helion.pl
KSIĘGARNIA INTERNETOWA · WYDAWNICTWO

IDŹ DO
PRZYKŁADOWY ROZDZIAŁ
SPIS TREŚCI

KATALOG KSIĄŻEK
KATALOG ONLINE
ZAMÓW DRUKOWANY KATALOG

TWÓJ KOSZYK
DODAJ DO KOSZYKA

CENNIK I INFORMACJE
ZAMÓW INFORMACJE O NOWOŚCIACH
ZAMÓW CENNIK

CZYTELNIA
FRAGMENTY KSIĄŻEK ONLINE

Java Server Pages.
Leksykon kieszonkowy

Autor: Hans Bergsten
Tłumaczenie: Adrian Nowak
ISBN: 83-7197-674-7
Tytuł oryginału: JavaServer Pages. Pocket Reference
Liczba stron: 118



Java Server Pages (JSP) służy do tworzenia dynamicznych stron WWW, umożliwiając harmonijne połączenie dokonań projektantów i programistów. Wykorzystuje wielkie możliwości serwletów Javy do tworzenia efektywnych, uniwersalnych aplikacji sieciowych. JSP pozwala na projektowanie prężnie działających stron WWW o dużych możliwościach i — co najważniejsze — nie wymaga zaawansowanych umiejętności w dziedzinie programowania w Javie.
JavaServer Pages. Leksykon kieszonkowy stanowi dodatek do bestsellera wydawnictwa O'Reilly JavaServer Pages, również autorstwa Hansa Bergstena. Książka zawiera szczegółowe informacje na temat składni i przetwarzania JSP, elementów dyrektyw, elementów standardowych akcji, elementów skryptowych, obiektów niejawnych, akcji specjalizowanych, plików TLD i archiwów WAR.
Autor jest założycielem firmy Gefion Software. Hans Bergsten był również aktywnym uczestnikiem grup roboczych opracowujących specyfikacje serwletów Javy i JSP. Jako członek komitetu kierującego projektem Apache Jakarta wniósł istotny wkład w implementację wzorcową Apache Tomcat.

Wydawnictwo Helion
ul. Chopina 6
44-100 Gliwice
tel. (32)230-98-63
e-mail: helion@helion.pl

Spis treści

Przetwarzanie JSP 6
Dyrektywy 9
Elementy akcji standardowych 14
Komentarze 29
Blokowanie interpretacji znaków 30
Elementy skryptowe 31
Obiekty niejawne 36
Akcje specjalizowane 71
Tworzenie pliku opisu biblioteki znaczników (TLD) 107
Pakowanie i instalacja biblioteki znaczników 112
Pliki archiwów WWW (WAR) 115

Akcje specjalizowane

Programista może rozszerzyć możliwości języka JSP, definiując akcje specjalizowane (ang. *custom actions*). Sposób ten pozwala na wykonanie dowolnych zadań nie przewidzianych w standardowych akcjach JSP, takich jak sprawdzenie poprawności danych lub zmiana sposobu ich prezentacji i lokalizacji.

Składnia stosowana do wykorzystania akcji specjalizowanej jest taka sama jak dla akcji standardowych: otwarcie znacznika (opcjonalnie z atrybutami), ciało znacznika i zamknięcie znacznika. Inne elementy oraz treść szablonowa mogą być zagnieżdżone w ciele.

A oto przykład:

```
<prefiks:nazwaAkcji atr1="value1" atr2="value2">
  Ciało znacznika
</prefiks:nazwaAkcji>
```

Jeżeli znacznik nie ma ciała, zamiast pełnego otwarcia i zamknięcia można zastosować notację skrótową:

```
<prefiks:nazwaAkcji atr1="value1" atr2="value2" />
```

Zanim jednak wykorzystamy akcję specjalizowaną na stronie JSP, musimy zadeklarować, w jakiej bibliotece znaczników jest

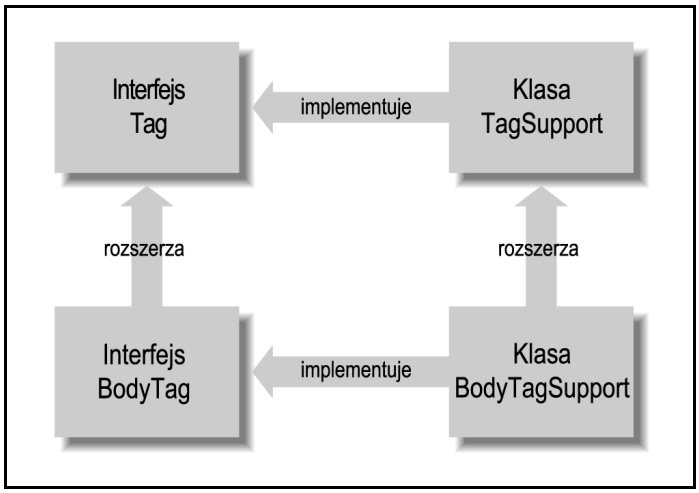
ona zawarta. W tym celu używamy dyrektywy `taglib`, wskazując bibliotekę i przypisując jej prefiks, za pomocą którego będziemy identyfikować akcje w obrębie strony.

Tworzenie akcji specjalizowanych

Akcja specjalizowana — a właściwie klasa obsługi znacznika (ang. *tag handler class*) dla akcji specjalizowanej — jest, ogólnie rzecz biorąc, komponentem JavaBean. Jego metody ustawiające właściwości odpowiadają atrybutom elementu akcji specjalizowanej. Ponadto klasa obsługi znacznika musi implementować jeden z dwóch interfejsów Javy zdefiniowanych w specyfikacji JSP.

Wszystkie interfejsy i klasy potrzebne do implementacji obsługi znacznika są zdefiniowane w pakiecie `javax.servlet.jsp.tagext`. Dwa podstawowe interfejsy to `Tag` i `BodyTag`. Interfejs `Tag` określa metody konieczne do implementacji dla dowolnej akcji. Interfejs `BodyTag` natomiast rozszerza `Tag`, dodając do niego metody wykorzystywane w celu uzyskania dostępu do ciała znacznika. Aby ułatwić utworzenie klasy obsługi znacznika, w API zdefiniowano dwie klasy pomocnicze: `TagSupport` i `BodyTagSupport` (relacje między nimi ukazuje rysunek 4.). Klasy te dostarczają domyślnej implementacji dla metod należących do odpowiednich interfejsów.

Specyfikacja definiuje zarówno interfejsy, jak też wyczerpujące ich implementację klasy pomocnicze. Jeżeli mamy już klasę oferującą pewien zestaw funkcji i chcielibyśmy udostępnić ją jako akcję specjalizowaną, możemy zaznaczyć, że klasa implementuje stosowny interfejs, i uzupełnić ją o metody określone w tym interfejsie. W praktyce jednak zalecane jest implementowanie własnych klas obsługi znaczników jako rozszerzeń klas



Rysunek 4. Podstawowe interfejsy rozszerzania znaczników i klasy obsługi

pomocniczych. W ten sposób otrzymujemy większość metod „za darmo", a istniejące klasy możemy wykorzystać, wywołując je z wnętrza klasy obsługi znacznika.

