
Getting started with JavaServer Faces 1.2, Part 1: Building basic applications

Skill Level: Introductory

[Richard Hightower \(righthouse@arc-mind.com\)](mailto:righthouse@arc-mind.com)
CTO
ArcMind

18 Dec 2007

Java™ Server Faces (JSF) technology, a server-side framework that offers a component-based approach to Web user-interface development, has come a long way. JSF 1.2 (incorporated into Java Enterprise Edition 5) has fixed some JSF pain points and added some nice features. This tutorial series covers how to get started with JSF 1.2. It's heavy on examples and light on theory — just what you need to get started quickly.

Section 1. Before you start

About this series

This tutorial series is about getting started with JavaServer Faces (JSF) technology, a server-side user-interface component framework for Java-based Web applications. The series is for developers who are new to JSF and want to come up to speed quickly — not just with JSF, but with using JSF components to reduce effort. The series covers just the essentials, with lots of examples.

JSF is a more-traditional GUI development environment like AWT, SWT, and Swing. One of its major benefits is that it makes Web development easier by putting the hard work on the framework developers, not the application developers. Granted, JSF itself is more complex than many other Web frameworks, but the complexity is hidden from the application developer. It is much easier to develop Web applications in JSF than in most other frameworks: it requires less code, less complexity, and

less configuration.

If you are doing Java server-side Web development, JSF is the easiest framework to learn. It is geared for creating Web applications (not Web sites per se). It allows you to focus on your Java code without handling request objects, session objects, request parameters, or dealing with complicated XML files. With JSF, you can get more things done more quickly than with other Java Web frameworks.

About this tutorial

This tutorial takes a basic approach to JSF development. You won't use fancy tools or IDE support in this tutorial (although tool support is a main benefit of JSF). You'll do bare-knuckled programming! I cover the essentials with just enough theory to keep the discussion going and keep you productively learning to use JSF to build Web applications. You might be surprised to learn that JSF is easier to program than other Java Web frameworks, even without fancy IDE tools.

Objectives

In this tutorial, you get an overview JSF's features and learn how to write a basic JSF application. You build a simple calculator application and, in subsequent iterations, improve its look and feel, modify its structure to add dependency injection, and implement JSF's navigation mechanism. In Part 2, you'll build custom converters, validators, and phase-listeners.

Who should take this tutorial?

If you are new to JSF, this tutorial is for you. Even if you have used JSF but have not tried out the JSF 1.2 features or have only used GUI tools to build JSF applications, you will likely learn a lot from both tutorials in this series.

Prerequisites

This tutorial is written for Java developers whose experience is at a beginning to intermediate level. You should have a general familiarity with using the Java language, with some GUI development experience.

System requirements

To run the examples in this tutorial, you need a Java development environment (JDK) and Apache Maven. It helps to have a Java IDE. Maven project files and Eclipse Java EE and Web Tools Project (WTP) project files are provided. See [Download](#) to obtain the example code.

Section 2. JSF for beginners

Like Swing and AWT, JSF is a development framework that provides a set of standard, reusable GUI components. JSF is used for building Web application interfaces. JSF provides the following development advantages:

- Clean separation of behavior and presentation
- Component-level control over statefulness
- Events easily tied to server-side code
- Leverages familiar UI-component and Web-tier concepts
- Offers multiple, standardized vendor implementations
- Excellent IDE support

A typical JSF application consists of the following parts:

- JavaBeans for managing application state and behavior
- Stateful GUI components
- Event-driven development (via listeners as in traditional GUI development)
- Pages that represent Model-View-Controller (MVC)-style views; pages reference *view roots* via the JSF component tree

You'll likely need to overcome some conceptual hurdles to use JSF, but doing so is well worth the effort. JSF's component state management; easy-to-use user-input validation; granular, component-based event handling; and easily extensible architecture will greatly simplify your Web development efforts. This section explains the most important of these features in detail.

A component-based architecture

Think stateful components

The biggest hurdle you might have is forgetting that JSF is a stateful component model. If you used Struts before, repeat after me: "JSF is not Struts. JSF is not Struts." I find people who have a Swing, AWT, Visual Basic, or Delphi GUI background learn JSF more quickly than people who have been using Struts for years and have never done GUI component development before. This tutorial is designed to help you think in terms of stateful components.

JSF provides component tags for every input field available in standard HTML. You can also write your own custom components for application-specific purposes or for combining multiple HTML components to form a composite — for example, a Data Picker component that consists of three drop-down menus. JSF components are stateful. Their statefulness is provided through the JSF framework. JSF uses components to produce HTML responses. Many third-party JSF GUI components are also available.

JSF includes:

- An event-publishing model
- A lightweight inversion-of-control (IoC) container
- Components for just about every other common GUI feature, including (but not limited to):
 - Pluggable rendering
 - Server-side validation
 - Data conversion
 - Page-navigation management

Being a component-based architecture, JSF is extremely configurable and extensible. Most JSF functions — such as navigation and managed-bean lookup — can be replaced with pluggable components. This degree of pluggability gives you considerable flexibility in building your Web application GUIs and allows you to incorporate other component-based technologies easily into your JSF development efforts. For example, you could replace JSF's built-in IoC framework with the more full-featured IoC/aspect-oriented programming (AOP) Spring framework for managed-bean lookups. I'll cover a lot of the advanced features in Part 2.

JSF and JSP technology

The user interface of a JSF application consists of JavaServer Pages (JSP) pages. Each JSP page contains JSF components that represent the GUI functionality. You use JSF custom tag libraries inside JSP pages to render the UI components, to register event handlers, to associate components with validators, to associate components with data converters, and more.

JSF has no knowledge of JSP

JSF has nothing to do with JSP per se. JSF works with JSP through a JSP tag library bridge. However, the life cycle of JSF is very different from the life cycle of JSP. Facelets fits JSF much better than JSP because Facelets was designed with JSF in mind, whereas integrating JSF and JSP has always been like forcing a square peg into a round hole. You should consider Facelets; Facelets features will be part of JSF 2.0. See [Resources](#) for more information on Facelets.

That said, the truth is that JSF is not bound to JSP technology inherently. In fact, the JSF tags used by JSP pages merely reference the components so they can be displayed. The components have a different life cycle from the JSP page.

