ESTÁNDAR DE PROGRAMACIÓN PARA CONSTRUCCIÓN EN JAVA

ESTÁNDAR 11 – 2DA EDICIÓN

Ruiz Cuervo Víctor Arturo
Sandoval Bravo Alejandro
Tapia Cruz Abner Jeffrey

Introducción

En el mundo de la programación no existen reglas que declaren cómo debe o no debe escribirse el código, al menos no más allá que las otorgadas por el mismo lenguaje como su sintaxis. Por esta razón, la ausencia de un control estricto sobre el código que se escribe ha sido un grande problema a través de los años, pues las prácticas utilizadas llegan a ser contraproducentes al momento de mantener o expandir sistemas. Sin embargo, los proyectos que son escritos teniendo en cuenta una estructura unificada y una serie de reglas, que buscan estandarizar buenas prácticas dentro del código, son más fáciles de trabajar tanto en desarrollo como en mantenimiento.

Para poder escribir un proyecto unificado se han creado los estándares de programación. Estos son convenciones que determinan la forma en la que codificaremos nuestros programas según el lenguaje de programación que utilicemos. Los estándares nos permiten crear programas legibles, y facilitan el proceso de mantenimiento y modificación de código. También permite, a otros programadores, reconocer el uso y finalidad de las variables o funciones con solo leer su nombre.

Teniendo en cuenta lo anterior, el siguiente documento establecerá el *Estándar de Programación para Construcción en Java*. Inspirado en el *47 Degrees Coding Standards* y el *Google Java Style Guide*, el *Estándar de Programación para Construcción en Java* busca abarcar los puntos esenciales para la creación de un proyecto de software sostenible, robusto y de calidad. Este estándar será utilizado para la creación del proyecto final de, la Experiencia Educativa, Principios de construcción de Software; del equipo once.

Repositorio: https://github.com/Phalord/PracticasProfesionales.

Formato

Llaves

Uso de llaves

Las llaves serán usadas siempre, así sea opcional como en el caso del *if*.

```
//Incorrect
if (isThisTrue())
    ejecutar();

//Correct
if (isThistrue()) {
    ejecutar();
}
```

Bloqueos: Estilo Kernighan and Ritchie

Las llaves seguirán el estilo “Llaves Egipcias” para los bloques de código.

Estilo Egipcio:

- No salto de línea antes de la llave de apertura
- Salto de línea después de la llave de apertura
- Salto de línea antes de la llave de cierre
- Lave de cierre después de la llave
- Los punto y coma después de la llave puede ser omitida, a menos que termine una declaración
- No hay salto de línea si después de la llave hay un “else” o una coma

```
//Example
return () -> {
    while (condition()) {
        method();
    }
};
```

```
return new MyClass() {
    @Override
    public void method() {
        if (condition()) {
            something();
        } catch (Exception exception) {
            recover();
        }
    }
};
```

Sangría de Bloque: +4 espacios

Cada que un bloque o una línea del bloque es abierta, la sangría incrementa por cuatro espacios. Cuando el bloque termina, la sangría regresa a su anterior nivel de sangría. La sangría aplica para el código como a comentarios a través del bloque.

```
//Example
try {
    something();
} catch (Exception exception) {
    recover();
}
```

Una declaración por Línea

Cada declaración es seguida de un salto de línea.

Límite de Columna: 100 caracteres

El código Java tiene un límite de columna de 100 caracteres. Un “carácter” es cualquier punto de código Unicode. A excepción de lo mostrado abajo, cualquier línea que pueda exceder este límite debe ser ajustado, de acuerdo con este estándar.

Excepciones:

- Líneas donde sea imposible obedecer el límite (por ejemplo, un método de referencia JSNI largo)
- Declaraciones package e import

Envoltura de Línea

Nota de Terminología: Cuando código que puede, legalmente, ocupar una sola línea de código es dividida en múltiples líneas, la actividad se llama “Envoltura de código”.

