

	<b>NORMATIVIDAD</b>		<b>Código:</b> NRM-STI-601
	<b>ESTÁNDARES DE PROGRAMACIÓN PARA EL DESARROLLO DE SOFTWARE CON TECNOLOGÍA JAVA</b>		<b>Versión:</b> 1.2
			<b>Fecha:</b> 29/05/2015

<b>NORMATIVIDAD</b>			
<b>Estándares de Programación para el Desarrollo de Software con tecnología Java</b>			
<b>SUB DIRECCIÓN DE DESARROLLO DE PROYECTOS - OSCE</b>			

 <small>Organismo Superior de las Certificaciones del Estado</small>	<b>NORMATIVIDAD</b>		<b>Código:</b> NRM-STI-601
	<b>ESTÁNDARES DE PROGRAMACIÓN PARA EL DESARROLLO DE SOFTWARE CON TECNOLOGÍA JAVA</b>		<b>Versión:</b> 1.2
			<b>Fecha:</b> 29/05/2015


### REVISIÓN HISTÓRICA

Fecha	Versión	Autoría	Descripción
11/06/2013	1.0	Nelson A. Vega Bazán Meléndez Milton García Llamoca	Creación del documento
01/04/2014	1.1	Milton García Llamoca	Modificación de documento.
01/06/2014	1.2	Milton García Llamoca.	Inclusión y clasificación de estándares.


 <b>OSCE</b> <small>Organismo Superior de las Certificaciones del Estado</small>	<b>NORMATIVIDAD</b>		<b>Código:</b> NRM-STI-601
	<b>ESTÁNDARES DE PROGRAMACIÓN PARA EL DESARROLLO DE SOFTWARE CON TECNOLOGÍA JAVA</b>		<b>Versión:</b> 1.2
			<b>Fecha:</b> 29/05/2015

## Contenido

<b>1. Introducción</b>	<b>5</b>
1.1. Propósito	5
1.2. Alcance	5
1.3. Definiciones, Acrónimos y Abreviaciones	5
1.4. Audiencia	5
<b>2. Estándares de programación Java</b>	<b>6</b>
2.1. Organización De Archivos	6
2.2. Archivos Fuentes Java	6
2.3. Comentarios iniciales	6
2.4. Paquete y declaraciones de importación	7
2.5. Clases y declaraciones de interfaz	7
2.6. Indentación	7
2.7. Longitud de la línea	7
2.8. Rompiendo líneas	7
2.9. Nomenclatura en los programas	8
2.9.1. Para Nombres en los programas:	8
2.9.2. Para Nombres de Funciones Miembros en los programas:	8
2.10. Nomenclatura para Paquetes	8
2.11. Nomenclatura para Clases, Interfaces	9
2.12. Sufijos en la Nomenclatura de Clases e Interfaces	9
2.13. Nomenclatura para atributos	9
2.14. Nomenclatura de Métodos	10
2.15. Nomenclatura para Variables Locales	10
2.16. Nomenclatura para Parámetros	10
2.17. Declaraciones	10
2.17.1. Cantidad por línea	10
2.17.2. Inicialización	11
2.17.3. Colocación	11
2.17.4. Declaraciones de clases e interfaces	11
2.18. Comentarios	12
2.18.1. Formatos de los comentarios de implementación	13
2.18.2. Comentarios de bloque	13
2.18.3. Comentarios de una línea	13
2.18.4. Comentarios de remolque	14
2.18.5. Comentarios de fin de línea	14
2.18.6. Comentarios de documentación	14
2.18.7. Para Tags “javadoc” en los programas:	15
2.19. Sentencias	15
2.19.1. Sentencias simples	15
2.19.2. Sentencias compuestas	16
2.19.3. Sentencias de retorno	16
2.19.4. Sentencias if, if-else, if else-if else	16
2.19.5. Sentencias for	16
2.19.6. Sentencias while	17
2.19.7. Sentencias do-while	17
2.19.8. Sentencias switch	17
2.19.9. Sentencias try-catch	18
2.20. Espacios en blanco	18
2.20.1. Líneas en blanco	18
2.20.2. Espacios en blanco	19

	<b>NORMATIVIDAD</b>		<b>Código:</b> NRM-STI-601
	<b>ESTÁNDARES DE PROGRAMACIÓN PARA EL DESARROLLO DE SOFTWARE CON TECNOLOGÍA JAVA</b>		<b>Versión:</b> 1.2
			<b>Fecha:</b> 29/05/2015

2.21.	Manejo de Errores y Validaciones	19
2.22.	Concatenación de cadenas	19
2.23.	Colecciones	20
2.24.	Métodos que devuelven una lista de objetos	20
2.25.	Métodos que reciben parámetros directos	21
2.26.	Uso de Valores Monetarios	21
2.27.	Hábitos de programación	21
2.27.1.	Referencias a variables y métodos de clase	21
2.27.2.	Constantes	22
2.27.3.	Asignaciones de variables	22
2.28.	Hábitos varios	22
2.28.1.	Paréntesis	22
2.28.2.	Valores de retorno	22
2.28.3.	Expresiones antes de '?' en el operador condicional	23
2.28.4.	Comentarios especiales	23
2.28.5.	Para Escritura de "código limpio" en los programas;	23
2.28.6.	Ejemplo de código Java	23
3.	Aplicaciones empresariales basadas en java	25
3.1.	Utilitarios	25
3.1.1.	Uso de Log's	25
3.2.	Conectividad	26
3.2.1.	Estándares JDBC	26
3.3.	Aplicaciones WEB	26
3.3.1.	Recursos WEB estáticos	26
3.3.2.	JavaScript	27
3.3.3.	Formularios WEB	27
3.3.4.	Componentes Plantillas - JSF Faceletes	27
3.3.5.	Globalización	28

	<b>NORMATIVIDAD</b>		<b>Código:</b> NRM-STI-601
	<b>ESTÁNDARES DE PROGRAMACIÓN PARA EL DESARROLLO DE SOFTWARE CON TECNOLOGÍA JAVA</b>		<b>Versión:</b> 1.2
			<b>Fecha:</b> 29/05/2015

## ESTÁNDARES DE PROGRAMACIÓN PARA EL DESARROLLO DE SOFTWARE CON TECNOLOGÍA JAVA

### 1. Introducción

#### 1.1. Propósito

Este documento describe los estándares de codificación, herramientas y productos a utilizar por los desarrolladores, así como documentar las buenas prácticas en la construcción de aplicaciones JAVA.

El código fuente debe ser desarrollado siguiendo estándares de desarrollo para facilitar su lectura y la modificación por cualquier miembro del equipo de desarrollo.

