MANUAL DE ESTÁNDARES DE DESARROLLO WEB

MANUAL DE DESARROLLO EN JAVA

Junta de Castilla y León

Coordinador de la Información: Área de Desarrollo y Mantenimiento
Dirección General de Telecomunicaciones y Transportes.

ÍNDICE DE CONTENIDOS

CAPITULO

1 Normativa Java

Este documento pretende describir una serie de estándares basados en la normativa de Sun Microsystems™ para escribir código Java robusto y fácil de mantener.

Se establecerán unas normas a seguir para codificar en Java, se recogerán las convenciones y sugerencias de codificación Java que sigue Javasoft en su código y que recomienda seguir a la comunidad de programadores. Cubre muchos aspectos, desde los nombres de los ficheros, a la organización de éste, comentarios, declaraciones, sentencias, espacios en blanco, convenciones de nomenclatura, prácticas de programación. Se incluye código de ejemplo que ilustra cada uno de los puntos descritos.

Estas convenciones son muy importantes por un gran número de razones, entre ellas:

- El 80% del coste del ciclo de vida de una pieza de software es el mantenimiento
- Raramente un software es mantenido por el autor original
- Las convenciones incorporan legibilidad al código, permitiendo que los ingenieros entiendan el nuevo código mucho más rápidamente
- Si se proporciona el software como un producto, ha de asegurarse que está igual de empaquetado y limpio que cualquier otro que se haya desarrollado.

Elección de la Normativa para Java

El mundo Java está dividido en dos grupos bien diferenciados dependiendo de la normativa que siguen:

- Estándares de codificación de Sun, explicado en el apartado 1.
- Estándares de codificación basados en notación húngara. Se trata de una convención para el nombrado de variables inventada por Charles Simonyi de Microsoft. Permite al desarrollador determinar el tipo de variable simplemente observando el nombre de la misma, puesta que éste está compuesto por un prefijo que describe el ámbito de la variable y otro prefijo que describe el tipo de la variable.

Las ventajas de la notación húngara son claras:

- La lectura del código es más cómoda ya que permite conocer el tipo de la variable y su ámbito.
- La necesidad de *casting* se identifica rápidamente.
- Los errores de tipo y ámbito se descubren mientras se escribe.

Se seleccionan los estándares de codificación de Sun como válidos y no la notación húngara debido a las siguientes razones:

- **Tipos muy definidos** - El lenguaje Java es un lenguaje muy tipado por lo que no se hace necesaria la identificación del tipo de la variable a través del nombre.
- **Métodos poco extensos** - Los métodos en Java tienen muy pocas líneas de código, debido a que la lógica deberá estar bien estructurada haciendo uso de las jerarquías y el concepto de herencia, por lo que la identificación de variables es rápida.
- **Nomenclatura de clases nuevas** - La notación húngara no dice nada del nombrado de variables que no sean tipos básicos. Por ejemplo, si se tiene una clase llamada *Empleado*, lo lógico sería utilizar el prefijo "emp" para las instancias, pero si otro programador desarrolla otra clase que extienda de la anterior y que se llame *SalarioEmpleado*, elegir el prefijo correcto no será una decisión fácil ni inmediata.

- **Mantenimiento del código** - Las ventajas de la notación húngara en el mantenimiento son dudosas. El mayor problema que tiene la notación húngara, desde nuestro punto de vista y debido a lo estricta que es, desde el principio se tiene que tener muy claro el ámbito y tipo de las variables, y ser muy consecuente con este nombrado. Sin embargo bien es sabido que realmente las variables cambian de tipo y, sobre todo, de ámbito respecto a la concepción inicial. Con notación húngara esto supone no sólo tener que cambiar el ámbito y tipo de la variable, sino cambiar el nombre de la misma en todo el código donde ésta aparezca, puesto que sino la notación húngara en vez de ser una ayuda se convertiría en un engaño que supondría un gran riesgo en el mantenimiento.

- **Claridad del código** - Según la normativa de Sun, el nombrado de las variables debe ser una descripción de su funcionalidad o contenido, al tener que añadir tantos prefijos con la notación húngara, el nombre se hace más oscuro (a parte de mucho más engorroso a la hora de escribirlo) y, por tanto, se rompe una de las reglas de Sun.

- **Tiempo de desarrollo** - Se incrementa el tiempo de desarrollo en la mayoría de los casos entre un 15%-20%, y el tiempo no es una variable despreciable en la ejecución de un proyecto. Además será preferible dedicar ese tiempo a un buen diseño del código que no a ser consecuente con un nombrado estricto de las variables.

Hay que tener en cuenta que no emplear la notación húngara no supone dejar las puertas abiertas a cualquier programador para que utilice un nombrado libre, sino que con la notación de Sun ya se está restringiendo este nombrado, como se puede ver en el siguiente punto en el apartado de nombrado. Por lo tanto es importante destacar que siguiendo la notación de Sun nuestro código Java será muy limpio, fácilmente comprensible y mantenible por otros desarrolladores siempre que el diseño del conjunto de clases sea óptimo, y es ahí donde, desde nuestro punto de vista, debemos enfocar todos nuestros esfuerzos.

Javadoc

Definición

Javadoc es una herramienta de generación de documentación automática a partir del **código fuente** de las aplicaciones desarrolladas en lenguaje Java. El resultado de la generación es un conjunto de ficheros en formato HTML que documenta el conjunto de los elementos que aparecen en el código, tanto clases, como sus métodos, campos, y enlaces entre todos ellos. A este conjunto de documentos se le conoce normalmente como *javadoc* de la aplicación que se ha desarrollado.

La herramienta se distribuye como una parte del Java SDK.

Componentes:

- API : com.sun.javadoc
- Aplicación Javadoc

Se propone como norma que los comentarios de documentación sigan siempre el estándar de javadoc. También es importante señalar que cualquier herramienta para la generación de código Java, como por ejemplo JDeveloper trae ya instalado este componente. De forma que es siempre un estándar a seguir para la generación de documentación.

Formato de comentarios de documentación

Los comentarios de documentación (doc comments) se usan para describir el comportamiento externo (Nivel de API) de clases, interfaces, constructores, métodos y campos. Sus delimitadores son /**...*/ y deben aparecer justo antes de la declaración de lo que es descrito. Deben cumplir:

- Los comentarios de clases o interfaces no deben ser sangrados.
- Los comentarios de miembros deben sangrarse 4 espacios.
- La primera línea debe contener solamente el delimitador /**
- El resto de las líneas deben comenzar por asterisco sangrado 1 espacio y separado del texto al menos por otro espacio, excepto la última, la cual debe contener solamente el delimitador final */, sangrado al mismo nivel que las líneas precedentes.
- Los bloques de comentarios de documentación deben estar separados del código precedente por una línea en blanco.
- Nunca deben utilizarse dentro de métodos o constructores, ya que javadoc asocia el bloque de comentario a la declaración inmediatamente posterior.

Ejemplo:

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

Contenido de los comentarios de documentación

La siguiente tabla muestra los elementos que deben tener los comentarios de documentación en cada uno de los componentes públicos o protegidos de los ficheros fuente. Esta lista define los mínimos, pero no limita el uso de otros tags de javadoc:

Elemento	Descripción	Uso
Clases e interfaces		
Descripción	Describe la funcionalidad general de la clase o interfaz, especificando los fundamentos generales para el uso de la misma.	Siempre
@autor	Autor de la clase o interfaz	Siempre
@deprecated	Indica si la clase o interfaz está obsoleta por lo que se debería usar una alternativa.	Cuando proceda
@see NombreClase	Indica una clase o interfaz relacionada con la que se está documentando. El nombre debe estar totalmente cualificado.	Cuando proceda
@see NombreClase#miembro	Indica un método o variable miembro relacionada con la clase que se está documentando. El nombre debe estar totalmente cualificado.	Cuando proceda
Métodos miembro		
Descripción	Explicación del propósito y uso del método.	Siempre
@deprecated	Indica que el método está obsoleto y debe usarse una alternativa	Cuando proceda
@exception nombre descripción	Describe las excepciones que lanza un método. El nombre de la clase debe ser completo. Un tag por excepción.	Cuando proceda
@param nombre descripción	Describe los argumentos del método documentado, incluyendo su tipo o clase. Un tag por parámetro.	Cuando proceda
@return descripción	Describe el valor que devuelve un método, indicando además el tipo o clase del valor y su posible uso.	Cuando proceda
@see NombreClase	Indica una clase o interfaz relacionado con el método que se está documentando. El nombre debe estar totalmente cualificado.	Cuando proceda
@see NombreClase#miembro	Indica un método miembro relacionada con la que está documentando. El nombre debe estar totalmente cualificado.	Cuando proceda
Campos		
Descripción	Indica el propósito del campo	Siempre
@see NombreClase	Indica una clase o interfaz relacionado con el campo que se está documentando. El nombre debe estar totalmente cualificado.	Cuando proceda
@see NombreClase#miembro	Indica un método miembro relacionada con el campo que se está documentando. El nombre debe estar totalmente cualificado.	Cuando proceda

Referencias

- Javadoc Tool Home Page (Sun Microsystems) (http://java.sun.com/j2se/javadoc/)

Normas Generales de Codificación en Java

Sangrado

Unidad de sangrado

La unidad de sangrado está formada por 4 espacios, no permitiéndose el uso de tabuladores[1] (su representación puede variar dependiendo de editores).

Longitud de líneas

Las líneas no deben ocupar más de 80 caracteres. En ciertos terminales o editores no se ven correctamente si su longitud es mayor.

Expresiones en varias líneas

Cuando una expresión es demasiado larga para que ocupe una única línea, su separación en varias líneas debe ajustarse a las siguientes reglas:

- Cortar tras una coma

Ejemplo:

```
metodo(expresion1, expresion2,
     expresion3);
```

- Cortar antes de un operador

Ejemplo:

```
resultado = expresion1 * (expresion2 + expresion3)
                  - expresion4
```

- El corte debe situarse en el menor nivel de profundidad posible

Ejemplo:

```
resultado = expresion1 * (expresion2
          + expresion3) - expresion4 //EVITAR

resultado = expresion1 * (expresion2 + expresion3)
          - expresion4 //CORRECTO
```

[1] La tabulación hace referencia al carácter de tabulación, no al uso de la tecla de tabulación, siempre que ésta escriba espacios y no tabuladores.

- Alinear el principio de la nueva línea con el principio de la subexpresión cortada en la línea anterior

Ejemplo:

```
resultado = expresion1 * (expresion2
                          + expresion3)
```

- Si la regla anterior produce un código confuso debido a que la sangría es demasiado grande, se puede cambiar por sangrado de 8 espacios.

