

Лекция 20

Java Server Faces

Подобно Swing и AWT, JSF представляет собой каркас разработки приложений, предоставляющий набор стандартных графических компонентов для создания интерфейсов.

Но при этом JSF ориентирована на создание Web-приложений.

Компонентная архитектура

JSF предоставляет описываемые тегами компоненты, соответствующие всем полям ввода в стандартном HTML.

Кроме этого можно создавать специальные компоненты для нужд приложения.

Все компоненты могут сохранять свое состояние – это поддерживается на уровне самой технологии JSF.

Кроме этого компоненты используются для динамической генерации HTML-страниц.

Являясь компонентной архитектурой, JSF легко расширяется и конфигурируется.

Большинство функций JSF, например, навигация или управление объектами JavaBean, могут быть реализованы встраиваемыми компонентами.

JSF и JSP

Интерфейс JSF-приложения состоит из страниц JSP, которые содержат компоненты, обеспечивающие функциональность интерфейса.

При этом нельзя сказать, что JSF неразрывно связана с JSP, т.к. теги, используемые на JSP-страницах всего лишь отрисовывают компоненты, обращаясь к ним по имени. Жизненный же цикл компонентов JSF не ограничивается JSP-страницей.

JSF не имеет непосредственного отношения к JSP: она лишь использует JSP через специальную библиотеку тегов – своего рода мост. Жизненный цикл компонент JSF сильно отличается от цикла JSP-страниц.

Жизненный цикл приложения JSF.

Фазы жизненного цикла:

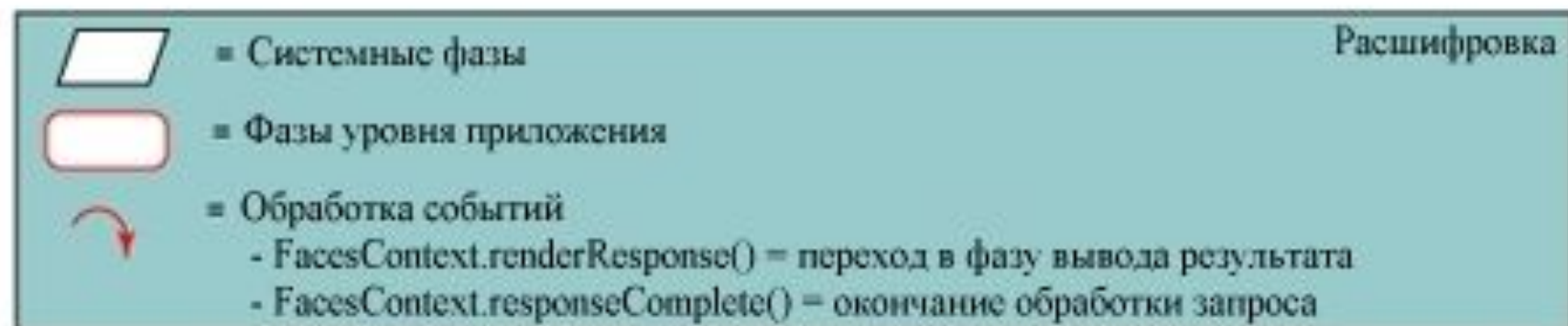
1. Восстановление представления
2. Использование параметров запроса; обработка событий
3. Проверка данных; обработка событий
4. Обновление данных модели; обработка событий
5. Вызов приложения; обработка событий
6. Вывод результата

На каждом этапе возможна обработка событий.

При этом фазы не обязательно следуют одна за другой в строго установленном порядке.

Более того, отдельные фазы можно пропускать или вовсе прерывать цикл.

Таким образом, схема жизненного цикла имеет вид:



Фаза 1: Восстановление представления

В первой фазе обработки — этапе восстановления представления — запрос поступает на вход сервлета FacesServlet.

Последний анализирует данные запроса и извлекает идентификатор представления, определяемый именем страницы JSP.

Идентификатор используется контроллером JSP для поиска компонентов, ассоциированных с данным представлением.

Если оно не существует, то контроллер создаст его, если существует - будет использовать существующее.