No existe una fórmula comprehensiva o determinística que muestre exactamente cómo envolver una línea en cada situación. Regularmente, existen varias formas válidas de envolver la misma pieza de código.

Aunque la razón típica para envolver código sea prevenir sobrepasar el límite de columna, incluso código que puede caber en una sola línea puede ser envuelto por decisión propia del autor.

Dónde saltar:

- Cuando se aplica un salto de línea a un operador sin-asignación, el salto se hace antes del símpolo.
    - Esto también aplica a los siguientes símbolos “parecidos a operadores”:
        - Separador punto (`.`)
        - Un ampersand en un tipo enlazado (<T extends Foo & Bar>)
        - Pipe en un bloque *catch* (catch (FooException | BarException e)
- Cuando una línea es saltada en un operador de asignación, regularmente, el salto viene después del símbolo, pero también se acepta hacerlo antes.
- El nombre de un método o un constructor permanece atado a su paréntesis de apertura que le sigue (*(*).
- Una coma (`,`) permanece atada al token que le precede
- Una línea nunca se salta adyacente a una flecha en una lambda, excepto que un salto venga inmediatamente después de la flecha si el cuerpo de la lambda consiste en una simple expresión sin llaves.

```
//Example
MyLambda<String, Long, Object> lambda =
  (String label, long value, Object obj) -> {
      //...
  };
Predicate<String> predicate = string ->
  longExpressionInvolving(string);
```

Espacio Vertical

Un único espacio aparece siempre. Entre miembros consecutivos o inicializadores de una clase: campos, constructores, métodos, clases anidadas, inicializadores estáticos, e inicializadores

Restricciones

Comparación de Booleanos

No invertir el significado natural del lenguaje; preferible usar “si el estado actual no está activado” que “si activo no es el estado actual”

```
//Incorrect
!active;

//Correct
active == false;
```

For loops VS for-each loop

Cuando se esté iterando en un elemento iterable, se prefiere hacer uso de:

```
//Incorrect
for (int i=0; i<names.lenght(); i++) {
    doSomething();
}

//Correct
for (String name: names) {
    doSomething();
}
```

Concatenación

Evitar el uso del operador “+” para concatenar cadenas. Usa estándares del lenguaje creados para esos propósitos como lo es *String.format( )* y *StringBuilder( ).*

```
//Incorrect
log.debug(“found ” + amount + “ items”);

//Correct
log.debug(String.format(“found %s items”, amount));
```

Excepciones

No comerse excepciones

Evitar el uso de la excepción de mayor nivel (*Exception)* dentro del *catch*. Excepción que se atrapa, excepción que debe ser manejada. No pueden existir bloques *catch* vacíos.

```
//Incorrect
try {
    do();
} catch (Throwable e) {
    log.error(e);
}

try {
    do();
} catch (SomethingWeCanHandle e) {

}

//Correct
try {
    do();
} catch (SpecificException specificException) {
    log.error(specificException);
    notifyUser(specificException);
} finally {
    cleanup();
}
```

Tipos generals

Evitar usar declaraciones tipo “Dios” con las clases que soporten genéricos.

```
//Incorrect
List people = Arrays.asList(“you”, “me”);

//Correct
List<String> people = Arrays.asList(“you”, “me”);
```

Uso de final

Al implementar servicios y clases que sean más que javabeans u objetos para transferir datos entre capas, mantente seguro de usar la palabra clave “final” para comunicar las intenciones con respecto a las subclases, el uso de constantes y valores que una vez declarados sean inmutables.

```
//Example
public final class ThisShouldNeverBeExtended() {
    //...
}

public final void neverOverrideThisMethod() {
    //...
}

private final int thisFieldvalueWillNeverChange;

final int thisLocalVariableWillNeverChangeOnceSet;
```

Nombra los valores de retorno como “result”

Considera el uso de “result” como el nombre de la variable de retorno. Esto facilita la dura tarea de depurar e incrementa la legibilidad del código.

```
//Example
public Object doSomething() {
    Object result = null;
    If (something) {
        result = new Object();
    }
    result;
}
```