You'll realize this the first time you modify a JSP page to change the attributes of a JSF component and reload the page ... and nothing happens. This is because the tag looks up the component in its current state. If the component already exists, the custom tag doesn't modify its state. The component model allows your controller code to change a component's state (for example, disable a text field), and when that view is displayed, the current state of your component tree is displayed.

A typical JSF application needs no Java code and very little Universal Expression Language (JSTL EL) code in the UI. As I noted previously, there are lots of IDE tools for building and assembling applications in JSF, and there is a third-party market for JSF GUI components. It is also possible to code JSF without the use of WYSIWYG tools (as you'll do in this tutorial), although JSF was designed with WYSIWYG IDE tools in mind.

We don't need no stinking WYSIWYG IDE support

Although JSF was designed with WYSIWYG IDE support in mind, you don't have to use WYSIWYG IDE support to get JSF's benefits. In fact, JSF is still much easier to use than most Java Web frameworks even if you code it by hand. If you program in Swing and you use a WYSIWYG IDE, then you will likely use such a tool with JSF. If you prefer to code Swing by hand, you will prefer to code JSF by hand. Bare-knuckled programming!

Improvements in JSP 2.1 and JSF 1.2

JSP 2.1 added many new features to support JSF, including the universal Expression Language (EL) API that JSF 1.2 also adds. With standard JSTL tags, you can now iterate through a list and render JSF components, something impossible with JSF 1.1 and JSP 2.0. See [Resources](#) for more details on changes that were made in JSP 2.1. (Even with the improvements, Facelets is a much better fit for JSF, and many ideas from Facelets are going into JSF 2.0.)

JSF and MVC

JSF is the result of lessons learned over several years of evolving Web-development techniques on the Java platform. This trend started with JSP technology, which despite its advantages made it too easy to mix Java code in with HTML (and HTML-like) pages. The next step up was the Model 1 architecture, which had developers pushing most back-end code into JavaBeans components and then importing the JavaBeans components into Web pages with the `<jsp:useBean>` tag. This worked well for simple Web applications, but many Java developers disliked JSP technology's incorporation of C++ features such as static includes. So the Model 2 architecture was introduced.

Essentially, the Model 2 architecture is a watered-down version of MVC for Web applications. In the Model 2 architecture, the controller is represented by servlets (or Actions), and display is delegated to JSP pages. Apache Struts is a simplified Model 2 implementation wherein Actions take the place of servlets. In Struts, the application's controller logic is separated from its data (represented by ActionForms). The main complaint against Struts is that it can feel more procedural than object-oriented ("COBOL for the Web"). WebWork and Spring MVC are two other Model 2 architectures that improve on Struts by being less procedural, but neither is as widely accepted as Struts. And neither offers a stateful component model as JSF does. (Struts 2 is built on top of WebWork, and the original Struts code base has been abandoned. Even Struts doesn't want Struts.)

The real issue with most Model 2 frameworks is that the event model is too simplistic (essentially a highly scaled-down MVC), and it has no stateful GUI components, which leaves too much of the work to the developer. A richer component and event model makes it easier to create the kinds of interactions most users expect. Like JSP technology, most Model 2 frameworks also make it too easy to mix HTML layout and formatting with GUI custom tags, which act loosely like components except they are not stateful. And some Model 2 architectures (like classic Struts) make the mistake of separating behavior and state, which leaves many Java

developers feeling like they're programming COBOL.

A richer MVC environment

JSF is not Struts; empty your glass so you can fill it

JSF is not a Model 2 framework. It's much more than that. Many people believe that because the original author of Struts was the JSF specification lead, that Struts skills can be used on a JSF project. Don't try to program JSF like Struts. You need to give up on some of your Struts skills and learn JSF skills.

JSF provides a component model and a richer MVC environment — much richer than Model 2 frameworks. JSF is much closer to a true MVC programming environment than the Model 2 architectures, although it still builds on top of a stateless protocol. JSF also facilitates building more fine-grained, event-driven GUIs than the Model 2 frameworks. Whereas JSF gives you a host of event options — menu item selected, button clicked, tree node expanded, and so on — most Model 2s rely on the more simple "request received."

JSF's fine-tuned event model allows your applications to be less tied to HTTP details and simplifies your development effort. JSF also improves somewhat on the traditional Model 2 architecture by making it easier to move presentation and business logic out of your controller and move business logic out of your JSP pages. In fact, simple controller classes aren't tied to JSF at all, making them easier to test. Unlike a true MVC architecture, the JSF model tier is unlikely to issue many events that must be resolved in more than one viewport (although Crank tries to do this, with support from JBoss ajax4JSF; see [Resources](#)). Again, this would be unnecessary because you're dealing with a stateless protocol. The system event for changing or updating a view is almost always (dare I say always?) a request from the user.