- Para las sentencias condicionales la sangría debería ser de 8 caracteres en lugar de alinearla con el principio de la subexpresión

Ejemplo:

```
  /* En este bloque no se distingue
  * la sentencia de las
  * condiciones
  */
 if (condicion1
     || condicion2
     || condicion3) {
     ejecutar();
  }


 /* En este sí */
 if (condicion1
       || condicion2
       || condicion3) {
     ejecutar();
 }
```

Comentarios

Información general

Existen 2 tipos de comentarios en Java:

- Comentarios de implementación: Aquellos que se usan para aclarar o especificar aspectos referidos a la implementación.
- Comentarios de documentación: Aquellos que se usan para describir el comportamiento externo de una clase, método o campo. Se usan para generar documentación de API mediante el programa JAVADOC (ver apartado 2)

Normas básicas

Antes de mostrar el formato para los comentarios en Java, es conveniente señalar varias reglas básicas que hay que seguir para obtener una buena documentación:

- En caso de querer documentar detalles de implementación de una clase o un método, nunca se deben usar comentarios de documentación. Se deben incluir comentarios de implementación tras la primera línea de la declaración del método o la clase.
- Los comentarios sólo deben contener información relevante para entender el código, evitando incluir información sobre características externas al mismo (ubicación en un directorio, instalación,...).
- Evitar la inclusión de comentarios que queden obsoletos con facilidad
- Evitar la inclusión de comentarios para duplicar información ya presente en el código de forma explícita.
- La frecuencia de comentarios implica a veces código de baja calidad. Si un fragmento de código necesita ser comentado exhaustivamente quizás sea bueno rescribirlo de forma más clara
- Evitar decorar el código. Hay que evitar en todo momento el uso de cajas de asteriscos o florituras de ese tipo, porque el desarrollador debe escribir un código limpio no bonito, además es una pérdida de tiempo que no aporta nada.
- Escribir el comentario antes de escribir el código. Esto ofrece al desarrollador la oportunidad de recapacitar sobré la funcionalidad del código.
- Comentar por qué se hacen las cosas y no qué se está haciendo. Muchas veces es muy sencillo de saber qué está haciendo el código, sin embargo no es sencillo averiguar el porqué, es decir, qué regla de negocio impulsa a hacer, por ejemplo, una comprobación, y eso es precisamente lo que interesa al desarrollador.

Formato de comentarios de implementación

Los comentarios de implementación pueden ser de 4 tipos: bloque, línea, rastreo y final de línea.

- Comentarios de bloque

  Los comentarios de bloque se usan para describir ficheros, métodos, estructuras de datos y algoritmos. Pueden ser usados al principio de cada fichero o antes de cada método, aunque se pueden incluir en otros lugares, dentro de métodos.

  Deben cumplir que:

  - Dentro de un método deben estar sangrados al mismo nivel que el código que describen.
  - Deben estar precedidos por una línea en blanco para separarlos del resto del código.

  Su aspecto debe ser el siguiente:

```
/*
 * Esto es un comentario de bloque.
 */
```

- Comentarios de línea

  Se distinguen varios tipos, aunque para todos ellos se propone la misma notación (//):

  - Comentarios de línea

    Los comentarios de línea deben:
    - Estar sangrados al mismo nivel que el código que les sigue
    - Ser precedidos por una línea en blanco para separarlos del resto del código.
    - Si el comentario no puede incluirse en una única línea, se deben usar bloques (según se describe en el apartado anterior de comentarios de bloque).

    Un ejemplo puede ser:

```
if (condicion) {

    // Hacemos lo que sea
    ....
}
```

- Comentarios de rastreo o fin de línea

Los comentarios de rastreo son comentarios que aparecen en la misma línea de código que describen. Deben cumplir que:

- Si en una porción de código existe más de un comentario de este tipo, deben estar sangrados al mismo nivel
- Deben estar sangrados lo suficientemente lejos de las sentencias para distinguirse claramente.
- Pueden usarse para comentar varias líneas de texto o fragmentos de código.

Ejemplo1:

```
if (a % 2 == 0) {
    return true;        // Ningún par es primo
} else {
    return esPrimo(a); // Sólo ejecutamos si a es impar
}
```

Ejemplo2:

```
if (condicion) {

    // Ejecutamos la sentencia
    ....
} else {
    return false;  // Explicamos por qué
}



//if (condicion) {
//
//    //Esto es una prueba
//    ...
//} else {
//    return false;
//}
```

Declaraciones

Una declaración por línea

Se debe declarar un único componente por línea.

Inicialización

- Las variables deben ser inicializadas en el mismo lugar en el que son declaradas.
- Excepción: Si el valor inicial de una variable depende de una ejecución posterior a la declaración, la inicialización se realizará tras dicha ejecución (no se deben inicializar variables de forma indiscriminada con valores que no tienen sentido).

Posicionamiento

- Las declaraciones deben estar agrupadas al principio de cada bloque (se entiende bloque como el fragmento de código delimitado por {...}).
- Evitar declaraciones locales con el mismo nombre que otras de nivel superior.

Ejemplo:

```
void metodo1() {
    int entero1 = 0;

    if (condicion) {
         int entero2 = 0;
         ...
    }
}
```

Declaraciones de clases y métodos

Se deben cumplir las siguientes reglas:

- No debe existir espacio entre el nombre del método y el paréntesis de apertura de la lista de argumentos.
- La llave abierta debe aparecer al final de la misma línea que la sentencia de la declaración, excepto si la declaración ocupa varias líneas en cuyo caso la llave aparecerá al final de la última línea de la declaración.
- La llave cerrada debe aparecer sola en una línea y sangrada a la altura del inicio de la declaración, excepto si la declaración está vacía, en cuyo caso las llaves abierta y cerrada deben aparecer en la misma línea ({}).

- Si la sentencia arroja excepciones, estas deben estar situadas en la misma línea, excepto si la línea supera los 80 caracteres, en cuyo caso las excepciones deben estar sangradas a la altura de la primera excepción en la lista del throws.
- Las declaraciones de métodos están separadas por una línea en blanco.

Ejemplo:

```
class Ejemplo {
      int x;
      int y;

      Ejemplo(int i, int j) {
          x = i;
          y = j;
      }
      void metodoVacio {}
      void metodoQueArrojaExcepciones() throws Exception1,
                                               Exception2,
                                               Exception3 {
       ...
         }
      }
```

Sentencias

Sentencias sencillas

- Una línea debe contener una única sentencia sencilla

Ejemplo:

```
i++;    //BIEN
j++;    //BIEN
i++; j++;    //MAL
```

Sentencias compuestas

Una sentencia compuesta es una sentencia que contiene un bloque de sentencias sencillas delimitado por {}. (if ... { sentencias }, for .... { sentencias })

- Las sentencias agrupadas deben estar sangradas un nivel más a la derecha que las sentencias compuestas que las contienen.
- La llave abierta debe estar al final de la línea de inicio de la sentencia compuesta.
- Las llaves deben delimitar cualquier sentencia (aunque sea sencilla), cuando formen parte de una estructura de control (tipo if-else o for). De

este modo se evita que, al añadir más sentencias, accidentalmente se introduzcan bugs por olvidar los paréntesis.

Sentencias return

Una sentencia de return no debe usar paréntesis a menos que su uso resalte el valor de retorno.

Ejemplos:

```
return a == b;
return ( (saludo != null) ? saludo : "Hola mundo");
```

if, if-else, if-else if-else

Deben escribirse de la siguiente manera:

```
if (condicion) {
    sentencias;
}

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

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

Sentencias for

- Una sentencia for debe escribirse de la siguiente manera:

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

- Una sentencia for vacía (aquella que no necesita un cuerpo ya que su funcionalidad completa se ejecuta entre la inicialización, la condición y la actualización) debería escribirse:

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

Sentencias while

- Una sentencia while debería escribirse:

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

- Una sentencia while vacía debería escribirse:

```
while (condicion) ;
```

Sentencias do-while

- Una sentencia do-while debe escribirse:

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

Sentencias switch

- Una sentencia switch debe escribirse:

```
switch (condicion) {
case ABC:
    sentencias;
    /* no rompe */
case DEF:
    sentencias;
    break;

case XYZ:
    sentencias;
    break;

default:
    sentencias;
    break;
}
```

- En cada bloque case que no rompa (no tiene break al final) es necesario indicarlo explícitamente para evitar confundirlo con un despiste. Esto se puede hacer escribiendo un comentario del tipo /* no rompe */ en el lugar en el que normalmente debería estar el break.
- Cada sentencia switch debe tener una etiqueta default.

Sentencias try-catch

- Una sentencia try-catch debe escribirse:

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

- Si la sentencia incluye un bloque finally

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

Uso de espacios en blanco

Líneas en blanco

Las líneas en blanco mejoran la lectura del código fuente por un humano. Las reglas de su uso son las siguientes:

- Dos líneas de código en blanco se deben usar en los siguientes casos:
  - Entre secciones de un fichero de código fuente (las secciones son: Comentarios iniciales, sentencias package e import y declaración de clase o interface).
  - Entre declaraciones de clases o interfaces.

- Una línea en blanco debería usarse en los siguientes casos:
  - Entre métodos
  - Entre la declaración de las variables locales de un método y la primera sentencia.
  - Antes de un comentario de bloque o de un comentario de línea.
  - Entre secciones lógicas de un método.

Espacios en blanco

El uso de espacios en blanco cumplirá las siguientes reglas:

- Si a una palabra clave le sigue un paréntesis, estos deben estar separados por un espacio en blanco.
- No se debe usar un espacio en blanco entre el nombre de un método y el paréntesis que abre la lista de argumentos. Esto diferencia las palabras clave de las llamadas.
- Tras las comas que separan argumentos en las listas debe aparecer un espacio en blanco.
- Se deben usar para separar operadores binarios de sus operandos.

Ejemplo:

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

- Nunca se deben usar para separar operadores unitarios de sus operandos.

Ejemplo:

```
a++;
b++;
```

- Las expresiones en una sentencia for deben estar separadas por espacios.

Ejemplo:

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

- Los operadores de casting y su objeto deben estar separados por espacios.

Ejemplo:

```
byte b = (byte) x;
```

Nombrado

A continuación se muestra el estándar para el nombrado de objetos que propone Sun. Se ha incluido una norma específica para el nombrado de variables que se correspondan con un campo de la base de datos que no es una norma de Sun específica, pero que consideramos muy útil para la Junta de Castilla y León.

Tipo de identificador	Reglas de nombrado	Ejemplos
Paquetes	Los nombres de los paquetes deben estar escritos en minúscula.	`com.sun.misc` `java.util`

Clases	Los nombres de las clases deben estar formados por una o varias palabras, con la primera letra de cada una en mayúsculas y el resto en minúsculas. Se deben evitar acrónimos excepto si su uso está más extendido que el de la forma larga (URL, HTML,...).	`class Documento`
		`class ClassLoader`
		`class URLConnection`

Entidades	Son clases que representan tablas de la base de datos. El nombre de la entidad será el de la tabla sin underscores y siguiendo la norma para genérica para clases.	`Nombre de la tabla:`
		`COMU_SITEMAS`
		`Nombre de la clase:`
		`ComuSistemas`

Interfaces	Siguen las mismas reglas que las clases.	`interface SQLData`

Métodos	Los métodos deben expresar acciones, por lo que deben ser verbos en infinitivo acompañados normalmente del objeto/objetos de la acción o de un adjetivo que los cualifique. La primera palabra estará escrita en minúsculas y el resto con la primera letra en mayúsculas.	`mostrarDetalleCliente();`
		`editarFormulario();`
		`abrirMinimizado();`

Variables	Los nombres de las variables pueden componerse de una o varias palabras con la primera de ellas en minúsculas y el resto con la primera letra en mayúscula. El nombre de una variable debe tener un uso mnemónico, de forma que una lectura ocasional permita intuir su función. El nombre debe ser además conciso, aunque deben evitarse los nombres de un solo carácter, excepto en las variables de uso temporal y función muy definida (índices, auxiliares,...)	`int i;` `String nombreCliente;`

Variables de campos de base de datos	Los nombres de las variables que mapeen campos de la base de datos, deberán ser un reflejo lo más exacto posible del nombre que tienen en la tabla.
	Se eliminarán los underscores y a partir de ahí se seguirá la notación general para variables.
		Nombre del campo:
		`URL_ENLACE`
		`VARCHAR2(200)`
		`A_TEXTO_SOLICITUD`
		`VARCHAR2(100)`
		`F_TIMESTAMP DATE`
		Nombre de la variable:
		`String urlEnlace;`
		`String aTextoSolicitud;`
		`Date fTimestamp;`

Constantes	Los nombres deben estar escritos en letra mayúscula y si se componen por varias palabras éstas deben estar separadas por guiones bajos (_).	`static final int` `NUM_CARACTERES = 4;`

Estructura de los ficheros fuente

Un fichero fuente debe cumplir las siguientes reglas:

- Debe contener una única clase o interfaz público.
- Si la clase pública tiene asociadas clases o interfaces privados, estos pueden estar en el mismo fichero que la clase pública, estando ésta siempre en primer lugar.
- El orden de las secciones debe ser el siguiente:
  - Comentarios iniciales
  - Sentencias de paquete y de import.
  - Declaraciones de clase e interfaz.

Comentarios iniciales

Todos los ficheros deben comenzar con un comentario de bloque en el que se listarán: el nombre de la clase, información acerca de la versión, fecha y copyright.

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

Sentencias de paquete e import

En primer lugar, tras el comentario inicial y separado por dos líneas, se debe escribir la sentencia package. Separada de ésta por una línea en blanco seguirá la lista de imports. Es buena práctica especificar el nombre de la clase en el import, para conocer a simple vista qué clases exactamente se están usando.

Declaración de clase o interfaz

A continuación se definen las partes que debe contener una declaración de clase/interfaz y el orden en el que deben aparecer.

	Parte de la declaración	**Comentario**
1	Comentario de documentación de la clase.	Ver apartado Javadoc para más información.
2	Sentencia de clase o interfaz	
3	Comentario de implementación de la clase	Debe contener la información sobre los detalles de implementación de la clase.
4	Variables de clase	Deben ordenarse entre sí según su visibilidad (de mayor a menor)
5	Variables de instancia	Deben ordenarse entre sí según su visibilidad (de mayor a menor)
6	Constructores	
7	Métodos	Deben ir agrupados según funcionalidad

Referencias

- Java Code Conventions (Sun Microsystems)
- Java Coding Standards (Scott Ambler)
- Thinking in Java 2nd Edition (Bruce Eckel)

Recomendaciones Generales de Codificación

Nombrado de métodos clase

Getters

Podemos denominar *Getters* a aquellos métodos que devuelven el valor de un campo. Normalmente se utilizará el prefijo "get" más el nombre del campo del que se está retornando el valor. Si el campo es de tipo boolean se puede utilizar el prefijo "es" o "tiene". Ejemplo:

```
getNombre()
getDireccion()
esValido()
tieneEmpleados()
```

Setters

Se denominan Setters a los métodos que modifican el valor de un campo. Se antepone el prefijo "set" al nombre del campo. Ejemplo:

```
setNombre()
setDireccion()
```

Contadores

Los contadores tendrán como sufijo el nivel de anidamiento del bucle sobre el que actúan, empezando por 1. Se pueden utilizar letras como i, j, k, aunque es recomendable usar un nombre del tipo "contador", "counter", "index", para facilitar su localización.

Variables auxiliares

Las variables auxiliares pueden tener un prefijo significativo que facilite su búsqueda e identificación, del tipo "tmp", "aux".

Llamadas a funciones del sistema

Todas las aplicaciones Java tienen una instancia de la clase Runtime que permitirá interaccionar con el sistema operativo donde la aplicación esté ejecutándose. Esto permitirá hacer llamadas a funciones del sistema a través del método exec(...).
Por política de seguridad, se establecerá como norma obligatoria no utilizar esta clase para hacer llamadas al sistema. Si en algún caso fuera necesario, deberá ser justificado y documentado, solicitando su validación al área competente.

Ejemplo:
Copiar un fichero con java.lang.Runtime:

```
try {
        Runtime.getRuntime().exec("cmd /c copy
                        c:\\fn d:\\fn.bak ");
}catch (IOException e) {
    ...
}
```

Copiar un fichero con java.io.File:

```
FileInputStream fileIn   = new FileInputStream( fn );
FileOutputStream fileOut = new FileOutputStream( fn + ".bak" );
int i;
while( true ){
        i = fileIn.read();
        if( i == -1 )break;
        fileOut.write(i);
}
fileIn.close();
fileOut.close();
```

Uso de this / super

Al acceder a variables de instancia de una clase, se debe utilizar la palabra clave ***this*** para hacer referencia a las variables de la propia clase.
Ejemplo:

```
public class MiClase {
    int i;
    public MiClase() {
        i = 10;
    }
    // Este constructor establece el valor de i
    public MiClase( int valor ) {
        this.i = valor;  // i = valor
    }
    public void Suma_a_i( int j ) {
        i = i + j;
    }
}
```

Si se necesita llamar al método padre dentro de una clase que ha reemplazado ese método, se deberá utilizar la palabra clave ***super*** para hacer referencia al método padre.
Ejemplo:

```
import MiClase;
public class MiNuevaClase extends MiClase {
    public void Suma_a_i( int j ) {
        i = i + ( j/2 );
        super.Suma_a_i( j );
    }
}
```

Gestión de memoria

En Java, como norma general, no será necesario realizar tareas de asignación, liberación u otro tipo de gestión de memoria. La asignación de memoria se realiza automáticamente, y la liberación la gestiona el *garbage collector*.

En algunos casos se recomienda tener un cuidado especial en el uso de la memoria, y más concretamente en:

- Uso de las clases Vector y ArrayList. Cuando se finaliza el uso de una instancia de alguno de las dos clases, el *garbage collector* la libera, pero no libera los objetos contenidos en ella. Para ello es necesario invocar el método que los vacía (*removeAllElements()*).
- Uso de métodos nativos en un entorno multithread. Se han visto casos de mal funcionamiento del *garbage collector* en estas situaciones, ya que no libera la memoria de objetos que contengan métodos nativos, después de finalizar el thread.

Orden de los métodos en una clase

Una forma de que las clases sean más fáciles de entender es escribirlas de una forma consistente. Lo más normal en Java es declarar una clase donde los métodos sigan el orden de los de más visibilidad a los de menos:

```
constructors
finalize()
public métodos
protected métodos
private métodos
private campos
```

Orden de las variables en una clase

Seguirán también el orden de las de más visibilidad a los de menos:

```
public variables
protected variables
private variables
```

Alineación en asignaciones

En caso de que haya varias asignaciones en líneas de código consecutivas, se pueden alinear los = verticalmente, de la siguiente manera:

```
int numeroCliente = 0;
int aux1          = -1;
```

Sustitución de caracteres no ASCII en nombrado

Original	Reemplazado
ñ	Nn
Ñ	NN
á	A
é	E
í	I
ó	O
ú	U
Á	A
É	E
Í	I
Ó	O
Ú	U
ü	U
Ü	U

Acceso público a variables estáticas y de instancia

No se debe dar acceso público a variables si no existe una buena razón para ello. En general ninguna variable debe tener acceso público excepto en casos, como por ejemplo, si la clase tiene la misma función que un “struct” de C/C++.

Referencia a variables o métodos estáticos

Se debe evitar hacer referencia a una variable o método estático usando el nombre de una instancia del mismo. En su lugar se debe usar el nombre de la clase.

Constantes

Las constantes numéricas no deben escribirse en el código directamente, excepto -1, 0, 1.

Asignaciones de variables

Evitar múltiples asignaciones en una única sentencia:

```
x = y = 1;    //Evitar
x = 1;        //Correcto
y = 1;

