Published on *Marco de Desarrollo de la Junta de Andalucía* (http://madeja.i-administracion.junta-andalucia.es/servicios/madeja)

Reglas de construcción en Java

- **Área:** Especificaciones de Codificación y Construcción
- **Tipo de pauta:** Directriz
- **Carácter de la pauta:** Obligatoria
- **Tecnologías:** Java

Código: LIBP-0018

Aplicar este conjunto de directrices a la hora de construir clases en Java con el fin de asegurar la calidad del código

A la hora de definir código en el proceso de construcción de un desarrollo es necesario tener claros los siguientes objetivos:

- Realizar un código claro, eficiente y estructurado que facilite un posible mantenimiento del mismo.
- Producir un código con el menor número de errores posibles.
- Realizar un código en base a un estándar de construcción que facilite la reutilización de componentes en la construcción.
- Documentar con eficiencia el código para facilitar la compresión del mismo.
- Tener en cuenta las cuestiones de rendimiento a la hora de programar, intentando encontrar soluciones que minimicen el consumo de recursos del sistema.

Pautas

Título	Carácter
Creación de clases innecesarias	No Recomendada
Cohesión de las clases	Obligatoria
Acoplamiento entre clases	Obligatoria
Clases finales	Recomendada
Atributos de clases finales	Obligatoria
Clases internas	Recomendada
Inicialización de clases	Obligatoria
Herencia	Obligatoria
Interfaces	Obligatoria
Tipos de interfaces	Recomendada
Interfaces frente clases abstractas	Obligatoria
Interfaces redundantes	No Recomendada
Clases del API Java	Obligatoria
Interfaz Serializable	Obligatoria
Creación de métodos innecesarios	No Recomendada
Funcionalidad de los métodos	Obligatoria
Métodos get/set	Obligatoria
Métodos constructores	Obligatoria
Constructores por defecto	Obligatoria
Método finalize()	Obligatoria
Existencia del método finalize()	Obligatoria
Método main()	Recomendada
Método clone()	Recomendada
Métodos hashCode() y equals()	Obligatoria

Comunicación entre hilos	Obligatoria
Métodos finales	Obligatoria
Declaración de atributos públicos	No Recomendada
Declaración de atributos protegidos	Obligatoria
Atributos estáticos	Obligatoria
Inicialización de atributos estáticos	Obligatoria
Modificador "final" en variables	Obligatoria
Variables nuevas	Obligatoria
Tipo de las variables para uso monetario	Recomendada
Simplificación de las condiciones	Obligatoria
Complejidad del código	Obligatoria
Instrucciones de tipo If	Recomendada
Instrucciones de tipo Switch	Obligatoria
Importación de paquetes	Obligatoria
Expresiones invariables en bucles	No Recomendada
Cadena de caracteres	Recomendada
Utilización del System.out	No Recomendada
Niveles de prioridad de log	Obligatoria

Creación de clases innecesarias

No crear clases que no se vayan a usar posteriormente

Sólo debemos crear las clases que sean realmente necesarias para nuestro desarrollo.

Volver al índice

Cohesión de las clases

Conseguir una alta cohesión

La cohesión es la medida que indica si una clase tiene una función bien definida dentro del sistema. El objetivo es enfocar de la forma más precisa posible el propósito de la clase, cada clase debe poseer un propósito claro y simple. No es conveniente mezclar varios propósitos funcionales dentro de una misma clase ya que puede provocar confusión, errores de interpretación y dificultar la detección de errores.

Volver al índice

Acoplamiento entre clases

Conseguir un bajo acoplamiento entre clases

Deberemos intentar que nuestras clases tengan un acoplamiento bajo. El acoplamiento entre clases es una medida de la interconexión o dependencia entre clases. Cuantas menos cosas conozca una clase de otra menor será su acoplamiento. Una clase debe conocer los métodos que ofrece otra, pero, por norma general, no los detalles de implementación o sus atributos.

Volver al índice

Clases finales

No declarar las clases como finales, excepto por motivos de seguridad

Al declarar una clase como final impedimos que se puedan crear subclases que hereden de ésta. Por este motivo, nunca se deberán crear clases finales. Tan sólo deben crearse clases finales por motivos de seguridad, para impedir que se puedan crear subclases que implementen alguna funcionalidad que pueda perjudicar a la aplicación o cuando todos los métodos constructores de la clase sean privados.