#### 1.2. Alcance

Este documento es de interés a todos los desarrolladores y empresas externas que desean conocer los estándares de codificación propuesta para el desarrollo de aplicaciones JAVA para el Organismo Supervisor de las Contrataciones del Estado - OSCE.

#### 1.3. Definiciones, Acrónimos y Abreviaciones


Las principales siglas utilizadas a lo largo de este documento, son:

- JAVA EE: Java Enterprise Edition
- JVM: Java Virtual Machine
- DTO: Data Transfer Object
- VO: Value Object
- EJB: Enterprise Java Bean
- ESB: Enterprise Service Bus
- RIA: Rich Internet Applications
- SOA: Service Oriented Architecture
- JPA: Java Persistence API
- POJO: Plain Old Java Object
- RMI: Remote Method Invocation
- JMS: Java Message Service
- WSDL: Web Service Description Language
- SOAP: Simple Object Access Protocol
- UDDI: Universal Description Discovery and Integration
- XSD: XML Schema Definition
- XSLT: Extensible Stylesheet Language Transformations
- WS Policy: Web Service Policy
- LDAP: Lightweight Directory Access Protocol
- ECM: Enterprise Content Management

#### 1.4. Audiencia

La audiencia objetivo del presente documento corresponde a:

- El equipo de arquitectura; el cual utilizará este documento para la revisión de los artefactos de desarrollo, velando que se sigan los estándares aquí establecidos.
- Los miembros del equipo desarrollo, los cuales encontrarán en este documento el detalle de la documentación de los bloques básicos, en los que se construirá gran parte de los elementos de las aplicaciones.
- Los miembros del equipo calidad; Encontrarán la información necesaria para crear y aplicar listas de revisión e inspección.

	NORMATIVIDAD		Código:	NRM-STI-601
	ESTÁNDARES DE PROGRAMACIÓN PARA EL DESARROLLO DE SOFTWARE CON TECNOLOGÍA JAVA		Versión:	1.2
			Fecha:	29/05/2015

- Los miembros del equipo mantenimiento, los cuales encontrarán en este documento el lineamiento base para la construcción de los componentes de software a ser mantenidos.

## 2. Estándares de programación Java

El propósito fundamental de los estándares de codificación es que el **Organismo Supervisor de Contrataciones del Estado** tenga una arquitectura y un estilo consistente en la construcción de aplicaciones JAVA. Esto ayudará a contar con sistemas de fácil entendimiento y mantenimiento.

### 2.1. Organización De Archivos

El archivo consta de secciones que deben ser separados por líneas en blanco y un comentario opcional que identifica cada sección.

Archivos de más de 2.000 líneas son engorrosos y deben ser evitados.

### 2.2. Archivos Fuentes Java

Cada archivo fuente Java contiene una clase o interface pública. Cuando las clases e interfaces son asociadas con clases públicas, se puede juntar en el mismo archivo como una clase pública. La clase pública puede ser la primera clase o interface en el archivo.

Los archivos fuentes de java tienen el siguiente orden.

- Sentencias de Package e Import.
- Declaración de clases e interfaces.

### 2.3. Comentarios iniciales


Todos los archivos de Fuentes pueden empezar con un comentario c-style que liste: una breve descripción de la clase; propiedad intelectual, año de creación y el nombre de la institución; Nombre del desarrollador o la empresa; información de la versión, fecha, etc. Ejemplo:

#### *Ejemplo de comentario para una clase*

```
/**
 * [Breve Información de la clase]
 *
 * <ul>
 * <li> [Copyright-año] [Nombre de la Institución dueña del código fuente].</li>
 * </ul>
 *
 * @author [Nombre y Apellidos del autor] o [Nombre de la Empresa que desarrollo el código]
 * @version [Información de la versión]
 * @since [Año de Creación de la clase]
 * @see [Referencia cruzada]
 *
 */
```

#### *Ejemplo de comentario para un método*

```
/**
 * Constructor para la serie de números aleatorios
 *
 * @param [nombreDelParametro] [Descripción]
 * @return [Descripción del parámetro devuelto]
 * @exception [Excepción lanzada]
 * @deprecated [Comentario del método que será obsoleto (propio de versiones anteriores) y que no se recomienda su uso.]
 */
```

	NORMATIVIDAD		Código:	NRM-STI-601
	ESTÁNDARES DE PROGRAMACIÓN PARA EL DESARROLLO DE SOFTWARE CON TECNOLOGÍA JAVA		Versión:	1.2
			Fecha:	29/05/2015

## 2.4. Paquete y declaraciones de importación

La primera línea que no sea la mayoría de los comentarios de los archivos de código fuente de Java es una sentencia de paquete. Después de eso, las declaraciones de importación pueden seguir. Ejemplo:

```
package pe.gob.osce.seguridad.be.model.jpaa;
import java.io.Serializable;
import java.util.Date;
```

## 2.5. Clases y declaraciones de interfaz

En la siguiente tabla se describen las partes de una declaración de clase o interfaz, en el orden en que deben aparecer.

Parte de una declaración Clase/Interface	Notas
Comentario de Clase/Interface documentación (/*...*/)	
Sentencias de clase o interface	
Comentario de Implementación de Clase/Interface (/*...*/), si fuera necesario	Este comentario debe contener información que no es apropiada para el comentario documentación.
Variables (static) de Clase	Primero las variables públicas, luego las protegidas, luego las de nivel de paquete y luego las privadas.
Variables Instanciadas	Primero públicas, luego protegidas, luego de nivel de paquete y luego privadas.
Constructores	
Métodos	Este método debe agrupar por funcionalidad más que por lugar o accesibilidad.

## 2.6. Indentación

EL código debe ser indentando de tal manera que sean fáciles de leer.

Se deben emplear cuatro espacios como unidad de indentación. La construcción exacta de la indentación (ajustarse al esquema de la herramienta de desarrollo) no se especifica. Los tabuladores deben ser exactamente cada 4 espacios.

## 2.7. Longitud de la línea

Evitar las líneas de más de 160 caracteres, ya que no son bien manejadas por muchos terminales y herramientas.

**Nota:** Para uso en la documentación deben tener una longitud inferior, generalmente no más de 120 caracteres.


## 2.8. Rompiendo líneas

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 (más a la derecha que el "padre") que de bajo nivel (más a la izquierda que el "padre").
- 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.

