Tema 13:
Convenciones al código

Fundamentos de Telemática

Índice

Convenciones de código para Java

¿Por qué tener convenciones de código?

• El 80% del coste del código de un programa va a su mantenimiento.

• Casi ningún software lo mantiene toda su vida el autor original.

• Las convenciones de código mejoran la lectura del software, permitiendo a _los ingenieros entender nuevo código mucho más rápido y más a fondo.

• Si distribuyes tu código fuente como un producto, necesitas asegurarte de que _está bien hecho y presentado como cualquier otro producto.

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

Recolección de reglas

- **Estilo**
- **Interfaces**
- **Depuración**
- **Pruebas**
- **Rendimiento**
- **Portabilidad**

Estilo (I)

- Emplee nombres descriptivos para variables globales, y nombre cortos para variables locales.
- Sea consistente.
- Emplee nombres activos para las funciones.
- Sea preciso.
- Utilice sangrías para mostrar la estructura.
- Utilice una forma natural para las expresiones.
- Emplee paréntesis para resolver la ambigüedad.
- Divida las expresiones complejas.
- Sea claro.
- Tenga cuidado con los efectos laterales.
- Use un estilo consistente para las sangrías y las llaves.
- Emplee convenciones para tener consistencia.
- Utilice else-if para decisiones múltiples.

Estilo (y II)

- Evite las macros de funciones.
- Ponga entre paréntesis el cuerpo y los argumentos de las macros
- Déle nombres a los números mágicos.
- Defina los números como constantes, no como macros.
- Utilice constantes de caracteres, no enteros.
- Use el lenguaje para calcular el tamaño de un objeto.
- No repita lo que ya es obvio.
- Comente las funciones y los datos globales.
- No comente el código malo, vuelva a escribirlo.
- No contradiga el código.
- Aclare, no confunda.

Interfaces

- Oculte los detalles de la implementación.
- No se asome a las espaldas del usuario.
- Haga la misma cosa de la misma forma en todas partes.
- Libere un recurso en la misma capa en que fue asignado.
- Detecte errores en un nivel bajo, manéjelos en un nivel alto.
- Use excepciones sólo para situaciones excepcionales.

Depuración

- Busque patrones familiares.
- Examine los cambios más recientes.
- No cometa el mismo error dos veces.
- Corríjalo ahora, no después.
- Obtenga una reconstrucción de la pila.
- Lea antes de teclear.
- Explíquele su código a alguien más.
- Haga que el error sea reproducible.
- Divida y conquistará.
- Estudie la numerología de los errores.
- Desplieque salidas para situar la búsqueda.
- Escriba código de autoverificación.
- Escriba un archivo de registro.
- Haga un dibujo.
- Emplee herramientas.
- Mantenga registros.

Pruebas

- Pruebe los límites del código.
- Pruebe las condiciones previas y posteriores.
- Utilice aserciones.
- Programe a la defensiva.
- Verifique las devoluciones con error.
- Pruebe incrementalmente.
- Pruebe las partes sencillas primero.
- Sepa cuáles salidas esperar.
- Verifique las propiedades de conservación.
- Compare implementaciones independientes.
- Mida la cobertura de las pruebas.
- Automatice las pruebas de regresión.
- Cree pruebas autocontenidas.

Rendimiento (I)

- Automatice las mediciones de tiempo.
- Utilice un generador de perfiles.
- Concéntrese en los puntos críticos.
- Haga un dibujo.
- Use un mejor algoritmo o estructura de datos.
- Active las optimizaciones del compilador.
- Afine el código.
- No optimice lo que no importa.
- Reúna las subexpresiones comunes.
- Reemplace operaciones costosas por baratas.
- Desdoble o elimine los ciclos.
- Almacene en la memoria caché los valores empleados frecuentemente.

Rendimiento (y II)

- Escriba un asignador de propósito general.
- Almacene en buffer las entradas y salidas.
- Maneje por separado los casos especiales.
- Precalcule los resultados.
- Use valores aproximados.
- Reescriba en un lenguaje de menor nivel.
- Ahorre espacio empleando el tipo de datos más pequeño posible.
- No guarde lo que pueda recalcular con facilidad.