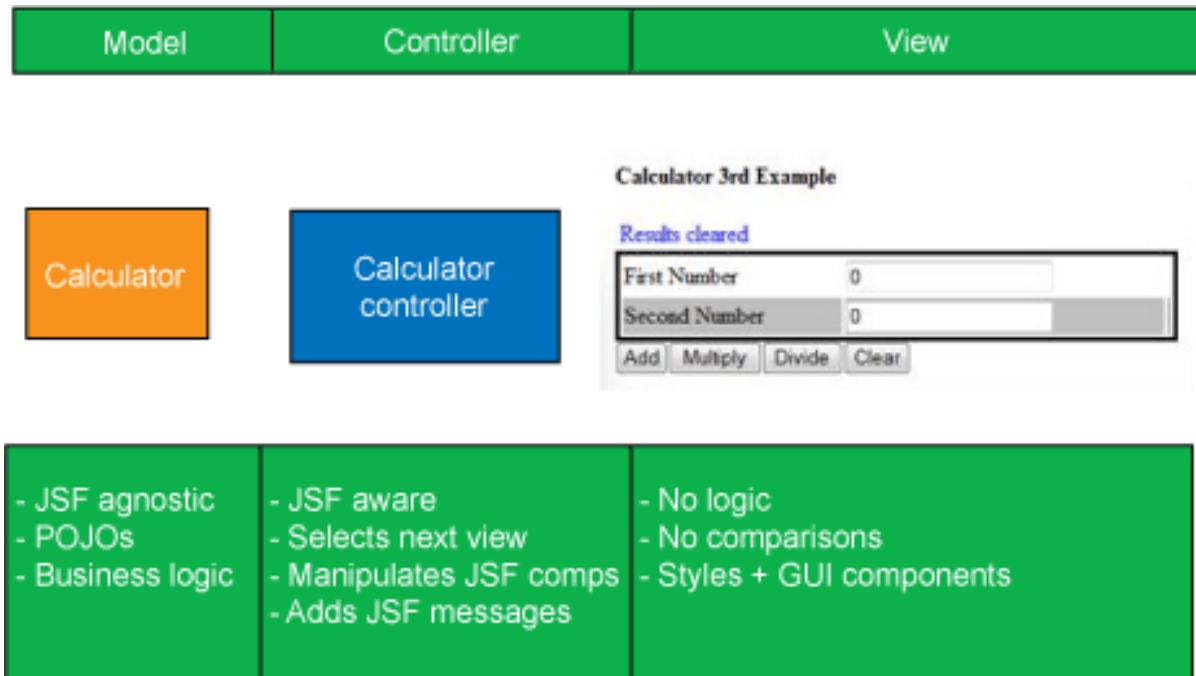
Details of JSF's MVC implementation

In JSF's MVC implementation, mapping managed beans mediate between view and model. Because of this, it's important to limit the business logic and persistence logic in the managed beans that are tied to JSF. One common alternative is to delegate business logic to the application model. In this case, the managed beans also map model objects where the view can display them (as properties of the managed bean). I tend to separate my managed beans into two categories: managed beans that are tied to JSF (controllers) and managed beans that are not tied to JSF (model objects).

Unlike JSP technology, JSF's view implementation is a stateful component model. The JSF view is composed of two pieces: the *view root* and JSP pages. The view root is a collection of UI components that maintain the UI's state. Like Swing and AWT, JSF components use the Composite design pattern to manage a tree of components (simply put: a container contains components; a container is a component). The JSP page binds UI components to JSP pages and allows you to bind field components to properties of backing beans (or properties of properties, more likely) and buttons to event handlers and action methods.

Figure 1 illustrates the structure of an example application (the one you're about to get to know in detail!) from a MVC point of view:

Figure 1. Sample application from an MVC perspective



If this first section on JSF has left you shaking your head a little, don't worry: you're over the worst hump. Getting into the conceptual framework of JSF is more than half the battle with implementing this technology — and you'll soon see that it's well worth the trouble. But that's enough theory: let's get into some bare-knuckled coding!

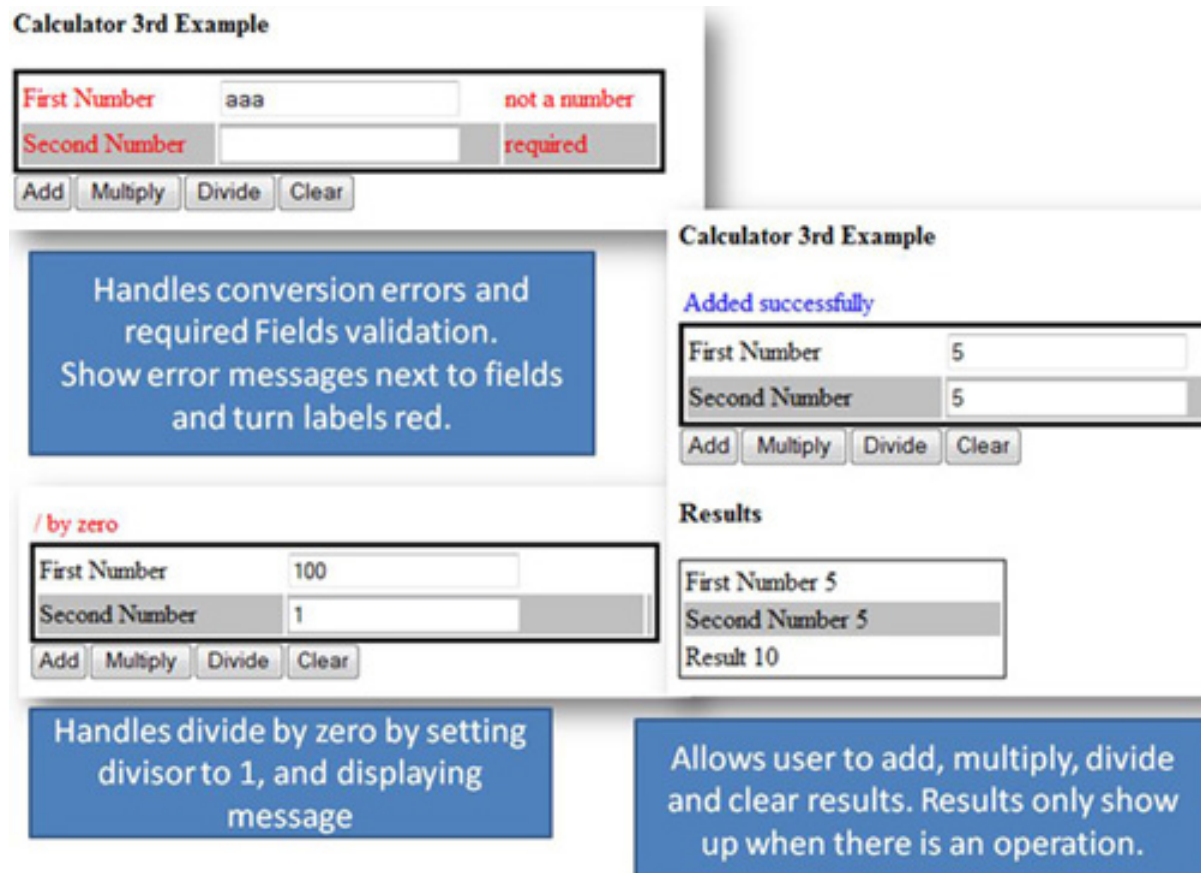
Section 3. A JSF example: Step by step

This section focuses on the step-by-step process of creating an application in JSF. The example application is a simple Calculator application that demonstrates the following aspects of using JSF technology:

- How to lay out a JSF application for deployment
- How to configure a web.xml file for JSF
- How to configure a faces-config.xml file for an application
- Writing managed beans (also known as model objects and controllers)
- Constructing the view using JSP technology
- Using custom tag libraries to construct the component tree in the view root
- Default validation of form fields

In subsequent sections, you'll improve the application over several iterations and make it more JSF savvy. Figure 2 shows an annotated view of what the final iteration of the example Calculator application will look like. See [Download](#) to get the application source.