Volver al índice

Atributos de clases finales

Asegurar que los atributos de las clases finales no estén definidos como **protected**

Los atributos de las clases finales deben declararse como públicos o privados, pero no deben declararse como protegidos. Esto se debe a que las clases finales no se pueden derivar, por lo que el uso del modificador de acceso protegido puede crear

confusiones.

Volver al índice

Clases internas

Usar clases internas cuando el grado de acoplamiento sea elevado

Se recomienda utilizar clases internas cuando el grado de acoplamiento entre ciertas clases sea muy elevado, pero sin descuidar el tamaño de dichas clases para no aumentar la complejidad. La clases internas no deben sobrepasar las 60 líneas de código.

Volver al índice

Inicialización de clases

Inicializar las clases y superclases en un estado conocido

Las clases y superclases deben inicializarse a un estado estable y conocido para evitar conflictos iniciales y que puedan aparecer ciclos dentro de los inicializadores estáticos que ocasionen errores graves en la aplicación.

Volver al índice

Herencia

Sólo se deben crear clases que hereden de otras cuando se vaya a ampliar la funcionalidad de la clase padre en la clase hija

La herencia consiste en la creación de clases que extienden de otras. Esto es, una clase que añade características propias al contenido de otra clase de la que hereda. Por lo tanto, sólo se deben crear clases que hereden de otras cuando se vaya a ampliar la funcionalidad de la clase padre en la clase hija.

Volver al índice

Interfaces

Usar las interfaces para establecer 'protocolos' entre las clases.

Debemos utilizar las interfaces para establecer protocolos entre las clases, ya que éstas permiten establecer la forma de una clase (nombres de métodos, listas de argumentos y tipos de retorno, pero no bloques de código). En ellas se especifica qué se debe hacer pero no su implementación. Serán las clases que implementen estas interfaces las que describan la lógica del comportamiento de los métodos.

Volver al índice

Tipos de interfaces

Crear diferentes tipos de interfaces para diferentes tipos de usuarios

Se recomienda la creación de diferentes tipos de interfaces según el tipo de usuario para proporcionar un sistema más comprensible desde las diversas perspectivas que puede haber en el mismo. De este modo conseguimos reducir el impacto por mantenimiento.

Volver al índice

Interfaces frente clases abstractas

Usar las interfaces en lugar de clases abstractas

Debemos promover el uso de interfaces, en lugar de clases abstractas, para aquellos casos en los que se tenga pensado dar distintas implementaciones a un mismo método. Esto se debe a que las interfaces son más flexibles que las clases abstractas, permitiendo herencia múltiple en Java. Se recomienda utilizar clases abstractas sólo cuando se implemente cierta funcionalidad que deba ser compartida por todas las subclases.

Volver al índice

Interfaces redundantes

Evitar crear interfaces redundantes

Existen clases que declaran e implementan una interfaz que también es implementada por una superclase. Esto es redundante

porque una vez que una superclase implementa una interfaz, todas las subclases de forma predeterminada también implementan esta interfaz.

Volver al índice

Clases del API Java

Usar o extender en la medida de lo posible las clases del API Java

Debemos usar o extender en la medida de lo posible las clases del API Java, ya que suelen ofrecer un rendimiento nativo de máquina que no se puede igualar utilizando una implementación Java propia. Por ejemplo, el método java.lang.System.arraycopy() es mucho más rápido a la hora de copiar un array de cualquier tamaño que si implementamos nuestro propio bucle para copiar cada uno de sus elementos.

Volver al índice

Interfaz Serializable

Definir el atributo serialVersionUID y crear un constructor vacío, si la clase tiene una superclase

Para optimizar el uso de la interfaz Serializable debemos definir el atributo **serialVersionUID** y crear un constructor vacío, si la clase tiene una superclase. Además, se declararán como private los métodos para la "serialización" o "deserialización", en caso de definirlos.

Volver al índice

Creación de métodos innecesarios

No crear métodos que no se vayan a usar posteriormente

Sólo debemos crear los métodos que sean realmente necesarios para nuestro desarrollo.

Volver al índice

Funcionalidad de los métodos

Minimizar las funcionalidades asignadas a cada método

Cada método debe poseer una funcionalidad clara y simple, por lo que debemos separar los métodos que cambian de estado de aquellos que los consultan. De esta manera, simplificamos el control de concurrencia y extensiones por herencia.