Biblioteka znaczników (ang. *tag library*) jest zbiorem akcji specjalizowanych. Oprócz plików klas obsługi znaczników musi ona zawierać plik opisu biblioteki znaczników (*Tag Library Descriptor*, TLD). Jest to plik XML definiujący odwzorowanie między nazwami specjalizowanych akcji a odpowiednimi klasami obsługi znaczników oraz opisujący atrybuty obsługiwane przez każdą ze specjalizowanych akcji. Pliki klas oraz plik TLD mogą być spakowane do archiwum JAR, co ułatwia instalację biblioteki.

Zanim zagłębimy się w zawiłe szczegóły, ustalmy, czego potrzeba, aby stworzyć, zainstalować i wykorzystać akcję specjalizowaną. Przede wszystkim musimy zaimplementować klasę obsługi znacznika, podobnie jak w przykładzie:

```
package com.mycompany;

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

public class HelloTag extends TagSupport {
    private String name = "World";

    public void setName(String name) {
        this.name = name;
    }

    public int doEndTag() {
        try {
            pageContext.getOut().println("Hello " +
            ↳name);
        }
        catch (IOException e) {} // Ignore it
        return EVAL_PAGE;
    }
}
```

Klasa obsługi znacznika zawiera metodę ustawiającą wartość atrybutu o nazwie `name`. Metoda `doEndTag()` (zdefiniowana w interfejsie `Tag`) podaje jako odpowiedź tekst „Hello” oraz wartość atrybutu `name`. Klasę tę należy skompilować i umieścić plik wynikowy w katalogu *WEB-INF/classes* omawianej aplikacji.

Następnie tworzymy plik TLD. Poniżej znajduje się minimalny plik TLD dla biblioteki zawierającej tylko jedną akcję specjalizowaną:

```
<?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">

<taglib>
  <tlibversion>1.0</tlibversion>
  <jspversion>1.1</jspversion>
  <shortname>test</shortname>

  <tag>
    <name>hello</name>
    <tagclass>com.mycompany.HelloTag</tagclass>
    <bodycontent>empty</bodycontent>
    <attribute>
      <name>name</name>
    </attribute>
  </tag>
</tablib>
```

Plik TLD przyporządkowuje nazwę akcji specjalizowanej `hello` klasie obsługi znacznika `com.mycompany.HelloTag` i definiuje atrybut `name`. Plik ten umieszczamy w katalogu *WEB-INF/tlds* aplikacji, nadając mu nazwę taką jak *mylib.tld*.

Teraz możemy już wykorzystać akcję specjalizowaną na stronie JSP, na przykład w taki sposób:

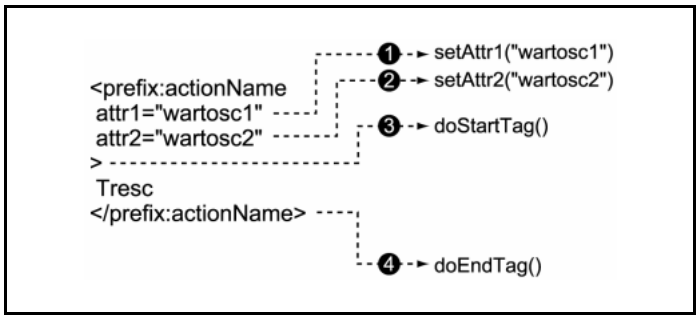
```
<%@ taglib uri="/WEB-INF/mylib.tld" prefix="test" %>
<html>
  <body bgcolor="white">
    <test:hello name="Hans" />
  </body>
</html>
```

Dyrektywa `taglib` wiąże plik TLD z prefiksem nazwy elementu `test`, użytego na naszej stronie do wywołania akcji specjalizowanej. Kiedy pojawia się żądanie strony, kontener sieciowy wykorzystuje TLD, aby stwierdzić, którą klasę należy

wykonać dla akcji specjalizowanej. Następnie wywołuje stosowne metody, co skutkuje włączeniem do odpowiedzi tekstu „Hello Hans".

Akcje specjalizowane, które nie przetwarzają ciała znacznika

Klasa obsługi znacznika jest przywoływana przez kontener sieciowy, ilekroć na stronie JSP występuje akcja specjalizowana. Wymaga ona dostępu do wszystkich informacji o żądaniu i o stronie, a także o wartościach atrybutów elementu akcji (jeżeli takie istnieją). Jako minimum klasa ta musi implementować interfejs `Tag`, który zawiera metody umożliwiające jej dostęp do informacji o stronie i o żądaniu, jak również metody wywoływane przez kontener w momencie napotkania otwarcia znacznika i zamknięcia znacznika. Jeśli chodzi o wartości atrybutów, to kontener sieciowy traktuje klasę obsługi znacznika jako komponent bean i wywołuje metody ustawiające właściwości, które odpowiadają atrybutom elementu akcji, co pokazano na rysunku 5.



Rysunek 5. Metody interfejsu Tag i metody ustawiania właściwości

Przeważnie klasa obsługi znacznika jest rozszerzeniem klasy `TagSupport` (która dostarcza domyślnej implementacji wszystkich metod interfejsu `Tag`) i przesłania tylko jedną z metod.

Zauważmy, że chociaż element akcji obsługiwany przez klasę implementującą interfejs `Tag` może mieć ciało, to jednak kontrola nad jego zawartością będzie bardziej ograniczona niż w przypadku zaimplementowania interfejsu `BodyTag`.

Interfejs Tag

Nazwa interfejsu:	`javax.servlet.jsp.tagext.Tag`
Rozszerza:	—
Implementowany przez:	klasy obsługi znaczników akcji specjalizowanych oraz `javax.servlet.jsp.tagext.TagSupport`

Opis

Interfejs `Tag` powinien być implementowany przez klasy obsługi znaczników, które nie wymagają dostępu do ciała odpowiedniego elementu akcji specjalizowanej i nie muszą po nim iterować.

Metody

`public int doEndTag() throws JspException`

Wykonuje odpowiednie działania, gdy napotykane jest domknięcie znacznika. Jeżeli metoda ta zwraca `SKIP_PAGE`, pozostała część strony nie jest wykonywana i następuje powrót z metody `_jspService()` klasy implementującej

stronę JSP. Jeżeli zwracana jest wartość `EVAL_PAGE`, wykonywany jest kod następujący po specjalizowanej akcji w metodzie `_jspService()`.

`public int doStartTag throws JspException`
Rozpoczyna działanie, kiedy napotykane jest otwarcie znacznika. Metodę tę wywołuje kontener sieciowy, gdy zostały już wywołane wszystkie metody ustawiające wartości właściwości. Zwracana wartość decyduje o tym, w jaki sposób traktowane jest ciało akcji, o ile takie istnieje. Jeżeli wynikiem jest `EVAL_BODY_INCLUDE`, kontener sieciowy przetwarza ciało i ewentualne elementy JSP, dodając efekty tego działania do odpowiedzi. Jeżeli zwracana jest wartość `SKIP_BODY`, ciało jest ignorowane.

