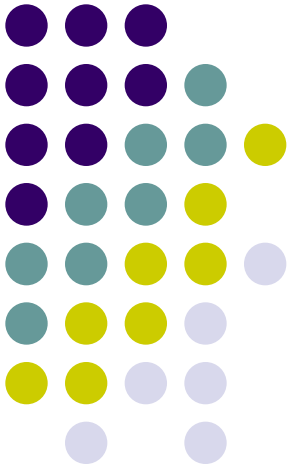
Прикладное программирование

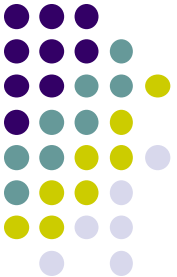
Лекция 17
Разработка собственных (custom) тегов



Курс читается при поддержке:

itransition
SOFTWARE DEVELOPMENT COMPANY

17. Custom-теги
Общее представление

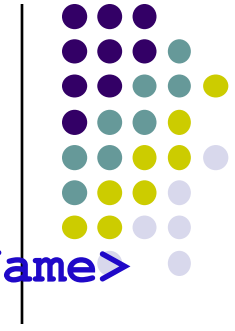


Custom-теги (теговые расширения JSP) предлагают средства инкапсуляции функциональности в JSP, и обладают следующими преимуществами:

- Повторное использование – каждый тег представляет собой небольшой компонент, предназначенный для решения узкоспециализированной задачи;
- Лёгкость восприятия – восприятие тегов является гораздо более простым чтения Java-кода;
- Лёгкость поддержания – устранение дублирования кода из приложения, т.е. вместо повторения одной функциональности на 10 страницах мы выделяем её в отдельный компонент и обращаемся к ней с 10 страниц.

17. Custom Tags

Терминология и основные понятия

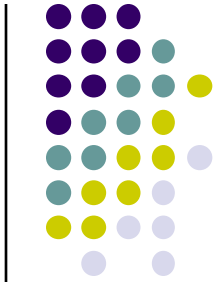


Тег – фрагмент разметки, начинающийся с **`<prefix:tagName>`** и заканчивающийся **`</prefix:tagName>`**.

Тело тега – разметка между открывающим и закрывающим элементами. Существуют следующие типы тел:

- *empty* – между начальным и конечным тегами ничего нет;
- *JSP* – тело тега может содержать что угодно (HTML, скриптлеты, другие теги и пр.);
- *scriptless* – тело тега аналогично JSP, но не может содержать скриптлеты (в случае обнаружения генерируется исключение);
- *tagdependent* – тело тега рассматривается как обычный текст, и тег имеет полный контроль над тем, как его тело включается или вычисляется на странице.

Атрибуты – дополнительные параметры тегов, записываются в открывающем тег элементе в виде пар **`attributeName="value"`**.



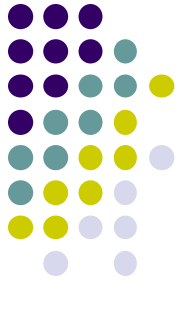
JavaBean – обычный Java-класс, имеющий ряд свойств и методы (сеттеры и геттеры) для работы с ними. Широко используются в JSP, т.к. стандарт JSP предлагает набор стандартных механизмов работы с JavaBeans.

Не смотря на то, что в теле методов JavaBean можно реализовать определённую логику, JavaBeans не обладают доступом к контексту выполнения веб-приложения (объектам **`pageContext`**, **`request`**, **`response`**, **`session`**).

JavaBeans предназначены для представления и хранения некоторого состояния объектов приложения.

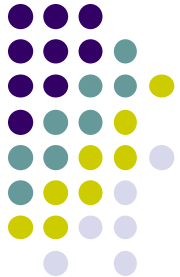
17. Custom Tags

Пример: обычный Bean и Bean с логикой



```
class PwdBean {
  private String login;
  private String pwd;
  public PwdBean() {…}
  public setLogin(
    String login) {
    this.login = login;
  }
  public String getPwd() {
    return pwd;
  }
}
```

```
class PwdBean {
  private String login;
  private String pwd;
  public PwdBean() { … }
  public setLogin(String login) {
    this.login = login;
  }
  public String getPwd() {
    // Подключиться к СУБД
    // Запросить из СУБД пароль
    // для логина login
    // String pwd = compute();
    return pwd;
  }
}
```

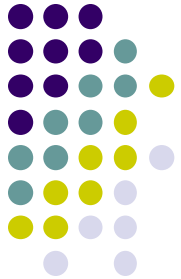


В отличие от JavaBeans, **теги** являются технологией, специфичной для веб-приложений. Они предназначены для генерации элементов представления, и поэтому напрямую инкапсулируют некоторое поведение. Кроме того, теги обладают сведениями об окружении, в котором функционируют (о контексте, неявно объявленных и доступных объектах и т.п.).

Теги используются для реализации действий над JavaBeans, а также логики представления информации.

Архитектура JSP позволяет создавать два типа тегов: простые и классические. С точки зрения разработчика, между ними существуют принципиальные различия.

В рамках подхода классических тегов (изначально существовавшего подхода) функциональность помещается в Java-классы, реализующие интерфейс **`javax.servlet.jsp.tagext.Tag`**. Эти классы, содержащие предоставляемую тегом функциональность, называются обработчиками тегов. Одной из основных проблем разработки классических тегов является сложность интерфейса **`Tag`**, жизненного цикла подобных тегов и семантики их использования в контейнере.