Volver al índice

Métodos get/set

Evitar el mal uso de los métodos get/set

Crear los métodos de acceso y consulta de datos necesarios, teniendo en cuenta que muchos de los atributos tienen dependencias entre ellos para mostrar un valor conjunto y la gestión individual de los mismos puede provocar errores.

Volver al índice

Métodos constructores

Dotar de la mínima funcionalidad a los métodos constructores

Los métodos constructores serán lo más simples posible, evitando llamadas a métodos reemplazables (overridable) y métodos que no sean finales, ya que éstos podrían ser redefinidos, causando errores en la construcción.

Volver al índice

Constructores por defecto

Definir un constructor por defecto

Debemos crear un constructor por defecto, sin parámetros, cuando existan constructores con argumentos en la clase. De esta forma facilitamos la carga dinámica de clases de tipo desconocido en tiempo de compilación, mejorando el rendimiento de la aplicación.

Volver al índice

Método finalize()

Declarar el método finalize como protected

Hay que declarar el método finalize como **protected**, en caso de sobreescribirlo, asegurando que se realizan acciones previas a la invocación del método **super.finalize()**. Además, habrá que evitar que este método pueda ser llamado por el recolector de basura, cuando no haya más referencias al objeto, y que contenga parámetros, ya que podría ocurrir que la máquina virtual no lo invocara.

Volver al índice

Existencia del método finalize()

Asegurar la existencia de un método finalize para las clases que crean recursos

El método **finalize** debe eliminar los recursos (objetos, referencias, etc.) creados por el constructor, normalmente en el orden inverso al que fueron creados.

Volver al índice

Método main()

Escribir un método main para cada clase relevante

Se recomienda crear un método **main** para facilitar los test y las pruebas de las clases.

Volver al índice

Método clone()

Sobreescribir el método clone cuando un objeto pueda ser copiado.

Se recomienda sobrescribir el método **clone()** cuando un objeto pueda ser copiado, ya que el método de la clase Object realiza una copia que puede no tener el nivel de profundidad buscado. A la hora de sobrescribir este método tenemos que tener en cuenta que la clase debe implementar la interfaz **Cloneable** y que debe lanzar la excepción **CloneNotSupportedException** para prevenir que la operación de clonación se ejecute si no se ha otorgado el permiso para ello.

Volver al índice

Métodos hashCode() y equals()

Asegurar que si una clase sobreesscribe el método hashCode() también sobreescribe el método equals()

Estos métodos son especialmente importantes si vamos a guardar nuestros objetos en cualquier tipo de colección: listas, mapas, etc. y más aun si los objetos que vamos a guardar en la colección son serializables.

Volver al índice

Comunicación entre hilos

Usar los métodos wait(), notify() y notifyAll()

Para realizar la comunicación entre hilos debemos usar los métodos **wait()**, **notify()** y **notifyAll()**.

Volver al índice

Métodos finales

Usar métodos finales cuando se quiera proteger de sobreescritura

Debemos crear métodos finales cuando queramos evitar que éstos sean sobreescritos por las subclases. Para ello utilizaremos la palabra clave **"final"** en la declaración del método.

Volver al índice

Declaración de atributos públicos

Nunca declarar como público un atributo de una clase

Se debe evitar el uso de atributos públicos, ya que no debemos dar el control de la estructura interna de la clase a desarrolladores externos. En su lugar, los atributos se declararán privados (**private**) excepto los que sean accesibles por herencia que deben ser declarados como protegidos (**protected**).

Volver al índice

Declaración de atributos protegidos

Favorecer el uso de atributos protegidos en lugar de privados

Los atributos se definirán como protegidos salvo que existan razones muy importantes como para negar el uso de un atributo en las subclases. Esto implica un mayor control sobre estos atributos ya que éstos pueden ser accedidos desde clases externas.

Volver al índice

Atributos estáticos

Minimizar el uso de atributos estáticos

Debemos minimizar el uso de atributos estáticos ya que, al tener un comportamiento similar a las variables globales, provocan que los métodos dependan más del contexto y sean menos reutilizables.

Volver al índice

Inicialización de atributos estáticos

Asegurar que los atributos estáticos tenga valores válidos

Hay que asegurar que las partes estáticas se inicializan correctamente ya que podemos obtener errores al poder invocarse sin necesidad de instanciar la clase.

Volver al índice

Modificador "final" en variables

Usar final para variables que no cambien de valor

Hay que utilizar el modificador "**final**" en aquellas variables que no se van a modificar para evitar que se realicen controles sobre ellas. Si añadimos el modificador "**static**" a estas variables las convertimos en constantes.

