IES de Teis
Avda. de Galicia, 101
36216 – Vigo
886 12 04 64
ies.teis@edu.xunta.es

A Anexo I: Guía de estilo

Índice

1. Introducción

Hay muchas formas de programar, pero no todas son igualmente comprensibles. Por eso, especialmente en compañías grandes, se suele elaborar una guía de estilo de programación que recoge un conjunto de reglas de codificación: como separar código en ficheros y directorios, como elegir a nombres para variables y funciones, cómo alinear la sangría de un bloque etc.

Una guía de estilo de programación sirve para unificar la manera de crear código. Un código desconocido es más fácil de entender si las mismas cosas se hacen de la misma manera. Un programador nuevo en el proyecto tarda menos tiempo en entenderlo.

Así, las convenciones de código son importantes para los programadores por un gran número de razones:

- El 80% del coste del código de un programa va en su mantenimiento.
- Casi ningún software lo mantiene toda su vida el autor original.
- Las convenciones de código mejoran la lectura del software, permitiendo entender código nuevo mucho más rápidamente 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.

2. Guía de estilo

2.1 Organización de los ficheros

- Se deben evitar ficheros de más de 2000 líneas.
- Cada fichero contiene una ÚNICA clase o interfaz pública. Cuando algunas clases o interfaces privadas están asociadas a una clase pública, pueden ponerse en el mismo fichero que la clase pública. La clase o interfaz pública debe ser la primera clase o interface del fichero.
- Estructura del fichero de una clase: indica el orden en el que deben aparecer las distintas secciones que forman el fichero.
  - Si la clase forma parte de un paquete la sentencia `package` debe ser la primera del fichero.
  - Si se importan paquetes, la sentencia `import` debe aparecer después de la sentencia `package`.
  - Comentario de documentación de la clase o interface (/**... */)
  - Sentencia `class` o `interface`.
  - 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`.

- Variables de instancia: primero las `public`, después las `protected`, después las de nivel de paquete (sin modificador de acceso), y después las `private`.
- Constructores.
- Métodos: se deben agrupar por funcionalidad más que por visión o accesibilidad. Así, si un método público usa dos métodos privados, se debería colocar primero el público seguido de los dos privados. La idea es que al leer el código se siga con facilidad la funcionalidad de los métodos.

2.2 Identificadores

- Los identificadores deben ser elegidos de tal manera que el solo nombre describa el uso que se dará dentro del programa, por tanto no es recomendable usar identificadores de una letra, excepto en el `for`, ni abreviaturas raras o ambiguas.

```
integralIndefinida        //Bien
intInd                    //Mal
```

- Para clases e interfaces se debe escoger como identificador un sustantivo. Usar minúsculas excepto para la letra inicial. Si el identificador consta de varias palabras colocarlas juntas, con la inicial de cada una en mayúsculas. Ejemplos:

```
class Cliente
class ClientePreferencial
```

- Para las variables y objetos utilizar identificadores en minúsculas. Si el identificador consta de varias palabras la primera de ellas se escribirá en minúsculas y la inicial de las demás en mayúsculas. Los nombres de variables no deben empezar con los caracteres subguión "_" 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.

```
double valorFinal = 0.0;  //Bien
int CantidadFinal = 0;    //Mal
```

- Para las constantes, el identificador debe usarse en mayúsculas. Si el identificador consta de varias palabras se separan con el carácter de subrayado. Ejemplos correctos serían:

```
final int MINIMO=100;
final double PRECISION=0.001;
final double ALTURA_MEDIA=29.5;
```

- En el caso de los métodos, el identificador debe ser un verbo y debe usarse en minúsculas. Si el identificador contiene varias palabras, la inicial de todas las posteriores a la primera va en mayúsculas. Ejemplos válidos serían,

```
public void introducir(double valor){
 - - - cuerpo del método - - -
}
public double obtenerMedia(){
 - - - cuerpo del método - - -
}
```

- Para el identificador de los paquetes se recomienda usar minúsculas.

2.3 Alineación y espacios en blanco

- Se debe usar cuatro espacios como unidad de indentación (o el tabulador establecido de NetBeans).
- Evitar líneas de más de 80 caracteres.
- Todas las sentencias dentro un bloque deben ir alineadas a la misma altura.
- Sangrar cada nuevo bloque de sentencias.
- Cuando una expresión no entre en una línea, romperla de acuerdo a los siguientes principios:
  - Romper después de una coma.
  - Romper antes de un operador.
  - Alinear la nueva línea con el comienzo de la expresión al mismo nivel de la línea anterior.

```
nombreLargo1 = nombreLargo2 * (nombreLargo3
               + nombreLargo4 - nombreLargo5)
               + 4 * nombreLargo6; // PREFERIDA
nombreLargo1 = nombreLargo2 * (nombreLargo3
                              + nombreLargo4
                              - nombreLargo) + 4; //NO
```

  - Ejemplos de cómo romper la llamada de un método:

```
unMetodo(expresionLarga1, expresionLarga2,
         expresionLarga3, expresionLarga4);
var = unMetodo1(expresionLarga1,
                unMetodo2(expresionLarga2,
                          expresionLarga3));
```

- Se deben usar líneas en blanco en las siguientes circunstancias:
  - Usar una línea en blanco para separar partes del código que sean lógicamente distintas.
  - Usar una línea en blanco para separar métodos.
  - Usar una línea en blanco entre las variables locales de un método y su primera sentencia.
  - Usar una línea en blanco antes de un comentario de bloque o de un comentario de una línea.
- Se deben usar espacios en blanco en las siguientes circunstancias:
  - Una palabra clave del lenguaje seguida por un paréntesis debe separarse por un espacio.

