Interface deben tener como sufijo “**Impl**”.

public interface DepredableIF { ... }

public class DuckImpl implements DepredableIF { ... }

DepredableIF duck = new DuckImpl();

## Prefijos para a*rrays*

Los arreglos se declaran agregando la letra “**a**” al prefijo ya establecido para su nombre.

private Client[] arrClientOrders; // array-client de pedidos

private String[] arrStrProductNames; // array-string de nombre de productos

prívate byte[] arrBytSyncFlags; // array-byte de flags de sincronización

## Variables locales.

* Mantener la misma indentación en un grupo de variables, para mejor visibilidad del código.

.

## Variables estáticas

Además de las características definidas para las [variables locales,](#id.44sinio) se aplican las siguientes reglas:

* Usar el prefijo "**s\_**" para identificarlas como variables estáticas.
* Agrupar con un comentario todas las declaraciones según su funcionalidad.

## Variables en *loops* e *iterations*

Los variables usadas como contadores de loops deben ser de una sola letra en minúscula y sin prefijo, empezando por las letras **i**, **j**, **k**, **m**, **n**... caso contrario nombres cortos que indiquen su propósito (ej.: “col” y “row” para indicar columna y fila, “x” y “y”” para indicar coordenadas, “h” y “w” para indicar altura y ancho).

No usar la letra “**l**” (ele minúscula) a fin de evitar confusión con el numero “1” ya que, dependiendo del *font* usado, ambos caracteres pueden verse similares.



Esta regla se aplica también a las variables usadas con Iteration:

for( Iterator i = suits.iterator(); i.hasNext(); ) {

for( Iterator j = ranks.iterator(); j.hasNext(); ) {

...

# Buenas prácticas de programación

## Variables

En lo posible se deben usar variables locales para minimizar problemas de concurrencia.

No asignar más de una variable en una misma sentencia ya que es tarea del compilador optimizar el código para un mejor rendimiento:

d = (a = b + c) + r; // ☹ NO

a = ( b + c ); // ☺ SI

d = ( a + r );

## Constantes

Usar siempre constantes para los valores fijos. El nombre de una constante debe ser escrito en mayúscula y usar el carácter *underscore* para separar palabras, más no al inicio o fin del mismo:

private static final STATE\_ACTIVE = 1; // ☺ SI

if( c == 1 ) { ... } // ☹ NO

if( c == STATE\_ACTIVE ) { ... } // ☺ SI

## Paréntesis

Usar paréntesis para definir la precedencia:

if( a == b && c == d || e == f ) { ... } // ☹ NO

if( (a == b && c == d) || (e == f) ) { ... } // ☺ SI

## Retorno de Valores

Evitar “*return*” dentro de condiciones simples y dentro de bloques *try-catch.*

Ejemplos 1:

if( blCondition ) { // ☹ NO

return true;

} else {

return false;

}

return blCondition; // ☺ SI

Ejemplo 2:

if( blCondition ) { // ☹ NO

return x;

} else {

return Y;

}

return ( blCondition ? x : y); // ☺ SI

Ejemplo 3:

try{

return 666; // ☹ NO

} catch {

return 667; // ☹ NO

}

Try {

x = 666;

} catch {

x = 667;

}

return x; // ☺ SI

## Operador ternario '? :'

Evitar utilizar múltiple expresiones condicionales dentro de la misma. Es preferible segmentar toda la operación en varias líneas para una mejor comprensión y fácil mantenimiento.

La condición debe ir siempre entre paréntesis:

x >=0 ? x : -x; // ☹ NO

( x >=0 ) ? x : -x; // ☺ SI

## Clases como parámetros de entrada

Tratar de minimizar la cantidad de parámetros de entrada de un método. Si por ejemplo el método updateClient() tiene 2 parámetros de entrada (sName, sLastName) y más adelante hay que agregar un tercer parámetro (sEmail), será necesario cambiar en el proyecto todas las llamadas al método. En cambio, al pasar una clase Client solo habrá que agregar el nuevo campo dentro de la clase sin alterar las llamadas al método.

public void updateClient( String sName, String sLastName, String sEmail )// ☹ NO

public void updateClient( Client client ) // ☺ SI

Nota: esta regla no se aconseja en programación JME ya que incrementa el consumo de memoria.

# Indentación

El objetivo de indentar el código fuente es agilizar su lectura y comprensión aprovechando los “patrones de lecturas”. Cuando se lee un texto la vista no se desliza de manera continua, lo hace saltando porciones de texto y captando información alrededor del punto enfocado. Los programadores con experiencia focalizan menos texto e interpretan más código justamente porque abarcan un mayor campo visual y saben que determinados elementos en el código fuente se sitúan en posiciones pre-establecidas.