Ejemplos:

```
public metodo (longExpression1, longExpression2, longExpression3,
```

	<b>NORMATIVIDAD</b>		<b>Código:</b> NRM-STI-601
	<b>ESTÁNDARES DE PROGRAMACIÓN PARA EL DESARROLLO DE SOFTWARE CON TECNOLOGÍA JAVA</b>		<b>Versión:</b> 1.2
			<b>Fecha:</b> 29/05/2015

```

                                longExpression4, longExpression5) {

var = function1(longExpression1,
                function2(longExpression2,
                           longExpression3));

longName1 = longName2 * (longName3 + longName4 - longName5)
+ 4 * longname6; // Preferir.

longName1 = longName2 * (longName3 + longName4
- longName5) + 4 * longname6; // Evitar

```

## 2.9. Nomenclatura en los programas

### 2.9.1. Para Nombres en los programas:

- Usar terminología aplicable al dominio.
- Usar combinaciones de Mayúsculas / minúsculas para facilitar la lectura de los nombres (por ejemplo: privilegioUsuario).
- Evitar preferentemente el uso de abreviaciones, por ejemplo: apm, app, num, etc.
- Evitar el uso de nombres demasiado largos (<20).
- Evitar el uso de nombres similares o con diferencia de una letra, por ejemplo persistentObject y persistentObjects.
- Respetar la nomenclatura de acrónimos estándar, por ejemplo: sqlDatabase y no sQLDatabase, etc.
- Iniciales según los tipos primitivos para variables locales, por ejemplo int iCode, boolean bType. Esto no aplica para variables de clase o instancia.
- Uso de "this." para variables de clase que no sean estáticos.
- Los nombres de los atributos de dominio sean completos sin anteposición de caracteres adicionales.

### 2.9.2. Para Nombres de Funciones Miembros en los programas:

- Setters y Getters deberán empezar con respectivos set y get en minúsculas.
- En el uso de Getters se utilizará convenciones para HAS, CAN e IS. Por ejemplo. para una variable tipo boolean llamada "empty" el metodo getEmpty() se convierte en isEmpty().
- Para Constructores, debe haber visibilidad de funciones miembros, así como documentación de las mismas.
- Documentación de Cabecera: paso de parámetros, que hace, como lo hace, variable de retorno, bugs, lista de excepciones, visibilidad, precondiciones, poscondiciones, historial de cambios.
- Documentación interna: estructuras de control, que hace, como lo hace, variables locales, fin de corchetes (ejemplo } // fin de while )

## 2.10. Nomenclatura para Paquetes

Se escribe sólo en minúsculas

El nombramiento debe seguir la regla:

prefijo.{sistema}.{nombremodulo}.{subparte}

Donde:


prefijo es "pe.gob.osce".

sistema es el nombre del sistema que se está desarrollando, por ejemplo seace, rnp, arbitraje, etc.

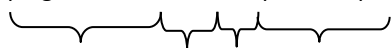
nombremodulo es el nombre del módulo a desarrollar, por ejemplo interfaces, configurador, etc.

subparte es el grupo de clases de un mismo tipo. La subparte a la vez pueden tener mas agrupaciones, según la arquitectura.



	<b>NORMATIVIDAD</b>		<b>Código:</b> NRM-STI-601
	<b>ESTÁNDARES DE PROGRAMACIÓN PARA EL DESARROLLO DE SOFTWARE CON TECNOLOGÍA JAVA</b>		<b>Versión:</b> 1.2
			<b>Fecha:</b> 29/05/2015

pe.gob.osce.seace.int.bp.dao.impl



prefijo sistema módulo subparte

Módulos predefinidos

Son los módulos predefinidos que dan soporte a las aplicaciones, en este caso se verán más adelante.

## 2.11. Nomenclatura para Clases, Interfaces


- Visibilidad (package vs. public)
- Documentación (propósito, bugs, historial de cambios)
- Declaración (calificadores)
- Orden de los miembros (+visible a -visible)
- Minimización de la interfaz (reducción de acoplamiento y flexibilidad)
- Usar importación con nombre completo del paquete (import java.util.Vector)
- Los nombres de clase deben empezar con una letra mayúscula y el resto de letras deben estar escritas en minúscula.
- Los nombres de clase no pueden ser verbos.
- Los nombres de clase no pueden contener espacios ni caracteres especiales, sólo son permitidas las letras de la “a” a la “z” y los números del 0 al 9.
- Si el nombre de clase está compuesto por más de una palabra, cada palabra debe empezar con mayúscula.

## 2.12. Sufijos en la Nomenclatura de Clases e Interfaces

- Los nombres de las clases que son implementaciones de las interfaces deberán tener el sufijo “Impl”
- Los nombres de clases tipo DAO (Data Access Object) deberán tener el sufijo “DAO”.
- Los nombres de clases tipo JPA no tendrán sufijo.
- Las clases de tipo Enum que los valores hagan mención a:
  - Estados deberán tener el sufijo “Estado”.
  - Tipo deberán tener el sufijo “Tipo”.
- Los nombres de clases de tipos de Session Beans: stateless, stateful deberán tener el sufijo:
- “DAOImpl” cuando es una implementación de una Interface y que la clase sea un DAO
- “Local” o “Remoto” cuando es interfaz.
- Los nombres de clases de tipo VO (Value Object) deberán tener el sufijo “VO”. Ejemplo UsuarioVO.
- Los nombres de clases de tipo DTO (Data Access Object) deberán tener el sufijo “DTO”. Ejemplo DatoDTO.
- Como regla general, aquellas clases que implementen una funcionalidad característica o un patrón de diseño definido diferente a los anteriormente expuestos, deben indicarlo mediante un sufijo en la denominación del nombre de la clase
- Los nombres de clases de tipo JSF Bean Administrados deberán tener el sufijo “Bean”.
- Los nombres de clases para pruebas unitarias deberán tener el sufijo “UT”. Ejemplo UsuarioUT
- Los nombres de clases para pruebas deberán tener el sufijo “Test”. Ejemplo UsuarioTest
- Los nombres de clases utilitarias deberán tener el sufijo “Util”. Ejemplo LetrasUtil.

## 2.13. Nomenclatura para atributos

- Los nombres de atributos deben empezar con una letra minúscula y el resto de letras deben estar escritas en minúscula.

 <small>Organismo Superior de las Comunicaciones del Estado</small>	<b>NORMATIVIDAD</b>		<b>Código:</b> NRM-STI-601
	<b>ESTÁNDARES DE PROGRAMACIÓN PARA EL DESARROLLO DE SOFTWARE CON TECNOLOGÍA JAVA</b>		<b>Versión:</b> 1.2
			<b>Fecha:</b> 29/05/2015

- Si el nombre de atributo está compuesto por más de una palabra, cada palabra debe empezar con mayúscula.
- Los nombres de atributo no pueden ser verbos.
- Los nombres de atributo no pueden contener espacios ni caracteres especiales, sólo son permitidas las letras de la “a” a la “z” y los números del 0 al 9.
- Debe haber Visibilidad de campos, es decir determinar si el campo es público, privado o protegido.
- Documentación de los campos.
- Inicialización de campos estáticos
- Acceso a los campos (Getters, Setters),
  - Inicialización,
  - Acceso a constantes,
  - Acceso a colecciones,
  - Acceso simultaneo a varios campos,
  - Visibilidad.
- Nombres de Campos,
  - Descriptor en Español
  - Nombre de variables (descriptor completo)
  - Nombre de constantes (en mayúsculas)
  - Nombre de colecciones (plural)

#### 2.14. Nomenclatura de Métodos

- Los nombres de método deben empezar usar la notación Camel.
- Los nombres de método deben ser verbos o palabras que identifiquen de manera general el objetivo del método
- Los nombres de método no pueden contener espacios ni caracteres especiales, sólo son permitidas las letras de la “a” a la “z” y los números del 0 al 9.
- Si el nombre de método requiere estar compuesto por más de una palabra, cada palabra debe empezar con mayúscula.