Figure 2. Final Calculator application



The Calculator application

Build by Maven 2 and Eclipse WTP

The default build environment for the example Calculator application is Apache Maven 2. In these examples, I've used the default layout for a Maven Web application. See [Resources](#) for instructions and JARs for running this tutorial's source code with Eclipse JEE, Tomcat, and Maven 2.

The goal of the initial Calculator application is to present a page that allows the user to enter two numbers and then add or multiply them.

The page has:

- A form
- Two text fields
- Two labels

- Two error-message locations
- Two Submit buttons
- A results panel

The text fields are for entering the numbers. The labels are for labeling the text fields. The error-message locations are to display validation or data-conversion error messages for the text fields. There are two JSP pages: `calculator.jsp` and `index.jsp`, which just redirects to `calculator.jsp`. A managed bean called `Calculator` serves as the model for `calculator.jsp`. This simple example does not have a controller layer other than what JSF provides.

Creating the application: Overview

To build the initial Calculator application in JSF you need to:

- Declare the Faces Servlet and add Faces Servlet mapping in the Web application deployment descriptor (`web.xml`) file
- Specify the `faces-config.xml` file in the `web.xml` file
- Create the `Calculator` class
- Declare the `Calculator` bean in the `faces-config.xml` file
- Create the `index.jsp` page
- Create the `calculator.jsp` page

The application uses the following directory layout:

```
+---src
  +---main
    +---java
    +---webapp
      +---pages
      +---WEB-INF
        +---lib
```

The Java code is under `src/main/java/`. The `web.xml` file is under the `src/main/webapp/WEB-INF` directory. The JSF configuration file is also under `src/main/webapp/WEB-INF`. The example application was created with the Eclipse IDE for Java EE Developers (Eclipse JEE), which includes a JSF project-creation wizard that creates the `web.xml` file and `faces-config.xml` file with the right entries. I assume you are going to use an application server that supports Java EE 5, that is,

that has the JSF and JSTL JAR files. See [Resources](#) for instructions on setting up Tomcat to run JSF, setting up Eclipse JEE to run Tomcat 6, and running the examples with Maven 2.

Declare the Faces Servlet and servlet mapping

To use the Faces Servlet, you first need to install it in your web.xml file, as shown in Listing 1:

Listing 1. Faces Servlet declaration in web.xml

```
<servlet>
  <servlet-name>Faces Servlet</servlet-name>
  <servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
  <load-on-startup>1</load-on-startup>
</servlet>
```

This is similar to most web.xml descriptors, except that you're giving control to the JSF servlet to handle requests instead of specifying your own servlet. All requests to JSP files that use an `<f:view>` tag (which the sample application does) must go through this servlet. Therefore, you need to add a mapping and load only the JSF-enabled JSP technology through that mapping, as shown in Listing 2:

Listing 2. Faces Servlet path mapping in web.xml

```
<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>
```

Listing 2 tells the Faces Servlet container to send all requests that start with `/faces/` or end with `*.jsf` to the Faces Servlet for processing. This allows JSF to initialize the Faces context and the view root before displaying the JSF page. The view root contains the JSF component tree. The Faces context is the way to interact with JSF.

This means that to load the Calculator application you use `http://localhost:8080/calculator/pages/calculator.jsf` or `http://localhost:8080/calculator/faces/pages/calculator.jsp` — not `http://localhost:8080/calculator/pages/calculator.jsp`. If you load the JSP page outside of the JSF context, JSF will not get a chance to initialize the Faces context

or the view root.

Specify the faces-config.xml file

If you name your Faces configuration file `faces-config.xml` and place it in your Web application's `WEB-INF` directory, then the Faces Servlet picks it up and uses it automatically (because it's the default). Alternatively, you can load one or more application-configuration files through an initialization parameter — `javax.faces.application.CONFIG_FILES` — in your `web.xml` file with a comma-separated list of files as the body. You will likely use the second approach for all but the simplest JSF applications. Because the example application is simple, take the default `faces-config.xml` file location of `/src/main/webapp/WEB-INF`.

Create the Calculator class

Now you'll create a POJO (plain old Java object) called `Calculator` that's not tied to JSF at all, and then bind it to JSF with method bindings and property bindings. This simple class has two properties: `firstNumber` and `secondNumber`.

My goal is to demonstrate how to get started with JSF, so I've kept the model object very simple. The model of this application is contained within one model object, shown in Listing 3. Later you'll split it into two classes: `controller` and `model`.

Listing 3. Calculator POJO

```
package com.arcmind.jsfquickstart.model;

/**
 * Calculator. Simple POJO.
 *
 * @author Rick Hightower
 */
public class Calculator {

    /** First number used in operation. */
    private int firstNumber = 0;

    /** Result of operation on first number and second number. */
    private int result = 0;

    /** Second number used in operation. */
    private int secondNumber = 0;

    /** Add the two numbers. */
    public void add() {
        result = firstNumber + secondNumber;
    }

    /** Multiply the two numbers. */
```

```
public void multiply() {
    result = firstNumber * secondNumber;
}

/** Clear the results. */
public void clear() {
    result = 0;
}

/* ----- properties ----- */

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;
}
}
```

Listing 3 is straightforward and does not require explanation; just read the code. Remember, though, that the `Calculator` POJO has nothing to do with JSF.

Declare the Calculator bean in the `faces-config.xml` file

Listing 4 shows the entire `faces-config.xml` file. As you can see, a good part of it just associates the file with the Java EE JSF XML schema. In `faces-config.xml`, the `<managed-bean>` element is used to declare a bean that JSF can bind to:

Listing 4. The `faces-config.xml` file containing managed-bean declaration

```
<?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
    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>
```

The bean declaration Listing 4 specifies is the name of the bean — `calculator` — with `<managed-bean-name>`. It specifies the fully qualified class name with `<managed-bean-class>`. The class must have a no-argument constructor.

The `<managed-bean>` element's `<managed-bean-scope>` specifies where JSF can find the bean: `request` scope. If you bind this bean name to a view (you'll do this later in the tutorial), and JSF can't find it, it will create it. This is done through JSF and the universal EL API. `request` scope is available for one request only. It is a good place to put beans that don't need to maintain state between page views.