Portabilidad

- Apéguese al estándar.
- Programe con la parte establecida del lenguaje.
- Tenga cuidado con los puntos problemáticos del lenguaje.
- Pruebe varios compiladores.
- Emplee bibliotecas estándar.
- Únicamente utilice las características disponibles en todas partes.
- Evite la compilación condicional.
- Coloque las dependencias del sistema en archivos separados.
- Oculte las dependencias del sistema tras las interfaces.
- Use texto para intercambio de datos.
- Utilice un orden de bytes fijo para el intercambio de datos.
- Cambie el nombre si cambia la especificación.
- Mantenga la compatibilidad con los datos y programas existentes.
- No suponga que es ASCII.
- No suponga que es inglés.

Nombres de ficheros

✓ **Extensiones de los ficheros**

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

✓ **Nombres comunes para ficheros**

Tipo de archivo	Extensión
`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.

Organización de los ficheros

➢ **Ficheros fuente Java**

Cada fichero fuente Java contiene una única clase o interfaz pública que debe ser la primera del fichero.

Los ficheros fuentes Java tienen la siguiente ordenación:

- Comentarios de comienzo.
- Sentencias **`package`** e **`import`**.
- Declaraciones de clases e interfaces.

Organización de los ficheros

✓ ***Comentarios de comienzo***

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

```
/*
 * Nombre de la clase
 *
 * Información de la versión
 *
 * Fecha
 *
 * Copyright
 */
```

✓ ***Sentencias* `package` e `import`**

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

```
package java.awt;
import java.util.*;
```

Organización de los ficheros

✓ ***Declaraciones de clases e interfaces***

Orden	Partes de la declaración de una clase o interfaz	Notas
1	Comentario de documentación de la clase o interfaz (`/**...*/`)	
2	Sentencia `class` o `interface`	
3	Comentario de implementación de la clase o interfaz si fuera necesario (`/*...*/`)	Este comentario debe contener cualquier información aplicable a toda la clase o interfaz que no era apropiada para estar en los comentarios de documentación de la clase o interfaz.
4	Variables de clase (`static`)	Primero las variables de clase `public`, después las `protected`, después las de nivel de paquete (sin modificador de acceso), y después las `private`.
5	Variables de instancia	Primero las `public`, después las `protected`, después las de nivel de paquete (sin modificador de acceso), y después las `private`.
6	Constructores	
7	Métodos	Estos métodos se deben agrupar por funcionalidad más que por visión o accesibilidad. Por ejemplo, un método de clase privado puede estar entre dos métodos públicos de instancia. El objetivo es hacer el código mas legible y comprensible.

Indentación

- **Se deben emplear cuatro espacios** como unidad de indentación.
  Los **tabuladores** deben ser exactamente cada **ocho espacios** (no cada cuatro).

- **Evitar las líneas de más de 80 caracteres**, ya que no son manejadas bien por muchas terminales y herramientas.

- **Ruptura de líneas largas**

  Cuando una expresión no entre en una línea, romperla de acuerdo con estos principios:

  - Romper después de una coma.
  - Romper antes de un operador.
  - Preferir roturas de alto nivel que de bajo nivel.
  - Alinear la nueva línea con el comienzo de la expresión al mismo nivel de la línea anterior.
  - Si las reglas anteriores llevan a código confuso o a código que se aglutina en el margen derecho, indentar justo 8 espacios en su lugar.

Indentación

Ruptura de la llamada a un método:

```
unMetodo(expresionLarga1, expresionLarga2,expresionLarga3,
        expresionLarga4, expresionLarga5);

var = unMetodo1(expresionLarga1,
             unMetodo2(expresionLarga2,
                     expresionLarga3));
```

Ruptura de líneas en expresiones aritméticas:

```
nombreLargo1 = nombreLargo2 * (nombreLargo3 + nombreLargo4 - nombreLargo5)
                  + 4 * nombreLargo6; //PREFERIDA

nombreLargo1 = nombreLargo2 * (nombreLargo3 + nombreLargo4
                                  - nombreLargo) + 4 * nombreLargo6; //EVITAR
```