#### 2.15. Nomenclatura para Variables Locales

- Nombre de streams (input / output)
- Nombre de contadores en loops
- Nombre de excepciones
- Declaración y documentación

#### 2.16. Nomenclatura para Parámetros


Mismos que para variables locales

#### 2.17. Declaraciones

##### 2.17.1. Cantidad por línea

- Se recomienda una declaración por línea, ya que facilita los comentarios. En otras palabras, se prefiere:  

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

	NORMATIVIDAD		Código: NRM-STI-601
	ESTÁNDARES DE PROGRAMACIÓN PARA EL DESARROLLO DE SOFTWARE CON TECNOLOGÍA JAVA		Versión: 1.2
			Fecha: 29/05/2015

Antes que: `int level, size;`

- No poner diferentes tipos en la misma línea. Ejemplo:  
`int foo, foobar[]; //ERROR!`

### 2.17.2. Inicialización

Intentar inicializar las variables locales donde se declaran. La única razón para no inicializar una variable donde se declara es si el valor inicial depende de algunos cálculos que deben ocurrir.

### 2.17.3. Colocación

- Poner las declaraciones solo al principio de los bloques (un bloque es cualquier código encerrado por llaves "{" y "}"). No esperar al primer uso para declararlas; puede confundir a programadores no preavisados y limitar la portabilidad del código dentro de su ámbito de visibilidad.
- 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!
    }
    ...
}
```

- En los bloques for crear e inicializar el objeto a utilizar fuera del for, no dentro para optimizar el uso de memoria.

**Correcto!!**

```
ObjectX obj = null;
for (ObjectY obj2 : lista) {
    obj = ...
}
```


**Evitar!!!**

```
for (ObjectY obj2 : lista) {
    ObjectX obj = ...
}
```

### 2.17.4. Declaraciones de clases e interfaces

Al codificar clases e interfaces de Java, se siguen las siguientes reglas de formato:

- 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 declaración.

	NORMATIVIDAD		Código:	NRM-STI-601
	ESTÁNDARES DE PROGRAMACIÓN PARA EL DESARROLLO DE SOFTWARE CON TECNOLOGÍA JAVA		Versión:	1.2
			Fecha:	29/05/2015

- 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.
- La asignación de objetos/valores en los métodos debe incluir un espacio en blanco después de la coma en la lista de argumentos, ejemplo: *método(a, b, c)*;

## 2.18. Comentarios

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

Los comentarios de implementación son aquellos que también se encuentran en C++, delimitados por `/*...*/`, y `//`. Los comentarios de documentación (conocidos como "doc comments") existen sólo en Java, y se limitan por `/**...*/`. Los comentarios de documentación se exportarán a archivos HTML con la herramienta javadoc.


Los comentarios de implementación son para comentar nuestro código o para comentarios acerca de una implementación particular. Los comentarios de documentación son para describir la especificación del código, libre de una perspectiva de implementación, y para ser leídos por desarrolladores que pueden no tener el código fuente a mano.

Se deben usar los comentarios para dar descripciones de código y facilitar información adicional que no es legible en el código mismo. Los comentarios deben contener sólo información que es relevante para la lectura y entendimiento del programa. Por ejemplo, información sobre como se construye el paquete correspondiente o en que directorio reside no debe ser incluida como comentario.

Son apropiadas las discusiones sobre decisiones de diseño no triviales o no obvias, pero evitar duplicar información que está presente (de forma clara) en el código ya que es fácil que los comentarios redundantes se queden desfasados. En general, evitar cualquier comentario que pueda quedar desfasado a medida que el código evoluciona.

**Nota:** La frecuencia de comentarios a veces refleja una pobre calidad del código. Cuando se sienta obligado a escribir un comentario considere reescribir el código para hacerlo más claro.

Los comentarios no deben encerrarse en grandes cuadrados dibujados con asteriscos u otros caracteres. Los comentarios nunca deben incluir caracteres especiales como backspace.

	NORMATIVIDAD		Código:	NRM-STI-601
	ESTÁNDARES DE PROGRAMACIÓN PARA EL DESARROLLO DE SOFTWARE CON TECNOLOGÍA JAVA		Versión:	1.2
			Fecha:	29/05/2015

Los comentarios deben añadir claridad al código.

Evitar la decoración de los comentarios (por ejemplo decoraciones COBOL).

Mantener los comentarios simples.

Toda clase debe empezar con un comentario de cabecera que indique el autor, la fecha de creación y una descripción breve del objetivo de la clase.

Los comentarios no deben utilizarse para líneas de código que ya dejaron de usarse que por lo general son funcionalidades de funcionalidades modificadas por algunas correcciones.

### 2.18.1. Formatos de los comentarios de implementación

Los programas pueden tener cuatro estilos de comentarios de implementación: de bloque, de una línea, de remolque, y de fin de línea.

### 2.18.2. Comentarios de bloque

Los comentarios de bloque se usan para dar descripciones de archivos, métodos, estructuras de datos y algoritmos. Los comentarios de bloque se podrán usar al comienzo de cada archivo o antes de cada método. También se pueden usar en otro lugares, tales como el interior de los métodos. Los comentarios de bloque en el interior de una función o método deben ser indentados al mismo nivel que el código que describen. Un comentario de bloque debe ir precedido por una línea en blanco que lo separe del resto del código.

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


*Los comentarios de bloque pueden comenzar con /\*-, que es reconocido por **indent(1)** como el comienzo de un comentario de bloque que no debe ser reformateado. Ejemplo:*