Klasa obsługi znacznika, która implementuje interfejs `BodyTag` (rozszerzający interfejs `Tag`), może zwrócić `EVAL_BODY_TAG` zamiast `EVAL_BODY_INCLUDE`. Kontener sieciowy tworzy wtedy instancję `BodyContent` i udostępnia ją klasie obsługi znacznika w celu przeprowadzenia działań na ciele elementu akcji.

`public Tag getParent()`
Zwraca rodzica klasy obsługi znacznika (egzemplarz obiektu `Tag` dla zewnętrznego elementu akcji, jeśli taki istnieje lub `null` w przeciwnym wypadku).

`public void release()`
Zwalnia wszystkie referencje przechowywane przez obiekt.

`public void setPageContext(PageContext pc)`
Ustawia referencję do bieżącego obiektu `PageContext`.

`public void setParent(Tag t)`

Ustawia referencję do rodzica klasy obsługi znacznika (obiektu `Tag` dla zewnętrznego elementu akcji).

Klasa TagSupport

Nazwa klasy:	`javax.servlet.jsp.tagext.TagSupport`
Rozszerza:	—
Implementuje:	`Tag, java.io.Serializable`
Implementowana przez:	wewnętrzne klasy, zależne od kontenera sieciowego. Większość kontenerów wykorzystuje wzorcową implementację klasy (stworzoną dla projektu Apache Jakarta)

Opis

`TagSupport` jest klasą pomocniczą, która dostarcza domyślnej implementacji dla wszystkich metod interfejsu `Tag`. W zamierzeniu ma być wykorzystywana jako nadklasa dla klas obsługi znaczników, które nie wymagają dostępu do ciał odpowiadających im elementów akcji specjalizowanych.

Konstruktor

`public TagSupport()`

Tworzy nową instancję o wskazanej nazwie i wartości.

Metody

`public int doEndTag() throws JspException`
Zwraca stałą `EVAL_PAGE`.

`public int doStartTag() throws JspException`
Zwraca stałą `SKIP_BODY`.

`public static final Tag findAncestorWithClass (Tag from, Class class)`
Zwraca instancję danej klasy, odnalezioną przez sprawdzanie kolejnych rodziców w strukturze zagnieżdżeń klasy obsługi znacznika (analogicznej do struktury zagnieżdżeń elementów akcji), począwszy od obiektu `Tag`. Jeśli poszukiwanie się nie powiedzie, zwraca wartość `null`.

`public String getId()`
Zwraca wartość atrybutu `id` (lub `null`, jeśli nie został on ustawiony).

`public Tag getParent()`
Zwraca rodzica instancji `Tag` (reprezentującego element akcji, który zawiera element odpowiadający tej instancji). Jeżeli instancja nie ma rodzica (tzn. jest na najwyższym poziomie w stronie JSP), metoda zwraca `null`.

`public Object getValue(String k)`
Zwraca wartość wskazanego atrybutu, ustawioną uprzednio metodą `setValue()`, lub `null`, jeśli jej nie odnajdzie.

`public java.util.Enumeration getValues()`
Zwraca obiekt `Enumeration` zawierający nazwy wszystkich atrybutów, których wartości zostały ustawione za pomocą metody `setValue()`.

`public void release()`
Zwalnia wszystkie referencje przechowywane przez obiekt.

`public void removeValue(String k)`
Zwalnia wartość ustawioną metodą `setValue()`.

`public void setPageContext(PageContext pageContext)`
Ustawia referencję do bieżącego obiektu `PageContext`.

`public void setId(String id)`
Ustawia wartość atrybutu `id`.

`public void setParent(Tag t)`
Ustawia referencję do rodzica klasy obsługi znacznika.

`public void setValue(String k, Object o)`
Ustawia dany atrybut przypisując mu wskazaną wartość. Podklasy mogą wykorzystywać tę metodę jako alternatywę dla stosowania zmiennych składowych.

Przykład

Przykładem akcji specjalizowanej, możliwej do zaimplementowania jako prosta klasa obsługi znacznika (tzn. implementująca jedynie interfejs `Tag`), jest akcja, która dodaje cookie do odpowiedzi HTTP. Nazwijmy tę akcję `<ora:addCookie>`. Klasa obsługi nazywa się `com.ora.jsp.tags.generic.AddCookieTag` i rozszerza klasę `TagSupport`, dziedzicząc większość implementacji metod interfejsu `Tag`:

```
package com.ora.jsp.tags.generic;

import javax.servlet.http.*;
import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;
import com.ora.jsp.util.*;

public class AddCookieTag extends TagSupport {
```

Akcja <ora:addCookie> ma dwa obowiązkowe atrybuty, name i value, oraz jeden opcjonalny, maxAge. Każdy atrybut jest reprezentowany przez zmienną składową i standardową metodę ustawiającą właściwość:

```
private String name;
private String value;
private String maxAgeString;

public void setName(String name) {
    this.name = name;
}

public void setValue(String value) {
    this.value = value;
}

public void setMaxAge(String maxAgeString) {
    this.maxAgeString = maxAgeString;
}
```

Wszystkie metody ustawiające zmieniają wartości odpowiadających im zmiennych składowych.

Celem naszej akcji specjalizowanej jest utworzenie nowego obiektu javax.servlet.Cookie z wartościami name, value i maxAge, określonymi przez atrybuty i dodanie cookie do odpowiedzi. Klasa obsługi znacznika nadpisuje metodę doEndTag(), aby wykonać to zadanie:

```
public int doEndTag() throws JspException {
    int maxAge = -1;
    if (maxAgeString != null) {
        try {
            maxAge = Integer.valueOf
            ↳(maxAgeString).
              intValue();
        }
```

```
            catch (NumberFormatException e) {
                throw new JspException("Invalid
                ↳maxAge: " +
                    e.getMessage());
            }
        }
        sendCookie(name, value, maxAge,
            (HttpServletResponse)
            ↳pageContext.getResponse());
        return EVAL_PAGE;
    }

    private void sendCookie(String name, String
    ↳value,
      int maxAge,
        HttpServletResponse res) {
        Cookie cookie = new Cookie(name, value);
        cookie.setMaxAge(maxAge);
        res.addCookie(cookie);
    }
```

Atrybut `maxAge` jest opcjonalny, zanim więc odpowiednia wartość `String` zostanie skonwertowana na typ `int`, sprawdzamy, czy została ona zdefiniowana. Podobne testy nie są konieczne dla zmiennych `name` i `value`, ponieważ wszystkie obowiązkowe atrybuty akcji specjalizowanej są weryfikowane automatycznie przez kontener sieciowy. Jeżeli nie podano wartości obowiązkowego atrybutu, kontener odmówi przetworzenia strony — można więc zawsze mieć pewność, że zmiennej odpowiadającej atrybutowi obowiązkowemu została przypisana wartość. Informacja, czy atrybut jest obowiązkowy, jest wyspecyfikowana w pliku TLD.