Представление содержит в себе все необходимые компоненты графического интерфейса.

Существуют три типа представлений, с которыми приходится иметь дело на данном этапе:

1.новое

2.изначальное

3.повторное

В случае нового представления JSF создает представление страницы Faces и связывает компоненты с обработчиками событий и валидаторами данных.

Затем оно сохраняется в объекте FacesContext.

Этот объект служит для хранения информации, необходимой для управления состоянием компонентов GUI в процессе обработки запросов.

FacesContext сохраняет состояние в свойстве `viewRoot`.

Данное свойство содержит все компоненты JSF, ассоциированные с данным идентификатором представления.

Изначальное представление означает, что данная страница вызывается впервые.

В этом случае JSF создает пустое представление, которое заполняется по мере обработки JSP-страницы.

Поэтому JSF сразу переходит к этапу отрисовки результата.

Если пользователь возвращается на ранее уже посещенную страницу, то представление, соответствующее этой странице, уже существует и просто должно быть восстановлено.

В этом случае JSF использует сохраненную информацию о состоянии представления для его воссоздания.

Фаза 2: Использование данных запроса

Главной задачей данного этапа является получение данных о состоянии каждого компонента.

Сами компоненты хранятся или создаются внутри объекта `FacesContext` вместе со своими значениями.

Значения компонентов, как правило, приходят в параметрах запроса, хотя могут извлекаться также из cookies и заголовков запроса.

Многие компоненты сохраняют значения параметра запроса в свойстве `submittedValue`.

Фаза 3: Проверка данных

Конвертация и валидация данных, как правило, выполняются в фазе проверки данных.

Компонент конвертирует и сохраняет значение свойства `submittedValue`.

Например, если данное поле связано со свойством типа `Integer`, то значение будет преобразовано к типу `Integer`.

Если конвертация проходит неудачно, создается соответствующее сообщение об ошибке, направляемое в очередь `FacesContext` для последующего вывода в фазе отрисовки результата.

Фаза 4: Обновление модели

В четвертой фазе жизненного цикла JSF происходит обновление данных модели путем изменения свойств серверных объектов `JavaBean`.

Обновляются только те свойства, которые привязаны к значениям компонентов.

Фаза 5: Вызов приложения

В этой фазе контроллер JSF вызывает приложение для обработки данных, полученных через форму.

Значения компонентов уже преобразованы к нужным типам, валидированы и сохранены в объектах модели.

Теперь их можно использовать для выполнения бизнес-логики приложения.

Фаза 6: Вывод результата

В заключительной фазе цикла происходит вывод представления вместе со всеми его компонентами и их текущими состояниями.

Теперь рассмотрим примеры JSF приложений.

Пример 1. Web калькулятор.

Необходимо создать Web калькулятор, который умеет выполнять простые арифметические операции.

Для начала главной задачей будет создание простого Web-калькулятора, интерфейс которого представляет собой страницу, предлагающую пользователю ввести два числа для последующего сложения или умножения.

Описание и отображение сервлета Faces

Перед использованием Faces-сервлета необходимо объявить его в файле web.xml

```
<servlet>
```

```
<servlet-name>Faces Servlet</servlet-name>
```

```
<servlet- class>
```

```
    javax.faces.webapp.FacesServlet
```

```
</servlet-class>
```

Данный тег означает, что сервлет будет загружен первым

```
<load-on-startup>1</load-on-startup>
```

```
</servlet>
```

Сервлет должен вызываться на каждый запрос со страниц JSP, на которых используется тег `<f:view>`. Для этого необходимо добавить отображение сервлета (servlet mapping), указывающее, что через него должны загружаться только JSP-страницы, использующие JSF.

```
<servlet-mapping>
```

```
  <servlet-name>Faces Servlet</servlet-name>
```

```
  <url-pattern>*.jsf</url-pattern>
```

```
</servlet-mapping>
```

```
<servlet-mapping>
```

```
  <servlet-name>Faces Servlet</servlet-name>
```

```
  <url-pattern>/faces/*</url-pattern>
```

```
</servlet-mapping>
```

Согласно отображению в web.xml, JSF контейнер будет вызывать сервлет Faces для обработки всех запросов, относительный URI которых начинается с /faces/ или заканчивается на *.jsf.