```
/*-
* Aquí tenemos un comentario de bloque con cierto
* formato especial que quiero que ignore indent(1).
*
* uno
* dos
* tres
*/
```

**Nota:** Si no se usa **indent(1)**, no se tiene que usar /\*- en el código o hacer cualquier otra concesión a la posibilidad de que alguien ejecute **indent(1)** sobre él.

### 2.18.3. Comentarios de una línea

Pueden aparecer comentarios cortos de una única línea al nivel del código que siguen. Si un comentario no se puede escribir en una línea, debe seguir el formato de los comentarios de bloque. (ver sección 6.1.1). Un comentario de una sola línea debe ir precedido de una línea en blanco. Aquí un ejemplo de comentario de una sola línea en código Java:

 <small>Organismo Superior de las Comunicaciones del Estado</small>	<b>NORMATIVIDAD</b>		<b>Código:</b> NRM-STI-601
	<b>ESTÁNDARES DE PROGRAMACIÓN PARA EL DESARROLLO DE SOFTWARE CON TECNOLOGÍA JAVA</b>		<b>Versión:</b> 1.2
			<b>Fecha:</b> 29/05/2015

```

if (condicion) {
    /* Código de la condicion. */
    ...
}

```

#### 2.18.4. 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. Aquí un ejemplo de comentario de remolque:

```

if (a == 2) {
    return TRUE; /* caso especial */
} else {
    return isPrime(a); /* caso general */
}

```

#### 2.18.5. 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; sin embargo, puede usarse en líneas consecutivas para comentar secciones de código. Ejemplos de los tres estilos:

```

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

```

```

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

```

#### 2.18.6. 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 `/** ... */`, con un comentario por clase, interface o miembro (método o atributo). Este comentario debe aparecer justo antes de la declaración:

```

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

```

Darse cuenta de que las clases e interfaces de alto nivel son están indentadas, mientras que sus

	NORMATIVIDAD		Código:	NRM-STI-601
	ESTÁNDARES DE PROGRAMACIÓN PARA EL DESARROLLO DE SOFTWARE CON TECNOLOGÍA JAVA		Versión:	1.2
			Fecha:	29/05/2015

miembros los están. La primera línea de un comentario de documentación (`/**`) para clases e interfaces no esta indentada, subsecuentes líneas tienen cada una un espacio de indentación (para alinear verticalmente los asteriscos). Los miembros, incluidos los constructores, tienen cuatro espacios para la primera línea y 5 para las siguientes.

Si se necesita dar información sobre una clase, interface, variable o método que no es apropiada para la documentación, usar un comentario de implementación de bloque o de una línea para comentarlo inmediatamente *después* de la declaración. Por ejemplo, detalles de implementación de una clase deben ir en un comentario de implementación de bloque *siguiendo* a la sentencia `class`, no en el comentario de documentación de la clase.

Los comentarios de documentación no deben colocarse en el interior de la definición de un método o constructor, ya que Java asocia los comentarios de documentación con la *primera declaración después* del comentario.

### 2.18.7. Para Tags “javadoc” en los programas:


Se colocan al inicio de cualquier definición de componentes (clase, método, etc.) para indicar sus características más relevantes a manera de documentación y deben indicarse con el siguiente propósito:

Tipo de Tag	Usado para	Propósito
@author name	Interfaces, Clases	Indica el autor de una porción de código.
@version text	Interfaces, Clases	Indica la versión de una porción de código
@param name description	Métodos	Describe los parámetros que se están pasando a un método, incluyendo el tipo / clase que se esta usando.
@return description	Métodos	Describe el parámetro que se esta devolviendo, incluyendo el tipo / clase que se esta usando.
@exception name description	Métodos	Describe la excepción que un método “tira” hacia la clase que lo invoco.
@see ClassName	Clases, Interfaces, Métodos, Atributos	Genera un <i>hiperlink</i> en la documentación a una clase específica.
@see ClassName#functionName	Clases, Interfaces, Métodos, Atributos	Genera un <i>hiperlink</i> en la documentación a método específico en una clase específica.
@deprecated	Interfaces, Clases, Métodos	Indicar que el API utilizado esta <i>deprecated</i> y que no deberá ser utilizado.

## 2.19. Sentencias

### 2.19.1. Sentencias simples

Cada línea debe contener como mucho una sentencia por ejemplo:  
`argv++; argc--; // EVITAR!`

 <small>Organismo Superior de las Coordinaciones del Estado</small>	<b>NORMATIVIDAD</b>		<b>Código:</b> NRM-STI-601
	<b>ESTÁNDARES DE PROGRAMACIÓN PARA EL DESARROLLO DE SOFTWARE CON TECNOLOGÍA JAVA</b>		<b>Versión:</b> 1.2
			<b>Fecha:</b> 29/05/2015

No usar el operador coma para agrupar múltiples sentencias a menos que sea en un “for” en obvias razones por ejemplo:

```
if (err) {
    Format.print(System.out, "error"), exit(1); //MAL!
}
```

### 2.19.2. Sentencias compuestas

Las sentencias compuestas son sentencias que contienen listas de sentencias encerradas entre llaves "{sentencias}". Ver las siguientes secciones para ejemplos.

- 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 principio 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 bugs accidentalmente por olvidar las llaves.

### 2.19.3. Sentencias de retorno

Una sentencia return con un valor no debe usar paréntesis a menos que hagan el valor de retorno más obvio de alguna manera, por ejemplo:

```
return;
return miDiscoDuro.size();
return (tamanyo ? tamanyo : tamanyoPorDefecto);
```

### 2.19.4. Sentencias if, if-else, if else-if else

La clase de sentencias **if-else** debe tener la siguiente forma:

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

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

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


### 2.19.5. 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 las cláusulas de inicialización, condición, y actualización) debe tener la siguiente forma:



 <small>Organismo Superior de las Certificaciones del Estado</small>	NORMATIVIDAD		Código:	NRM-STI-601
	ESTÁNDARES DE PROGRAMACIÓN PARA EL DESARROLLO DE SOFTWARE CON TECNOLOGÍA JAVA		Versión:	1.2
			Fecha:	29/05/2015

**for** (*inicialización; condición; actualización*);

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

#### 2.19.6. Sentencias while

Una sentencia **while** debe tener la siguiente forma:

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

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

```
while (condición);
```

#### 2.19.7. Sentencias do-while

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

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

#### 2.19.8. 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: /* El uso del default es obligatorio */
    sentencias;
    break;
}
```

Cada vez que un caso se propaga (no incluye la sentencia “**break**”), añadir un comentario donde la sentencia “**break**” se encontraría normalmente. Esto se muestra en el ejemplo anterior con el comentario */\* este caso sepropaga \*/*.

Cada sentencia **switch** debe incluir un caso por defecto (**default**).

 <small>Organismo Superior de las Comunicaciones del Estado</small>	NORMATIVIDAD		Código:	NRM-STI-601
	ESTÁNDARES DE PROGRAMACIÓN PARA EL DESARROLLO DE SOFTWARE CON TECNOLOGÍA JAVA		Versión:	1.2
			Fecha:	29/05/2015

El “**break**” en el caso por defecto es redundante, pero prevee que se propague por error si luego se añade otro caso.