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

  - Debe aparecer un espacio en blanco después de cada coma en las listas de argumentos.

```
miMetodo(arg1, arg2, arg3);
```

  - Todos los operadores binarios excepto "." se deben separar de sus operandos con espacios en blanco. Los espacios en blanco no deben separar los operadores unarios, incremento ("++") y decremento ("--") de sus operandos.

```
a = c + d;
a = (a + b) / (c * d);

while (d++ == s++) {
    n++;
}

printSize("el tamaño es " + foo + "\n");
```

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

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

  - Los "Cast"s deben ir seguidos de un espacio en blanco.

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

2.4 Variables y constantes

- Se recomienda definir una variable por línea, ya que facilita los comentarios.

```
int horas = 0, minutos = 0; //Mal si hay que documentar
int horas = 0;              //Bien
int minutos = 0;            //Bien
```

- No poner diferentes tipos en la misma línea:

```
int foo, fooarray[];      //Mal
```

- Declarar las variables al principio del bloque en el que se vayan a utilizar. En los métodos, declarar las variables al principio del método. Intente inicializar todas las variables que se declaren. La única excepción serán los índices de bucles `for`, que en Java se pueden declarar en la sentencia `for`.

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

2.5 Sentencias de control

- No usar saltos incondicionales. No usar la sentencia `break` excepto en la sentencia `switch`, donde es forzoso hacerlo.
- No usar la sentencia `continue`.
- La palabra clave de la sentencia de control irá separada de un espacio respecto de los paréntesis ().
- La llave de apertura de bloque ({) deberá ir separada con un espacio al final de la línea inicial de la sentencia.
- La llave de fin de bloque (}) se colocará en la línea siguiente a la finalización del bloque.
- Poner cada nuevo bloque de sentencias entre llaves aunque conste de una sola sentencia.
- La sintaxis para sentencias `if` debe tener la siguiente forma:

```
if (condicion) {
    sentencias;
}

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

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

- La sintaxis para sentencias `for` debe tener la siguiente forma:

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

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

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

- La sintaxis para sentencias `while` debe tener la siguiente forma:

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

- La sintaxis para sentencias `do-while` debe tener la siguiente forma:

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

- La sintaxis para sentencias `switch` debe tener la siguiente forma:

```
switch (condicion) {
    case A:
        sentencias;
        /* este caso se propaga */
    case B:
        sentencias;
        break;
```

```
    case C:
        sentencias;
        break;
    default:
        sentencias;
    break;
}
```

- La sintaxis para sentencias `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 haya completado con éxito o no.

```
try {
    sentencias;
} catch (ExceptionClass e) {
    sentencias;
} finally {
    sentencias;
}
```

2.6 Métodos

- Todo método excepto `main` debe empezar con un comentario en formato Javadoc.
- El propósito del método debe ser único.
- No dejar espacio en blanco entre el identificador del método y el paréntesis "(" que abre su lista de parámetros.

```
public void nombreMétodo(int valor1, double valor2) //OK
public void nombreMétodo (int valor1, double valor2)//NO
```

- La llave de apertura "{" aparece al final de la misma línea de la sentencia de declaración después de un espacio en blanco.

- La llave de cierre "}" empieza en 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() {}
    ...
}
```

- No usar más de un `return` en cada método y colocarlo al final del método.
- Después de cada método se debe dejar una línea en blanco.

2.7 Modificadores de acceso

El orden de los modificadores debe seguir la especificación del lenguaje Java :

- Los modificadores de acceso las clases deben seguir este orden:

```
public protected private abstract static final strictfp
```

- Los modificadores de acceso de los campos deben seguir este orden:

```
public protected private static final transient volatile
```

- Los modificadores de acceso de los métodos deben seguir este orden:

```
public protected private abstract static final synchro-
nized native strictfp
```

Java Language Specification: http://docs.oracle.com/javase/specs/

2.8 Documentación y comentarios

- Se debe documentar todo el código.
- Se deben incluir comentarios Javadoc para la cabecera de una clase, donde se incluirá la descripción de la clase y el autor.

- Se deben incluir comentarios Javadoc para la cabecera de los métodos, donde se incluirá el propósito del método, el significado de los parámetros, el significado de la información devuelta con `return`, las excepciones y la fecha de la última modificación.
- En el cuerpo de los métodos deben añadirse comentarios para indicar el propósito de tareas que no sean evidentes.
- En los comentarios de implementación de varias líneas el cierre del comentario se hará en una nueva línea:

```
/*Este sería un comentario de implementación que ocupa
más de una línea de código.
*/
```

3. Hábitos de programación

- No hacer ninguna variable de instancia o clase pública sin una buena razón.
- Evitar usar un objeto para acceder a una variable o método de clase (`static`). Usar el nombre de la clase en su lugar.

```
metodoDeClase(); //OK
UnaClase.metodoDeClase(); //OK
unObjeto.metodoDeClase(); //EVITAR!
```

- Evitar asignar el mismo valor a varias variables en la misma sentencia. Es difícil de leer.

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

- No usar asignación embebidas como un intento de mejorar el rendimiento en tiempo de ejecución. Ese es el trabajo del compilador.

```
d = (a = b + c) + r; // EVITAR!
//En su lugar:
a = b + c;
d = a + r;
```

- 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
```

- Intentar hacer que la estructura del programa se ajuste a su intención.

```
if (expresionBooleana) {
    return true;
} else {
    return false;
}

//En su lugar se debe escribir
return expressionBooleana;
```

- Si una expresión contiene un operador binario antes de ? en el operador ternario ?: , se debe colocar entre paréntesis.

```
(x >= 0) ? x : -x;
```

 También se puede consultar la siguiente guía de estilo: https://github.com/twitter/commons/blob/master/src/java/com/twitter/common/styleguide.md

 Otra guía de estilo: http://javaranch.com/styleLong.jsp#class