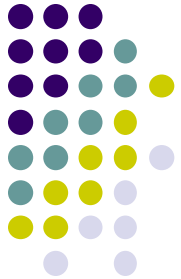


Реакцией на требования разработчиков по упрощению процесса разработки и использования тегов стали два новых способа создания тегов: теговые файлы и простые теги.

Теговый файл представляет фрагмент JSP, содержащий разметку или JSP-код, который планируется многократно использовать. Этот фрагмент делается доступным посредством тега. В прошлом аналогичный результат достигался вынесением фрагментов кода из JSP в отдельный файл и подключения его по мере необходимости.

В рамках подхода **простых тегов** обработчики тегов строятся на основе классов, реализующих интерфейс **`javax.servlet.jsp.tagext.SimpleTag`**.

17. Custom Tags
Теговые файлы



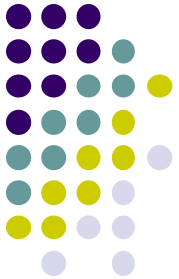
Для определения тега с помощью тегового файла следует желаемый фрагмент разметки (который будет представлять тег) сохранить в отдельном файле с расширением *.tag* в папке */WEB-INF/tags*. В качестве имени тега будет использоваться имя файла.

Перед использованием теги на основе теговых файлов также необходимо подключить. Полагая, что файлы находятся в папке */WEB-INF/tags*, можно сказать, что папка *tags* представляет множество тегов (или библиотеку). Исходя из этого, подключить теги можно с помощью директивы **`taglib`**, например:

```
<%@ taglib prefix="tags" tagdir="/WEB-INF/tags" %>
<tags:copyright/>
```

17. Custom Tags

Определение атрибутов в теговых файлах



При использовании теговых файлов существует возможность передачи им некоторых параметров, подставляемых в шаблонную разметку, определяемую тегом. Для задания перечня атрибутов в начало тегового файла следует вставить следующие директивы:

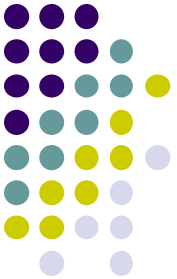
```
<%@ attribute name="<Имя>" required="true|false"
rtexprvalue="true|false" %>
```

Пример: Тег определён в файле *highlight.tag*

```
<%@ attribute name="color"
required="true" rtexprvalue="false" %>
<font color="${color}">
<strong><jsp:doBody/></strong>
</font>
```

17. Custom Tags

Передача атрибутов при использовании теговых файлов

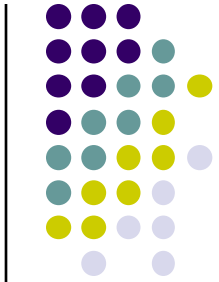


Передача аргументов при использовании теговых файлов может осуществляться как с использованием нотации встроенных атрибутов, например:

```
<tags:highlight color="#ff0000">Будьте
внимательны</tag:highlight>
```

так и с использованием нотации вложенных тегов:

```
<tags:highlight>
   <jsp:attribute name="color">green</jsp:attribute>
   <jsp:body>
      Жили-были старик со старухой, и была у них
внучка Ряба, та ещё курица. Пошла как-то внучка в
музей, и «снесла» оттуда яичко Фаберже, золотое.
Продали они его «перекупам», и стали жить-поживать,
добра наживать!
   </jsp:body>
</tags:highlight>
```

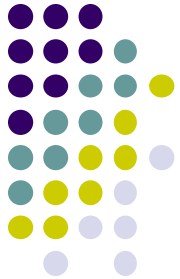


В отличие от теговых файлов, логика простых тегов инкапсулируется в Java-классах, реализующих интерфейс **`javax.servlet.jsp.tagext.SimpleTag`**. Объявление такого интерфейса служит двум целям:

- предоставляет тегу сведения об окружении, в котором он выполняется;
- предоставляет метод для выполнения инкапсулированной функциональности.

Характеристики тега, такие как его имя и список атрибутов, определяются в специальном дескрипторе библиотеки.