Create the index.jsp page

The purpose of the `index.jsp` page in the Calculator application is to ensure that the `calculator.jsp` page loads in the JSF context so that the page can find the corresponding view root. Listing 5 shows the `index.jsp` page:

Listing 5. The index page, redirecting to calculator.jsp

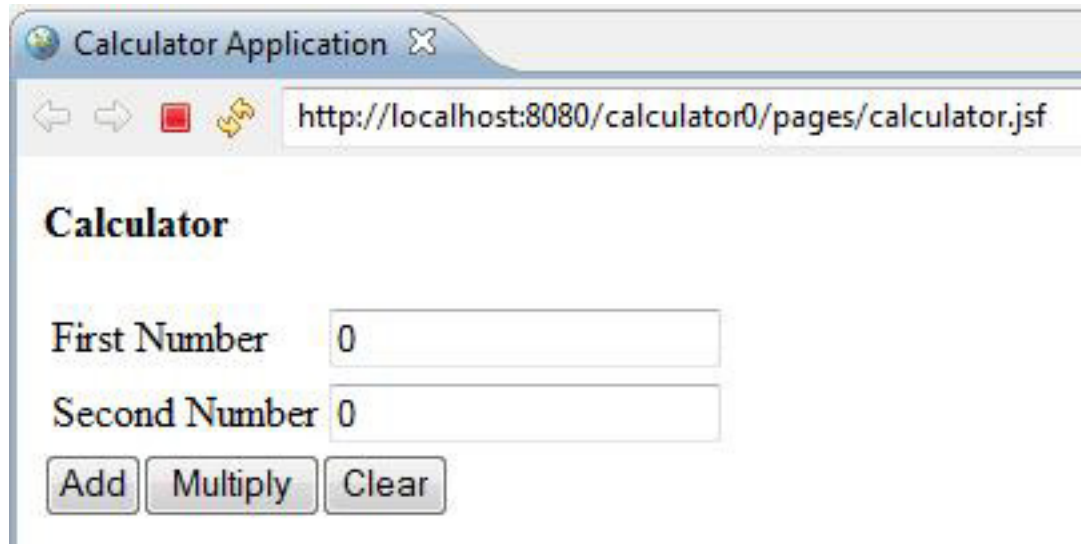
```
<jsp:forward page="/faces/calculator.jsp" />
```

All this page does is redirect the user to `calculator.jsp` under the `faces` Web context. This puts the `calculator.jsp` page under the JSF context path, where it can find its view root.

Create the calculator.jsp page

The `calculator.jsp` page is the heart of the Calculator application's view. This page takes two numbers entered by the user, as shown in Figure 3:

Figure 3. First Calculator application running in Eclipse JEE/WTP



The entire code for this page is in Listing 6:

Listing 6. /src/main/webapp/calculator.jsp

```
<?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>
  </f:view>
</body>
</html>
```



```
</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>
```

Note that most of this file is plain old HTML (XHTML to be exact). You can use HTML inside of `<f:view>`, `<h:form>`, and `<h:panelGroup>` tags. It is a common myth that you can't mix HTML with JSF tags. You can in many cases. You cannot use HTML inside of the `<h:commandButton>`, which takes only other components as children in its body.

Because this page is complex, I'll show you how to build it step by step.

Declare taglibs

You start by declaring the taglibs for JSF, as shown in Listing 7:

Listing 7: Importing the taglibs in calculator.jsp

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

Listing 7 tells the JSP engine that you want to use the two JSF taglibs `html` and `core`. The `html` taglib contains all the tags for dealing with forms and other HTML-specific items. The `core` taglib contains all the logic, validation, controller, and other tags specific to JSF.

The `<f:view>` tag

Once you've laid out the page in normal HTML, you want to tell the JSF system that you're going to be using JSF to manage your components. You do this by using the

`<f:view>` tag, which informs the container that you're using JSF to manage the components inside it.

Without `<f:view>`, JSF cannot build the component tree and later cannot look up the component tree that was already created. Use the `<f:view>` tag as shown in Listing 8:

Listing 8. calculator.jsp's `<f:view>` tag

```
<f:view>
  <h:form id="calcForm">
    ...
  </h:form>
</f:view>
```

The first line in Listing 8 is the declaration of `<f:view>`, telling the container that it is managed by JSF.

Inside `<f:view>`: The `<h:form>` tag

The second line in Listing 8 is the `<h:form>` tag telling JSF that you want an HTML form here. During the render phase, the components contained within the form component are looked up and asked to render themselves, whereupon they generate standard HTML to the output. You can lay out the form components any way you like. Listing 9 is the layout for the Calculator application's input fields:

Listing 9. Inside calculator.jsp's `<h:form>` tag: The input fields

```
<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>
```

Again note the heavy use of HTML. You could lay out your application with `spans`, `divs`, `tables`, or whatever else you want. JSF is designer friendly. There are even tools that allow you to use JSF in Dreamweaver (see [Resources](#)). JSF is even more

designer friendly when you use it with Facelets (which you can use with JSF 1.1 and higher and will be part of JSF 2.0).

Notice in Listing 9 that both `inputTexts` use a JSF EL (JavaServer Faces Expression Language) value expression for the `value` attribute (for example, `value="#{calculator.firstNumber}"`). At first blush this looks a lot like JSTL EL. However, the universal EL code actually associates the fields with the corresponding values of the backing bean's properties. This association is reflexive; that is, if `firstNumber` were 100, then 100 would show up when the form was displayed. Likewise, if the user submitted a valid value such as 200, then 200 would be the new value of the `firstNumber` property (assuming a conversion and validation pass, which you'll learn more about later).

In addition to the fields, the `calcForm` is associated with two actions using three `commandButtons`, as shown in Listing 10:

Listing 10. Inside `calculator.jsp`'s `<h:form>` tag: The buttons

```
<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>
```

About stylesheets

The look and feel of all JSF components is declared through stylesheet classes. Each component has a style to associate inline styles, and a `styleClass` to associate the component with a `stylesheet` class. The `<panelGrid>`, which you'll use in the tutorial's next section, also has style attributes to associate styles with rows and columns. In that section, you'll use Cascading Style Sheets (CSS) with JSF.

The code in Listing 10 binds the three buttons to the `calculator` class's `add()`, `multiply()`, and `clear()` methods, so that when a button is clicked, its respective method is called (assuming that validation and conversion are performed successfully).

By default, JSF validates the form before executing any of the action methods (such as `add()` or `multiply()`). However, if you use `immediate="true"`, as Listing 10 does for the Clear button, then JSF skips the validation phase and executes the method directly (more or less — more details to follow) without validating the form.