Klasa obsługi znacznika powinna także implementować metodę `release()`, aby zwolnić wszystkie przechowywane referencje:

```
public void release() {
    name = null;
    value = null;
```

```
        maxAgeString = null;
        super.release();
    }
```

Metoda `release()` zostaje wywołana w momencie, gdy obiekt klasy obsługi znacznika nie jest już potrzebny. Klasa `AddCookieTag` ustawia wszystkie swoje właściwości na `null`, po czym wywołuje `super.release()`, aby klasa `TagSupport` mogła zadziałać tak samo. W rezultacie zostają udostępnione obiekty, które były przyporządkowane właściwościom, mechanizmowi odśmiecania pamięci.

Metodą klasy `TagSupport` niepotrzebną w tym przykładzie, lecz pożyteczną w innych sytuacjach, jest metoda `findAncestorWithClass()`. Może ona zostać wykorzystana przez klasę obsługi w celu odnalezienia rodzica zagnieżdżonego elementu akcji, a następnie wywołania metod implementowanych przez jego klasę obsługi dla dostarczenia lub pobrania niektórych informacji. W ten sposób można by na przykład operować na elementach `<jsp:param>` zagnieżdżonych wewnątrz elementów standardowych akcji JSP: `<jsp:forward>` oraz `<jsp:include>`. Równoważna akcja specjalizowana dla zagnieżdżonych parametrów byłaby zaimplementowana jako klasa obsługi znacznika, wykorzystująca metodę `findAncestorWithClass()` w poniższy sposób:

```
import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;
public class ParamTag extends TagSupport {
    private String name;
    private String value;

    public void setName(String name) {
        this.name = name;
    }
```

```
    public void setValue(String value) {
        this.value = value;
    }

    public int doEndTag() throws JspException {
        Tag parent = findAncestorWithClass(this,
          ParamParent.class);
        if (parent == null) {
            throw new JspException("Akcja param
            ↳nie jest umieszczona" +
                "wewnatrz akcji obslugiwanego
                ↳typu");
        }
        ParamParent paramParent = (ParamParent)
        ↳parent;
        paramParent.setParam(name, URLEncoder.
          encode(value));
        return EVAL_PAGE;
    }
}
```

Akcje specjalizowane, przetwarzające ciało znacznika

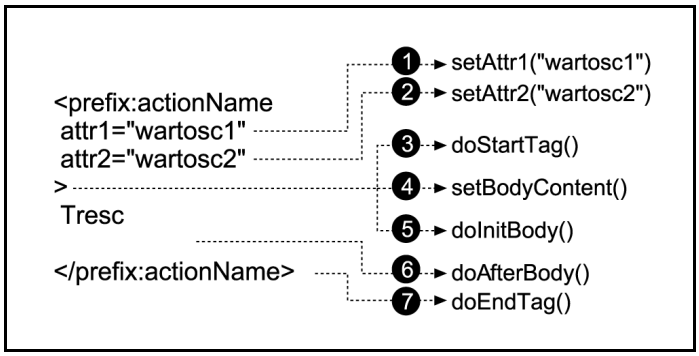
Jak widać, nietrudno jest stworzyć klasę obsługi znacznika, która nie zajmuje się ciałem elementu akcji. Jeżeli ma ona przetwarzać ciało, potrzebnych jest jeszcze kilka dodatkowych metod. Zostały one zdefiniowane w interfejsie `BodyTag`, rozszerzającym interfejs `Tag`.

Treść elementu akcji możne zostać wykorzystana w wielu przypadkach — na przykład wtedy, gdy mamy do czynienia z danymi wejściowymi zajmującymi kilka linijek tekstu. Załóżmy, że tworzymy akcję specjalizowaną mającą wykonać instrukcję SQL, wyspecyfikowaną przez autora strony. Polecenia SQL są przeważnie długie, lepiej więc pozwolić, by autor napisał je w treści znacznika, zamiast zmuszać go do upchnięcia ich w jednej

linii, co jest wymagane w przypadku atrybutów akcji. Można również wykorzystać ciało elementu akcji, przetwarzając je w jakiś szczególny sposób przed dodaniem do odpowiedzi (na przykład zmienić treść XML według arkusza stylów XSL, wyspecyfikowanego jako atrybut).

Podobnie jak dla interfejsu `Tag`, również dla interfejsu `BodyTag` istnieje klasa `BodyTagSupport` implementująca wszystkie jego metody oraz kilka metod narzędziowych.

Klasa obsługi znacznika implementująca interfejs `BodyTag` traktowana jest początkowo tak samo, jak klasa implementująca interfejs `Tag`: kontener wywołuje wszystkie metody ustawiania właściwości oraz metodę `doStartTag()`. W tym momencie jednak postępowanie odbiega od poprzedniego schematu, co widać na rysunku 6.



Rysunek 6. Metody interfejsu BodyTag

Dodatkowe metody `setBodyContent()`, `doInitBody()` i `doAfterBody()` umożliwiają klasie dostęp do ciała elementu i dają sposobność jego przetworzenia, co opisano poniżej.

Interfejs BodyTag

Nazwa klasy:	`javax.servlet.jsp.tagext.BodyTag`
Rozszerza:	`javax.servlet.sjp.tagext.Tag`
Implementowany przez:	klasy obsługi znaczników specjalizowanych akcji oraz `javax.servlet.jsp.tagext.BodyTagSupport`

Opis

Interfejs `BodyTag` musi być implementowany przez każdą klasę obsługi znacznika, która wymaga dostępu do ciała elementu i odpowiadającej znacznikowi akcji specjalizowanej — na przykład, aby przekształcić zawartość w pewien sposób przed dołączeniem jej do odpowiedzi. Również klasy, które mają iterować po ciele, muszą implementować ten interfejs.

Metody

`public int doAfterBody() throws JspException`

Wywoływana jest za każdym razem, kiedy kończy się wartościowanie ciała. Jeżeli metoda ta zwraca `EVAL_BODY_TAG`, ciało jest wartościowane ponownie, przeważnie po zmianie wartości wykorzystywanych zmiennych. Jeżeli zwraca `SKIP_BODY`, następuje wywołanie metody `doEndTag()`.

Metoda `doAfterBody()` nie jest wywoływana, jeśli ciało elementu jest puste lub jeśli metoda `doStartTag()` zwróciła stałą `SKIP_BODY`.

`public void doInitBody() throws JspException`

Przeprowadza przygotowania do wartościowania ciała elementu. Jest wywoływana przez implementację strony jednokrotnie przy każdym wystąpieniu akcji, po otrzymaniu nowego obiektu `BodyContent` i ustawieniu referencji do niego metodą `setBodyContent()`, natomiast przed przeprowadzeniem wartościowania ciała elementu.

Metoda `doInitBody()` nie jest wywoływana, jeśli ciało elementu jest puste lub metoda `doStartTag()` zwróciła stałą `SKIP_BODY`.