Благодаря этому будет корректно инициализироваться контекст Faces и корневой элемент представления перед показом страниц JSF.

Таким образом, URL для загрузки Калькулятора должен быть либо

<http://localhost:8080/calculator/calculator.jsf>

либо

<http://localhost:8080/calculator/faces/calculator.jsp>

Создание класса Calculator

Класс реализующий соответствующую бизнес логику имеет вид:

```
package aaa;  
public class Calculator{  
    private int firstNumber = 0;  
    private int result = 0;  
    private int secondNumber = 0;  
  
    public void add(){  
        result = firstNumber + secondNumber;  
    }  
  
    public void multiply() {  
        result = firstNumber * secondNumber;  
    }  
  
    public void clear() { result = 0; }
```

```
public int getFirstNumber() { return firstNumber; }
```

```
public void setFirstNumber(int firstNumber) {  
    this.firstNumber = firstNumber; }
```

```
public int getResult() { return result; }
```

```
public void setResult(int result) { this.result = result; }
```

```
public int getSecondNumber() {  
    return secondNumber;  
}
```

```
public void setSecondNumber(int secondNumber) {  
    this.secondNumber = secondNumber;  
}
```

```
}
```

Объявление объекта Calculator в файле faces-config.xml

Для объявления управляемых объектов JavaBean служит элемент `<managed-bean>`.

Файл `faces-config.xml` имеет вид:

```
<?xml version="1.0" encoding="UTF-8"?>
<faces-config xmlns="http://java.sun.com/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation=
    "http://java.sun.com/xml/ns/javaee/web-facesconfig_1_2.xsd"
    version="1.2">

<managed-bean>
  <managed-bean-name>calculator</managed-bean-name>
  <managed-bean-class>
    com.arcmind.jsfquickstart.model.Calculator
  </managed-bean-class>
  <managed-bean-scope>request</managed-bean-scope>
</managed-bean>
</faces-config>
```

Объявление управляемого объекта состоит из двух частей: имени объекта — `calculator` —, задаваемого с помощью элемента `<managed-bean-name>`, и полного имени класса (элемент `<managed-bean-class>`).

При этом класс управляемого объекта обязан содержать конструктор без параметров.

Кроме вышеперечисленных элементов существует еще один —

`<managed-bean-scope>`, который определяет, где JSF будет искать объект.

В данном случае это `request`.

Т.е. область видимости запроса.

В jsf 2 и выше можно не использовать файл **faces-config.xml** а вместо него использовать аннотации.

Т.е. в файле calculator.java написать:

```
@ManagedBean(name="calculator")
```

```
@RequestScoped
```

```
public class Calculator {
```

```
.....
```

```
}
```

Создание страницы index.jsp

Страница index.jsp необходима, чтобы гарантировать, что calculator.jsp будет загружена в контексте JSF.

Сама страница имеет вид:

```
<jsp:forward page="/faces/calculator.jsp" />
```

Создание страницы calculator.jsp

Данная страница занимает центральное место в слое представления Калькулятора.

Страница содержит элементы для ввода двух чисел.

Данная страница имеет вид:

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h"%>
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0
                                Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
  <title>Calculator Application</title>
</head>
<body>
```

```
<f:view>
  <h:form id="calcForm">
    <h4>Calculator</h4>
    <table>
      <tr>
        <td><h:outputLabel value="First Number"
                          for="firstNumber" />
        </td>
        <td><h:inputText id="firstNumber"
                        value="#{calculator.firstNumber}"
                        required="true" />
        </td>
        <td><h:message for="firstNumber" /></td>
      </tr>
```

```
<tr>
  <td><h:outputLabel value="Second Number"
                    for="secondNumber" />
  </td>
  <td><h:inputText id="secondNumber"
                  value="#{calculator.secondNumber}" required="true" />
  </td>
  <td><h:message for="secondNumber" /></td>
</tr>
</table>
```