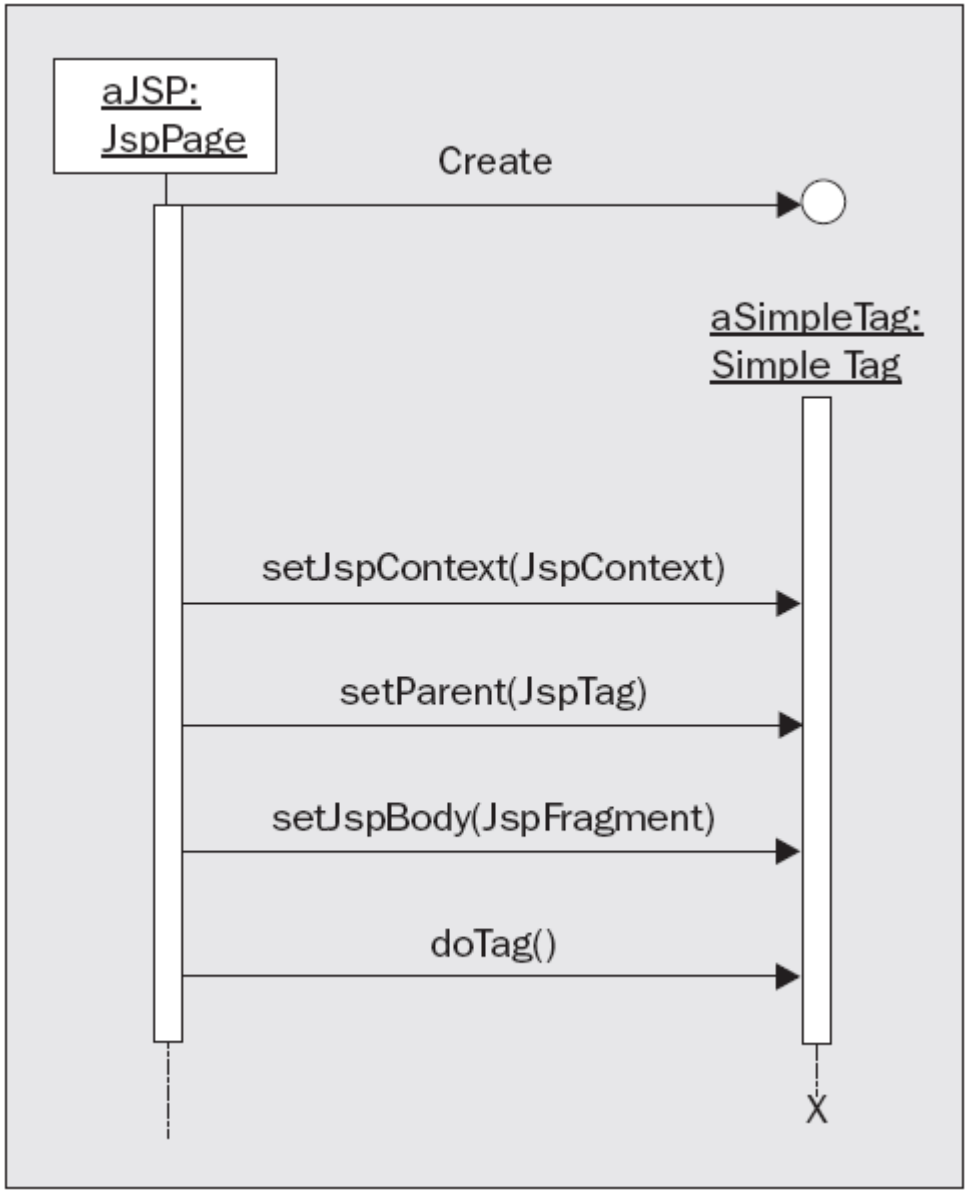
При использовании тега на странице создаётся экземпляр класса обработчика тега и выполняется обращение к его методам.

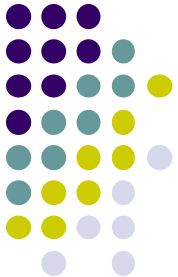


```
package javax.servlet.jsp.tagext;
public interface SimpleTag extends JspTag {
    public void doTag()
                throws JspException, IOException;
    public JspTag getParent();
    public void setJspBody(
                        JspFragment jspBody);
    public void setJspContext(
                        JspContext jspContext);
    public void setParent(JspTag parent);
}
```

17. Custom Tags

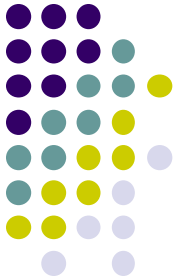
Жизненный цикл простого тэга





Хотя реализация методов интерфейса **`SimpleTag`** и не является сложной задачей, для удобства спецификация JSP предлагает класс **`javax.servlet.jsp.tagext.SimpleTagSupport`**, который предлагает стандартную реализацию для всех объявленных методов.

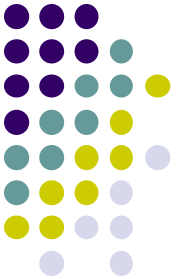
17. Custom Tags
Пример простого тега



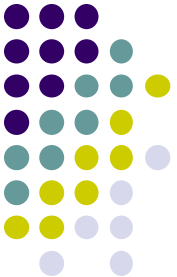
```
public class DateTimeTag extends
SimpleTagSupport {
  public void doTag()
          throws JspException, IOException {
    DateFormat df =
      DateFormat.getDateTimeInstance(
      DateFormat.MEDIUM, DateFormat.MEDIUM);
    // Вывести дату на страницу
    getJspContext().getOut().write(
                        df.format(new Date()));
  }
}
```

17. Custom Tags

Создание дескриптора тега



```
<?xml version="1.0" encoding="UTF-8" ?>
<taglib xmlns="…" xmlns:xsi="…"
xsi:schemaLocation="…" version="2.1">
<description>Моя первая библиотека
тегов</description>
<jsp-version>2.1</jsp-version>
<tlib-version>1.0</tlib-version>
<short-name>myLib</short-name>
<uri>http://ivan.ivanov/java/taglib/myLib</uri>
<tag>
  <name>datetime</name>
  <tag-class>bsu.rfe.DateTimeTag</tag-class>
  <body-content>empty</body-content>
  <description>Выводит текущие дату и
время</description>
</tag>
</taglib>
```