Volver al índice

Variables nuevas

Usar variables nuevas y controlar el número de las mismas

Se crearán las variables que sean necesarias, controlando que no se creen más de las debidas, en lugar de reutilizar las variables definidas que no se volverán a usar dentro del código.

Volver al índice

Tipo de las variables para uso monetario

Usar la clase java.math.BigDecimal para aquellas variables de uso monetario

Se recomienda utilizar la clase **java.math.BigDecimal**, proporcionada por Java, en aquellas variables que tengan un uso monetario ya que permite realizar cálculos con punto flotante con la precisión requerida. No se recomienda el uso de los tipos **float** o **double** ya que éstos introducen pequeños márgenes de imprecisión que pueden producir errores en los cálculos.

Volver al índice

Simplificación de las condiciones

Simplificar la complejidad de las condiciones expresadas mediante el uso de operadores lógicos

Se recomienda no anidar más de tres operadores lógicos a la hora de crear una condición dentro del código, ya que la concatenación de más operadores lógicos puede provocar una disminución significativa en el rendimiento y en el mantenimiento de la aplicación.

Volver al índice

Complejidad del código

Comprobar la complejidad ciclomática del código contra un límite especificado

Se debe comprobar la complejidad ciclomática del código contra un límite especificado, midiendo el número de instrucciones

del tipo **if, while, do, for, catch, switch, case** y los operadores **&&** y **||** en el cuerpo de un constructor o el método inicializador de la clase.

Volver al índice

Instrucciones de tipo If

No podrán contener un bloque de código vacío y deberán utilizarse para sentencias lógicas cuyo valor cambie

Las instrucciones de tipo **If** no podrán contener un bloque de código vacío y deberán utilizarse para sentencias lógicas cuyo valor cambie, evitando asignar un valor lógico a una variable dentro del bloque. Además, las instrucciones **If** que sean colapsables entre sí deberán sustituirse por un operador lógico que maneje la condición.

Volver al índice

Instrucciones de tipo Switch

Siempre tendrán un caso por defecto

Las instrucciones de tipo **Switch** siempre tendrán un caso por defecto y no se permitirán bloques de código vacíos.

Volver al índice

Importación de paquetes

Evitar la importación de paquetes usando '*'

Hay que evitar la importación de paquetes usando **'*'**, ya que puede dificultar el seguimiento de las dependencias y provocar duplicidades de paquetes importados. Además, debemos importar sólo aquellos paquetes que se vayan a usar, evitando realizar importaciones de paquetes **'sun.*'** ya que éstos no son portables y tienden a cambiar.

Volver al índice

Expresiones invariables en bucles

Eliminar las expresiones invariables de los bucles

Debemos extraer del interior de los bucles todas las expresiones cuya ejecución produzca siempre el mismo resultado.

Volver al índice

Cadena de caracteres

Usar la clase StringBuffer cuando se trabaje con cadenas de caracteres

Se recomienda usar la clase **StringBuffer** cuando se vayan a manipular, de manera intensiva (reemplazando caracteres, añadiendo o suprimiendo, etc.), cadenas de caracteres, ya que usar la clase **String** resulta poco conveniente.

Volver al índice

Utilización del System.out

Emplear la función System.out o similares para enviar mensajes a consola

No se debe hacer invocación directa a consola, por lo que se descartan mecanismos de log como:

```
System.out.println("Consultando el API");
```

Este tipo de llamadas sólo serán aceptadas en pruebas unitarias de JUnit.

Volver al índice

Niveles de prioridad de log

Utilizar el nivel de log adecuado para cada entorno

En caso de realizar un seguimiento a un nivel muy bajo, la ejecución de las aplicaciones se puede ralentizar, y el log se convertiría en ilegible. Por tanto en tiempo de pruebas o desarrollo se puede utilizar el nivel más bajo de log, DEBUG, pero una vez la aplicación se encuentre en un entorno de producción se recomienda utilizar solo el nivel ERROR o WARN.

Volver al índice

Recursos

Área: Desarrollo » Especificaciones de Codificación y Construcción

Código	Título	Tipo	Carácter
RECU-0745	Implementación de reglas de construcción en Java	Ejemplo	Recomendado
RECU-0749	Niveles de Prioridad de Logging	Ficha	Obligatorio