```
<div>
  <h:commandButton action="#{calculator.add}"
                  value="Add" />
  <h:commandButton action="#{calculator.multiply}"
                  value="Multiply" />
  <h:commandButton action="#{calculator.clear}"
                  value="Clear" immediate="true"/>
</div>
</h:form>
```

```
<h:panelGroup rendered="#{calculator.result != 0}">
  <h4>Results</h4>
  <table>
    <tr>
      <td> First Number ${calculator.firstNumber} </td>
    </tr>
    <tr>
      <td> Second Number ${calculator.secondNumber} </td>
    </tr>
    <tr>
      <td> Result ${calculator.result} </td>
    </tr>
  </table>
</h:panelGroup>
</f:view>
</body>
</html>
```

Рассмотрим данную страницу подробнее.

Объявление библиотек тегов.

Написание страницы начинается с объявления библиотек тегов для JSF

```
<%@ taglib uri="http://java.sun.com/jsf/html"
                                prefix="h" %>
<%@ taglib uri="http://java.sun.com/jsf/core"
                                prefix="f" %>
```

Библиотека `html` включает в себя теги для работы с формами и другими элементами **HTML**.

Все теги, реализующие логику, валидацию данных, контроллер и т.д., включены в библиотеку **core**.

<f:view>

Если необходимо явно указать, что для работы с элементами интерфейса будет использоваться JSF, то для этого используется тег `<f:view>`, информирующий JSP-контейнер о том, что компоненты будут управляться JSF.

Без этого JSF не сможет построить дерево компонентов или получить доступ к ранее созданному дереву.

Пример использования `<f:view>`:

<f:view>

<h:form id="calcForm"> ... </h:form>

</f:view>

<h:form>

Тег `<h:form>` указывает JSF, что в этом месте необходима форма HTML.

Во время отрисовки страницы JSF находит ранее созданные дочерние компоненты формы и вызывает их методы для генерации соответствующего HTML-кода.

Тег `<h:outputLabel>` задает метку для соответствующего поля ввода.

Рассмотрим пример

```
<h:form id="calcForm">
```

```
  <h:outputLabel value="First Number"  
                                     for="firstNumber" />
```

```
  <h:inputText id="firstNumber"  
                value="#{calculator.firstNumber}" />
```

```
</h:form>
```

Соответствующий HTML код будет иметь вид:

```
<form id="calcForm" name="calcForm" method="post"
      action="/calculator/calculator.jsf">
  <label for="calcForm:firstNumber">
    First Number
  </label>
  <input id="calcForm:firstNumber"
        name=" calcForm:firstNumber" type="text" value=""/>
</form>
```

```
<h:message>
```

Рассмотрим пример:

```
<h:inputText id="username" required="#{true}"  
             value="#{userBean.user.username}"  
             errorStyle="color:red" />
```

```
<h:message for="username" />
```

Соответствующий HTML код будет иметь вид:

```
<input type="text" id="form:username"  
       name="form:username" value="" />  
<span style="color:red"> "username": Value is  
  
required.</span>
```

<h:panelGroup>

Результаты операций сложения и умножения выводятся внутри тега <f:view> с помощью элемента <h:panelGroup>.

Можно свободно использовать выражения JSP внутри <h:panelGroup>.

Кроме того, аналогично другим компонентам JSF этот тег содержит атрибут `rendered`, управляющий выводом содержимого.

Таким образом, секция отображения результата выводится только если значением выражения `calculator.result != 0 is` является `true`.

Развертывание и запуск приложения.

Предположим, что Tomcat установлен в папке

C:\tomcat

Тогда развертывание будет иметь вид:

1. Создать директорию calculator в папке
c:\tomcat\webapps\
2. В директории calculator создать папку WEB-INF
3. В WEB-INF создать директорию classes, а в ней
папку aaa
4. Скомпилировать, используя javac Calculator.java и
поместить файл Calculator.class в папку
c:\tomcat\webapps\calculator\WEB-INF\classes\aaa\

5. Поместить файлы web.xml и faces-config.xml в каталог

c:\tomcat\webapps\calculator\WEB-INF\

6. Поместить файлы index.jsp и calculator.jsp в каталог c:\tomcat\webapps\calculator\

7. Скачать пакет mojarra по адресу <https://javaserverfaces.dev.java.net/> и скопировать оттуда файлы jsf-impl.jar и jsf-api.jar в каталог c:\tomcat\lib\, а также скопировать в этот же каталог пакеты jstl.jar и standard.jar.

8. Запуск приложения:

<http://localhost:8080/calculator/calculator.jsf>

Тогда в браузере получим:

Calculator

First Number

Second Number

Results

First Number 3
Second Number 5
Result 15

Использование компонента `<h:panelGrid>`

Рассмотрим обновленную версию формы для ввода данных:

```
<h:form id="calcForm">
  <h4>Calculator</h4>
  <h:panelGrid columns="3">
    <h:outputLabel value="First Number" for="firstNumber" />
    <h:inputText id="firstNumber"
      value="#{calculator.firstNumber}" required="true" />
    <h:message for="firstNumber" />
    <h:outputLabel value="Second Number" for="secondNumber" />
    <h:inputText id="secondNumber"
      value="#{calculator.secondNumber}" required="true" />
    <h:message for="secondNumber" />
  </h:panelGrid>
```

Компонент `<h:panelGrid>` может содержать только дочерние компоненты, в отличие от `<h:form>`, `<f:view>` и `<h:panelGroup>`, внутрь которых можно также помещать обычные фрагменты HTML.

Данный элемент `<h:panelGrid>` на странице включает в себя три колонки, поэтому добавление более трех дочерних компонентов приведет к появлению новой строки.

Например, если написать
`<h:panelGrid columns="2">`

То в браузере получим

Calculator

First Number

Second Number

Оформление интерфейса с помощью CSS

CSS можно использовать вместе с `<h:panelGrid>`. Для этого нужно импортировать таблицу CSS и можно начинать использовать стили внутри `<h:panelGrid>`.

Прежде всего необходимо объявить импортирование таблицы стилей

```
<head>
```

```
<title>Calculator Application</title>
```

```
<link rel="stylesheet" type="text/css"
```

```
href="<%=request.getContextPath()%>
```

```
/css/main.css" />
```

```
</head>
```

Таблица стилей имеет вид:

```
.oddRow { background-color: white; }
```

```
.evenRow { background-color: silver; }
```

```
.formGrid { border: solid #000 3px; width: 400px; }
```

```
.resultGrid { border: solid #000 1px; width: 200px;}
```

Применение этих стилей к `<h:panelGrid>` имеет вид:

```
<h:panelGrid columns="3"
```

```
    rowClasses= "oddRow, evenRow"
```

```
    styleClass="formGrid">
```

```
...
```

```
</h:panelGrid>
```

Чтобы применить стили к строкам таблицы необходимо установить значение атрибута `rowClasses` в `oddRow`, `evenRow`.

Стиль, определяющий внешнюю рамку для таблицы, применяется с помощью атрибута `styleClass(styleClass="formGrid")`

Переопределение текстов сообщений

В JSF 1.2 были добавлены атрибуты `requiredMessage` и `conversionMessage`, таким образом теперь можно переопределять текст сообщений индивидуально для каждого компонента.

```
<h:outputLabel value="First Number" for="firstNumber" />
<h:inputText id="firstNumber"
    label="First Number"
    value="#{calculator.firstNumber}"
    required="true"
    requiredMessage="required"
    converterMessage="not a valid number" />
<h:message for="firstNumber" />

<h:outputLabel value="Second Number" for="secondNumber" />
<h:inputText id="secondNumber"
    label="Second Number"
    value="#{calculator.secondNumber}"
    required="true"
    requiredMessage="required"
    converterMessage="not a valid number" />
<h:message for="secondNumber" />
```

Тогда в браузере получим:

Calculator

First Number	<input type="text" value="sss"/>	not a valid number
Second Number	<input type="text" value="4"/>	
<input type="button" value="Add"/>	<input type="button" value="Multiply"/>	<input type="button" value="Clear"/>

Глобальное управление сообщениями

Сообщениями можно управлять глобально, для этого следует определить набор ресурсов в файле `faces-config.xml` и использовать его для хранения текстов сообщений.