### 2.19.9. Sentencias try-catch

Una sentencia **try-catch** debe tener la siguiente forma:

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

Una sentencia **try-catch** debe tener el bloque **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;
}
```

Con Java 7:

```
try {
    sentencias;
} catch (ExceptionClass1 e1, ExceptionClass2 e2,...) {
    sentencias;
} finally {
    sentencias;
}
```

Cuando la sentencia catch involucre multiples excepciones usar el siguiente esquema

```
try {
    ...
} catch (NamingException e) {
} catch (RemoteException e) {
} catch (Exception e){
} finally {
}
```


## 2.20. Espacios en blanco

### 2.20.1. 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 archivo fuente
- Entre las definiciones de clases e interfaces.

 <small>Organismo Superior de las Certificaciones del Estado</small>	<b>NORMATIVIDAD</b>		<b>Código:</b> NRM-STI-601
	<b>ESTÁNDARES DE PROGRAMACIÓN PARA EL DESARROLLO DE SOFTWARE CON TECNOLOGÍA JAVA</b>		<b>Versión:</b> 1.2
			<b>Fecha:</b> 29/05/2015

- 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.

### 2.20.2. Espacios en blanco

Se deben usar espacios en blanco en las siguientes circunstancias:

- Una palabra clave del lenguaje seguida por un paréntesis debe separarse por un espacio. Ejemplo:

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

Notar que no se debe usar un espacio en blanco entre el nombre de un método y su paréntesis de apertura. Esto ayuda a distinguir palabras claves de llamadas a métodos.

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

```
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. Ejemplo:

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

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


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

### 2.21. Manejo de Errores y Validaciones

Se van a considerar dos tipos de validaciones, las del lado del cliente, que se trabajarán con javascript y las validaciones del lado del servidor a través de la lógica de negocio.

### 2.22. Concatenación de cadenas

La concatenación de cadenas String no debe realizarse con la unión de varias cadenas, pues esto consume recursos de memoria innecesariamente. Se debe utilizar la clase StringBuilder para realizar la concatenación.

	NORMATIVIDAD		Código:	NRM-STI-601
	ESTÁNDARES DE PROGRAMACIÓN PARA EL DESARROLLO DE SOFTWARE CON TECNOLOGÍA JAVA		Versión:	1.2
			Fecha:	29/05/2015

Modo incorrecto de concatenar cadenas:

String cadenaFinal = "Esta es " + "una " + "concatenación " + "incorrecta";

Modo correcto de concatenar cadenas:

```
Stringbuilder sb = new StringBuilder ();
sb.append("Esta es ");
sb.append("una ");
sb.append("concatenación ");
sb.append("correcta");
String cadenaFinal = sb.toString();
```

## 2.23. Colecciones

Para el manejo de colecciones de objetos, las siguientes clases deben ser evitadas, y usar las nuevas clases recomendadas por SUN:

No usar estas clases	Utilizar estas clases
Hashtable	HashMap
Vector	ArrayList
Enumeration	Iterator

## 2.24. Métodos que devuelven una lista de objetos

Cuando se codifican métodos que devuelven una lista de objetos, no se debe devolver un tipo "List" o "ArrayList" porque pueden causar confusión o retraso en el programa llamador al tratar de "averiguar" que objeto es el que se está recibiendo. Lo que se debe devolver es un arreglo con el tipo específico de objeto utilizado.

### Ejemplo de método incorrecto:

```
public ArrayList getPaises() {
    PaisBean p1 = new PaisBean();
    PaisBean p2 = new PaisBean();


    ArrayList lista = new ArrayList();
    lista.add(p1);
    lista.add(p2);
    return lista;
}
```

### Ejemplo de método correcto:

```
public PaisBean[] getPaises() {
    PaisBean p1 = new PaisBean();
    PaisBean p2 = new PaisBean();

    ArrayList lista = new ArrayList();
    lista.add(p1);
    lista.add(p2);

    PaisBean[] arregloPais = (PaisBean[]) lista.toArray(new PaisBean[0]);
    return arregloPais;
}
```

	NORMATIVIDAD		Código:	NRM-STI-601
	ESTÁNDARES DE PROGRAMACIÓN PARA EL DESARROLLO DE SOFTWARE CON TECNOLOGÍA JAVA		Versión:	1.2
			Fecha:	29/05/2015

## 2.25. Métodos que reciben parámetros directos

Cuando se codifican métodos que reciben parámetros específicos y que son de poco conocimiento para el programa llamador, estos parámetros deben estar colocado EN LA CLASE A SER LLAMADA como variables ESTATICAS Y DE TIPO CONSTANTE y se recomienda que sean INT o CHAR para que se puedan diferenciar mejor en la codificación con la sentencia “switch”.

Ejemplo incorrecto, en donde se observa que el programa llamador va a tener dificultad para entender “que enviar” en el parámetro de entrada “tipoExpediente”:

```
public class ExpedienteBean {

    public void procesarServicio(String tipoExpediente) {
        //realizar un proceso según el tipo de asiento recibido
        if (tipoExpediente.equals("TRA"))
            //hacer algo
        if (tipoExpediente.equals("PRO"))
            //hacer otra cosa
    }
}
```

Ejemplo correcto, en donde se observa que el programa llamador puede utilizar las constantes que ofrece la clase llamada para identificar con precisión que parámetro debe enviarle:

```
public class ExpedienteBean {

    public static final int TIPO_TRAMITE = 0;
    public static final int TIPO_PROCEOS = 1;

    public void procesarExpediente(int tipoExpediente) {
        //realizar un proceso según el tipo de asiento recibido
        switch (tipoExpediente)
        case TIPO_TRAMITE:
            //hacer algo
        case TIPO_PROCEOS:
            //hacer otra cosa
    }
}
```

## 2.26. Uso de Valores Monetarios


Se debe usar **BigDecimal** para guardar valores monetarios. No usar uso del tipo de dato Float y Double para almacenar valores monetarios, esto es debido a que no guardan correctamente los decimales.