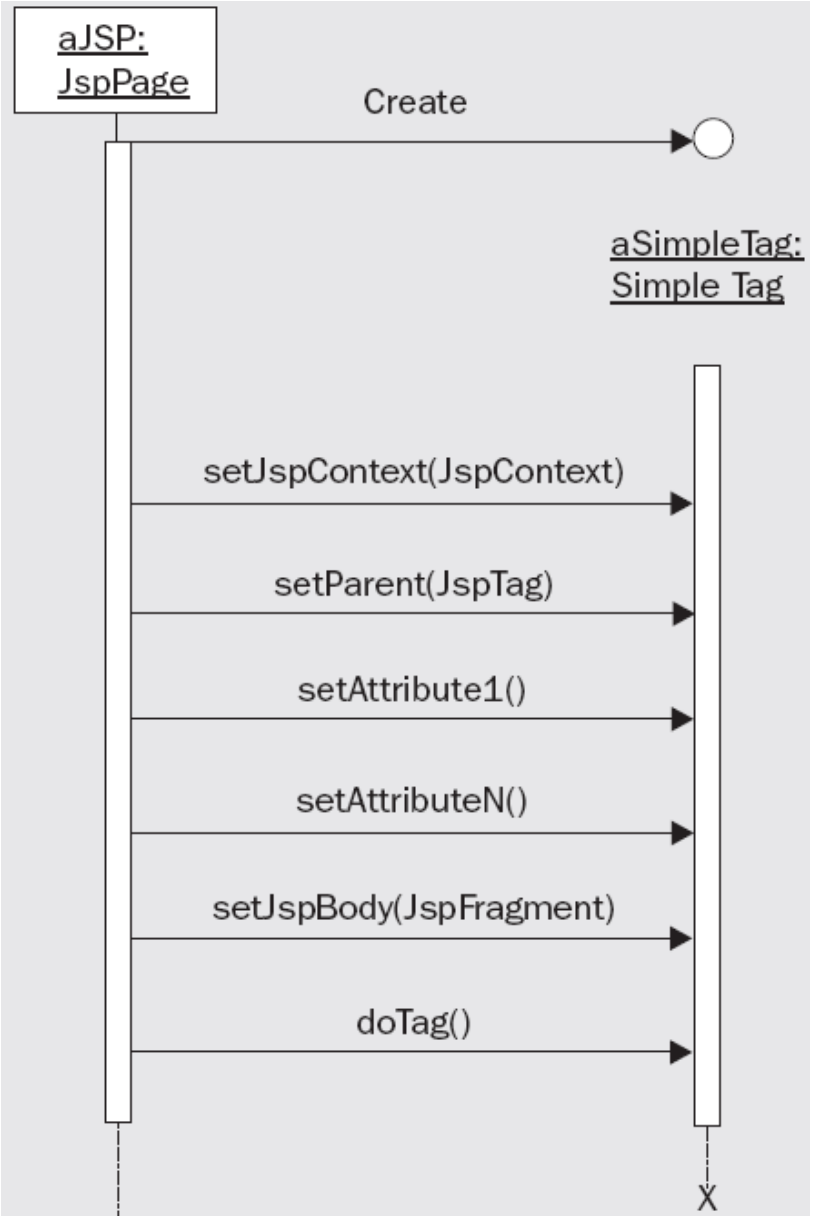


Передача в экземпляры классов обработчиков тегов значений атрибутов осуществляется посредством определения свойства и метода-сеттера для каждого поддерживаемого атрибута. Методы-сеттеры должны соответствовать стандартной конвенции именования JavaBean'ов, т.е. для поддержки атрибута с именем **`name`** типа **`String`** обработчик тега должен объявлять метод со следующей сигнатурой:

```
public void setName(String s);
```

17. Custom Tags

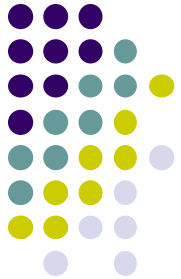
Жизненный цикл простого тега с атрибутами



17. Custom Tags

Соответствие типов атрибутов

Тип в Java	Сигнатура метода	Замечания
`String`	`public void setName(String s)`	
`char`	`public void setName(char c)`	Используется 1-я буква атрибута
`Character`	`public void setName(Character c)`	
`boolean`	`public void setName(boolean b)`	Если строка – `true` (без учёта регистра), то значение свойства – «истина»
`Boolean`	`public void setName(Boolean b)`	
`XXX = Byte, byte, Short, short, Integer, int, Long, long, Float, float, Double, double`	`public void setName(XXX var)`	Преобразование строки в любой числовой тип проводится по стандартным правилам Java преобразования строк в числа.

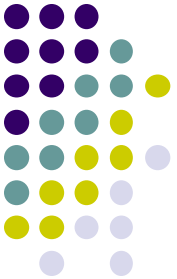


Тегу может передаваться и значение атрибута, являющееся ссылкой на объявленный в контексте JSP-страницы объект. В этом случае обработчик тега должен декларировать метод-сеттер:

```
public void setPropertyName(MyObjectType o)
```

17. Custom Tags

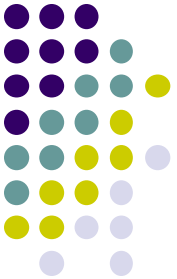
Описание атрибутов в дескрипторе тегов



Если тег поддерживает какие-либо атрибуты, то это необходимо указать в его описании в дескрипторе библиотеки тегов. Для каждого поддерживаемого тега необходимо добавить следующий блок:

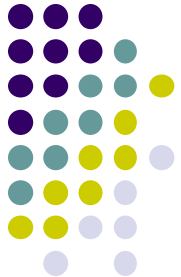
```
<attribute>
    <name>path</name>
    <required>true</required>
    <rtexprvalue>true</rtexprvalue>
</attribute>
```

- **name** – содержит имя атрибута,
- **required** – указывает на обязательность его задания;
- **rtexprvalue** (Runtime Expression Value) определяет, должно ли быть значение атрибута фиксированным (явно заданным) или вычисляться как результат некоторого выражения.