```
<?xml version="1.0" encoding="UTF-8"?>
<faces-config xmlns="http://java.sun.com/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation=
    "http://java.sun.com/xml/ns/javaee/web-
      facesconfig_1_2.xsd" version="1.2">
<b><application>
  <message-bundle>messages</message-bundle>
</application>
...

```

При этом необходимо определить файл
message.properties:

javax.faces.component.UIInput.REQUIRED=required

javax.faces.converter.IntegerConverter

.INTEGER=not a valid number

При этом в теге `<h:inputText>` нет необходимости
писать `requiredMessage`, `converterMessage`.

Файл `message.properties` должен находиться в
директории

`c:\tomcat\webapps\calculator\WEB-INF\classes\`

Добавление контроллера

Добавим метод деления в класс calculator

```
public class Calculator {  
    private int firstNumber = 0;  
    private int result = 0;  
    ...  
    public void divide() {  
        this.result = this.firstNumber/this.secondNumber;  
    }  
    public void clear () { result = 0; }  
    ...  
}
```

Создадим класс-контроллер

CalculatorController, содержащий ссылку на класс Calculator.

CalculatorController также связан с тремя компонентами JSF - другими словами, он зависит от классов, являющихся частью JSF

- **resultsPanel** – компонент типа UIPanel
- **firstNumberInput** – компонент типа UIInput
- **secondNumberInput** – компонент типа UIInput

Таким образом, класс CalculatorController имеет вид:

```
package aaa;  
import javax.faces.application.FacesMessage;  
import javax.faces.component.UIInput;  
import javax.faces.component.UIPanel;  
import javax.faces.context.FacesContext;  
  
public class CalculatorController {  
    private Calculator calculator;  
    private UIPanel resultsPanel;  
    private UIInput firstNumberInput;  
    private UIInput secondNumberInput;
```

```
public String add() {
    FacesContext facesContext = FacesContext.getCurrentInstance();
    try {
        calculator.add();
        resultsPanel.setRendered(true);
        facesContext.addMessage(null,
            new FacesMessage( FacesMessage.SEVERITY_INFO,
                "Added successfully", null));
    }catch (Exception ex) {
        resultsPanel.setRendered(false);
        facesContext.addMessage(null,
            new FacesMessage(FacesMessage.SEVERITY_ERROR,
                ex.getMessage(), null));
    }
    return null;
}
```

```
public String multiply() {
    FacesContext facesContext = FacesContext.getCurrentInstance();
    try {
        calculator.multiply();
        resultsPanel.setRendered(true);
        facesContext.addMessage(null,
            new FacesMessage( FacesMessage.SEVERITY_INFO,
                "Multiplied successfully", null));
    }catch (Exception ex) {
        resultsPanel.setRendered(false);
        facesContext.addMessage(null,
            new FacesMessage(FacesMessage.SEVERITY_ERROR,
                ex.getMessage(), null));
    }
    return null;
}
```

```
public String divide() {
    FacesContext facesContext = FacesContext.getCurrentInstance();
    try {
        calculator.divide();
        resultsPanel.setRendered(true);
        facesContext.addMessage(null,
            new FacesMessage(FacesMessage.SEVERITY_INFO,
                "Divided successfully", null));
    } catch (Exception ex) {
        resultsPanel.setRendered(false);
        if (ex instanceof ArithmeticException) {
            // Т.к. поле secondNumberInput связано с полем secondNumber, то поле
            // secondNumber будет равно 1.
            secondNumberInput.setValue(Integer.valueOf(1));
        }
        facesContext.addMessage(null,
            new FacesMessage(FacesMessage.SEVERITY_ERROR,
                ex.getMessage(), null));
    }
    return null;
}
```