Indentación en declaraciones de métodos:

```
//INDENTACION CONVENCIONAL
unMetodo(int unArg, Object otroArg, String todaviaOtroArg,
         Object yOtroMas) {
   ...
}

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

Indentación

Ruptura de líneas para sentencias `if` (8 espacios, ya que la indentación convencional de 4 espacios hace difícil ver el cuerpo):

```
//NO USAR ESTA INDENTACION
if ((condicion1 && condicion2)
    || (condicion3 && condicion4)
    ||!(condicion5 && condicion6)) { //MALOS SALTOS
    hacerAlgo(); //HACEN ESTA LINEA FACIL DE OLVIDAR
}

//USAR ESTA INDENTACION
if ((condicion1 && condicion2)
        || (condicion3 && condicion4)
        ||!(condicion5 && condicion6)) {
    hacerAlgo();
}

//O USAR ESTA
if ((condicion1 && condicion2) || (condicion3 && condicion4)
        ||!(condicion5 && condicion6)) {
    hacerAlgo();
}
```

Indentación

Ruptura de expresiones ternarias (todas válidas):

```
alpha = (unaLargaExpresionBooleana) ? beta : gamma;

alpha = (unaLargaExpresionBooleana) ? beta
                                    : gamma;

alpha = (unaLargaExpresionBooleana)
        ? beta
        : gamma;
```

Comentarios

Los programas Java pueden tener dos tipos de comentarios: comentarios de implementación y comentarios de documentación.

• Los comentarios de implementación son para comentar nuestro código o para comentarios acerca de una implementación particular. Se encuentran delimitados por /*...*/, y //.

• Los comentarios de documentación son para describir la especificación del código, y para ser leídos por desarrolladores que pueden no tener el código fuente a mano. Se limitan por /**...*/.

Comentarios de Implementación

Los programas pueden tener cuatro estilos de comentarios de implementación:

• de bloque

• de una línea

• de remolque

• de fin de línea

Comentarios de Implementación

Comentarios de bloque

Se usan al comienzo de cada fichero y antes de cada método o el interior de éstos para dar descripciones de ficheros, métodos, estructuras de datos y algoritmos.

Debe ir precedido por una línea en blanco. EJ:

```
/*
 * Aquí hay un comentario de bloque.
 */
```

Los comentarios de bloque pueden comenzar con `/*-`, como el comienzo de un comentario de bloque que no debe ser reformateado.
EJ:

```
/*-
 * Aquí tenemos un comentario de bloque con cierto
 * formato especial
 *
 *      uno
 *              dos
 *                      tres
 */
```

Comentarios de Implementación

Comentarios de línea

Pueden aparecer comentarios cortos de una única línea indentados al nivel del código que siguen.

Si un comentario no se puede escribir en una única línea, debe seguir el formato de los comentarios de bloque.

Un comentario de una sola línea debe ir precedido de una línea en blanco.

EJ:

```
if (condicion) {

        /* Código de la condicion. */
        ...
}
```

Comentarios de Implementación

Comentarios de remolque

Pueden aparecer comentarios muy pequeños en la misma línea que describen, pero deben ser movidos lo suficientemente lejos para separarlos de las sentencias.

Si más de un comentario corto aparece en el mismo trozo de código, deben ser indentados con la misma profundidad.

EJ:

```
if (a == 2) {
        return TRUE;           /* caso especial */
} else {
        return esPrimo(a);     /* caso cuando a es impar */
}
```

Comentarios de Implementación

Comentarios de fin de línea

El delimitador de comentario `//` puede convertir en comentario una línea completa o una parte de una línea.

No debe ser usado para hacer comentarios de varias líneas consecutivas.

Puede usarse en líneas consecutivas para comentar secciones de código.

EJ:

```
if (foo > 1) {
        // Hacer algo.
        ...
}
else {
        return false; // Explicar aquí por que.
}

//if (bar > 1) {
//
//      // Hacer algo.
//      ...
//}
//else {
//      return false;
//}
```

Comentarios de Documentación

Los comentarios de documentación describen clases Java, interfaces, constructores, métodos y atributos.

Cada comentario de documentación se encierra con los delimitadores de comentarios `/**...*/`:

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

La primera línea de un comentario de documentación (/**) para clases e interfaces no está indentada, sucesivas líneas tienen cada una un espacio de indentación (para alinear verticalmente los asteriscos).

Comentarios de Documentación