`public void setBodyContent (BodyContent b)`

Ustawia referencję do obiektu `BodyContent` utworzonego dla bieżącego znacznika. Ta metoda nie zostanie wywołana, jeżeli ciało jest puste lub jeśli metoda `doStartTag()` zwróciła stałą `SKIP_BODY`.

Klasa BodyTagSupport

Nazwa klasy:	`javax.servlet.jsp.tagext.BodyTagSupport`
Rozszerza:	`javax.servlet.jsp.tagext.TagSupport`
Implementuje:	`BodyTag`
Implementowana przez:	wewnętrzne klasy, zależne od kontenera sieciowego. Większość kontenerów wykorzystuje wzorcową implementację klasy (stworzoną dla projektu Apache Jakarta)

Opis

`BodyTagSupport` jest klasą pomocniczą, która dostarcza domyślnych implementacji dla wszystkich metod interfejsu `BodyTag`. W zamierzeniu powinna być wykorzystywana jako nadklasa dla klas obsługi wymagających dostępu do ciała elementu, odpowiadającej znacznikowi akcji specjalizowanej.

Konstruktor

`public BodyTagSupport()`
Tworzy nową instancję `BodyTagSupport`.

Metody

`public int doAfterBody() throws JspException`
Zwraca `SKIP_BODY`.

`public int doEndTag() throws JspException`
Zwraca `EVAL_PAGE`.

`public void doInitBody()`
W tej klasie nie wykonuje żadnego działania.

`public BodyContent getBodyContent()`
Zwraca przypisany do tego egzemplarza obiekt `BodyContent`.

`public JspWriter getPreviousOut()`
Zwraca obiekt `JspWriter`, zewnętrzny dla obiektu `BodyContent` przypisanego temu egzemplarzowi.

`public void release()`

Usuwa referencje do wszystkich obiektów przechowywanych przez tę składową.

`public void setBodyContent(BodyContent bodyContent)`

Ustawia referencję do obiektu `BodyContent` — przypisanego egzemplarzowi — jako zmienną składową.

Klasa BodyContent

Nazwa klasy:	`javax.servlet.jsp.tagext.BodyContent`
Rozszerza:	`javax.servlet.jsp.JspWriter`
Implementuje:	—
Implementowana przez:	wewnętrzne klasy zależne od kontenera

Opis

Kontener tworzy instancję klasy `BodyContent` w celu przechowania wyniku przetworzenia ciała elementu akcji, jeśli odpowiadająca tej akcji klasa obsługi implementuje interfejs `BodyTag`. Kontener udostępnia obiekt `BodyContent` klasie obsługi znacznika przez wywołanie metody `setBodyContent()`.

Konstruktor

`protected BodyContent(JspWriter e)`

Tworzy nowy egzemplarz obiektu, przypisując mu jako zewnętrzny obiekt `JspWriter` wartość argumentu.

Metody

`public void clearBody()`
Usuwa całą treść znajdującą się w buforze obiektu.

`public void flush() throws java.io.IOException`
Nadpisuje zachowanie odziedziczone po `JspWriter` tak, aby zawsze zgłaszany był wyjątek IOException, ponieważ wywołanie `flush()` dla obiektu `BodyContent` nie ma sensu.

`public JspWriter getEnclosingWriter()`
Zwraca zewnętrzny obiekt `JspWriter`; innymi słowy, jest to obiekt `JspWriter` najwyższego poziomu na stronie lub przypisany rodzicowi klasy obsługi znacznika.

`public abstract java.io.Reader getReader()`
Zwraca bieżący obiekt jako obiekt `Reader` z zawartością otrzymaną przez wartościowanie ciała elementu.

`public abstract String getString()`
Zwraca bieżący obiekt jako `String` z zawartością otrzymaną przez wartościowanie ciała elementu.

`public abstract void writeOut (java.io.Writer out) throws java.io.IOException`
Wypisuje zawartość obiektu `BodyContent` do obiektu `Writer`.

Przykład

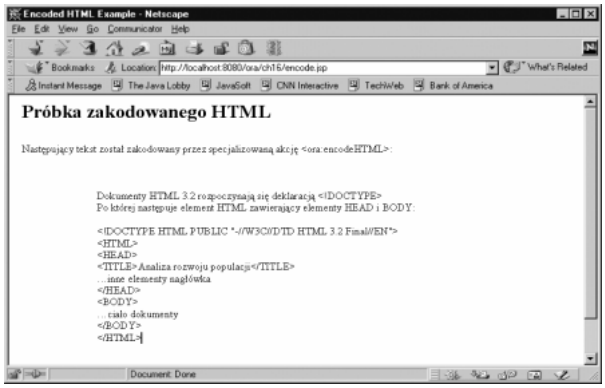
Rozpatrzmy przykład klasy obsługi znacznika rozszerzającej klasę `BodyTagSupport`. Klasa `EncodeHTMLTag` odpowiada akcji specjalizowanej o nazwie `<ora:encodeHTML>`. Akcja ta

odczytuje ciało elementu; zastępuje ona wszystkie znaki o specjalnym znaczeniu w HTML, takie jak cudzysłowy, znaki mniejszości i większości oraz ampersandy, odpowiadającymi im symbolami HTML (`'`, `"`, `<`, `>`, `&`), po czym wstawia otrzymany ciąg do odpowiedzi. Następujący przykład pokazuje, w jaki sposób można wykorzystać tę akcję na stronie JSP:

```
<%@ page language="java" %>
<%@ taglib uri="/orataglib" prefix="ora" %>
<html>
  <head>
    <title>Próbka zakodowanego HTML</title>
  </head>
  <body>
    <h1> Próbka zakodowanego HTML </h1>
    Następujący tekst został zakodowany przez
    ↳specjalizowaną akcję
    &lt;ora:encodeHTML&gt;:
    <pre>
      <ora:encodeHTML>
        Dokumenty HTML 3.2 rozpoczynają się
        ↳deklaracją <!DOCTYPE>
        po której następuje element HTML
        ↳zawierający
        elementy HEAD i BODY:

        <!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML
        ↳3.2 Final//EN">
        <HTML>
        <HEAD>
        <TITLE>Analiza rozwoju populacji</TITLE>
        ... inne elementy nagłówka
        </HEAD>
        <BODY>
        ... ciało dokumentu
        </BODY>
        </HTML>
      </ora:encodeHTML>
    </pre>
  </body>
</html>
```

Należy zwrócić uwagę, że ciało akcji `<ora:encodeHTML>` na przykładowej stronie JSP zawiera elementy HTML. Jeżeli znaki specjalne nie zostaną skonwertowane na symbole, przeglądarka zinterpretuje je jako HTML i na ekranie ukaże się wynik tej interpretacji zamiast właściwych znaczników. Dzięki konwersji wykonanej przez akcję specjalizowaną strona zostaje jednak przetworzona poprawnie (rysunek 7.).