El éxito de una indentación radica en que la ubicación de los textos siga un patrón definido permitiendo a los ojos llegar rápidamente los puntos más importantes en el código fuente.

## Llaves

Es recomendable el uso de llaves e indentación de acuerdo a los siguientes ejemplos, quedando a criterio el uso de una nueva línea después de abrir una llave.

Ejemplos:

if( var1 == var2 ) {

tmp1 = 666;

tmp2 = 667;

} else if( var3 == var4 ) {

tmp1 = 668;

tmp2 = 669;

}

if( var1 == var2 ) { tmp1 = 666; tmp2 = 667; }

else if(var3 == var4 ) { tmp1 = 667; tmp2 = 668; }

if( var1 == var2 ) tmp1 = 666;

else if( var1 == var3 ) tmp1 = 667;

else if( var1 == var4 ) tmp1 = 668;

if( var1 == var2 ) { tmp1 = 666; tmp2 = 667; }

if( var3 == var4 ) { tmp1 = 668; tmp2 = 669; }

if( var5 == var6 ) { tmp1 = 670; tmp2 = 671; }

if( var1 == var2 ) tmp1 = 666;

if( var3 == var4 ) tmp1 = 667;

if( var5 == var6 ) tmp1 = 668;

## Espacios

Separar los argumentos de la llamada al método agregando un espacio después del paréntesis izquierdo y antes del paréntesis derecho:

String.valueOf( fNumber ).indexOf( “666” ); // ☺ SI

En el caso de *casting,* no hay necesidad de agregar espacios entre los paréntesis:

(String)result.getObject(); // ☺ SI

En las sentencias con operadores, agregar un espacio antes y después de cada operador:

nTotal = ( iResult1 + iResult2 + iResult3 ); // ☺ SI

En el caso de operadores unarios, no usar separación con un espacio:

nCount++; // ☺ SI

## Declaraciones extensas

Cuando la declaración de una variable exceda considerablemente la longitud de la pantalla, se debe separar en dos o más líneas el tipo de variable del nombre y del valor de inicialización:

private Thread              m\_thread;

private boolean             m\_blRunning;

private SenderProcessUDP    m\_spUDP;

private CountryBean         m\_countryBean;

private LinkedHashMap< Integer,PacketBean>

m\_hmSendQueue = new LinkedHashMap< Integer,PacketBean >();

private HashMap< Integer, LinkedHashMap< Integer,PacketBean > >

m\_hmSendQueueMMID = new HashMap< Integer, LinkedHashMap< Integer,PacketBean > >();

private HashMap< Integer, LinkedHashMap< Integer,PacketBean > >

m\_hmWaitAckQueueMMID = new HashMap< Integer,LinkedHashMap< Integer,PacketBean > >();

# Logs

Si bien la tendencia es utilizar la misma librería en todos los proyectos, es posible que en algunos no sea así. Es por lo tanto conveniente que el formato de los datos registrados no dependa de la librería usada. Estandarizando el formato se agiliza el análisis, el seguimiento de cada proceso y eventual procesamiento con apropiados tools.

La estandarización y la menor dependencia son parte de las buenas prácticas en programación ya que permiten la automatización de procesos y autonomía.



## Log de Métodos

Dependiendo de la necesidad y nivel de criticidad de un método, hay que *loggear* la entrada y salida al método con el texto “**[*method*] << ENTER**” y “**[*method*] >> EXIT**” (notar los espacios antes y después de los caracteres **<<** y **>>**).

Ejemplo:

public int nextFocus()

{

m\_log.info( “[nextFocus] << ENTER” );

int iRes = STATE\_NONE; // asignacion permitida por ser la única

int iType = -1; // tipo del control

iType = getType();

switch( iType ) {

case CTRL\_EDIT: iRes = STATE\_FOUND; break;

case CTRL\_SOFTMENU: iRes = STATE\_SKIP; break;

default: iRes = STATE\_ERROR; break;

}

m\_log.info( “[nextFocus] >> EXIT” );

return iRes;

}

Nota: aplicando el criterio indicado en el punto [Uso de llaves,](#id.3as4poj) en el ejemplo se optó por programar en el bloque switch-case cada alternativa en una sola línea.

No todos los metodos deben registrar la entrada y salida. El programador debe determinar si sera util y necesario hacerlo. Recordar que poca o excesiva informacion en un log solo entorpece el trabajo de mantenimiento, hay que registrar solo información útil.