```
public String clear() {
    FacesContext facesContext = FacesContext.getCurrentInstance();
    try {
        calculator.clear();
        resultsPanel.setRendered(false);
        facesContext.addMessage(null,
            new FacesMessage(FacesMessage.SEVERITY_INFO,
                "Results cleared", null));
    } catch (Exception ex) {
        resultsPanel.setRendered(false);
        facesContext.addMessage(null,
            new FacesMessage(FacesMessage.SEVERITY_ERROR,
                ex.getMessage(), null));
    }
    return null;
}
```

```
public String getFirstNumberStyleClass() {  
    if (firstNumberInput.isValid()) {  
        return "labelClass"; }  
    else {  
        return "errorClass";  
    }  
}
```

resultsPanel представляет собой секцию для вывода результатов арифметических операций.

resultsPanel.setRendered(true) делает панель результата видимой для пользователя.

JSF предоставляет средства для показа сообщений пользователям о статусе той или иной операции.

Для добавления сообщений используется класс `FacesContext`, благодаря которому они впоследствии могут быть выведены с помощью тега `<h:messages>`.

JSF хранит объект класса `FacesContext` в переменной `ThreadLocal`.

К ней можно получить доступ, вызвав статический метод

```
static FacesContext getCurrentInstance()  
getCurrentInstance(),
```

Например, в методе add() в FacesContext добавляются сообщения, которые должны быть доступны до конца обработки запроса.

```
public String add() {
    FacesContext facesContext = FacesContext.getCurrentInstance();
    try {
        calculator.add();
        facesContext.addMessage(null,
            new FacesMessage( FacesMessage.SEVERITY_INFO,
                "Added successfully", null));
        ...
    }catch (Exception ex) {
        facesContext.addMessage(null,
            new FacesMessage(FacesMessage.SEVERITY_ERROR,
                ex.getMessage(), null)); //протоколирование
                                        ИСКЛЮЧЕНИЯ.
        ....
    }
    return null;
}
```

В данном примере сообщения представляют собой объекты класса `FacesMessage`, добавляемые в объект `facesContext`.

При этом уровень важности сообщения зависит от возникновения исключений при выполнении операции суммирования.

Если оно прошло успешно, то уровень будет `INFO`, в противном случае – `ERROR`.

Сообщения выводятся на странице с помощью тега `<h:messages>`:

```
<h:messages infoClass="infoClass"  
            errorClass="errorClass"  
            layout="table"  
            globalOnly="true"/>
```

errorClass – атрибут устанавливает CSS стиль, который будет применяться для сообщений уровня “error”.

infoClass - атрибут устанавливает CSS стиль, который будет применяться для сообщений уровня “info”.

Для вывода только глобальных сообщений, т. е. не относящихся к конкретному компоненту, можно установить значение атрибута `globalOnly` в `true`.

Страница calculator.jsp будет иметь следующий вид
(на этой странице осуществляется привязка
firstNumber к firstNumberInput, а также secondNumber
к secondNumberInput):

.....

```
<f:view>
```

```
<h:form id="calcForm">
```

```
<h4>Calculator 3</h4>
```

```
<h:messages infoClass="infoClass"  
            errorClass="errorClass"  
            layout="table"  
            globalOnly="true"/>
```

```
<h:panelGrid columns="3"  
            rowClasses="oddRow, evenRow"  
            styleClass="formGrid">
```

```
<h:outputLabel  
  value="First Number"  
  for="firstNumber"  
  styleClass=#{calculatorController.firstNumberStyleClass}"/>
```

```
<h:inputText id="firstNumber"  
  label="First Number"  
  value="#{calculatorController.calculator.firstNumber}"  
  required="true"  
  binding="#{calculatorController.firstNumberInput}" />
```

```
<h:message for="firstNumber" errorClass="errorClass"/>
```



```
<h:outputLabel id="snl"  
  value="Second Number"  
  for="secondNumber"  
  styleClass=  
    "#{calculatorController.secondNumberStyleClass}"/>
```