Rysunek 7. Kod HTML przetworzony przez akcję <ora:encodeHTML>

Poza statycznym tekstem ciało akcji może zawierać dowolny element JSP. Bardziej realistyczny przykład wykorzystania tej akcji to umieszczenie na stronie JSP tekstu pobranego z bazy danych, bez przejmowania się tym, w jaki sposób znaki specjalne będą zinterpretowane przez przeglądarkę. Klasa obsługi znacznika jest całkiem banalna, co ukazano poniżej:

```
package com.ora.jsp.tags.generic;

import java.io.*;
import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;
import com.ora.jsp.util.*;

public class EncodeHTMLTag extends BodyTagSupport {

    public int doAfterBody() throws JspException {
        BodyContent bc = getBodyContent();
        JspWriter out = getPreviousOut();
        try {
            out.write
            (toHTMLString(bc.getString()));
        }
        catch (IOException e) {} // Ignore
        return SKIP_BODY;
    }

    private String toHTMLString(String in) {
        StringBuffer out = new StringBuffer();
        for (int i = 0; in != null && i <
        in.length();
          i++) {
            char c = in.charAt(i);
            if (c == '\'') {
                out.append("&#39;");
            }
            else if (c == '\"') {
                out.append("&#34;");
            }
            else if (c == '<') {
                out.append("&lt;");
            }
            else if (c == '>') {
                out.append("&gt;");
            }
            else if (c == '&') {
                out.append("&amp;");
            }
```

```
                else {
                    out.append(c);
                }
            }
            return out.toString();
        }
    }
```

Akcja nie ma żadnych atrybutów, w klasie nie są więc potrzebne żadne zmienne składowe ani metody je ustawiające. Można wykorzystać wszystkie metody interfejsu `BodyTag` implementowane przez klasę `BodyTagSupport`, z wyjątkiem metody `doAfterBody()`.

W metodzie `doAfterBody()` korzystamy z dwu metod narzędziowych dostarczanych przez klasę `BodyTagSupport`. Metoda `getBodyContent()` zwraca referencję do obiektu `BodyContent` zawierającego wyniki przetworzenia ciała akcji. Metoda `getPreviousOut()` zwraca obiekt `BodyContent` dla akcji zewnętrznej (jeśli taka istnieje) lub główny obiekt `JspWriter`, jeżeli akcja znajduje się na najwyższym poziomie w stronie.

Można by się zastanawiać, dlaczego metoda ta nosi nazwę `getPreviousOut()`, a nie `getOut()`. Nazwa ta ma na celu podkreślenie tego, że chcemy wykorzystać obiekt przypisany jako wyjście dla elementu *zewnętrznego* w hierarchii zagnieżdżonych elementów akcji. Załóżmy, że na stronie występują następujące elementy:

```
<xmp:foo>
    <xmp:bar>
      Jakiś tekst szablonowy
    </xmp:bar>
</xmp:foo>
```

Kontener sieciowy najpierw tworzy obiekt `JspWriter` i przypisuje go zmiennej `out` strony. Kiedy napotyka akcję `<xmp:foo>`, tworzy obiekt `BodyContent` i tymczasowo przyporządkowuje go zmiennej `out`. Potem tworzy kolejny obiekt `BodyContent` dla akcji `<xmp:bar>` i ponownie przypisuje go zmiennej `out`. Kontener zapamiętuje tę hierarchię przypisań. Tekst szablonowy i to, co przekazują na wyjście elementy standardowych akcji, trafia do bieżącego obiektu wyjścia. Każdy element może uzyskać dostęp do przypisanego mu obiektu `BodyContent`, wywołując metodę `getBodyContent()` i odczytując zawartość. Dla elementu `<xmp:bar>` zawartością jest tekst szablonowy. Po przetworzeniu jego treści może zapisać wynik w ciele znacznika `<xmp:foo>`, docierając do przypisanego mu obiektu `BodyContent` przez wywołanie metody `getPreviousOut()`. Ostatecznie element `<xmp:foo>` przetwarza treść dostarczoną mu przez `<xmp:bar>` i wypisuje ją do obiektu wyjścia najwyższego poziomu: obiektu `JspWriter`, do którego dostęp otrzymuje poprzez wywołanie metody `getPreviousOut()`.

Klasa obsługi znacznika w tym przykładzie konwertuje wszystkie znaki specjalne, jakie napotka w przypisanym sobie obiekcie `BodyContent`, za pomocą metody `toHTMLString()`. Wykorzystując metodę `getString()` uzyskuje dostęp do zawartości `BodyContent` i używa jej jako argumentu dla `toHTMLString()`. Wynik jest zapisywany do obiektu `JspWriter` otrzymanego przez wywołanie `getPreviousOut()`.

Przedstawiona powyżej przykładowa metoda `doAfterBody()` zwraca stałą `SKIP_BODY`, co sygnalizuje kontenerowi, że w następnej kolejności powinien wywołać `doEndTag()`. W klasie obsługi akcji, przeprowadzającej iterację po treści znacznika,

metoda `doAfterBody()` może zwrócić `EVAL_BODY_TAG`. Wtedy kontener przeprowadza ponowne wartościowanie ciała, zapisując wynik do obiektu `BodyContent` odpowiadającego elementowi, i wywołuje metodę `doAfterBody()`. Proces ten jest powtarzany, dopóki metoda `doAfterBody()` nie zwróci stałej `SKIP_BODY`.

Akcje tworzące obiekty

Akcje mogą współpracować ze sobą za pośrednictwem obiektów dostępnych w standardowych zasięgach JSP (strony, żądania, sesji i aplikacji). Przykład tego typu współpracy stanowi działanie trzech standardowych akcji JSP: `<jsp:useBean>`, `<jsp:setProperty>` i `<jsp:getProperty>`. Akcja `<jsp:useBean>` tworzy nowy obiekt i udostępnia go w jednym z zasięgów JSP. Dwie pozostałe mogą następnie uzyskać dostęp do właściwości obiektu, szukając go w zasięgach. Poza udostępnieniem obiektu w jednym z zasięgów, akcja `<jsp:useBean>` czyni go także osiągalnym jako zmienną skryptową, tak, aby mogły mieć do niego dostęp elementy skryptowe w obrębie strony.

Według specyfikacji JSP 1.1 zmienna utworzona przez akcję musi mieć nazwę taką, jak wartość atrybutu `id`. Wartość ta musi być unikalna w obrębie strony. Skoro będzie wykorzystywana jako zmienna skryptowa, musi również stosować się do reguł nazewnictwa zmiennych języka skryptowego. W Javie oznacza to, że zaczyna się literą, po której następuje kombinacja liter i cyfr, i nie może zawierać znaków specjalnych, takich jak kropka lub znak dodawania. Atrybut używany w innej akcji do odwoływania się do tej zmiennej może mieć dowolną nazwę, jednak według konwencji przyjętej dla akcji standardowych nosi on nazwę `name`.