## 2.27. Hábitos de programación

### 2.27.1. 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
```

	NORMATIVIDAD		Código:	NRM-STI-601
	ESTÁNDARES DE PROGRAMACIÓN PARA EL DESARROLLO DE SOFTWARE CON TECNOLOGÍA JAVA		Versión:	1.2
			Fecha:	29/05/2015

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

### 2.27.2. Constantes

Las constantes numéricas (literales) no se deben codificar directamente, excepto -1, 0, y 1, que pueden aparecer en un bucle **for** como contadores.

### 2.27.3. Asignaciones de variables

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

```
fooBar.fChar = barFoo.lchar = '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 lo rechaza)
...
}
```

se debe escribir:

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

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

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

se debe escribir:

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

## 2.28. Hábitos varios

### 2.28.1. 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
```

### 2.28.2. Valores de retorno

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

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

	NORMATIVIDAD		Código:	NRM-STI-601
	ESTÁNDARES DE PROGRAMACIÓN PARA EL DESARROLLO DE SOFTWARE CON TECNOLOGÍA JAVA		Versión:	1.2
			Fecha:	29/05/2015

}

*en su lugar se debe escribir*

**return** *expression*Booleana;

*Similarmente,*

**if** (*condicion*) {

**return** *x*;

}

**return** *y*;

se debe escribir:

**return** (*condicion* ? *x* : *y*);

### 2.28.3. 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;*

### 2.28.4. 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.

### 2.28.5. Para Escritura de “código limpio” en los programas;

- Documentación interna
- Identación (ajustarse al esquema de la herramienta) y párrafos.
- Puntuación de sentencia multi-líneas
- Espacios en blanco
- Uso de la regla de la “lectura en 30 segundos”. Evaluación subjetiva en la que una persona distinta del programador debe entender el concepto general de una porción de código en una sola lectura.
- Orden de las operaciones realizadas en una porción de código.

### 2.28.6. Ejemplo de código Java

**package** pe.gob.osce.seguridad.be.model.vo;

**import** java.io.Serializable;

**public class** UsuarioVO **implements** Serializable {

/\*\*


\*

\*/

**private static final long** *serialVersionUID* = -4956655117298568477L;

**private** String *login*;

**private** String *clave*;

	<b>NORMATIVIDAD</b>		<b>Código:</b> NRM-STI-601
	<b>ESTÁNDARES DE PROGRAMACIÓN PARA EL DESARROLLO DE SOFTWARE CON TECNOLOGÍA JAVA</b>		<b>Versión:</b> 1.2
			<b>Fecha:</b> 29/05/2015

```

    public UsuarioVO() {
    }

    public UsuarioVO(String login, String clave) {
        this.login = login;
        this.clave = clave;
    }

    public String getLogin() {
        return login;
    }


    public void setLogin(String login) {
        this.login = login;
    }

    public String getClave() {
        return clave;
    }

    public void setClave(String clave) {
        this.clave = clave;
    }
}

```



	<b>NORMATIVIDAD</b>		<b>Código:</b> NRM-STI-601
	<b>ESTÁNDARES DE PROGRAMACIÓN PARA EL DESARROLLO DE SOFTWARE CON TECNOLOGÍA JAVA</b>		<b>Versión:</b> 1.2
			<b>Fecha:</b> 29/05/2015

### 3. Aplicaciones empresariales basadas en java

#### 3.1. Utilitarios

##### 3.1.1. Uso de Log's

El framework de log a usar es "Log4j", elegido por su facilidad de uso y por ser de uso libre. Su modo de uso en el sistema se describe a continuación:

##### *Para el desarrollador:*

Para que una clase pueda utilizar la funcionalidad del framework "Log4j", debe tener un atributo "log" de tipo org.apache.log4j.Logger declarado de la siguiente manera:

```
package pe.gob.osce.seguridad.dao;

import org.apache.log4j.Logger;

public class UsuarioDAO {
    private Logger log = Logger.getLogger(UsuarioDAO.class);
    //.....
}
```


Una vez asignado ese atributo, se puede utilizar el objeto "log" para escribir en el log.

##### Uso de Log

Los mensajes a grabar en el log se suscriben a cuatro niveles, según su importancia y gravedad:

Prioridad	Nivel	Descripción
Baja	Debug (Menor)	Usado por los desarrolladores para registrar eventos y/o mensajes a un nivel técnico. Esto incluye mensajes comprensibles únicamente por el propio desarrollador y que faciliten la depuración y/o rastreo de errores en la aplicación.  Ejemplo: "ingreso al método procesarExpediente con el valor "234".
Baja	Info	Se inscriben en este nivel mensajes que puedan ser comprendidos por personal no-técnico y que requieren baja atención. Se incluyen aquí los eventos normales propios del sistema.  Ejemplo: "Ingreso de usuario 48484848 con fecha DATE : 20 may 2013 - 17:15:13"
Media	Warn	Indican mensajes que requieren la atención del usuario y/o encargado del mantenimiento y que quizás genera complicaciones en el futuro.  Ejemplo: "Almacenamiento ha llegado al 80% de su capacidad"
Alta	Error (Mayor)	En este nivel se graban los mensajes que requieren atención inmediata del usuario y/o encargado de la administración de la aplicación para una pronta solución.  Ejemplo: "Servidor CMS de Alfresco no disponible"

Para grabar estos mensajes en el archivo de log, basta con llamar a los métodos de la clase Logger, existen cuatro métodos, uno por cada nivel:

	<b>NORMATIVIDAD</b>		<b>Código:</b> NRM-STI-601
	<b>ESTÁNDARES DE PROGRAMACIÓN PARA EL DESARROLLO DE SOFTWARE CON TECNOLOGÍA JAVA</b>		<b>Versión:</b> 1.2
			<b>Fecha:</b> 29/05/2015

1. log.debug(...)
2. log.info(...)
3. log.warning(...)
4. log.error(...)

Si lo que se desea es grabar en el log un error, incluyendo su información de stacktrace, se debe escribirlo en el log de esta manera:

```
try{
...
}
catch (Exception ex) {
    log.error("mensaje",ex)
}
```

Además la llamada a los métodos de log de nivel DEBUG deben estar precedidos por la llamada al método "isDebugEnabled()" para verificar si es válido enviar el mensaje. La razón de esto es evitar realizar el envío de mensajes a log cuando el nivel "DEBUG" no ha sido habilitado, mejorando así su desempeño.

Ejemplos:

```
public void verificarUsuario(UsuarioSesion us) {
    //.....
    if(log.isDebugEnabled()){
        log.debug("Entrando a metodo verificarUsuario");
    }
    //.....
}
```

El nivel del log para todas las aplicaciones debe estar habilitado por defecto en WARN y subir a INFO o DEBUG cuando sea requerido.

### 3.2. Conectividad

#### 3.2.1. Estándares JDBC

Los nombres de los recursos JDBC en lo que corresponde "DataSource"

**/jdbc/<contexto-sistema>/<usuario>**


**Donde:**

- contexto-sistema** : contexto del sistema al cual pertenecerá
- usuario** : nombre del usuario de base de datos dueño del esquema o con privilegios suficientes para acceder a los objetos requeridos dentro de la base de datos.