Viewing the results: The `<h:panelGroup>` tag

Last, within the `<f:view>` tag, you show the results of the add and multiply operations with a `<h:panelGroup>`, shown in Listing 11:

Listing 11. Showing the results

```
<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>
```

Again you mostly use HTML. Notice that you can mix JSP style expressions in the body of your `<h:panelGroup>`. The `<h:panelGroup>` has a `rendered` expression, as do all JSF components. Thus the results section appears only if the expression `calculator.result != 0` is true. For this reason, when users first load the calculator page, this section does not show. When they enter values, the results section shows (as long as the results are not zero). (The problem with this expression is that it puts logic in the view. And, what if you want the user to be able to enter `0 + 0` (or `7 * 0`) and show the result? You'll fix this later in the tutorial.)

Run the application

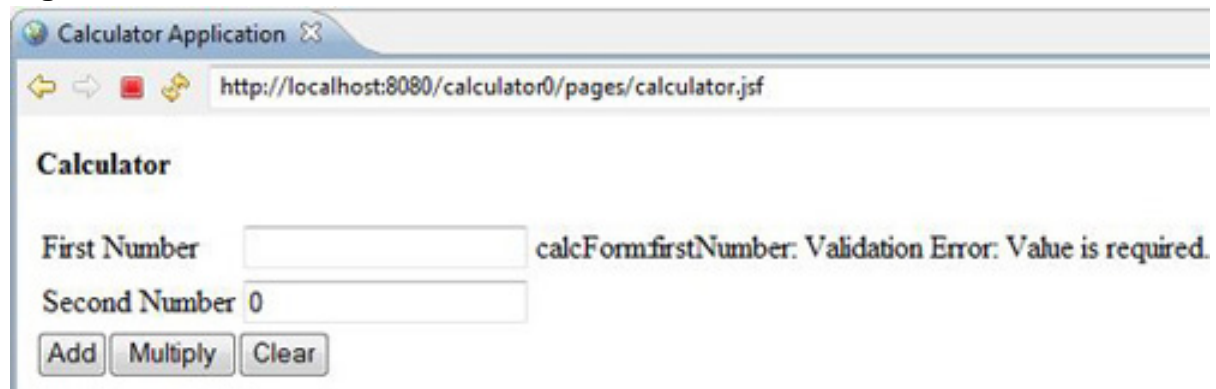
Is Struts easier than JSF?

I estimate that it would take at least twice the effort to create a classic Struts version of the simple JSF application you built here. Using Struts, you would need two action classes for the two buttons, each requiring its own set of action mappings. You would also need an action mapping to load the first page, at least assuming you were following the Model 2 recommendation. To mimic JSF's default error handling and validation you would have to configure Struts to use the validator framework or implement the equivalent in the `validate` method on an `ActionForm`. You would also need to declare a `DynaValidatorForm` in the Struts configuration, or create an `ActionForm` and override the `validate` method, or use the subclass of the `ValidatorForm` with the hooks into validator framework. And finally, you would probably need to configure some forwards (possibly two sets for each action) or global forwards to be used by all the actions. Friends don't let friends use classic Struts on new applications.

To run this application, go to the page where the WAR file is mapped (on my box this `http://localhost:8080/calculator0/`). This causes the `index.jsp` file to load the `calculator.jsp` page under the JSF context. If you are using Eclipse WTP and you have the server set up, you right-click the `calculator.jsp` page in the Navigator and choose the **Run As > Run On Server** option.

If you enter some invalid text (for example, `abc`) in either the First Number field or the Second Number field and submit, you're taken back to the `/calculator.jsp` view, and an error message is displayed next to the corresponding field. If you leave either field blank and submit, you're taken back to the `/calculator.jsp` view, and an error message is displayed next to the corresponding field. Thus, you can see that some validation is nearly automatic in JSF merely by specifying that the fields are required and binding the fields to `int` properties. Figure 4 shows how the application deals with validation and data-conversion errors:

Figure 4. Validation and data-conversion errors



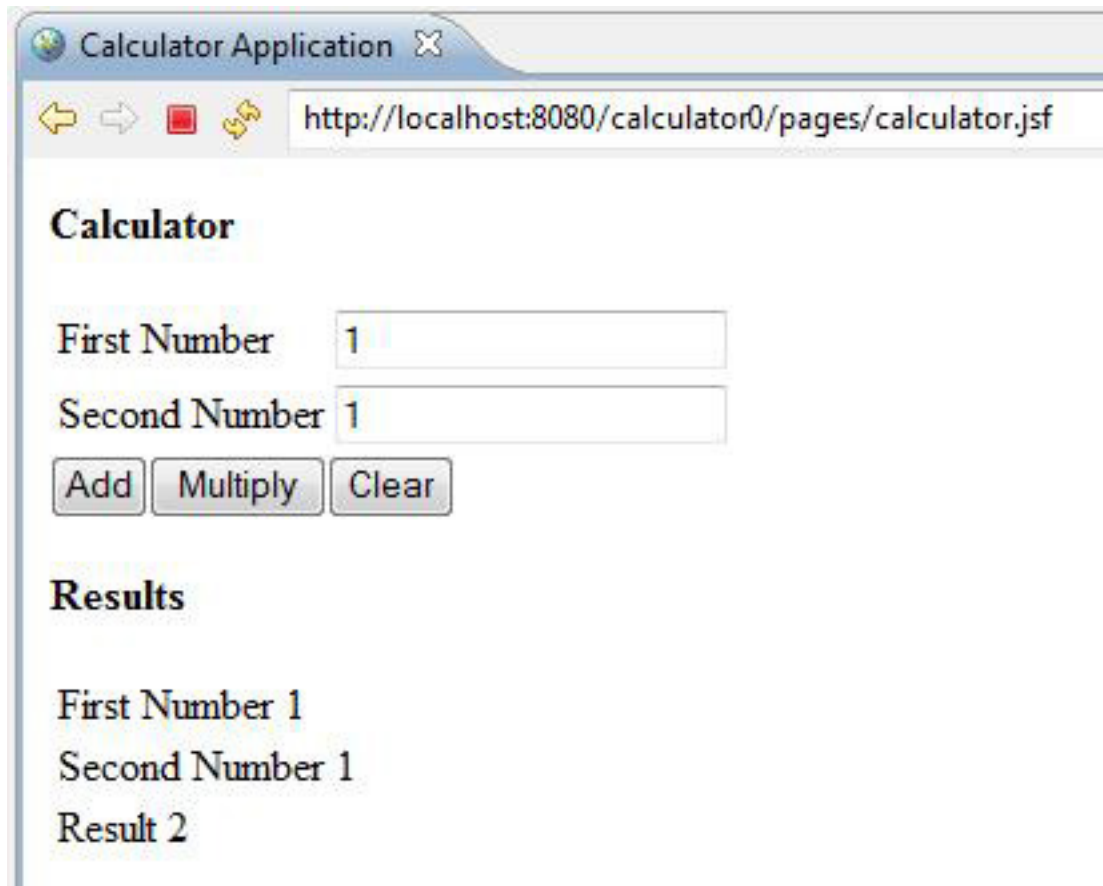
Notice the weird error messages. The default error messages for conversion and required values are not as user friendly:

- `calcForm:firstNumber: 'abc' must be a number between -2147483648 and 2147483647 Example: 9346.`
- `calcForm:secondNumber: Validation Error: Value is required.`

You'll fix these error messages in the next section.

Once you enter and submit two values whose sum or product isn't zero, the results section appears, as shown in Figure 5:

Figure 5. Results panel



Section 4. Improving the Calculator example

In this section, you'll improve the Calculator application's appearance and simplify it by using JSF techniques. You'll learn how to use CSS, set internationalization (I18N) messages, and improve the application's look and feel in other ways. You'll also improve on the default error messages, which leave a lot to be desired.

Use a panel grid

In the preceding section, you used a lot of HTML to do layout. With HTML, you can lay out the page exactly how you want. However, a Web application's layout might not be as important as, say, the layout of a marketing brochure. To simplify the Calculator application's GUI quite a bit, for now you'll use a `<h:panelGrid>` to lay

out items in a grid.

Listing 12 shows how the input-form code changes to use the `<h:panelGrid>`:

Listing 12. Changing the form to use the `<h:panelGrid>`

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

Listing 13 shows how the results section changes to use the `<h:panelGrid>`:

Listing 13. Changing the results section to use the `<h:panelGrid>`

```
<h:panelGroup rendered="#{calculator.result != 0}">
  <h4>Results</h4>
  <h:panelGrid columns="1">
    <h:outputText value="First Number #{calculator.firstNumber}"/>
    <h:outputText value="Second Number #{calculator.secondNumber}"/>
    <h:outputText value="Result #{calculator.result}"/>
  </h:panelGrid>
</h:panelGroup>
```

You've eliminated almost 20 lines of code (about a third), making it a little easier to read. And now this application has a higher level of "JSFness," which some people (for example, most developers) like, and others (Web designers) hate. You need to balance what you do according to your project's needs. Projects vary, and so do the need for absolute control (pure HTML layout) and easy-to-maintain applications (more JSFness).

Here is where things get a little tricky. An `<h:panelGrid>` can contain only components, whereas `<h:form>`, `<f:view>`, and `<h:panelGroup>` can contain both HTML and components. The first `<h:panelGrid>` in the form has three columns, so as soon as you add an extra component, it goes to the next row. The second `<h:panelGrid>` contains only one column, so each component is added to the next row. The `<h:panelGrid>` renders a table, so the output is nearly identical to what you had before. (To the user it is identical.) To reiterate: you can't add HTML to a `<h:panelGrid>`. It won't render it where you expect. It takes only components.

Dress up the GUI with CSS

If you know HTML and CSS, then you know how you would augment the look and feel of the first version of the Calculator application. The `<h:panelGrid>` allows you to do this as well. You'll import a CSS style sheet and then use it with the `<h:panelGrid>`. You'll make the `<h:panelGrid>` show up with a border and alternating silver and white rows.

First, import the stylesheet, as shown in Listing 14:

Listing 14. Importing the style sheet

```
<head>
<title>Calculator Application</title>
  <link rel="stylesheet" type="text/css"
        href="<%=request.getContextPath()%>/css/main.css" />
</head>
```

Listing 15 is the stylesheet:

Listing 15. CSS style sheet

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

evenRow {
  background-color: silver;
}

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

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

Listing 15 defines the `oddRow` and `evenRow` styles, making odd rows white and even rows silver.

Now apply these styles to the `panelGrid`s. Listing 16 adds them to the form `panelGrid`:

Listing 16. Using style classes with the form `panelGrid`


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

By setting the `rowClasses="oddRow, evenRow"` attribute, Listing 16 applies the `oddRow` and `evenRow` styles to the form. The `styleClass="formGrid"` draws the border around the table. The results `<h:panelGrid>`, shown in Listing 17, is similar, with a smaller border:

Listing 17. Using style classes with the results panelGrid

```
<h:panelGrid columns="1" rowClasses="oddRow, evenRow"
  styleClass="resultGrid">
  ...
</h:panelGrid>
```

Figure 6 shows how the Calculator application looks now:

Figure 6. Calculator with some style

Calculator 2

First Number	<input type="text" value="5"/>
Second Number	<input type="text" value="5"/>
<input type="button" value="Add"/> <input type="button" value="Multiply"/> <input type="button" value="Clear"/>	

Results

First Number 5
Second Number 5
Result 10

I've only scratched the surface of the styles that the `<panelGrid>` supports. Refer to the tag lib API link in [Resources](#) for more details.

Start cleaning up the error messages

The error messages are great if your users are propellerheads or techno weenies. Otherwise they're not user friendly. You can improve them several ways. You can start by adding a label, as shown in Listing 18:

Listing 18. Adding a label

```
<h:inputText id="firstNumber" label="First Number" ... />
...
<h:inputText id="secondNumber" label="Second Number" .../>
...
```

Notice that you use the `label="First Number"` attribute in the `h:inputText` field. Now the message text is what you see in Figure 7:

Figure 7. Messages with labels

Calculator 2

First Number	<input type="text" value="aa"/>	First Number: 'aa' must be a number between - 2147483648 and 2147483647 Example: 9346
Second Number	<input type="text"/>	Second Number: Validation Error: Value is required.
<input type="button" value="Add"/> <input type="button" value="Multiply"/> <input type="button" value="Clear"/>		

The label names are no longer property names, so they are more user friendly. But why you would want labels at all, given that error messages are laid out right next to the fields anyway? Also, the label messages are very long. You can shorten them by using the code in Listing 19:

Listing 19. Showing summary messages instead of detail

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

```
<h:message for="firstNumber" showSummary="true" showDetail="false"/>
```

Notice that Listing 19 sets the `showSummary` and `showDetail` of the `h:message` component as `showSummary="true" showDetail="false"`. This yields "First Number: 'aaa' must be a number consisting of one or more digits." and "Second Number: Validation Error: Value is required." for conversion and requirement of the `firstNumber` and `secondNumber` input, respectively. This still doesn't work well, though. Read on for a better alternative.