Considera a los setters y getters para acceder a los campos

Dentro de la misma clase, considere usar getters y setters para acceder a los valores de variables para que siempre se apliquen en la inicialización diferida y otra lógica prevista implementada en los getters y setters.

```java
//Incorrect
public void changeName(String name) {
    this.name = name;
}


//Correct
public void changeName(String name) {
    setName(name);
}
```

Seguridad

Pruebas

Uso de la metodología de TDD. El Desarrollo Guiado por Pruebas, o Test Driven Development por sus siglas en inglés, consta en escribir primero los casos de prueba unitarios; para estos se toma una parte del código y comprobar su funcionamiento.

Excepciones

Captura de Excepciones

Las excepciones deben ser atrapadas en un bloque try-catch

Las respuestas típicas son registrara, o si se considera “imposible”, deben ser lanzadas como AssertionError.

```java
//Example
try {
    mySQLConnection.readProperties();
} catch (SQLException sqlException) {
    logger.log(Level.SEVERE, sqlException.getMessage());
}
```

Miembros estáticos

Cuando se revisa una referencia a un miembro de una clas estática, se califica con el nombre de esa clase, no con una referencia o expresión del tipo de clase.

Convención de Nombres

Reglas en común para todos los identificadores

Los identificadores solo usarán caracteres de ASCII y, en unos cuantos casos a mencionar, subrayado. Así, cada nombre de identificador válido coincide con la expresión regular “\w+”.

Reglas por tipo de identificador

Clases

Los nombres de las clases se escriben en PascalCase. Normalmente se utilizan sustantivos o frases nominales. Por ejemplo, “Character” o “ImmutableList”.

Las interfaces también se pueden nombrar con sustantivos o frases nominales (por ejemplo “IList”), pero, en ocasiones pueden ser adjetivos o frases adjetivas (por ejemplo, “IReadable”). Todas las interfaces comenzarán su nombre con una *i* mayúscula (I).

Las clases de prueba se nombran iniciando con el prefijo “Test”, seguido del nombre de la clase, Por ejemplo, “TestHash” o “TestHashIntegration”.

Métodos

Normalmente son verbos o frases verbales. Por ejemplo “sendMessage” o “stop”. Se les puede agregar un guion bajo a los métodos de prueba para separar los componentes lógicos del nombre.

Constantes

Los nombres de constantes usan CONSTANT_CASE: Todas las letras en mayúsculas y cada palabra separada por un guion bajo.

Las constantes son campos estáticos cuyo contenido es inmutable y cuyos métodos no tienen efectos secundarios notables. Esto incluye datos primitivos, cadenas, tipos inmutables y colecciones de estos últimos. Si alguno de los estados observables de la instancia se puede cambiar, no es una constante.

Campos no constantes

Normalmente, son sustantivos o frases nominales. Por ejemplo “computedValues” o “index”.

Variables locales

Incluso cuando son fianles e inmutables, estas variables no se consideran constantes y no se deben diseñar así.

Camel Case

Para convertir una frase a camelCase, se comienza con el nombre en forma de prosa. Después, se sigue el siguiente proceso:

1. Convertir la frase a ASCII simple y eliminar apóstrofes
2. Dividir el resultado en palabras divididas por espacios y cualquier puntación restante.
3. Convertir todo a minúscula.
4. Unir todas las palabras en un solo identificador
5. Escribir en mayúscula la primera letra de cada palabra del identificador a excepción de la primera.
6. Unir todas las palabras en un solo identificador

Prosa	Correcto	Incorrecto
“XML HTTP Request”	`xmlHTTPRequest`	`XMLHTTPRequest`
“New customer ID”	`newCustomerID`	`newCustomerId`
“Inner Stopwatch”	`innerStopwatch`	`innerStopWatch`
“Supports IPv6 on iOS?”	`supportsIPv6OnIOS`	`supportsipv6onios`
“import file”	`importFile`	`ImportFile`