```
/**
 * Returns an Image object that can then be painted on the screen.
 * The url argument must specify an absolute {@link URL}. The name
 * argument is a specifier that is relative to the url argument.
 * <p>
 * This method always returns immediately, whether or not the
 * image exists. When this applet attempts to draw the image on
 * the screen, the data will be loaded. The graphics primitives
 * that draw the image will incrementally paint on the screen.
 *
 * @param url an absolute URL giving the base location of the image
 * @param name the location of the image, relative to the url argument
 * @return the image at the specified URL
 * @see Image
 */
public Image getImage(URL url, String name) {
    try {
        return getImage(new URL(url, name));
    } catch (MalformedURLException e) {
        return null; }
    }
```

Comentarios de Documentación

```
getImage

public Image getImage(URL  url, String  name)

                    Returns an Image object that can then be
                painted on the screen.
                    The url argument must specify an absolute URL.
                The name argument
                    is a specifier that is relative to the url
                argument. This method
                    always returns immediately, whether or not the
                image exists.
                    When this applet attempts to draw the image on
                the screen, the
                    data will be loaded. The graphics primitives
                that draw the image
                    will incrementally paint on the screen.

     Parameters:
        url - an absolute URL giving the base location of the image
        name - the location of the image, relative to the url argument
Returns:
        the image at the specified URL
See Also:
        Image
```

Comentarios de Documentación

```
Order of Block Tags
Include block tags in the following order:
```

* @param (classes, interfaces, methods and constructors only)
* @return (methods only)
* @exception (@throws is a synonym added in Javadoc 1.2)
* @author (classes and interfaces only, required)
* @version (classes and interfaces only, required. See footnote 1)
* @see
* @since
* @serial (or @serialField or @serialData)
* @deprecated (see How and When To Deprecate APIs)

Ejemplo

```
/**
 * Graphics is the abstract base class for all graphics contexts
 * which allow an application to draw onto components realized on
 * various devices or onto off-screen images.
 * A Graphics object encapsulates the state information needed
 * for the various rendering operations that Java supports.  This
 * state information includes:
 * <ul>
 * <li>The Component to draw on
 * <li>A translation origin for rendering and clipping coordinates
 * <li>The current clip
 * <li>The current color
 * <li>The current font
 * <li>The current logical pixel operation function (XOR or Paint)
 * <li>The current XOR alternation color
 *     (see <a href="#setXORMode">setXORMode</a>)
 * </ul>
 * <p>
 * Coordinates are infinitely thin and lie between the pixels of the
 * output device.
 * Operations which draw the outline of a figure operate by traversing
 * along the infinitely thin path with a pixel-sized pen that hangs
 * down and to the right of the anchor point on the path.
 * Operations which fill a figure operate by filling the interior
 * of the infinitely thin path.
 * Operations which render horizontal text render the ascending
 * portion of the characters entirely above the baseline coordinate.
 * <p>
 * Some important points to consider are that drawing a figure that
 * covers a given rectangle will occupy one extra row of pixels on
 * the right and bottom edges compared to filling a figure that is
 * bounded by that same rectangle.
 * Also, drawing a horizontal line along the same y coordinate as
 * the baseline of a line of text will draw the line entirely below
 * the text except for any descenders.
 * Both of these properties are due to the pen hanging down and to
 * the right from the path that it traverses.
 * <p>
 * All coordinates which appear as arguments to the methods of this
 * Graphics object are considered relative to the translation origin
 * of this Graphics object prior to the invocation of the method.
 * All rendering operations modify only pixels which lie within the
 * area bounded by both the current clip of the graphics context
 * and the extents of the Component used to create the Graphics object.
 *
 * @author      Sami Shaio
 * @author      Arthur van Hoff
 * @version     %I%, %G%
 * @since       1.0
 */
public abstract class Graphics {