Для доступа к телу тега используется метод **getJspBody()**. Добавив в контекст JSP какие-либо дополнительные переменные, тег может попросить JSP-контейнер обработать содержание тела тега и отправить результат в JSP, например:

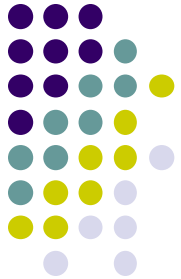
```
Iterator it = items.iterator();
while (it.hasNext()) {
  name = (String) it.next();
  JspFragment jspBody = getJspBody();
  if (jspBody != null) {
    getJspContext().setAttribute("name", name);
    jspBody.invoke(getJspContext().getOut());
  }
}
```



В отличие от теговых файлов и простых тегов, добвленных в спецификации JSP 2, существовавший изначально подход к созданию тегов называется теперь классическим, и, несмотря на большую сложность, по-прежнему используется в некоторых сценариях по причине большей гибкости.

Основные отличия классических тегов от простых заключаются в другом реализуемом интерфейсе – **`javax.servlet.jsp.tagext.Tag`** – и другом подходе к реализации класса-обработчика.

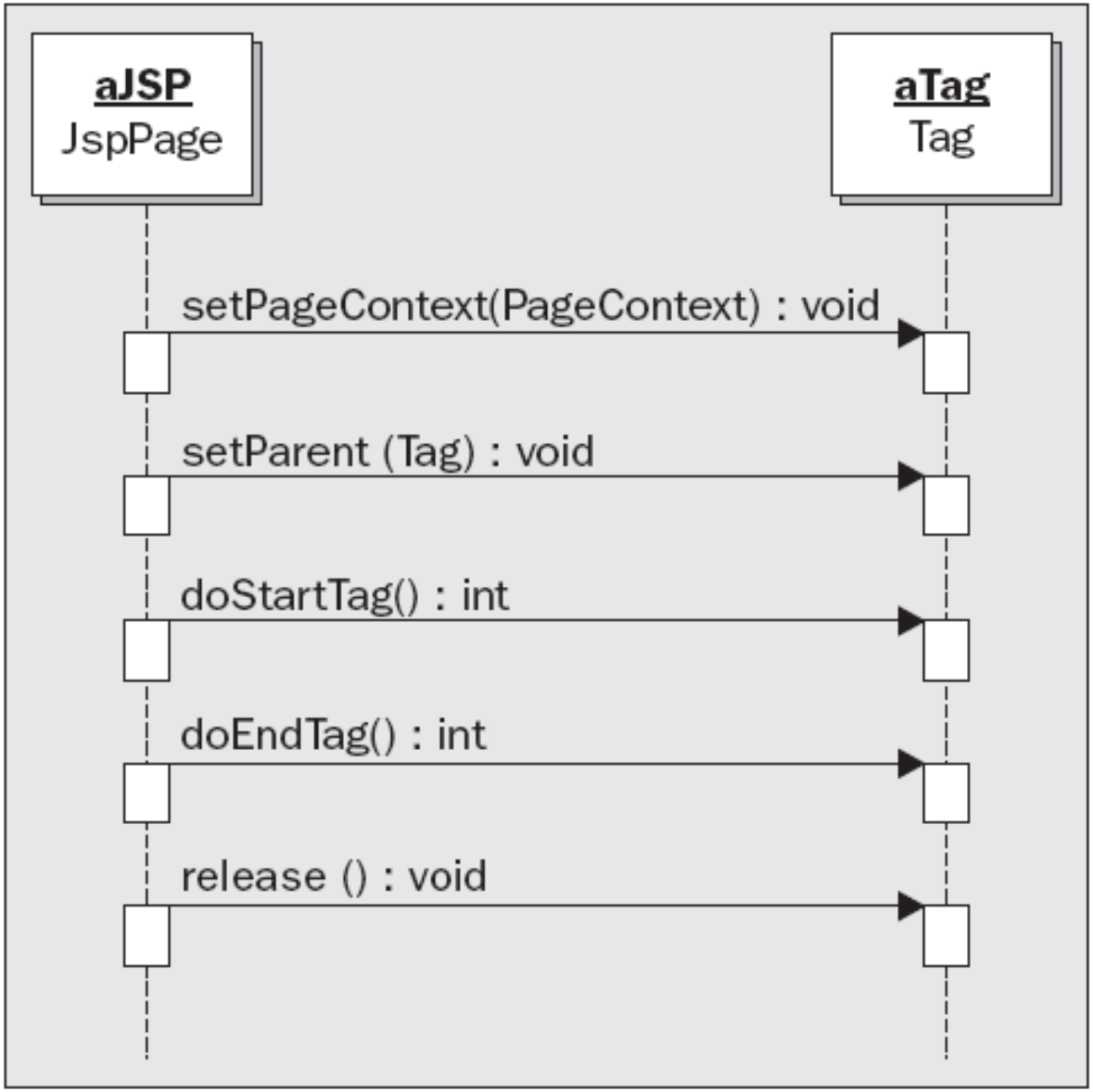
17. Custom Tags
Интерфейс Tag



```
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 pc);
   void setParent(Tag t);
   int doStartTag() throws JspException;
   int doEndTag() throws JspException;
   void release();
}
```

17. Custom Tags

Жизненный цикл классического тега



17. Custom Tags
Обработка классического тега

Метод **doStartTag()** вызывается при обнаружении на странице открывающего элемента тега. Значение, возвращаемое методом (простое целое), указывает JSP, как продолжать обработку.