### 3.3. Aplicaciones WEB

#### 3.3.1. Recursos WEB estáticos

- Los recursos estáticos de las aplicaciones web deben seguir la siguiente estructura.
  - <context-root>/resources/<contexto-app>/<version>/<grupo>/css
  - <context-root>/resources/<contexto-app>/<version>/<grupo>/images

	<b>NORMATIVIDAD</b>		<b>Código:</b> NRM-STI-601
	<b>ESTÁNDARES DE PROGRAMACIÓN PARA EL DESARROLLO DE SOFTWARE CON TECNOLOGÍA JAVA</b>		<b>Versión:</b> 1.2
			<b>Fecha:</b> 29/05/2015

<context-root>/resources/<contexto-app>/<version>/<grupo>/js

#### Ejemplo:

```
.../portal-static/resources/arbitraje/1.0/gnr/css
.../portal-static/resources/arbitraje/1.0/gnr/images
.../portal-static/resources/arbitraje/1.0/gnr/js
.../portal-static/resources/arbitraje/1.0/controversia/css
.../portal-static/resources/arbitraje/1.0/controversia/images
.../portal-static/resources/arbitraje/1.0/controversia/js
```

### 3.3.2. JavaScript

- Los utilitarios javascript de uso general en la aplicación web debe ser centralizada en archivos de recursos "\*.js" y ubicados en el proyecto de recursos estáticos.
- Las funcionalidades javascript debe ser centralizadas en objetos/clases que usaran como nombre el prefijo "js\*" con notación camel y deberán ser definidos a través de "JavaScript Object Literal".

```
var jsUtilitario = {
    var1: true,
    var2: "very interesting",
    method1: function () {
        alert(this.var1)
    },
    method2: function (msg) {
        alert(msg)
    }
};
```

- Un archivo javascript "\*.js" puede contener varias clases "js\*".

### 3.3.3. Formularios WEB

- Los nombres de los directorios que conforman la URL de los recursos web deben ser escritas con minúscula.
- Los nombres de los recursos web (formularios, servicios REST) escritas con notación CamelCase en su modalidad " lowerCamelCase", ejemplo:

<b>Correcto:</b>	admCatalogoCubso.xhtml
<b>Incorrecto:</b>	AdmCatalogoCubso.xhtml

- Todos los nombres de los JSF Managed Bean deben usar notación CamelCase en su modalidad " lowerCamelCase", ejemplo:

<b>Correcto:</b>	AdmCatalogoCubsoBean (Clase) => admCatalogoCubsoBean (Instancia)
<b>Incorrecto:</b>	AdmCatalogoCubsoBean (Clase) => AdmCatalogoCubsoBean (Instancia)


### 3.3.4. Componentes Plantillas - JSF Facelets

- Los recursos JSF Facelets deben ser ubicados en el directorio "META-INF\resources\" del binario que contendrá estos recursos.

META-INF\resources\<contexto-app>\ui\<grupo>\<composite>

...

META-INF\resources\seace\cubso\ui\gnr\selectorAtributos.xhtml

	<b>NORMATIVIDAD</b>		<b>Código:</b> NRM-STI-601
	<b>ESTÁNDARES DE PROGRAMACIÓN PARA EL DESARROLLO DE SOFTWARE CON TECNOLOGÍA JAVA</b>		<b>Versión:</b> 1.2
			<b>Fecha:</b> 29/05/2015

META-INF\resources\seace\cubso\ui\gnr\selectorCatalogo.xhtml  
META-INF\resources\seace\cubso\ui\cns\filtroCatalogo.xhtml

### 3.3.5. Globalización

- Los nombres de los archivos de internacionalización deben ser estructurado de la siguiente manera:

<b>&lt;contexto-app&gt;-be.properties</b>	Usado en el contexto de los proyectos *.be para tipificación de datos.
<b>&lt;contexto-app&gt;-bp.properties</b>	Usado en el contexto de los proyectos *.bp para mensajes de procesos.
<b>&lt;contexto-app&gt;-ui.properties</b>	Usado en el contexto de los proyectos *.ui para etiquetas de formularios

#### Ejemplo:

seace-logrec-be.properties  
seace- logrec-bp.properties  
seace- logrec-ui.properties

- Los códigos de los archivos deben ser comentados y estructurados según el siguiente formato:

**[seace-logrec-bp.propertie]**

**#Estado de Proceso de Carga Masiva**

**<contexto-app>.be.<tipo/enumerado/constante>.<nombre>.<valor>=<valor>**

seace.logrec.be.type.estadoPcm.solicitado=Solicitado

seace.logrec.be.type.estadoPcm.procesando=Procesando

seace.logrec.be.type.estadoPcm.procesadoExito=Procesado con Exito

seace.logrec.be.type.estadoPcm.procesadoError=Procesado con Error

seace.logrec.be.type.estadoPcm.cancelado=Cancelado

**[seace- logrec -bp.properties]**

**#Carga Masiva**

**<contexto-app>.bp.<proceso/actividad>.<accion/evento>.<atributo>.<resultado>=<valor>**

seace.logrec.bp.cargaMasivaRdc.validar.ruc.noCorresponde=<valor>

seace.logrec.bp.cargaMasivaRdc.validar.titulo.requerido=<valor>

seace.logrec.bp.cargaMasivaRdc.validar.archivo.requerido=<valor>

seace.logrec.bp.cargaMasivaRdc.validar.archivo.nodbf=<valor>

seace.logrec.bp.cargaMasivaRdc.registrar.exito=<valor>

seace.logrec.bp.cargaMasivaRdc.registrar.exito.segunda.parte=<valor>

seace.logrec.bp.cargaMasivaRdc.registrar.exito.tercera.parte=<valor>


seace.logrec.bp.cargaMasivaRdc.publicar.noExiste=<valor>

**[seace- logrec ui.properties]**

**#Registro de Documento**

**<contexto-app>.ui.<proceso/actividad/formulario>.title=<Valor>**

seace.logrec.ui.regDocumentoCompra.title =Valor

	<b>NORMATIVIDAD</b>		<b>Código:</b> NRM-STI-601
	<b>ESTÁNDARES DE PROGRAMACIÓN PARA EL DESARROLLO DE SOFTWARE CON TECNOLOGÍA JAVA</b>		<b>Versión:</b> 1.2
			<b>Fecha:</b> 29/05/2015

```

<contexto-app>.ui.<proceso/actividad/formulario>.<seccion>.<atributo/control>.label=<valor>
seace.logrec.ui.regDocumentoCompra.criterio.tipoContratacion.label=<valor>
seace.logrec.ui.regDocumentoCompra.criterio.fechaEmision.label=<valor>
seace.logrec.ui.regDocumentoCompra.resultado.documento.label=<valor>
seace.logrec.ui.regDocumentoCompra.resultado.fechaEmision.label=<valor>
seace.logrec.ui.regDocumentoCompra.resultado.contratista.label=<valor>

```

- Las declaraciones de los EJB dentro de los JSF Managed Bean debe realizarse de la siguiente manera:

```

@EJB
private transient MiEjbLocal

```