    /**
     * Draws as much of the specified image as is currently available
     * with its northwest corner at the specified coordinate (x, y).
     * This method will return immediately in all cases, even if the
     * entire image has not yet been scaled, dithered and converted
     * for the current output device.
     * <p>
     * If the current output representation is not yet complete then
     * the method will return false and the indicated
     * {@link ImageObserver} object will be notified as the
     * conversion process progresses.
     *
     * @param img       the image to be drawn
     * @param x         the x-coordinate of the northwest corner
     *                  of the destination rectangle in pixels
     * @param y         the y-coordinate of the northwest corner
     *                  of the destination rectangle in pixels
     * @param observer  the image observer to be notified as more
     *                  of the image is converted.  May be
     *                  <code>null</code>
     * @return          <code>true</code> if the image is completely
     *                  loaded and was painted successfully;
     *                  <code>false</code> otherwise.
     * @see             Image
     * @see             ImageObserver
     * @since           1.0
     */
    public abstract boolean drawImage(Image img, int x, int y,
                                      ImageObserver observer);
```

Ejemplo

```
    /**
     * Dispose of the system resources used by this graphics context.
     * The Graphics context cannot be used after being disposed of.
     * While the finalization process of the garbage collector will
     * also dispose of the same system resources, due to the number
     * of Graphics objects that can be created in short time frames
     * it is preferable to manually free the associated resources
     * using this method rather than to rely on a finalization
     * process which may not happen for a long period of time.
     * <p>
     * Graphics objects which are provided as arguments to the paint
     * and update methods of Components are automatically disposed
     * by the system when those methods return.  Programmers should,
     * for efficiency, call the dispose method when finished using
     * a Graphics object only if it was created directly from a
     * Component or another Graphics object.
     *
     * @see       #create(int, int, int, int)
     * @see       #finalize()
     * @see       Component#getGraphics()
     * @see       Component#paint(Graphics)
     * @see       Component#update(Graphics)
     * @since     1.0
     */
    public abstract void dispose();

    /**
     * Disposes of this graphics context once it is no longer
     * referenced.
     *
     * @see       #dispose()
     * @since     1.0
     */
    public void finalize() {
                dispose();
    }
}
```

Declaraciones

➢ **Cantidad por línea**

Se recomienda una declaración por línea, ya que facilita los comentarios.

Se prefiere

```
int nivel; // nivel de indentación
int tam;   // tamaño de la tabla
```

antes que

```
int nivel, tam;
```

No poner diferentes tipos en la misma línea. Ejemplo:

```
int foo, fooarray[]; //ERROR!
```

Nota: Los ejemplos anteriores usan un espacio entre el tipo y el identificador. Una alternativa aceptable es usar tabuladores, por ejemplo:

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

• **Inicialización**

Inicializar las variables locales donde se declaran salvo si el valor inicial depende de algunos cálculos previos.

Declaraciones

➢ **Colocación**

Poner las declaraciones sólo al principio de los bloques.

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

La excepción de la regla son los índices de bucles `for`, que en Java se pueden declarar en la sentencia `for`:

```
for (int i = 0; i < maximoVueltas; i++) { ... }
```

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

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

Declaraciones

➢ **Declaraciones de clases e interfaces**

• Ningún espacio en blanco entre el nombre de un método y el paréntesis "(" que abre su lista de parámetros.

• La llave de apertura "{" aparece al final de la misma línea de la sentencia de declaración.

• La llave de cierre "}" empieza una nueva línea indentada para ajustarse a su sentencia de apertura correspondiente, excepto cuando no existen sentencias entre ambas, que debe aparecer inmediatamente después de la de apertura "{".

```
class Ejemplo extends Object {
      int ivar1;
      int ivar2;

      Ejemplo(int i, int j) {
             ivar1 = i;
             ivar2 = j;
      }

      int metodoVacio() {}
      ...
}
```

• Los métodos se separan con una línea en blanco.

Sentencias

- **Sentencias simples**
- **Sentencias compuestas**
- **Sentencias return**
- **Sentencias if, if-else, if else-if else**
- **Sentencias for**
- **Sentencias while**
- **Sentencias do-while**
- **Sentencias switch**
- **Sentencias try-catch**

Sentencias simples

Cada línea debe contener como máximo una sentencia. Ejemplo:

```
argv++;           // Correcto
argc--;           // Correcto
argv++; argc--;   // EVITAR!
```

Sentencias compuestas

Las sentencias compuestas son sentencias que contienen listas de sentencias encerradas entre llaves "`{ sentencias }`".

• Las sentencias encerradas deben indentarse un nivel más que la sentencia compuesta.

• La llave de apertura se debe poner al final de la línea que comienza la sentencia compuesta; la llave de cierre debe empezar una nueva línea y ser indentada al mismo nivel que el comienzo de la sentencia compuesta.

• Las llaves se usan en **TODAS** las sentencias, **incluso las simples**, cuando forman parte de una estructura de control, como en las sentencias `if-else` o `for`. Esto hace más sencillo añadir sentencias sin incluir errores accidentales por olvidar las llaves.

Sentencias return

Una sentencia `return` con un valor no debe usar paréntesis a menos que hagan el valor de retorno más obvio de alguna manera. Ejemplo:

```
return;

return miDiscoDuro.size();

return (tamanyo ? tamanyo : tamanyoPorDefecto);
```

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) {
        sentencias;
} else if (condicion) {
        sentencias;
} else{
        sentencias;
}
```

Nota: Las sentencias `if` usan siempre llaves {}. Evitar la siguiente forma, propensa a errores:

```
if (condicion) //EVITAR! ESTO OMITE LAS LLAVES {}!
       sentencia;
```

Sentencias for

Una sentencia `for` debe tener la siguiente forma:

```
for (inicializacion; condicion; actualizacion) {
       sentencias;
}
```

Una sentencia `for` vacía (una en la que todo el trabajo se hace en la inicialización, condición, y actualización) debe tener la siguiente forma:

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

Nota: Al usar el operador coma en la inicialización o actualización de una sentencia `for`, evitar usar más de tres variables. Si se necesita, usar sentencias separadas antes de bucle `for` (para la inicialización) o al final del bucle (para la actualización).

Sentencias while

Una sentencia `while` debe tener la siguiente forma:

```
while (condicion) {
       sentencias;
}
```

Una sentencia `while` vacía debe tener la siguiente forma:

```
while (condicion);
```

Sentencias do-while

Una sentencia `do-while` debe tener la siguiente forma:

```
do {
       sentencias;
} while (condicion);
```

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.

Cada sentencia `switch` debe incluir un caso por defecto. El `break` en el caso por defecto es redundante, pero prevé que se propague por error si luego se añade otro caso.

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;
}
```

Espacios en blanco

- **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 o de un comentario de una línea.
• Entre las distintas secciones lógicas de un método para facilitar la lectura.

Espacios en blanco

- **Espacios en blanco**

Se deben usar espacios en blanco en las siguientes circunstancias:

• Una palabra reservada del lenguaje seguida por un paréntesis.

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

• Después de cada coma en las listas de argumentos.

• Todos los operadores binarios excepto el punto “.”

• Los operadores unarios como el incremento (“++”) y el decremento ("--") **NO**.

```
a += c + d;
a = (a + b) / (c * d);
while (d++ == s++) {
      n++;
}
prints("el tamaño es " + foo + "\n");
```

Espacios en blanco

- **Espacios en blanco (continuación)**

• Las expresiones en una sentencia `for` se deben separar con espacios en blanco.

```
for (expr1; expr2; expr3)
```

• Los "cast" deben ir seguidos de un espacio en blanco. Ejemplos:

```
miMetodo((byte) unNumero, (Object) x);
miMetodo((int) (cp + 5), ((int) (i + 3))
                              + 1);
```

Convenios de nombrado

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

Tipo de Identificador	Reglas de nombrado	Ejemplos

Paquetes	El prefijo del nombre de un paquete se escribe siempre con letras ASCII en minúsculas, y debe ser uno de los nombres de dominio de alto nivel, actualmente com, edu, gov, mil, net, org, o uno de los códigos ingleses de dos letras que identifican cada país como se especifica en el ISO Standard 3166, 1981.
	
	Los sucesivos componentes del nombre del paquete variarán de acuerdo a las convenciones de nombres internas de cada organización. Dichas convenciones pueden especificar que algunos nombres de los directorios correspondan a divisiones, departamentos, proyectos, máquinas o nombres de usuarios.
		`com.sun.eng`
		
		`com.apple.quicktime.v2`
		
		`edu.cmu.cs.bovik.cheese`

Convenios de nombrado

Tipo de Identificador	Reglas de nombrado	Ejemplos

Clases	Los nombres de las clases deben ser sustantivos, cuando son compuestos tendrán la primera letra de cada palabra que lo forma en mayúsculas. Intentar mantener los nombres de las clases simples y descriptivos. Usar palabras completas, evitar acrónimos y abreviaturas (a no ser que la abreviatura sea mucho más conocida que el nombre completo, como URL o HTML).	`class Cliente;`
		
		`class ImagenAnimada;`

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

Métodos	Los métodos deben ser verbos, cuando son compuestos tendrán la primera letra en minúscula, y la primera letra de las siguientes palabras que lo forma en mayúscula.	`ejecutar();`
		
		`ejecutarRapido();`
		
		`obtenerFondo();`

Convenios de nombrado

Tipo de Identificador	Reglas de nombrado	Ejemplos

Variables	Empezarán con minúscula y las palabras internas que lo forman (si son compuestas) empiezan con su primera letra en mayúsculas. Los nombres de variables no deben empezar con los caracteres guión bajo "_" o signo del dólar "$", aunque ambos están permitidos por el lenguaje.
	
	Los nombres de las variables deben ser cortos pero con significado. La elección del nombre de una variable debe ser un mnemónico, designado para indicar a un observador casual su función. Los nombres de variables de un solo carácter se deben evitar, excepto para variables índices temporales. Nombres comunes para variables temporales son `i`, `j`, `k`, `m` y `n` para enteros; `c`, `d`, y `e` para caracteres.
		`int i;`
		
		`char c;`
		
		`float miAnchura;`

Convenios de nombrado

Tipo de Identificador	Reglas de nombrado	Ejemplos

Constantes	Los nombres de las variables declaradas como constantes de clase y las constantes ANSI deben ir totalmente en mayúsculas separando las palabras con un guión bajo ("_"). (Las constantes ANSI se deben evitar, para facilitar su depuración.)	`static final int MIN = 1;`
		
		`static final int MAX = 9;`
		
		`static final int NUM = 5;`

Hábitos de programación

- **Proporcionando acceso a variables de instancia y de clase**

No hacer ninguna variable de instancia o clase pública sin una buena razón.

Hábitos de programación

- **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!
```

- **Constantes**

Las constantes numéricas (literales) no deberían ser codificadas directamente, excepto -1, 0, y 1, que pueden aparecer en un bucle `for` como contadores.

Hábitos de programación

- **Asignaciones de variables**

Evitar asignar el mismo valor a varias variables en la misma sentencia, dificulta su lectura. Ejemplo:

```
fooBar.fChar = barFoo.Ichar = 'c'; // EVITAR!
```

No usar el operador de asignación en un lugar donde se pueda confundir con el de igualdad. Ejemplo:

```
if (c++ = d++) {              // EVITAR! (Java no lo permite)
...
}

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

No usar asignaciones incrustadas como un intento de mejorar el rendimiento en tiempo de ejecución. Ése es el trabajo del compilador. Ejemplo:

```
d = (a = b + c) + r; // EVITAR!

a = b + c;
d = a + r;
```

Hábitos de programación

- **Hábitos varios**

- ***Paréntesis***
- ***Valores de retorno***
- ***Expresiones antes de `?' en el operador condicional***
- ***Comentarios especiales***

Hábitos Varios

• *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
```

Hábitos Varios

• *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);
```

Hábitos Varios

• *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;
```

Hábitos Varios

• *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.

Ejemplo Final

```
/*
 * @(#)Bla.java 1.82 99/03/18
 *
 * Copyright (c) 1994-1999 Sun Microsystems, Inc.
 * 901 San Antonio Road, Palo Alto, California.
 * All rights reserved.
 *
 * Más información y descripción del Copyright.
 *
 */

package java.bla;

import java.bla.blabla.BlaBla;

/**
 * La descripción de la clase viene aquí.
 *
 * @version datos de la versión (numero y fecha)
 * @author Nombre Apellido
 */
public class Bla extends OtraClase {
    /* Comentario de implementación de la clase.*/

    /** 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 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...
    }
}
```