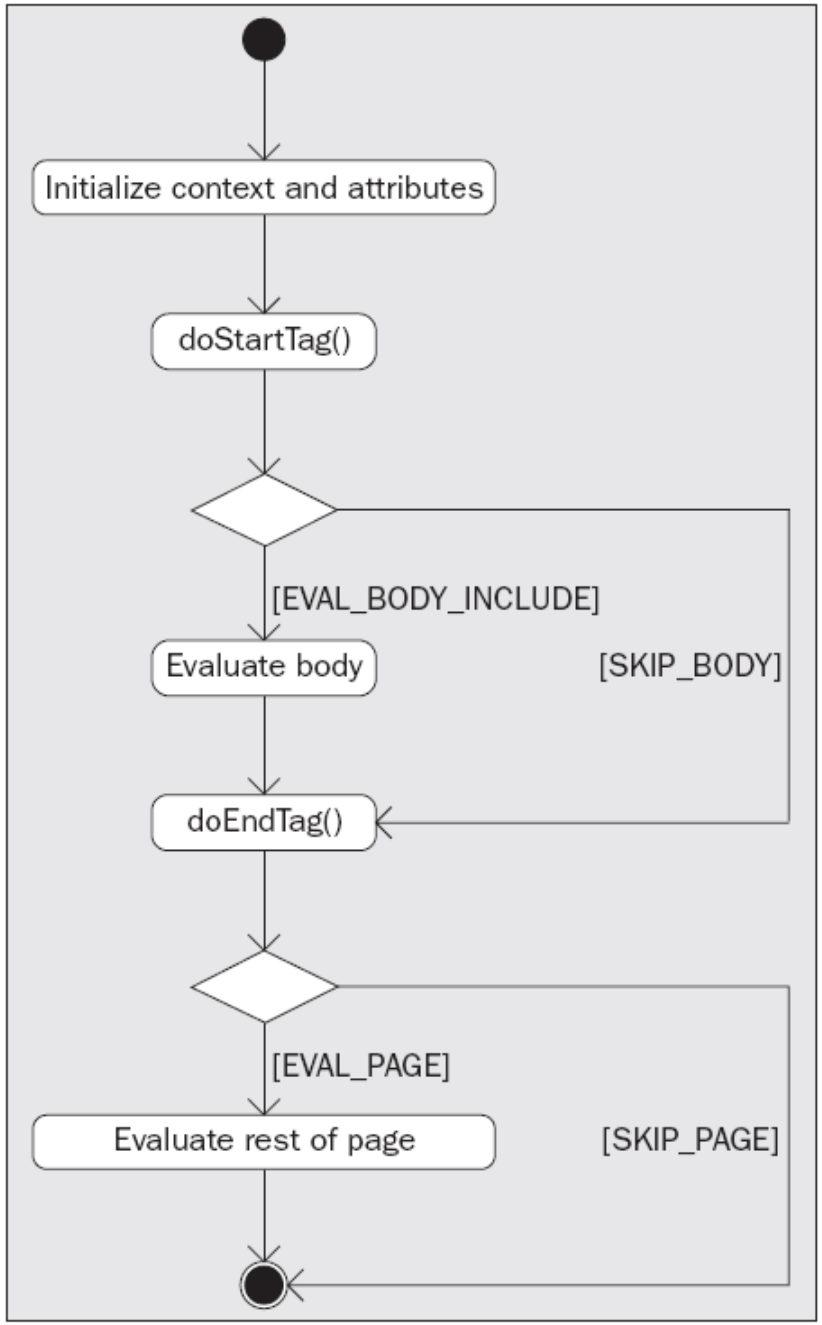
- **SKIP_BODY** указывает, что после вызова **doStartTag()** тело тега должно быть проигнорировано, после чего должна начаться обработка **doEndTag()**.
- **EVAL_BODY_INCLUDE** указывает, что тело тега должно быть обработано и вставлено на страницу.

Метод **doEndTag()** вызывается при обнаружении закрывающего элемента тега. Значение, возвращаемое методом, также сообщает JSP, как поступить дальше:

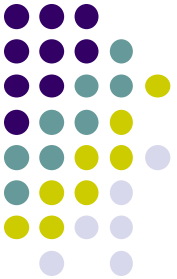
- **SKIP_PAGE** указывает на необходимость прекращения обработки страницы;
- **EVAL_PAGE** указывает на необходимость продолжения.

17. Custom Tags

Схема выполнения тега



Вспомогательный класс TagSupport

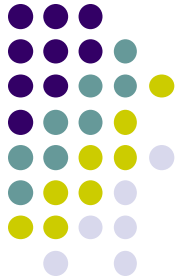


Несмотря на то, что интерфейс **`Tag`** содержит большее количество методов, чем **`SimpleTag`**, его реализация является тривиальной.

Тем не менее, для удобства спецификация JSP предлагает класс **`TagSupport`** для использования в качестве стартовой точки при построении собственных обработчиков тегов. Его реакциями по умолчанию являются **`SKIP_BODY`** в методе **`doStartTag()`** и **`EVAL_PAGE`** в методе **`doEndTag()`**.