x = (a = b + c) + r;    //Evitar
a = b + c;              //Correcto
x = a + r;
```

Referencias

- Java Code Conventions (Sun Microsystems)
- Java Coding Standards (Scott Ambler)
- Writing Robust Java Code (AmbySoft Coding Standards for Java v17.01d)

Servidor de aplicaciones Tomcat de Apache

Introducción

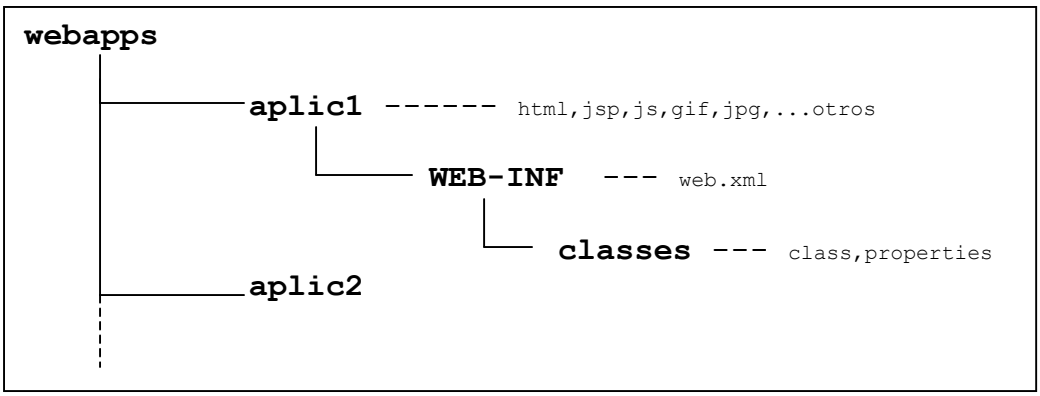
Tomcat es la implementación de referencia para las tecnologías Java Servlets y JavaServer Pages. Desarrollado bajo el proyecto Jakarta Project en la Apache Software Foundation, es gratuito y en formato open-source. Tomcat permite al usuario desarrollar y ejecutar sus servlets y sus páginas JSP bajo los estandares marcados por las especificaciones, y asegurar la compatibilidad con cualquier otro entorno que sea compatible con esas especificaciones.

Además, el servidor Tomcat será la base de la nueva versión del Oracle9iAS Application Server, sustituyendo de esta forma el actual servidor JServ que incorpora el iAS. Es por ello la necesidad de incluir en este documento una descripción del Tomcat y sus formas de uso.

Desarrollo de aplicaciones bajo Tomcat

Las tecnologías Java Servlets y Java Servler Pages están lo suficientemente estandarizadas para que los desarrollos puedan funcionar en distintos motores de servlets. La única particularidad es la estructuración de los componentes desarrollados a lo largo de los directorios del servidor, que hace que las aplicaciones no sean portables entre distintos servidores sin una adaptación previa de la estructura de directorios a los mismos.

En el caso de Tomcat, al ser el servidor de referencia, esa estructura es común a otros servidores, aunque no está estandarizada. La estructura se basa en la centralización de las aplicaciones en un mismo directorio, ***webapps***, que agrupa distintos subdirectorios, cada uno correspondiente a una aplicación, que a su vez mantienen entre ellos una estructura común:



Veamos cada una de las partes en detalle:

- ***Webapps*** – Directorio que agrupa las distintas aplicaciones web.

- ***Applic1*** – Directorio de una aplicación web que contiene todos los componentes propios de esa aplicación. Bajo este directorio se agrupan todos los componentes accesibles directamente desde el navegador del cliente (componentes estáticos como páginas html, ficheros js, gif, jpeg, recursos estáticos como ficheros pdf, docs,... y componentes dinámicos como páginas jsp)

- ***WEB-INF*** – Directorio que contiene todos aquellos recursos que no deben ser directamente accesibles desde el navegador del cliente. En este directorio se sitúa el fichero ***web.xml***, que contiene la configuración propia de la aplicación web (este fichero se describe más adelante, en el apartado "Ficheros de configuración de Tomcat").

- ***WEB-INF/classes*** – Directorio que contiene todos aquellos ficheros accesibles por el cargador de clases de la máquina virtual java. Este directorio se incluye automáticamente en el CLASSPATH del servidor Tomcat, sin necesidad de incluirlo en el entorno de arranque del servidor. Entre estos ficheros se encuentran los .class y los .properties. Aquí se sitúan los servlets (ficheros .class) y otras clases adicionales de la aplicación, sin que necesariamente tengan que ser servlets.

Estructura de directorios

Una vez conocida la estructura de las aplicaciones web, necesaria para poder desarrollar, veamos la estructura del resto de los directorios de Tomcat:

Nombre directorio	Descripción
bin	Contiene los scripts de arrancar/parar
conf	Contiene varios ficheros de configuración incluyendo server.xml (el fichero de configuración principal de Tomcat) y web.xml que configura los valores por defecto para las distintas aplicaciones desplegadas en Tomcat.
doc	Contiene varia documentación sobre Tomcat
lib	Contiene varios ficheros jar que son utilizados por Tomcat. Sobre UNIX, cualquier fichero de este directorio se añade al classpath de Tomcat.
logs	Aquí es donde Tomcat sitúa los ficheros de log.
webapps	Contiene las aplicaciones Web.

Adicionalmente Tomcat creará los siguientes directorios:

Nombre directorio	Descripción
work	Generado automáticamente por Tomcat, este es el sitio donde Tomcat sitúa los ficheros intermedios (como las páginas JSP compiladas) durante su trabajo. Si borramos este directorio mientras se está ejecutando Tomcat no podremos ejecutar páginas JSP.
classes	Podemos crear este directorio para añadir clases adicionales al classpath. Cualquier clase que añadamos a este directorio encontrará un lugar en el classpath de Tomcat.

Scripts de Tomcat

Tomcat es un programa Java, y por lo tanto es posible ejecutarlo desde la línea de comandos, después de configurar varias variables de entorno. Sin embargo, configurar cada variable de entorno y seguir los parámetros de la línea de comandos usados por Tomcat es tedioso y propenso a errores. En su lugar, el equipo de desarrollo de Tomcat proporciona unos pocos scripts para arrancar y parar Tomcat fácilmente.

Nombre script	Descripción
tomcat	El script principal. Configura el entorno apropiado, incluyendo CLASSPATH, TOMCAT_HOME y JAVA_HOME, y arranca Tomcat con los parámetros de la línea de comando apropiados.
startup	Arrancar tomcat en segundo plano. Acceso directo para tomcat start
shutdown	Para tomcat (lo apaga). Acceso directo para tomcat stop;

Ficheros de configuración de Tomcat

La configuración de Tomcat se basa en dos ficheros:

1. **server.xml** - El fichero de configuración global de Tomcat.
2. **web.xml** - Configura los distintos contextos en Tomcat.

server.xml

server.xml es el fichero de configuración principal de Tomcat. Sirve para dos objetivos:

- Proporcionar configuración inicial para los componentes de Tomcat.
- Especifica la estructura de Tomcat, lo que significa, permitir que Tomcat arranque y se construya a sí mismo ejemplarizando los componentes especificados en server.xml.

web.xml

A modo de introducción, veamos un ejemplo de fichero web.xml con una breve descripción de sus partes:

- Inicio y final del fichero. Todas las etiquetas relativas a la configuración deberán estar contenidas por estas etiquetas, y si no hubiera configuración necesaria en la aplicación, las etiquetas no contendrán nada:

```
<web-app>

<!—Configuración de la aplicación-->

</web-app>
```

- Configuración de los servlets. Según los campos que se documenten :

```
<servlet>
         <servlet-name>nombre_servlet</servlet-name>
         <servlet-class>clase_del_servlet</servlet-class>
         <init-param>
            <param-name>nombre_parametro1</param-name>
                              <param-value>valor_parámetro1</param-
                value>
                        </init-param>
                --> 
    </servlet>
```

- Mapeo de los servlets. Configura el URI con el que responden los servlets a las peticiones del cliente :

```
<servlet-mapping>
        <servlet-name>nombre_servlet1</servlet-name>
        <url-pattern>/uri_servlet1</url-pattern>
</servlet-mapping>
<servlet-mapping>
        <servlet-name>nombre_servlet2</servlet-name>
        <url-pattern>/uri_servlet2/*.extension</url-pattern>
 </servlet-mapping>
```

- Configuración de seguridad :

```
<security-constraint>
         <web-resource-collection>
         <!—Configuración de recursos web de seguridad→
         </web-resource-collection>
         <auth-constraint>
         <!—Configuración de opciones de autentificación→
         </auth-constraint>
</security-constraint>
```

- Otras opciones: Existen otras opciones de configuración que permiten adaptar el servidor Tomcat a las necesidades de la aplicación. Podemos encontrar una detallada descripción de web.xml y la estructura de la aplicación web (incluyendo la estructura de directorios y su configuración) en los capítulo 9, 10 y 14 de la Servlet API Spec en la site de Sun Microsystems.

Wars en Tomcat

Toda aplicación en Tomcat se encuentra agrupada en archivos WAR ("Web-ARchives"), la estructura de un WAR es definida por Sun, la cual **debe ser** implementada en cualquier producto de "Servlet Engine"(Web-Container).

La instalación por defecto de Tomcat contiene varios archivos WAR, con ejemplos y documentación, que se encuentran en el directorio /webapps.

La estructura de un Archivo WAR es idéntica a la aplicación que contiene, solo que empaquetada en un archivo comprimido:

- **/ *.html *.jsp *.css** : Este directorio base contiene los elementos que comúnmente son utilizados en un sitio, Documentos en HTML , JSP's , CSS("Cascading Style Sheets") ,etc.
- **WEB-INF/web.xml** : Contiene elementos de seguridad de la aplicación así como detalles sobre los Servlets que serán utilizados dentro de la misma.
- **/WEB-INF/classes/** : Contiene las clases Java adicionales a las del JDK que serán empleadas en los JSP's y Servlets
- **/WEB-INF/lib/**: Contiene los JAR's que serán utilizados por su aplicación.

Este tipo de estructura permite portabilidad a las diversas aplicaciones que son desarrolladas en Java, esto es , si se genera una aplicación para Foros o transacciones financieras estas se pueden contener en un archivo WAR, concentrando cualquier clase Java dentro de /WEB-INF/classes, parámetros específicos en /WEB-INF/web.xml , archivos JAR's en /WEB-INF/lib/ y los diversos JSP's,CSS y documentos HTML bajo el directorio raíz del archivo WAR.

Instalación de Tomcat en el Apache Web Server

El servidor Tomcat está desarrollado como motor de servlets, lo cual garantiza un correcto funcionamiento de los servlets y las páginas jsp de las aplicaciones web. Pero en casi todos los casos las aplicaciones contienen también un gran número de recursos estáticos, como son páginas html, imágenes,... que el Tomcat gestiona de una forma demasiado lenta, ya que no está preparado para ello.

En los casos en que sea necesario usar Tomcat como motor de servlets en un entorno operacional, se puede configurar una arquitectura en la que los recursos estáticos sean gestionados por un servidor http, que proporcionará la flexibilidad, robustez y rapidez de la que el Tomcat carece.

Tomcat está preparado para su integración con los principales servidores http disponibles (Apache Web Server, Internet Information Server y Netscape Enterprise Server) aunque se obtienen los mejores resultados usándolo conjuntamente con el Apache Web Server.

Actualmente la integración de Tomcat con Apache es posible por medio de dos módulos de Apache: mod_jk y mod_jserv.

- El adaptador para jserv, como el proyecto JServ, actualmente se ha abandonado, y solo sigue abierto para tareas de mantenimiento. Es un adaptador demasiado complejo, anticuado y está solo preparado para su uso en Apache.

- El adaptador para jk es mucho más rápido, más sencillo de configurar y es compatible con los principales servidores http.

Para la instalación en detalle del módulo mod_jk se puede consultar el documento "tomcat-apache-howto.html" contenido en la documentación instalada en Tomcat por defecto.

Recomendaciones Generales para Programación JDBC

- En ningún caso un applet deberá realizar una conexión jdbc directa.
- Utilizar variables de substitución (variables Host) para expresar una cláusula SQL. Ejemplo:

```
PreparedStatement pstmt = con.prepareStatement("UPDATE EMPLOYEES
                      SET SALARY = ? WHERE ID = ?");
pstmt.setBigDecimal(1, 153833.00)
pstmt.setInt(2, 110592)
pstmt.execute();
```

- Utilizar órdenes PreparedStatement en lugar de Statement.
- Liberar en todos los casos los recursos abiertos (ResultSet, Statement, PreparedStatement y Connection).
- No utilizar nunca la cláusula « select * ».
- Utilizar el commit explícito « setAutoCommit(false) » para las órdenes de actualización.
- Componer las cláusulas SQL con la ayuda de un StringBuffer cuando sea necesario.

  Ejemplo:

```
StringBuffer query = new StringBuffer("select ");
query.append("c_tema, ");
query.append("d_tema, ");
query.append("b_borrado ");
query.append("from comu_noticias where c_tema = ?");
```

- Utilizar la API executeBatch() para reagrupar las órdenes de actualización de un tratamiento elemental compuesto por múltiples actualizaciones. Llevar a cabo las actualizaciones después de cada llamada.
- Para tratamientos de actualización masivos (tipo batch), construir lotes de tamaño « razonable » (20 a 50 órdenes de actualización).

Convenciones para JDeveloper

Introducción

Oracle JDeveloper™ provee un completo ambiente Java para desarrollar y desplegar aplicaciones en el rango que abarca desde clientes Java y HTML hasta componentes de negocio basados en servidor para la plataforma de computación Internet.

Oracle JDeveloper™ con Oracle Business Components for Java es una herramienta para el desarrollo de aplicaciones completamente funcional que ofrece soporte integrado para construir aplicaciones de comercio electrónico (e-business) end-to-end para Internet. Oracle JDeveloper™ ofrece un ambiente integrado para que los desarrolladores elaboren, depuren, desplieguen, reutilicen y customicen en forma productiva aplicaciones Java y XML multicapa basadas en componentes.

Son muchos los elementos disponibles en JDeveloper, por lo que se hace necesario un nombrado consistente y completo.

Los estándares de codificación definidos anteriormente para Java son y deben ser totalmente aplicables para JDeveloper, por lo tanto en este apartado sólo se añadirán convenciones para el nombrado de elementos propios de JDeveloper que no son elementos puros de Java.

Ficheros

JDeveloper utiliza una estructura múltiple de directorios en el disco duro para almacenar código Java. Existen estructuras separadas para el código fuente, código compilado, documentación y ficheros HTML.

JDeveloper siempre mostrará un nombre por defecto (ejemplo: Untitled.jws) la primera vez que se salve el fichero, por lo que el desarrollador ha de asegurarse de cambiar este nombre por otro descriptivo del contenido del fichero. En el caso de ficheros Java, éste tendrá como nombre el de la clase, por lo tanto nos remitimos a las normas Java para su nombrado.

A continuación se muestra una tabla con los diferentes tipos de ficheros con los que nos podemos encontrar:

FILES TYPE	EXTENSION
Business Components	.xml / .java
Código Java	.java
Conexiones y otras propiedades	.properties
Ficheros HTML y XML	.html / .xml respectivamente
Java Server Pages (JSPs)	.jsp
Librerías	.jar / .zip
Paquetes	Ninguna extensión
Proyectos	.jpr
Workspaces	.jws

Workspaces

Estos ficheros se deberán nombrar con un nombre descriptivo de su contenido y el sufijo WS.
Ejemplo: EmpDeptWS
Este sufijo será de mucha utilidad cuando estamos tratando con una lista larga de ficheros.

Los ficheros de los workspaces se pueden almacenar donde se quiera, pero se deberán almacenar, como norma, en el mismo directorio que el código fuente, y es recomendable nombrarle igual que el directorio con el sufijo WS y la extensión .jws

Proyectos

Estos ficheros se nombrarán con un nombre descriptivo y el sufijo PRJ.
Ejemplo: EmpDeptPRJ

Librerías y Paquetes

Las librerías no son más que una colección de paquetes almacenados en un fichero comprimido de extensión .zip o .tar. No existe ninguna convención para el nombrado de estas librerías.

En JDeveloper se puede dar a cada librería un nombre de fichero y un alias que será más fácil de leer. Mientras que el nombre del fichero tiene que seguir las convenciones para el nombrado de ficheros del sistema operativo, el alias no. Además el alias puede englobar más de una librería, permitiendo almacenar en una librería lógica un grupo de ficheros físicos.

Los paquetes se corresponden con directorios (o carpetas) de la estructura de ficheros del sistema operativo y normalmente no llevan sufijos.

Nombrado Ficheros HTML

JDeveloper genera los siguientes ficheros HTML:

- **HTML para llamar a Applets**. Estos ficheros ejecutan un applet en el navegador. Cuando se usa el Applet Wizard de JDeveloper, los ficheros HTML generados son automáticamente nombrados utilizando la siguiente regla de concatenación:

  Nombre del Proyecto + "_" + Nombre de la clase del Applet
  Ejemplo: *BusPRJ_BusApplet.html*

  Estos ficheros son almacenados en el directorio raíz HTML, que está organizado de forma muy diferente a otros directorios en JDeveloper. No está basado en paquetes sino que está organizado por proyectos.

- **Javadoc**. JDeveloper nombra a los ficheros generados por Javadoc con el mismo nombre que las clases que representan con extensión .html.

Business Components

Existen diferentes elementos:

- **Entidades.** La convención para el nombrado es la misma que se definió en el apartado de Normas Generales de Codificación para Java.

- **Atributos.** La convención para el nombrado es la misma que se definió en el apartado de Normas Generales de Codificación para Java.

- **Asociaciones.** Su nombrado será el resultado de la concatenación del nombre de la constraint de la clave foránea y la palabra "Assoc". Por ejemplo: *EmpDeptFkAssoc*.

- **Dominios.** Los dominios viene definidos por el usuario. Cuando se crea un dominio, se deberá reemplazar el nombre "Domain" que viene por defecto por otro descriptivo. Se deberá escribir la primera letra en mayúsculas y añadir el sufijo Domain. Por ejemplo: *MaxSalaryDomain*. El dominio tendrá dos ficheros asociados, uno (.xml) para la definición de los datos y otro (.java) para el comportamiento. Estos ficheros son nombrados automáticamente con el nombre del dominio. Por ejemplo: MaxSalaryDomain.xml y MaxSalary.java.

- **Módulos de aplicación.** Su nombrado será el resultado de la concatenación del nombre del paquete y la palabra "Module". Por ejemplo: *myempappletModule*.

- **Vistas.** Su nombrado será el resultado de la concatenación del nombre de la entidad y la palabra "View". Por ejemplo: *EmpleadoView*.

- **Links a vistas.** Su nombrado sigue la misma convención que las asociaciones pero añadiendo el sufijo "Link" en vez de "Assoc".

UI Components

Para este tipo de componentes es interesante utilizar un nombrado que concatene la descripción del objeto con el tipo de objeto como sufijo.

Por ejemplo, si vamos a añadir un java.awt.Button que va a tener la funcionalidad de salir de la aplicación, éste se podrá nombrar como salirButton. SI utilizamos Swing en vez de awt, el componente es javax.swing.JButton, por lo que el objeto se nombraría como salirJButton.

En algunos tipos de componentes se podrá utilizar una abreviación del tipo si éste es muy largo, por ejemplo si se quiere incluir un GridControl para la tabla Empleados, el objeto se nombraría como empleadosGrid y no hace falta nombrarlo como empleadosGridControl, que sería demasiado largo.

Referencias

- Oracle JDeveloper 3 Handbook

CAPITULO

2 Normas Generales de Codificación JSP

La intención del documento es la de presentar una serie de normas y recomendaciones acerca de la codificación JSP. No consistirá en un catálogo de los tags propios del lenguaje JSP, y no pretende por tanto servir como referencia de la sintaxis del lenguaje como tal. Se presupone por parte del lector un conocimiento básico de la sintaxis JSP.

La mayoría de los desarrolladores JSP deberán seguir la misma regla a la hora de escribir su código :

"Una página JSP se deberá parecer a una página HTML con poco contenido dinámico y no a un programa Java con alguna marca HTML".

Hay dos reglas más que parten de esta primera premisa. Todo ello nos llevará a una clara separación de la presentación y contenido que es lo que se pretende:

- Asegurarse de que la lógica de negocio está donde tiene que estar. Los scripts JSP, es decir, el código Java embebido en la página JSP no sólo obscurece la página, sino que no es reutilizable y no sigue las reglas de una modelo orientado a objetos.

- Asegurarse de que la presentación está donde tiene que estar. La presentación forma parte de las JSPs, pero hay que evitar generar HTML con código Java.

Introducción

JavaServer Pages™ (JSP™) combinan HTML con fragmentos de Java para producir páginas web dinámicas. Cada página es automáticamente compilada a servlet por el motor de JSP , en primer lugar es recogida y a continuación ejecutada.

JSP tiene gran variedad de formas para comunicarse con las clases de Java, servlets, applets y el servidor web; por esto se puede aplicar una funcionalidad a nuestra web a base de componentes.

Una página JSP es archivo de texto simple que consiste en contenido HTML o XML con elementos JSP. Cuando un cliente pide una página JSP del sitio web y no se ha ejecutado antes, la página es inicialmente pasada al motor de JSP, el cual compila la página convirtiéndola en Servlet, la ejecuta y devuelve el contenido de los resultados al cliente.

Es posible ver el código del servlet generado, este código debe estar en el directorio que se informa en la estructura de directorios del servidor.

Si nos fijamos en este archivo podemos encontrar las siguientes clases:

- JSPPage
- HttpJspPage

Ellas definen la interface para el compilador de páginas JSP.

Nos encontramos también tres métodos:

- JspInit()
- JspDestroy()
- _jspService(HttpServletRequest request, HttpServletResponse response)

Los dos primeros métodos pueden ser definidos por el autor de la página JSP, pero el tercer método es una versión compilada de la página JSP, y su creación es responsabilidad del motor de JSP.

El código fuente de una página JSP incluye:

1. Directivas: Dan información global de la página, por ejemplo, importación de estamentos, página que maneja los errores o cuando la página forma parte de una sesión, en el ejemplo anterior informamos del tipo de script de Java.
2. Declaraciones: Sirven para declarar métodos y variables.
3. Scripts de JSP: Es el código Java embebido en la página.

4. Expresiones de JSP: Formatea las expresiones como cadenas para incluirlas en la página de salida. Estos elementos siguen una sintaxis como XML, así se obtiene un significado con una presentación totalmente separada de la lógica. Un buen ejemplo es `<jsp:useBean.../>` el cual busca o crea una instancia de un bean. Con el mecanismo de extensiones de tag se tiene la posibilidad de definir tags con acciones similares y poner la funcionalidad en una librería de tags.

Ventajas ofrecidas por JSP

Los beneficios ofrecidos por JSP como alternativa a la generación de contenido dinámico para la web se resumen en los siguientes apartados.

Mejoras en el rendimiento

- Utilización de procesos ligeros (hilos Java) para el manejo de las peticiones.
- Manejo de múltiples peticiones sobre una página .jsp en un instante dado.
- El contenedor servlet puede ser ejecutado como parte del servidor web.
- Facilidad para compartir recursos entre peticiones (hilos con el mismo padre: servlet container).

Soporte de componentes reutilizables

- Creación, utilización y modificación de JavaBeans del servidor.
- Los JavaBeans utilizados en páginas .jsp pueden ser utilizados en servlets, applets o aplicaciones Java.

Separación entre código de presentación y código de implementación

- Cambios realizados en el código HTML relativos a cómo son mostrados los datos, no interfieren en la lógica de programación y viceversa.

División del trabajo

- Los diseñadores de páginas pueden centrarse en el código HTML y los programadores en la lógica del programa.
- Los desarrollos pueden hacerse independientemente.
- Las frecuentes modificaciones de una página se realizan más eficientemente.

Reglas generales de sintaxis

Elementos JSP

La sintaxis JSP está basada en XML. Todos los elementos basados en la sintaxis XML tienen un tag de inicio que incluye el nombre del elemento (posiblemente con atributos), un cuerpo opcional, y su correspondiente tag de fin. Ejemplo:

```
<mytag attr1="valor atributo" ... >
cuerpo
</mytag>
```

ó

```
<mytag attr1="valor atributo" ... />
```

Los tags JSP serán sensibles a mayúsculas y minúsculas al igual que XML. Los scripts JSP y directivas seguirán la sintaxis: `<% .... %>`.

Inicio y fin de tags

Los elementos que tienen inicio y fin de tag (con cuerpo) deberán empezar y terminar en el mismo fichero. No se puede empezar un tag en un fichero y terminarlo en otro.

Esto se aplica también a los elementos con sintaxis alternativa, Por ejemplo, los fragmentos de código `<% scriptlet %>`, los caracteres de apertura `<%` y cierre `%>` deberán estar en el mismo fichero.

Elementos vacíos

Siguiendo las especificaciones XML, un elemento descrito por un tag vacío es indistinguible de uno que use tag de inicio, fin y cuerpo.

Valores de los atributos

Siguiendo las especificaciones XML, los valores de los atributos deberán siempre aparecer entre comillas. Se puede usar tanto comillas simples como dobles. Las entidades `&apos;` y `&quot;` son permitidas para describir comillas simples y dobles.

Espacios en blanco

En HTML y XML los espacios en blanco no suelen ser significativos, con algunas excepciones. Una excepción es que un fichero XML debe empezar con `<?xml` sin poder dejar ningún espacio en blanco.

La especificación al respecto seguirá el comportamiento definido para XML: todos los espacios en blanco dentro del cuerpo del documento no serán significativos pero se reservarán.

Por ejemplo, como las directivas no generan datos, ocurriría lo siguiente:

Entrada:

	`<?xml versión 1.0 ?>`
	`<%@ page buffer=”8k” %>`
	`Resto del documento`

Salida:

	`<?xml versión 1.0 ?>`
`Línea vacía`	
	`Resto del documento`

Manejo de errores

Hay dos fases en el proceso de generación de un JSP:

- Traducción (o compilación) de la página JSP.
- Procesamiento de la página.

Los errores se pueden dar en cualquiera de las dos fases anteriores. Se pretende en este apartado describir cómo tratar estos errores.

Errores en tiempo de compilación

Puede haber fallos en la traducción de una página JSP en su correspondiente clase de implementación. Como resultado de esto, el desarrollador se encontrará con errores de compilación que le serán reportados a través del navegador.

Aunque la forma exacta de este reporte varía de una implementación a otra, la mayoría de los contenedores JSP reportan un error que se basa en un código en el servlet JSP. Por ejemplo, para protocolos http, aparecerá el código de error 500 en la mayoría de los casos.

Sin embargo cuando se trata de errores de compilación por mala escritura del código, este error se mostrará de forma bastante clara en la navegador pudiendo el desarrollador encontrar el error fácilmente.

Errores en el procesamiento

Durante el procesamiento de la petición del cliente, pueden ocurrir errores arbitrarios en tiempo de ejecución. Estos errores utilizan el mecanismo de excepciones del lenguaje Java. Estas excepciones se pueden capturar y tratar en el cuerpo de la JSP.

Sin embargo, cualquier excepción no capturada se transformará en una redirección a la errorPage definida en la directiva `<@page...errorPage=""..>`.

El `java.lang.Throwable` que describe la excepción se almacenará en `el javax.ServletRequest` usando el método `putAttribute()`, con el nombre `javax.servlet.jspException`. Los nombres que empiezan por `"java"` o `"javax"` están reservados por las diferentes especificaciones de la plataforma Java, el prefijo `"javax.servlet"` es usado por las especificaciones JSP/Servlet.

Cómo añadir páginas de error

Aunque las llamemos páginas de error, las páginas especializadas JSP que describimos aquí realmente muestran información sobre excepciones. Para añadir páginas de error que muestren información de excepciones a una aplicación web, se deberán seguir estos pasos:

1. Escribimos nuestro Bean (o bean enterprise, servlet, u otro componente) para que lance ciertas excepciones bajo ciertas condiciones.

2. Usamos un sencillo mecanismo de seguimiento en nuestro componente para ayudarnos a obtener información sobre lo que estaba haciendo el usuario cuando la excepción fue lanzada. (Si nos movemos en el desarrollo de aplicaciones J2EE, nuestra aplicación podrá grabar el estado, que es la mejor forma de proporcionar información).

3. El fichero JSP usa una directiva page con errorPage que selecciona el nombre de un fichero JSP que mostrará un mensaje al usuario cuando ocurre una excepción.

4. Escribir un fichero de página de error, usando una directiva page con `isErrorPage="true".`

5. En el fichero de la página de error, usa el objeto exception para obtener información sobre la excepción.

6. Usamos mensajes informativos, en nuestra página de error o incluida desde otros ficheros, para darle al usuario un mensaje relevantemente informativo sobre lo que el usuario estaba haciendo cuando se lanzó la excepción.

Comentarios

Todos los desarrolladores experimentados usan Javadoc, pero las JSP no ofrecen esta herramienta.

El primer paso para una buena documentación JSP es ubicar el código Java en beans y manejadores de tags en vez de en las JSP. Esto permitirá usar el javadoc para generar la documentación de la mayoría de la lógica.

En cualquier caso, habrá dos tipos de comentarios en una página JSP, comentarios para la página JSP en sí misma, comentando lo que la página está haciendo, y comentarios que se pretenda que aparezcan en el documento resultado que se manda al cliente.

Comentarios de salida al cliente

La sintaxis es la siguiente:

```
<!-- comentarios... -->
```

El contenedor JSP no interpretará estos comentarios, serán incluidos en la salida de la página .jsp, y por lo tanto serán visibles en el código fuente que le llegue al cliente.

Es el tipo estándar de comentario para HTML y XML. Se puede incluir contenido dinámico en el comentario (expresiones JSP), y el valor de la expresión aparecerá como parte del comentario en el cliente:

```
<!-- comentarios <%= expresión %> más comentarios ... -->
```

Comentarios JSP

Estos comentarios sólo pueden ser vistos examinando la página .jsp original. Son de la forma:

```
<%-- comentarios... --%>
```

Una forma alternativa de introducir un comentario en JSP es hacerlo como dictan las normas Java para ello:

```
<% // Esto es un comentario %>

<% /* Esto es un comentario */ %>
```

Convenciones para Escape

Scripts

Un literal `%>` se escribiría como `%\>`

Texto

Un literal `<%` se escribiría como `<\%`

Atributos

- Comilla simple ‘ se escribiría como \’.
- Comilla doble “ se escribiría como \”.
- Un `\` se escribiría como `\\`.
- Un `%>` se escribiría como `%\>`.
- Un `<%` se escribiría como `<\%`.

Uso de includes

Hay un peligro de duplicación de código en las páginas JSP. La forma más eficaz de evitarlo es utilizar los `includes` para introducir fragmentos JSP que puedan ser reutilizables en muchas páginas. Hay dos formas de incluir datos:

Sintaxis	Qué es	Fase	Objeto	Descripción
`<%@ include file=... %>`	directiva	Compilación	estático	El contenido es incluido en tiempo de compilación.
`<jsp:include page=... />`	acción	Request-time	dinámico	El contenido es incluido sin ser parseado

include estático

Se trata de una directiva que nos permite incluir ficheros en el momento en que la página JSP es traducida a un servlet. Ejemplo:

```
<%@ include file="list.jsp %>
```

La URL especificada normalmente se interpreta como relativa a la página JSP a la que se refiere, pero, al igual que las URLs relativas en general, podemos decirle al sistema que interpreta la URL relativa al directorio home del servidor Web empezando la URL con una barra invertida. Los contenidos del fichero incluido son analizados como texto normal JSP, y así pueden incluir HTML estático, elementos de script, directivas y acciones.

include dinámico

Se trata de una acción que nos permite insertar ficheros en una página que está siendo generada. Ejemplo:

```
<jsp:include file="include.jsp"  flush="true" />
```

Al contrario que la directiva include, que inserta el fichero en el momento de la conversión de la página JSP a un Servlet, esta acción inserta el fichero en el momento en que la página es solicitada. Esto se paga un poco en la eficiencia, e imposiblita a la página incluida de contener código JSP general (no puede seleccionar cabeceras HTTP, por ejemplo), pero se obtiene una significante flexibilidad.

También se podrá utilizar como una subtag del <jsp:include> el tag <jsp:param>, para pasarle parámetros a la página incluida.

Ejemplo:

```
<jsp:include file="include.jsp  flush="true" %>
    <jsp:param name="Param1" value="param1" />
    <jsp:param name="Param2" value="param2" />
</jsp:include>
```

Uso de Tag Extensions

El uso de tag Extensions es una característica muy potente del JSP 1.1. Incrementa la habilidad de mantener la lógica en Java y no son mas que clases java que implementan o heredan de determinadas clases. Dependiendo de si la etiqueta tendrá cuerpo o no, se debe implementar una interfaz u otra.

Ventajas de su uso

- Potencian la reutilización de código usado frecuentemente.
- Permiten limpiar de código las JSP sustituyendo la mayoría del código embebido por etiquetas. Esto mejora el mantenimiento y la legibilidad de las páginas.
- Permiten separar más los papeles en el desarrollo de JSP. Esto permite un mayor paralelismo en el trabajo.
- Cuando ya se disponga de una buena biblioteca de taglibs se acelerará el desarrollo de páginas.
- Debido a que están basadas en XSL permiten, además de simplemente añadir código a la página, transformarla con gran potencia. Un ejemplo de las ventajas de que proporciona el hecho de que sean hojas de transformación XSL es que podemos definir etiquetas para invocar de forma sencilla métodos con signaturas complejas. Si preparamos valores por defecto para cada argumento podemos comprobar que atributos se nos pasan en la etiqueta y para los que falten colocar el valor por defecto.

Directiva taglib

La directiva **`taglib`** se utilizará para notificar al contenedor JSP que la página utilizará una o más librerías de etiquetas personalizadas. Una librería de etiquetas es una colección de etiquetas personalizadas, que pueden ser utilizadas para extender la funcionalidad básica de JSP:

```
<%@ taglib uri="URIdeLaLibreriaDeEtiquetas" prefix="prefijoEtiquetas" %>
```

El valor del atributo **uri** indica la localización del TLD (Tag Library Descriptor), mientras que el atributo **prefix** especifica el identificador del espacio de nombres en XML que será anexado a todas la etiquetas de esa librería que se utilicen en la página .jsp.

Etiquetas personalizadas:

```
<prefijoEtiqueta:nombre atributo="valor"+ ... />
<prefijoEtiqueta:nombre atributo="valor"+ ... >
  otras etiquetas
</prefijoEtiqueta:nombre>
```

Por razones de seguridad, la especificación JSP permite utilizar solamente TLD's del sistema de archivos local.

Ejemplo de definición de un Tag

Una etiqueta es una clase Java que implementa un interface especializado. Se usa para encapsular la funcionalidad desde una página JSP. Una etiqueta puede o no tener cuerpo. Para definir una sencilla etiqueta sin cuerpo, nuestra clase debe implementar el interface **`Tag`**.

A continuación se muestra un ejemplo del código fuente del interface Tag que hay que implementar:

```
public interface Tag {
   public final static int SKIP_BODY = 0;
   public final static int EVAL_BODY_INCLUDE = 1;
   public final static int SKIP_PAGE = 5;
   public final static int EVAL_PAGE = 6;

   void setPageContext(PageContext pageContext);
   void setParent(Tag parent);
   Tag getParent();
   int doStartTag() throws JspException;
   int doEndTag() throws JspException;
   void release();
}
```

Todas las etiquetas deben implementar el interface Tag (o uno de sus interfaces) como si definiera todos los métodos que el motor de ejecución JSP llama para ejecutar una etiqueta.

Método	Descripción
setPageContext (PageContext pc)	A este método lo llama el motor de ejecución JSP antes de doStartTag, para seleccionar el contexto de la página.
setParent(Tag parent)	Invocado por el motor de ejecución JSP antes de doStartTag, para pasar una referencia a un controlador de etiqueta a su etiqueta padre.
getParent	Devuelve un ejemplar Tag que es el padre de esta etiqueta.
doStartTag	Invocado por el motor de ejecución JSP para pedir al controlador de etiqueta que procese el inicio de etiqueta de este ejemplar.
doEndTag	Invocado por el motor de ejecución JSP después de retornar de doStartTag. El cuerpo de la acción podría no haber sido evaluado, dependiendo del valor de retorno de doStartTag.
release	Invocada por el motor de ejecución JSP para indicar al controlador de etiqueta que realice cualquier limpieza necesaria.

Ahora, veamos una etiqueta de ejemplo que cuando se le invoca imprime un mensaje en el cliente.

Hay unos pocos pasos implicados en el desarrollo de una etiqueta personalizada. Estos pasos son los siguientes:

1. Desarrollar el controlador de etiqueta
2. Crear un descriptor de librería de etiqueta
3. Probar la etiqueta

Desarrollar el controlador de etiqueta

Un controlador de etiqueta es un objeto llamado por el motor de ejecución JSP para evaluar la etiqueta personalizada durante la ejecución de una página JSP que referencia la etiqueta. Los métodos del controlador de etiqueta son llamados por la clase de implementación en varios momentos durante la evaluación de la etiqueta. Cada controlador de etiqueta debe implementar un interface especializado.

Ejemplo: `HelloTag.java`

```
package tags;

import java.io.*;
import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;

public class HelloTag implements Tag {
   private PageContext pageContext;
   private Tag parent;

   public HelloTag() {
        super();
   }

   public int doStartTag() throws JspException {
       try {
           pageContext.getOut().print(
                           "This is my first tag!");
       } catch (IOException ioe) {
           throw new JspException("Error:
                   IOException while writing to client"
                   + ioe.getMessage());
       }
       return SKIP_BODY;
   }

   public int doEndTag() throws JspException {
        return SKIP_PAGE;
   }

   public void release() {
   }

   public void setPageContext(PageContext pageContext) {
        this.pageContext = pageContext;
   }

   public void setParent(Tag parent) {
        this.parent = parent;
   }

   public Tag getParent() {
        return parent;
   }
}
```

Los dos métodos importantes a observar en HelloTag son doStartTag y doEndTag. El primero es invocado cuando se encuentra la etiqueta de inicio. En este ejemplo, este método devuelve SKIP_BODY porque es una simple etiqueta sin cuerpo. el método doEndTag es invocado cuando se encuentra la etiqueta de cierre. En este ejemplo devuelve SKIP_PAGE porque no queremos evaluar el resto de la página, de otro modo debería devolver EVAL_PAGE.

Descriptor de librería de etiqueta

El siguiente paso es especificar cómo ejecutará la etiqueta el motor de ejecución JSP . Esto puede hacerse creando un "Tag Library Descriptor" (TLD), que es un documento XML.

Ejemplo: `mytaglib.tld`

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE taglib
  PUBLIC "-//Sun Microsystems, Inc.//
  DTD JSP Tag Library 1.1//EN"
  "http://java.sun.com/j2ee/dtds/
  web-jsptaglibrary_1_1.dtd">

<!-- a tag library descriptor -->

<taglib>
   <tlibversion>1.0</tlibversion>
   <jspversion>1.1</jspversion>
   <shortname>first</shortname>
   <uri></uri>
   <info>A simple tab library for the
   examples</info>

  <tag>
    <name>hello</name>
    <tagclass>tags.HelloTag</tagclass>
    <bodycontent>empty</bodycontent>
    <info>Say Hi</info>
  </tag>
</taglib>
```

Primero especificamos la versión de la librería de etiquetas y la versión de JSP. La etiqueta `<shortname>` indica como se va a referencia la librería de etiquetas desde la página JSP. La etiqueta `<uri>` puede usarse como un identificador único para nuestra librería de etiquetas.

En este TLD, sólo tenemos una etiqueta llamada hello cuya clase se especifica usando la etiqueta `<tagclass>`. Sin embargo, una librería de etiquetas puede tener tantas etiquetas como queramos. El `<bodycontent>` nos dice que esta etiqueta no tiene cuerpo; de otro modo se produciría un error. Por otro lado, si queremos evaluar el cuerpo de la etiqueta, el valor debería ser:

- tagdependent: lo que significa que cualquier cuerpo de la etiqueta sería manejado por la propia etiqueta, y puede estar vacío.
- JSP: lo que significa que el contenedor JSP evaluaría cualquier cuerpo de la etiqueta, pero también podría estar vacío.

Probar la etiqueta

El paso final es probar la etiqueta que hemos desarrollado. Para usar la etiqueta, tenemos que referenciarla, y esto se puede hacer de tres formas:

1. Referenciar el descriptor de la librería de etiquetas de una librería de etiquetas desempaquetada. Por ejemplo:

```
<@ taglib uri="/WEB-INF/jsp/mytaglib.tld" prefix="first" %>
```

2. Referenciar un fichero JAR que contiene una librería de etiquetas. Por ejemplo:

```
<@ taglib uri="/WEB-INF/myJARfile.jar" prefix='first" %>
```

3. Definir una referencia a un descriptor de la librería de etiquetas desde el descriptor de aplicaciones web (web.xml) y definir un nombre corto para referenciar la librería de etiquetas desde al JSP. Para hacer esto, abrimos el fichero: c:\tomcat\webapps\examples\web-inf\web.xml y añadimos las siguientes líneas antes de la última línea, que es <web-app>:

```
    <taglib>
 <taglib-uri>mytags</taglib-uri>
 <taglib-location>/WEB-INF/jsp/
 mytaglib.tld</taglib-location>
    </taglib>
```

Ahora, escribimos un JSP y usamos la primera sintaxis. Lo podremos ver en el siguiente ejemplo:

Ejemplo: `Hello.jsp`

```
<%@ taglib uri="/WEB-INF/jsp/mytaglib.tld" prefix="first" %>
<HTML>
<HEAD>
<TITLE>Hello Tag</TITLE>
</HEAD>

<BODY bgcolor="#ffffcc">

<B>My first tag prints</B>:

<first:hello/>

</BODY>
</HTML>
```

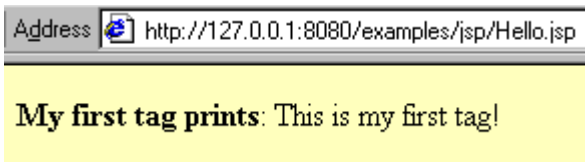
El taglib se usa para decirle al motor de ejecución JSP donde encontrar el descriptor para nuestra etiqueta, y el prefix especifica cómo se referirá a las etiquetas de esta librería. Con esto correcto, el motor de ejecución JSP reconocerá cualquier uso de nuestra etiqueta a lo largo del JSP, mientras que la precedamos con el prefijo first como en `<first:hello/>`.

Alternativamente, podemos usar la segunda opción de referencia creando un fichero JAR. O podemos usar la tercera opción simplemente reemplazando la primera línea del ejemplo 4 con la siguiente línea:

```
<%@ taglib uri="mytags" prefix="first" %>
```

Básicamente, hemos usado el nombre mytags que se ha añadido a web.xml, para referenciar a la librería de etiquetas. Para el resto de los ejemplos de esta página utilizaremos este tipo de referencia.

Ahora, si solicitamos Hello.jsp desde nuestro navegador, veríamos algo así:



JDeveloper Data Tag Library

JDeveloper, a partir de la versión 3.2, ofrece una facilidad de manejo de las JSP gracias a la librería de tags para el acceso de datos que incluye (DataTags.tld).

El acceso a los datos asociados a los Business Components, aportan una manera muy rápida y fácil de construir aplicaciones JSP, de forma que se consigue tener un control completo sobre el diseño HTML y las operaciones de acceso a base de datos.

Además Oracle JDeveloper 3.2 también ofrece un conjunto de wizards para trabajar de una forma más eficiente y rápida si cabe con los BC4J JSP Data Tags. Aparece también el DataPage Wizard que ofrecerá además la posibilidad de generar páginas JSP basadas en los BC4J Views.

Oracle JDeveloper 3.2 BC4J Data Tag Library, por lo tanto, permitirá:

- Un manejo eficiente de los Application Modules.
- Presentación, navegación y edición de los BC4J.
- Construcción de formularios HTML asociados a datos de una forma muy rápida y eficaz.

Uso de JSP Beans

La manera más sencilla de reducir el código Java en una página JSP es usar uno o más JSP Beans.

La acción `<jsp:useBean>` nos permite cargar y utilizar un JavaBean en la página JSP. Esta es una capacidad muy útil porque nos permite utilizar la reutilización de las clases Java sin sacrificar la conveniencia de añadir JSP sobre servlets solitarios.

El sintaxis más simple para especificar que se debería usar un Bean es:

```
<jsp:useBean id="name" class="package.class" />
```

Esto normalmente significa "instancia un objeto de la clase especificada por class, y lo asocia a una variable con el nombre especificado por id". Sin embargo, también podemos especificar un atributo scope que hace que ese Bean se asocie con más de una sóla página. En este caso, es útil obtener referencias a los beans existentes, y la acción jsp:useBean especifica que se instanciará un nuevo objeto si no existe uno con el mismo nombre y ámbito.

Ahora, una vez que tenemos un bean, podemos modificar sus propiedades mediante `<jsp:setProperty>`, o usando un scriptlet y llamando a un método explícitamente sobre el objeto con el nombre de la variable especificada anteriormente mediante el atributo id. Con los beans, cuando decimos "este bean tiene una propiedad del tipo X llamada foo", realmente queremos decir "Esta clase tiene un método getFoo que devuelve algo del tipo X, y otro método llamado setFoo que toma un X como un argumento". La acción `<jsp:setProperty>` nos permitirá suministrar un valor explícito, dando un atributo param para decir que el valor está derivado del parámetro de la petición nombrado, o sólo lista las propiedades para indicar que el valor debería derivarse de los parámetros de la petición con el mismo nombre que la propiedad. Leemos las propiedades existentes en una expresión o scriptlet JSP llamando al método `getXxx`, o más comúnmente, usando la acción `jsp:getProperty`.