```
<h:inputText id="secondNumber"  
  label="Second Number"  
  value="#{calculatorController.calculator.secondNumber}"  
  required="true"  
  binding="#{calculatorController.secondNumberInput}"/>
```

```
<h:message for="secondNumber" errorClass="errorClass"/>
```

```
</h:panelGrid>
```

```
<div>
  <h:commandButton action="#{calculatorController.add}"
                  value="Add" />
  <h:commandButton
    action="#{calculatorController.multiply}"
    value="Multiply" />
  <h:commandButton
    action="#{calculatorController.divide}"
    value="Divide" />
  <h:commandButton
    action="#{calculatorController.clear}"
    value="Clear"
    immediate="true"/>
</div>
</h:form>
```

```
<h:panelGroup binding="#{calculatorController.resultsPanel}"
              rendered="false">
<h4>Results</h4>
<h:panelGrid columns="1"
             rowClasses="oddRow, evenRow"
             styleClass="resultGrid">
<h:outputText value= "First Number
                #{calculatorController.calculator.firstNumber}"/>
<h:outputText value="Second Number
                #{calculatorController.calculator.secondNumber}"/>
<h:outputText value="Result
                #{calculatorController.calculator.result}"/>
</h:panelGrid>
</h:panelGroup>
</f:view>
```

Описание контроллера в файле

faces-config.xml имеет вид:

```
<application>
```

```
    <message-bundle>messages</message-bundle>
```

```
</application>
```

```
<managed-bean>
```

```
    <managed-bean-name>
```

```
        calculatorController
```

```
</managed-bean-name>
```

```
<managed-bean-class>
```

```
        aaa.CalculatorController
```

```
</managed-bean-class>
```

```
<managed-bean-scope>request</managed-bean-scope>
  <managed-property>
    <property-name>calculator</property-name>
    <value>#{calculator}</value>
  </managed-property>
</managed-bean>
<managed-bean>
  <managed-bean-name>calculator</managed-bean-name>
  <managed-bean-class>
    aaa.Calculator
  </managed-bean-class>
  <managed-bean-scope>none</managed-bean-scope>
</managed-bean>
```

Поскольку в данном случае объект calculator будет доступен только через внешний объект calculatorController, его необходимо поместить в область видимости none. Это означает, что объект не будет помещен ни в какую специальную область видимости после создания.

```
<managed-bean>  
  <managed-bean-name>  
    calculator  
  </managed-bean-name>  
  <managed-bean-class>  
    aaa.Calculator  
  </managed-bean-class>  
  <managed-bean-scope>  
    none  
  </managed-bean-scope>  
</managed-bean>
```

Объект `calculatorController` будет помещен в область видимости `request`.

При этом ссылка на `calculator` будет передана в `calculatorController`.

Это делается с помощью выражения `#{calculator}` в теге `<managed-property>`.

Таким образом, JSF создаст экземпляр класса `Calculator` и передаст его в метод `setCalculator` класса `CalculatorController`

```
<managed-bean>
  <managed-bean-name>
    calculatorController
  </managed-bean-name>
  <managed-bean-class>
    aaa.CalculatorController
  </managed-bean-class>
  <managed-bean-scope>
    request
  </managed-bean-scope>
  <managed-property>
    <property-name>calculator</property-name>
    <value>#{calculator}</value>
  </managed-property>
</managed-bean>
```


Объект `calculator` используется классом `CalculatorController`, но при этом сам остается “чистым”, т.е. никоим образом не привязанным к JSF.

Классы модели всегда желательно держать независимыми от библиотек, подобных JSF, изолируя весь JSF-зависимый код внутри контроллера, в данном случае – классе `CalculatorController`.

Это значительно облегчает тестирование и повторное использование модели.

В случае jsf 2 можно использовать аннотации

```
@ManagedBean(name="calculatorController")  
@RequestScoped  
public class CalculatorController{  
    @ManagedProperty(name="calculator",  
                        value="#{calculator}")  
    private Calculator calculator;  
    .....  
}
```

```
@ManagedBean(name="calculator")
```

```
@NoneScoped
```

```
public class Calculator {
```

```
.....
```

```
}
```