17. Custom Tags

Пример классического тега



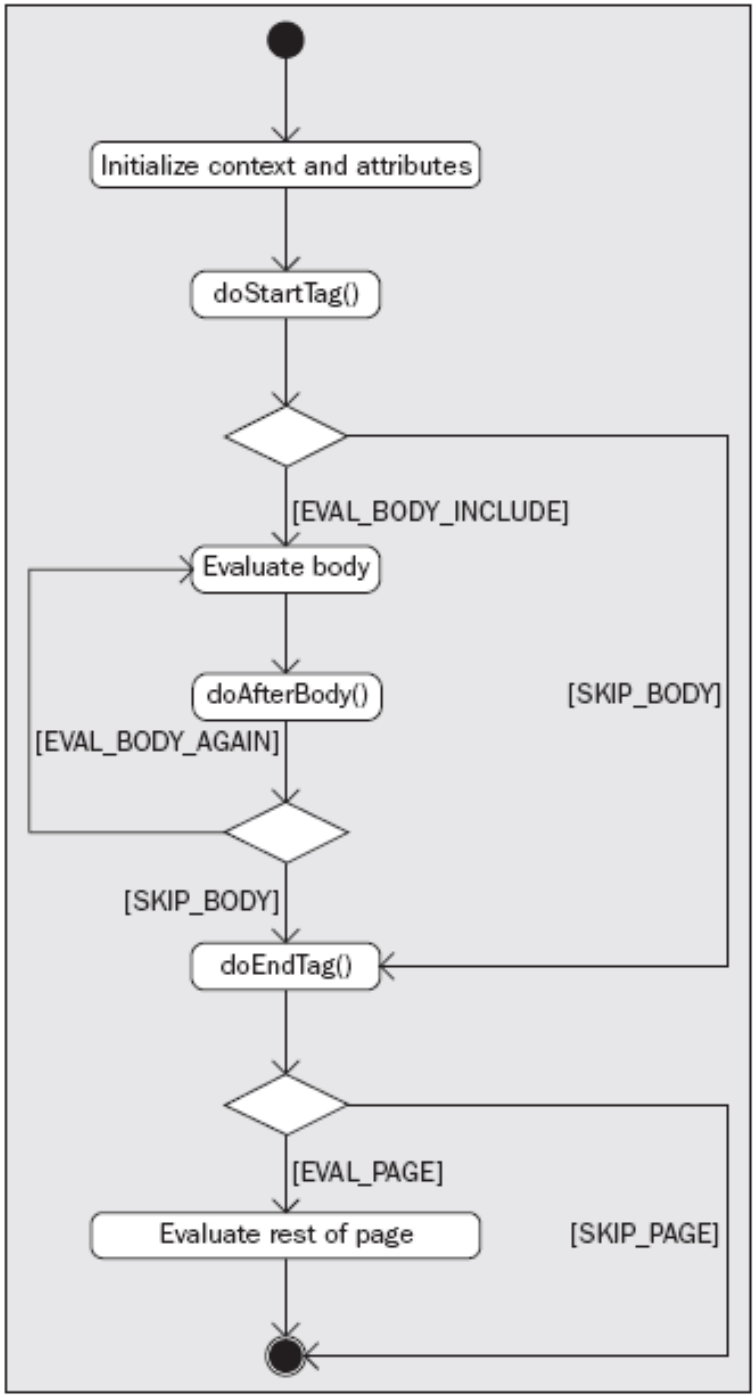
```
public class DateTimeTag extends TagSupport {
  public int doStartTag() throws JspException {
    DateFormat df =
      DateFormat.getDateTimeInstance(
         DateFormat.MEDIUM, DateFormat.MEDIUM);
    try {
      pageContext.getOut().write(
                          df.format(new Date()));
    } catch (IOException ioe) {
      throw new
             JspTagException(ioe.getMessage());
    }
    return SKIP_BODY;
  }
}
```

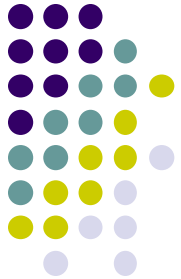
Ещё одним различием простых и классических тегов является возможность простого тега многократно обработать своё тело с помощью нескольких вызовов метода **invoke()** передаваемого в объекте **JspFragment** тела тега. С интерфейсом **Tag** это невозможно – возвращаемое из **doStartTag()** значение определяет, нужна обработка тела тега или нет, но не позволяет запросить многократную его обработку. Для преодоления этой проблемы в спецификации JSP 1.2 был добавлен интерфейс **IterationTag**.

```
public interface IterationTag extends Tag {
  public final static int EVAL_BODY_AGAIN=2;
  int doAfterBody() throws JspException;
}
```