W celu utworzenia zmiennej skryptowej akcja specjalizowana musi współpracować z kontenerem sieciowym. Aby zrozumieć, jak działa ten mechanizm, warto przypomnieć, że strona JSP jest przekształcana przez kontener w serwlet. W pierwszej kolejności kontener generuje kod, w którym deklarowana jest zmienna skryptowa dla tworzonego serwletu, i przypisuje tej zmiennej wartość. Aby to zrobić, musi znać nazwę zmiennej i jej typ w Javie. Informację tę należy dostarczyć kontenerowi za pośrednictwem podklasy `TagExtraInfo` dla akcji specjalizowanej. Kontener wywołuje metodę `getVariableInfo()` podklasy `TagExtraInfo` zdefiniowanej dla specjalizowanej akcji, kiedy konwertuje stronę JSP na serwlet. Metoda ta zwraca tablicę instancji `VariableInfo`, dostarczając informacji wymaganych dla zmiennych tworzonych przez akcję specjalizowaną. Następnie klasa obsługi znacznika musi umieścić obiekt w jednym z zasięgów JSP, wykorzystując metodę `setAttribute()` obiektu `PageContext`. Wygenerowany kod używa potem metody `findAttribute()`, aby odzyskać obiekt i przypisać go zmiennej skryptowej.

Klasa TagExtraInfo

Nazwa klasy:	`javax.servlet.jsp.tagext.TagExtraInfo`
Rozszerza:	—
Implementuje:	—
Implementowana przez:	wewnętrzne klasy zależne od kontenera. Większość kontenerów wykorzystuje wzorcową implementację klasy, stworzoną dla projektu Apache Jakarta

Opis

Podklasa klasy `TagExtraInfo` musi być stworzona i zadeklarowana w TLD dla akcji specjalizowanych, które tworzą zmienne skryptowe lub wymagają dodatkowego czasu przy translacji na sprawdzenie poprawności atrybutów znacznika. Kontener sieciowy tworzy egzemplarz obiektu `TagExtraInfo` w fazie tłumaczenia.

Konstruktor

`public TagExtraInfo()`
Tworzy nową instancję klasy `TagExtraInfo`.

Metody

`public TagInfo getTagInfo()`
Zwraca instancję `TagInfo` dla akcji specjalizowanej związanej z bieżącym egzemplarzem `TagExtraInfo`. Referencja do obiektu `TagInfo` jest ustawiana przy wykorzystaniu metody `setTagInfo()`, wywoływanej przez kontener sieciowy.

`public VariableInfo[]`
`getVariableInfo(TagData data)`
Zwraca tablicę `VariableInfo[]` zawierającą informacje o zmiennych skryptowych, utworzonych przez klasę obsługi znacznika związaną z bieżącym egzemplarzem `TagExtraInfo`. Domyślna implementacja zwraca pustą tablicę. Podklasa musi nadpisać tę metodę, jeśli odpowiadająca jej klasa obsługi tworzy zmienne skryptowe.

`public boolean isValid(TagData data)`
Zwraca `true`, jeśli zbiór wartości atrybutów wyspecyfikowanych dla akcji specjalizowanej związanej z bieżącym

egzemplarzem `TagExtraInfo` jest poprawny, w przeciwnym wypadku — `false`. Domyślna implementacja zwraca `true`. Podklasa może nadpisać tę metodę, jeżeli sprawdzenie poprawności dokonane przez kontener sieciowy na podstawie informacji zawartych w TLD jest niewystarczające.

`public void setTagInfo(TagInfo tagInfo)`
Ustawia referencję do obiektu `TagInfo` dla tego egzemplarza. Tę metodę wywołuje kontener sieciowy przed wykorzystaniem jakiejkolwiek innej metody.

Klasa VariableInfo

Nazwa klasy:	`javax.servlet.jsp.tagext.VariableInfo`
Rozszerza:	—
Implementuje:	—
Implementowana przez:	wewnętrzne klasy zależne od kontenera. Większość kontenerów wykorzystuje wzorcową implementację klasy, stworzoną dla projektu Apache Jakarta

Opis

Instancje `VariableInfo` są tworzone przez podklasy `TagExtraInfo` w celu opisania każdej zmiennej skryptowej stworzonej przez odpowiednią klasę obsługi znacznika.

Konstruktor

`public VariableInfo(String varName, String className, boolean declare, int scope)`
Tworzy nową instancję według wyspecyfikowanych wartości.

Metody

`public String getClassName()`
Zwraca typ Javy dla zmiennej skryptowej.

`public boolean getDeclare()`
Zwraca wartość `true`, jeśli kontener sieciowy tworzy instrukcję deklaracji dla zmiennej skryptowej; w przeciwnym wypadku zwraca `false` (wykorzystywana, jeśli zmienna została już zadeklarowana przez inną klasę obsługi znacznika i jest jedynie aktualizowana przez klasę obsługi odpowiadającą podklasie `TagExtraInfo`, która stworzyła ten egzemplarz `VariableInfo`).

`public int getScope()`
Zwraca jedną ze stałych: `AT_BEGIN` (udostępniającą zmienną skryptową od otwarcia znacznika do końca strony JSP), `AT_END` (udostępniającą zmienną od zamknięcia znacznika do końca strony JSP) lub `NESTED` (udostępniającą zmienną jedynie między otwarciem i zamknięciem znacznika).

`public String getVarName()`
Zwraca nazwę zmiennej.

Przykład

Poniżej podano przykład podklasy `TagExtraInfo` dla akcji specjalizowanej, która tworzy zmienną o nazwie wskazanej przez atrybut `id` oraz typie Javy wyspecyfikowanym przez atrybut `className`:

```
package com.ora.jsp.tags.generic;
import javax.servlet.jsp.tagext.*;
public class UsePropertyTagExtraInfo
  extends TagExtraInfo {
  public VariableInfo[] getVariableInfo(TagData
  ↳data) {
    return new VariableInfo[] {
      new VariableInfo(
        data.getAttributeString("id"),
        data.getAttributeString("className"),
        ↳true,
        VariableInfo.AT_END)
    };
  }
}
```

Kontener sieciowy wywołuje metodę `getVariableInfo()` podczas fazy translacji. Zwraca ona tablicę obiektów `VariableInfo`, po jednym dla każdej zmiennej wprowadzonej przez klasę obsługi znacznika.

Klasa `VariableInfo` to prosty komponent bean z czterema właściwościami, inicjalizowanymi przez wartości przekazywane jako argumenty do konstruktora: `varName`, `className`, `declare` i `scope`; `varName` jest nazwą zmiennej skryptowej, a `className` to jej klasa.

Właściwość `declare` jest typu `boolean`, przy czym wartość `true` oznacza, że akcja tworzy nową zmienną skryptową (czyli do generowanego serwletu trzeba dodać instrukcję deklaracji zmiennej). Wartość `false` oznacza, że zmienna została stworzona wcześniej przez inną akcję lub inne wystąpienie tej samej akcji, a zatem wygenerowany kod zawiera już stosowną deklarację. W tym przypadku kontener jedynie przyporządkowuje zmiennej nową wartość.