## Log de Métodos sobrecargados

Para diferenciar la invocación de métodos sobrecargados, agregar al nombre del método *loggeado* un identificador único entre paréntesis, creado de acuerdo a las siguientes alternativas:

* Abreviando el tipo de parámetro (1, 2 o máximo 3 letras)
* Indicando la cantidad de parámetros (format #p, ej. 5p = 5 parametros)

Ejemplos:

public void dummyMethod( Client client, Product prod ) {

m\_log.info( "[dummyMethod(c,p)] << ENTER ); //

m\_log.info( "[dummyMethod(2p)] << ENTER ); // ☺ SI, unico con 2 parametros

}

public void dummyMethod( Client client, Product prod, int iValue) {

m\_log.info( "[dummyMethod(c,p,i)] << ENTER );

m\_log.info( "[dummyMethod(3p)] << ENTER ); // ☺ SI, unico con 3 parametros

}

public void dummyMethod( Client cli, Product pro, Integer iCode) {

m\_log.info( "[dummyMethod(c,p,I)] << ENTER );

m\_log.info( "[dummyMethod(3p)] << ENTER ); // ☹ NO, ya existe con 3 parametros

}

Recordar que el objetivo es generar un identificador unico que facilite la identificacion del metodo sobrecargado. No especificar los parametros



## Log de Clases y Métodos

Si existe más de una clase con métodos con nombre similar, hay que loggear la entrada y salida al método con el texto “**[*class*/*method*] << ENTER**” y “**[*class*/*method*] >> EXIT**” para indicar a que clase pertenece.

Ejemplo:

public class CtrlComboBox extends Ctrl

{

public int nextFocus()

{

m\_log.info( “[CtrlComboBox/nextFocus] << ENTER” );

...

m\_log.info( “[CtrlComboBox/nextFocus] >> EXIT” );

return iRes;

}

}

## Log de Excepciones

Hay que agregar el prefijo “**\*EX** “ a la excepción para agilizar su ubicación en el log.

Ejemplo 1:

public void nextFocus ()

{

Log.info(“[CtrlComboBox/nextFocus] << ENTER”);

try {

...

} catch ( NullPointerException ex ) {

m\_log.error( “\*EX " + ex );

}

...

Log.info(“[CtrlComboBox/nextFocus] >> EXIT”);

}

Ejemplo 2:

public void nextFocus ()

{

...

try {

...

} catch ( NullPointerException ex ) {

m\_log.error( “[CtrlComboBox/nextFocus] \*EX " + ex );

}

...

}

En el Ejemplo 1 se muestra como loggear una excepción en un método que registra su entrada y salida usando solo el prefijo establecido.

En el Ejemplo 2, el método no necesita registrar la entrada y salida por lo que es necesario especificar la clase y método antes del prefijo establecido.

Tener en cuenta que en procesos concurrentes es necesario incluso identificar el nombre del *thread*. Ver la seccion [Log de Threads.](#id.1v1yuxt)



## Log de *Threads*

Normalmente es suficiente loggear únicamente el nombre del método pero si existen procesos concurrentes hay que loggear la entrada y salida al método con el texto “**[*thread*/*class*/*method*] << ENTER**” y “**[*thread*/*class*/*method*] >> EXIT**“.

Ejemplo:

m\_log.info( "["+Thread.currentThread().getName()+"/GRT/lockUpdate] << ENTER" );

También se aconseja, según el caso, crear *threads* con nombre propio para identificarlos rápidamente. En este caso es necesario agregar el prefijo “**Thread-**“ al nombre del *thread* para mantener el estándar de Java:

m\_thread = new Thread( this, "Thread-GRT" ); // Graphx Rendered Thread

m\_thread.start();

m\_log.info( "["+Thread.currentThread().getName()+"/GRT/lockUpdate] << ENTER" );

Esta regla aplica a todos los casos de Log. Queda a criterio del programador saber combinarlos, recordando que debe encontrar un balance entre lo práctico y lo efectivo.



## Log de Input y Output

Indicar la dirección de la transferencia anteponiendo los caracteres “**<-**“ (*input*) y “**->**” (*output*) a los datos que se desean visualizar en el log. Notar que se pone mucho énfasis en la legibilidad.

* El bloque de código que realiza el envío y recepción de datos debe contener únicamente la secuencia de métodos de I/O para reconocer a primera vista el protocolo de transmisión correspondiente, agilizándose así el mantenimiento.
* Inicializar las variables correspondientes con valores fácilmente identificables como negativos o incompletos y que difieran de los reales esperados.
* El texto a visualizar debe indicar el nombre real de la variable.