17. Custom Tags

Жизненный цикл тега-итератора



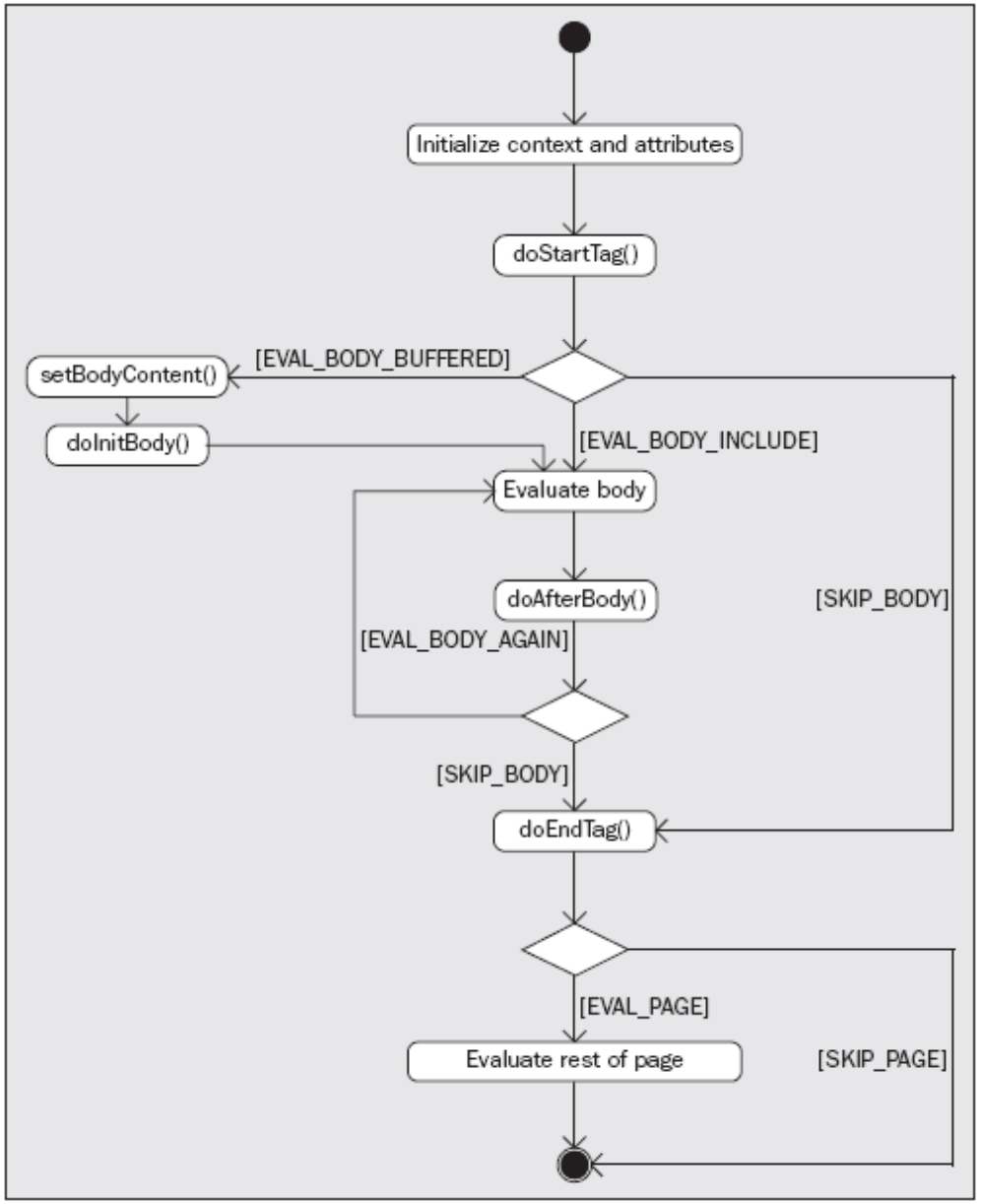


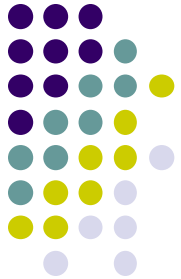
До настоящего момента рассматривались классические теги, способные обработать своё тело нуль, один или более раз. Это особенно полезно в случаях, когда тело тега является тривиальным, т.е. не требуется его преобразования перед выводом на страницу. Если же подобные преобразования необходимы, то следует говорить о реализации интерфейса **`javax.servlet.jsp.tagext.BodyTag`**.

```
public interface BodyTag extends IterationTag {
  public final static int EVAL_BODY_BUFFERED=2;
  void setBodyContent(BodyContent b);
  void doInitBody() throws JspException;
}
```

17. Custom Tags

Жизненный цикл тега с телом

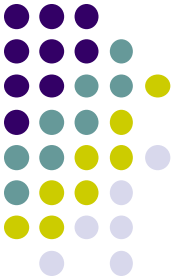




Если метод **doStartTag()** возвратил значение **EVAL_BODY_BUFFERED**, то вызывается метод **setBodyContent()**, чтобы обработчик тега смог получить ссылку на объект **BodyContent** и позже воспользоваться им

Класс **BodyContent** является наследником класса **JspWriter** и может рассматриваться в роли некоторого буфера, куда записывается выводимое содержание. С технической точки зрения, при обработке метода **setBodyContent()** JSP-контейнер заменяет стандартный поток вывода **JspWriter** на объект **BodyContent**. Это означает, что с данного момента и до конца обработки тега всё содержание будет записываться не на страницу, а во временный буфер.

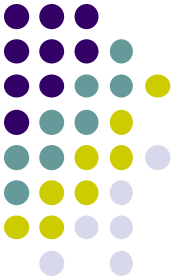
Для настройки каких-либо параметров перед обработкой тела тега вызывается метод **doInitBody()**.



Так как функциональность класса **`BodyTag`** несколько отличается от предоставляемой другими классическими тегами, в качестве стартовой точки для разработки классических тегов с телом применяется класс **`BodyTagSupport`**.

17. Custom Tags

Пример использования: фильтрация содержания



```
public class EmailAddressFilterTag extends
BodyTagSupport {
  public int doEndTag() throws JspException {
    if (bodyContent != null) {
      try {
        String content = bodyContent.getString();
        content = filter(content);
        bodyContent.clearBody();
        bodyContent.print(content);
        bodyContent.writeOut(getPreviousOut());
      } catch (IOException ioe) {
        throw new JspTagException(ioe.getMessage());
      }
    }
    return EVAL_PAGE;
  }
}
```