Właściwość `scope` nie ma nic wspólnego z zasięgami JSP, z którymi mieliśmy do czynienia do tej pory (zasięgi strony,

żądania, sesji i aplikacji). Definiuje ona natomiast, gdzie nowa zmienna będzie dostępna dla elementów skryptowych JSP. Wartość `AT_BEGIN` oznacza, że zmienna jest dostępna począwszy od otwarcia znacznika akcji, `AT_END` zaś — dopiero po zamknięciu znacznika. Z kolei stała `NESTED` udostępnia zmienną jedynie w ciele akcji, pomiędzy otwarciem i zamknięciem znacznika. Wartość `scope` kontroluje zatem, w jakim miejscu jest generowany kod deklaracji zmiennej i przypisania jej wartości, a klasa obsługi znacznika musi zadbać o to, by zmienna była dostępna w jednym ze standardowych zasięgów JSP w stosownym czasie; np. wykorzystując metodę `doStartTag()` dla zasięgów `AT_BEGIN` i `NESTED` i w metodzie `doEndTag()` dla zasięgu `AT_END`. Dla obiektu `BodyTag` przeprowadzającego iteracje po ciele wartość może także być aktualizowana w metodzie `doAfterBody()`, aby dostarczać nowej wartości dla każdej iteracji.

Sprawdzanie poprawności atrybutów

W poprzednim przykładzie klasa `UsePropertyTagExtraInfo` przypisywała właściwościom `varName` i `className` komponentu bean `VariableInfo` wartości atrybutów `id` oraz `className`, wyspecyfikowane przez autora na stronie JSP. Działo się to przy wykorzystaniu kolejnej prostej klasy o nazwie `TagData`, przekazywanej jako argument do metody `getVariableInfo()`. Obiekt `TagData` jest tworzony przez kontener w celu dostarczenia podklasy `TagExtraInfo`, zawierającej informacje o wszystkich atrybutach akcji dostarczonych przez autora strony JSP.

Obiekt `TagData` jest także przekazywany jako argument do metody `isValid()` klasy `TagExtraInfo`. Metoda ta jest wywoływana przez kontener WWW podczas fazy translacji, aby

umożliwić użytkownikowi zaimplementowanie własnych reguł sprawdzania poprawności atrybutów specjalizowanej akcji. Kontener przeprowadza proste sprawdzenie poprawności opierając się na informacjach zawartych w pliku TLD. Akcja specjalizowana może mieć jednak opcjonalne atrybuty, które wykluczają się wzajemnie lub są od siebie zależne. Zachodzi wtedy potrzeba implementacji metody `isValid()` w podklasie `TagExtraInfo` i dostarczenia własnego kodu sprawdzającego poprawność.

Klasa `TagData` ma dwie interesujące metody. Metoda `getAttributeString()` zwraca po prostu wyszczególniony atrybut jako `String`. Jednak wartości niektórych atrybutów mogą być określone poprzez wyrażenie JSP (atrybut „w czasie żądania"), nie zaś poprzez obiekt `String`. Ponieważ taka wartość nie jest znana w fazie translacji, klasa `TagData` udostępnia metodę `getAttribute()`, aby zaznaczyć, czy atrybut ma wartość dosłowną, nadawaną w czasie żądania, czy też w ogóle nie został ustawiony. Metoda `getAttribute()` zwraca wartość typu `Object`. Jeżeli wartość atrybutu jest określana w czasie żądania, zwracany jest specjalny obiekt `REQUEST_TIME_VALUE`. W przeciwnym wypadku metoda zwraca obiekt `String` lub `null`, jeśli atrybut nie został ustawiony.

Klasa TagData

Nazwa klasy:	`javax.servlet.jsp.tagext.TagData`
Rozszerza:	—
Implementuje:	`Cloneable`

Implementowana przez: wewnętrzne klasy zależne od kontenera. Większość kontenerów wykorzystuje wzorcową implementację klasy, stworzoną dla projektu Apache Jakarta

Opis

Instancje `TagData` są tworzone przez kontener WWW w fazie translacji. Dostarczają one informacji na temat wartości atrybutów (wyspecyfikowanych dla specjalizowanej akcji) podklasie `TagExtraInfo` (jeśli taka istnieje) odpowiedniej klasy obsługi znacznika.

Konstruktory

`public TagData(Object[][] atts)`
: Tworzy nową instancję z parami nazw atrybutów i wartości, określonymi przez macierz `Object[][]`. Element 0 każdej tablicy `Object[]` zawiera nazwę, a element 1 — wartość lub stałą `REQUEST_TIME_VALUE` (jeżeli wartość atrybutu została zdefiniowana jako określana w czasie żądania, lub wyrażenie JSP).

`public TagData(java.util.HashTable attrs)`
: Tworzy nową instancję z parami nazw atrybutów i wartości, określonymi przez tablicę haszującą `HashTable`.

Metody

`public Object getAttribute(String attName)`
: Zwraca wartość wskazanego atrybutu jako `String` lub obiekt `REQUEST_TIME_VALUE` (jeżeli wartość atrybutu jest określana w czasie żądania lub jako wyrażenie JSP).

`public String getAttributeString(String attName)`

Zwraca wartość wskazanego atrybutu jako `String`. Zgłaszany jest wyjątek `ClassCastException`, jeżeli wartość atrybutu jest definiowana w czasie żądania lub stanowi wyrażenie JSP.

`public String getId()`

Zwraca wartość atrybutu o nazwie `id` jako `String` (lub `null`, jeśli go nie odnajdzie).

`public void setAttribute(String attName, Object value)`

Przypisuje wskazanemu atrybutowi podaną wartość.

Przykład

Kiedy kontener WWW sprawdził wszystko, co mógł skontrolować samodzielnie na podstawie informacji o atrybutach zawartych w pliku TLD, szuka podklasy `TagExtraInfo` zdefiniowanej dla akcji specjalizowanej przez element `<teiclass>`. Jeżeli została ona zdefiniowana, kontener umieszcza wszystkie informacje o atrybutach w egzemplarzu `TagData` i wywołuje metodę `isValid()` podklasy `TagExtraInfo`:

```
public boolean isValid(TagData data) {
    // Mutually exclusive attributes
    if (data.getAttribute("attr1") != null &&
        data.getAttribute("attr2" != null) {
        return false;
    }

    // Dependent optional attributes
    if (data.getAttribute("attr3") != null &&
        data.getAttribute("attr4" == null) {
        return false;
    }
    return true;
}
```

Podklasa `TagExtraInfo` może wykorzystać obiekt `TagData`, aby sprawdzić, czy wszystkie zależności między atrybutami zostały zachowane, tak jak w podanym powyżej przykładzie. Niestety, w JSP 1.1 nie istnieje sposób, aby wygenerować stosowny komunikat o błędzie; metoda może jedynie zwrócić wartość `false`, aby zaznaczyć, że coś jest nie w porządku. Miejmy nadzieję, że w kolejnych wersjach JSP ta niedogodność zostanie usunięta.