Page Beans

Estos beans están íntimamente relacionados con la JSP que los usan. Sólo se deberán usar en varias páginas si éstas representan el mismo contenido en distinto formato.

Normalmente estos beans definirán accesores (getters) para cada contenido a visualizar en la página. Estos getters devolverán valores primitivos u objetos más complejos, y la página JSP será la responsable de mostrar estos objetos (si son complejos, normalmente se hará a través de tags personalizados).

Estos beans podrán tratar los parámetros del request, pero no deberán tratar lógica de negocio compleja. (por ejemplo, no deberían diseñarse para manejar redirecciones).

Session Beans

Estos beans ya no están ligados a la representación de la página, sino que podrán hacer referencia a recursos de lógica de negocio o a cualquier información, permitiendo al usuario mantener en la sesión.

Por ejemplo, un session bean se podrá utilizar para mantener el nombre y email del usuario, y así poder hacer referencia a esta información desde cualquier otro bean para sacar más datos.

Los session beans nunca se deberán utilizar para procesamiento del request.

Application Beans

Estos beans mantienen el estado para todos los usuarios de una aplicación. Se usan menos que los dos anteriores, pero pueden ser útiles para minimizar el consumo de memoria y mejorar el rendimiento, especialmente cuando muchos usuarios necesiten acceder a los mismos datos.

Application Beans se suelen usar para sólo lectura de datos. Cuando se quieren utilizar para lectura-escritura, se tendrá que hacer un tratamiento de sincronización y bloqueos.

Ventajas y desventajas

La regla más importante con beans es la de que no deberán generar HTML. Tampoco deberán procesar lógica de negocio que no sea realmente necesaria para la representación JSP.

También es importante ser precavido con el uso de la sesión puesto que hoy en día es muy usual la implementación de clustering de servidor, con lo que se perderá la sesión sino hacemos nada para evitarlo.

Objetos implícitos

En una página JSP se tiene acceso a algunos objetos que están siempre disponibles sin necesidad de haberlos declarado anteriormente. Estos son:

Objeto	Del tipo	Lo que representa	Scope
request	Subtipo de javax.servlet.ServletRequest Ejemplo: Javax.servlet.HttpServletRequest	Petición del cliente	request
response	Subtipo de javax.servlet.ServletResponse Ejemplo: javax.servlet.HttpServletResponse	Página JSP de respuesta	page
pageContext	javax.servlet.http.PageContext	El motor de JSP utiliza la clase Factory para devolver la implementación de clase PageContext del servidor. Esta clase PageContext es inicializada con los objetos response y request y algunos atributos de la directiva de la página (erropage, session,buffer and autoflush) y facilita los otros objetos implícitos para la pagina de petición.	page
session	javax.servlet.http.HttpSession	El objeto se sesión HTTP asociado a la petición del cliente	session
application	javax.servlet.ServletContext	El contexto servlet obtenido de la configuración del servlet (getServletConfig(). getContext())	application
out	javax.servlet.jsp.JspWriter	Objeto que representa la salida de texto por pantalla.	page
config	javax.servlet.ServletContext	El objeto ServletConfig de la página JSP	page
page	java.lang.Object	Es la forma que tiene la página para referirse a si misma. Se usa como alternativa al objeto this.	page

Además, en una página de error, se podrá acceder al objeto `exception` descrito a continuación:

Objeto	Del tipo	Lo que representa	Scope
exception	java.lang.Throwable	La excepción que no fue capturada en la página que invocó la página de error	page

Estos objetos tienen la funcionalidad común de establecer y recuperar valores de atributos como mecanismo de intercambio de información. Los métodos estándares para manejo de atributos son los que se muestran a continuación:

Método	Descripción
setAttribute(key, value)	Asocia un atributo con su valor correspondiente.
getAttributeNames()	Retorna los nombres de todos los atributos asociados con la sesión.
getAttribute(key)	Retorna el valor del atributo asociado con key.
removeAttribute(key)	Borra el atributo asociado con key

Elementos de script

Declaraciones

Se usan para declarar variables o métodos válidos en Java. Se inserta fuera del método `service()` del servlet que se genera a partir de la página JSP.

Un uso importante de la declaración de métodos es el manejo de eventos relacionados con la inicialización y destrucción de la página .jsp. La inicialización ocurre la primera vez que el contenedor JSP recibe una petición de una página .jsp concreta. La destrucción ocurre cuando el contenedor JSP descarga la clase de memoria o cuando el contenedor JSP se reinicia.

Sintaxis:

```
<%! Esto es una declaración %>
```

El espacio en blanco después de "`<%!`" y antes de "`%>`" es opcional.

Ejemplo:

```
<%! int i = 0; %>
<%! public String f(int i) { if (i<3) return ("..."); .. %>
```

Scriplets

Los scriptlets JSP nos permiten embeber segmentos de código java dentro de una página JSP. El código embebido se inserta directamente en el servlet generado que se ejecuta cuando se pide la página. Este scriptlet usa las variables declaradas en las directivas descritas arriba. Los Scriptlets van encerradas entre etiquetas <% y %>.

Debemos terminar una sentencia de lenguaje con un punto y coma si el lenguaje los requiere. Cuando escribamos un scriptlet podemos usar cualquiera de los objetos o clases implícitos de JSP importados por la directiva page, declarada en una declaración, o nombrada en una etiqueta <jsp:useBean> .

Sintaxis:

```
<% Esto es un scriplet %>
```

El espacio en blanco después de "`<%`" y antes de "`%>`" es opcional.

Ejemplo:

```
<% if(Calendar.getInstance().get(Calendar.AM_PM) == Calendar.AM) { %>
```

```
    Buenos días
<% } else {%>
    Buenas tardes
<% } %>
```

Expresiones

La expresiones JSP nos permiten recuperar dinámicamente o calcular valores a insertar directamente en la página JSP.

Todo resultado de una expresión es convertido a un String antes de ser añadido a la salida de la página. Las expresiones no terminan con un punto y coma (;).

Sintaxis:

```
<%= Esto es una expresión %>
```

El espacio en blanco después de "<%=" y antes de "%>" es opcional.

Ejemplo:

```
<% new java.util.Date()).toLocaleString() %>
```

Recomendaciones generales

- Todas las páginas JSP deberán ser de sólo lectura de datos, con un bean asociado que sea el que de el modelo. La manera de representar los datos al usuario puede cambiar muchas veces, sin embargo la estructura del contenido cambiará mucho menos.

- Pensar verticalmente. Primero hay que identificar los datos que van a ser representados en la JSP y después diseñar el bean que proveerá del modelo de datos a la página. JSPs y JSP Beans se han de diseñar conjuntamente.

- Para páginas más complejas, hay que pensar y diseñar en términos de grupos de JSPs que colaboren a ofrecer pequeños fragmentos de funcionalidad cada una de ellas, en vez de pensar en una JSP como algo individual.

- La lógica condicional ha de situarse en páginas JSP que hagan de controladores en vez de en la propia página JSP (es decir, en la vista). Por ejemplo, si una parte substancial del JSP varía dependiendo de una condición, este JSP será un candidato para rehacerse y convertirse en un controlador, redireccionando éste a dos páginas separadas.

- Será conveniente seguir una convención para el nombrado de páginas JSP, dependiendo de qué tipo de ficheros estemos hablando:

    - Controlador JSP: `xxxxController.jsp`
    - JSP incluida: `_nombreDescriptivoDelFragmento.jsp`
    - Beans de página: `<pagename>Bean`
    - Beans de sesión: `xxxxSessionBean`
    - Manejadores de tags y clases relacionadas:

      `xxxxTag, xxxxTagExtraInfo`

- Las directivas de import deberán evitarse. El uso de imports provoca que se instancien otras clases a parte de los JSP beans. Por lo tanto, en vez de usar una directiva como ésta:

```
<%@ page import="com.java.util.*" %>
```

  será mejor referirse a cada clase escribiendo el nombre completo del paquete, por ejemplo:

```
<%@ java.util.List lista = new java.util.LinkedList(); %>
```

- Las páginas JSP no deberán acceder directamente a los parámetros del request como norma general. Para ello, deberán utilizarse controladores o beans. Esto es porque el acceso a los parámetros del request requieren normalmente chequeo de errores que complicaría la página JSP.

- Las páginas JSP no deberían acceder a los ficheros de propiedades o usar JNDI. Son los beans los que deberán hacerlo (por ejemplo, para proveer al sistema de internacionalización).

- Evitar el diseño de páginas que muestran un formulario y procesan al mismo tiempo el resultado. Utilizar para el resultado otra JSP.

- Evitar duplicación de código en un JSP. Esto significará que el diseño ha sido pobre. Este código duplicado se deberá transformar en un include, bean o tag extensión para hacerle reutilizable en toda la aplicación. Se recomienda utilizar includes estáticos frente a los dinámicos.

- Los JSP beans nunca han de generar código HTML. Precisamente JSP fue diseñada para evitar este tipo de técnicas.

- El uso de `out.println()` se deberá evitar en las JSP. Este uso provoca que se piense en las páginas JSP como programas en vez de páginas web, y esto es bastante confuso.

- La capa JSP nunca deberá acceder directamente a los datos, es decir, accesos JDBC y EJB, sino que los beans de las JSPs deberán acceder a objetos de la sesión que correspondan a la capa de la lógica de negocio.

- Los scriptlets nunca deberán superar las 5 líneas de longitud.

- Los JSP beans de la sesión no deberán contener una cantidad excesiva de datos.

- Si se utiliza `<jsp:forward>` o `<jsp:include>` y se quieren pasar valores a otra página externa, se deberá usar siempre que sea posible la directiva `<jsp:param>`. Si esto no fuera posible se deberá usar el método `setAttribute()` en el primer caso y un include estático en el segundo.

- Se deberán usar tag extensions para eliminar lógica de las páginas JSP.

- La directiva `<jsp:forward>` deberá usarse con cuidado. En realidad equivale a un `goto`, y hará que se pierda la idea de JSP como vista. Se deberá usar en ese caso un controlador o un bean.

- En general, las declaraciones y los scriptlets no se deberían usar para crear variables a las que se haga referencia dentro de la misma página.

- Los métodos `jspInit()` y `jspDestroy()` deberán usarse sólo si es estrictamente necesario. Usar estos métodos hace que la página JSP parezca más un servlet que una página web.

- Evitar manejo de excepciones en las JSP. Cada JSP deberá usar una página de error para manejar las excepciones. Las excepciones se deberán capturar en la capa JSP pero no por la página JSP en sí misma si es posible.

Referencias

- JavaServer Pages Specification 1.2.
- JavaServer Pages. Dynamically generated web content (http://java.sun.com/products/jsp/index.html).
- Web Application Development with JSPTM and XML, Part III: Developing JSP Custom Tags, Java Developer ConnectionSM, August 2001
- Java Software Tutorial.
- Core Servlets and JavaServer Pages Tutorial (PDF).
- Jakarta-Taglibs.
- The binary and source distributions have a tutorial in the directory jakarta-taglibs/doc.
- Orion Server Tutorial