Ejemplo 1 (situación normal):



Buena legibilidad y de producirse un error es fácilmente ubicable en el log.

try {

// entrada de datos del producto // ☺ SI

ProductCode = dis.readInt(); m\_log.info( "<- iProductCode: " + iProductCode );

sProductName = dis.readUTF(); m\_log.info( "<- sProductName: " + sProductName );

dProductPrice = dis.readDouble(); m\_log.info( "<- dProductPrice: " + dProductPrice );

dProductDisc = dis.readDouble(); m\_log.info( "<- dProductDisc: " + dProductDisc );

} catch ( Exception ex ) {

Log.error( "[method] " + ex );

}

Ejemplo 2 (alternativa con pocos datos transferidos o protocolo condicionado):



Buena legibilidad y de producirse un error es fácilmente ubicable en el log.

try {

// recibe Codigo de Control del servlet // ☺ SI

bCodeCtrl = dis.readByte();

m\_log.info( "<- bCodeCtrl: " + bCodeCtrl );

// recibe indicador Tipo (0=Data, 1=Error)

bSyncType = dis.readByte();

m\_log.info( "<- bSyncType: " + bSyncType );

} catch ( Exception ex ) {

Log.error( "[method] " + ex );s

}

Ejemplo 3:



Pésima legibilidad del código y se complica la interpretación del protocolo.

try {

// entrada de datos del producto

iProductCode = dis.readInt();

m\_log.info( "<- iProductCode: " + iProductCode ); // ☹ NO

sProductName = dis.readUTF();

m\_log.info( "<- sProductName: " + sProductName );

sProductPrice = dis.readUTF();

m\_log.info( "<- dProductPrice: " + dProductPrice );

} catch ( Exception ex ) {

m\_log.error( "[method] " + ex );

}

Ejemplo 3:



Buena legibilidad del código pero al ocurrir un error el log no se registra.

try {

// entrada de datos del producto

IProductCode = dis.readInt();

sProductName = dis.readUTF();

sProductPrice = dis.readUTF();

m\_log.info( "<- iProductCode: " + iProductCode ); // ☹ NO

m\_log.info( "<- sProductName: " + sProductName );

m\_log.info( "<- dProductPrice: " + dProductPrice );

} catch ( Exception ex ) {

m\_log.error( "[method] " + ex );

## Log de caracteres no visibles

En algunos casos puede ser útil marcar el inicio y el fin de un texto a ser loggeado, especialmente si este puede contener caracteres no visibles (ejemplo: BEL, TAB, LF, CR, espacio, etc.). Para tal fin, agregar “\*\*\*” (3 asteriscos) antes y después de loggear el contenido de una variable.

Ejemplo:

m\_log.info( “<- sToken: \*\*\*" + sToken + “\*\*\*” );

# Comentarios

## Caracteres permitidos

Los comentarios deben redactarse únicamente con caracteres alfanuméricos y símbolos visibles del ASCII 7-bit: **a – z**, **A – Z**, **0 – 9, *underscore*, *dollar*, *pipe*, etc.**

Para maximizar la compatibilidad entre plataformas y utilitarios, evitar el uso de **tildes** y **ñ**:

private byte bConnection; // flag de la conexión ☹ NO

private String sCompany; // nombre de la compañía ☹ NO

## Pendientes y Código de prueba

Iniciar un comentario con uno de los siguientes *keywords* para indicar situaciones especiales en el código fuente:

* // @XXX el código es usado solo para pruebas durante el desarrollo y debe ser posteriormente comentado o removido antes del pase a producción.
* // @FIXME el código contiene errores y necesita ser corregido.
* // @TODO tareas pendientes por realizar.

Ejemplos:

// @FIXME getScreenWidth() no calcula bien el tamano de la pantalla

private int getScreenWidth() {

...

}

private int getScreenWidth() {

...

iWidth = 320; // @XXX hardcoded iWidth = 320

...

return iWidth;

}

IDE NetBeans:

* Presionar las teclas CTRL+6 para visualizar en la ventana “*Tasks*” todos los comentarios. Seleccionar “*Scope*” para indicar el alcance.
* Tratar de que el comentario no supere 80 caracteres por línea. Este límite es señalado por una línea vertical de color rosa.

NetBeans interpreta los *keywords* sin necesidad del prefijo @. Sin embargo la razón de su uso es para evitar *mismatches* cuando se realicen búsquedas con otros editores (ej. Notepad++). Esto aplica también para los pendientes anotados en XLSX, DOCX y TXT.