Override message text

JSF 1.2 added a `requiredMessage` and a `conversionMessage`, so you can override the messages on a case-by-case basis, as shown in Listing 20:

Listing 20. Using `requiredMessage` and `converterMessage` to shorten messages

```
<!-- First Number-->
<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" />
<!-- Second Number-->
<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" />
```

Notice that the `h:inputTexts` in Listing 20 adds `requiredMessage="required" converterMessage="not a valid number"`. Now this looks nice, and the messages make sense in the context of the `<h:panelGrid>`: they are laid out next to the fields, so the user knows by context where they apply (see Figure 8):

Figure 8. Shorter messages

Calculator 2

First Number	<input type="text" value="aa"/>	not a number
Second Number	<input type="text"/>	required
<input type="button" value="Add"/> <input type="button" value="Multiply"/> <input type="button" value="Clear"/>		

A problem with this approach is that you need to add `requiredMessage` and `converterMessage` to every `inputText` field. For this simple example, that is not a problem at all. For a real application, though, it could be a large maintenance problem and surely breaks the DRY (don't repeat yourself) principle.

Change the messages globally

To change the messages globally, you can define a resource bundle in the `faces-config.xml` file and use it to redefine the default messages, as shown in Listing 21:

Listing 21. Configuring messages in `faces-config.xml`

```
<?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
    http://java.sun.com/xml/ns/javaee/web-facesconfig_1_2.xsd"
  version="1.2">
  <application>
    <message-bundle>messages</message-bundle>
  </application>
  ...
</faces-config>
```

The `message.properties` file has the entries shown in Listing 22:

Listing 22. Messages resource bundle (`messages.properties`)

```
javax.faces.component.UIInput.REQUIRED_detail=required
javax.faces.converter.IntegerConverter.INTEGER_detail=not a valid number
```

Now the messages are changed globally whenever a field is required or an integer conversion fails.

Note: If you are using Eclipse JEE, be sure to add `src/main/resources/messages.properties` as a source folder.

The improvements you've made in this section have increased the application's GUI logic, so in the next section, you'll add a `CalculatorController` class that injects the `Calculator` class.

Section 5. Adding a controller

Now you'll refactor the application so you don't bind the pure Java `Calculator` object to JSF and pollute its POJOness. You'll do this by creating a controller class and injecting the pure model object into the controller class. The controller class will be JSF aware, and the model class will not.

This section covers:

- Using JSF's dependency-injection container
- Working with JSF `facesContext`
- Adding `FacesMessages`
- Using the `h:messages`
- Binding components to the controller

You'll work through each of these steps. Then I'll backtrack and explain each step in more detail.

Add a `divide()` method to `Calculator`

First, add a `divide()` method to the `Calculator`, as shown in Listing 23, so you can recover from divide-by-zero exceptions and add a `FacesMessage` to display to the user:

Listing 23. Adding a `divide()` method to the `Calculator` POJO

```
package com.arcmind.jsfquickstart.model;

/**
 * Calculator. Simple POJO.
 *
 * @author Rick Hightower
 */
public class Calculator {

    /** First number used in operation. */
    private int firstNumber = 0;

    /** Result of operation on first number and second number. */
    private int result = 0;

    ...

    /** Divide the two numbers. */
    public void divide() {
        this.result = this.firstNumber / this.secondNumber;
    }

    /** Clear the results. */
    public void clear () {
        result = 0;
    }

    ...
}
```

Create the controller class

Next, add a new class called `CalculatorController` that takes the `Calculator` POJO.

`CalculatorController` has three JSF components bound to it. It is highly *JSF aware*. It also recovers from exceptions by putting `FacesMessages` in the `FacesContext`.

The three JSF components bound to `CalculatorController` are:

- `resultsPanel`, which is a `UIPanel`
- `firstNumberInput`, which is a `UIInput`
- `secondNumberInput`, which is a `UInput`

Figure 9 shows how the calculator application handles errors:

Figure 9. Divide-by-zero exception

/ by zero

First Number	<input type="text" value="100"/>
Second Number	<input type="text" value="1"/>
<input type="button" value="Add"/> <input type="button" value="Multiply"/> <input type="button" value="Divide"/> <input type="button" value="Clear"/>	

Figure 10 shows how the application reports status:

Figure 10. Results displaying with status message

Calculator 3rd Example

Added successfully

First Number	<input type="text" value="5"/>
Second Number	<input type="text" value="5"/>
<input type="button" value="Add"/> <input type="button" value="Multiply"/> <input type="button" value="Divide"/> <input type="button" value="Clear"/>	

Results

First Number 5
Second Number 5
Result 10

CalculatorController, shown in Listing 24, keeps the JSF peanut butter out of your POJO chocolate. It is JSF aware, has components bound to it, and puts error

and status messages into the `facesContext`.

Listing 24. The JSF-aware CalculatorController

```
package com.arcmind.jsfquickstart.controller;

import javax.faces.application.FacesMessage;
import javax.faces.component.UIInput;
import javax.faces.component.UIPanel;
import javax.faces.context.FacesContext;
import com.arcmind.jsfquickstart.model.Calculator;

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) {
            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";
    }
}
}
//remove simple props

```

Update calculator.jsp

Next, update calculator.jsp as shown in Listing 25 to display error messages and bind to calculatorController instead of the Calculator POJO directly:

Listing 25. Updated calculator.jsp

```

<?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>
    <link rel="stylesheet" type="text/css"
        href="<%=request.getContextPath()%>/css/main.css" />
</head>
<body>
<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">
  <%-- First Number--%>
  <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" />

  <%-- Second Number--%>
  <h:outputLabel id="sn1" 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>

</body>
</html>

```

Map the controller into faces-config.xml

Next you need to map the new controller into faces-config.xml and inject calculator into it, as shown in Listing 26:

Listing 26. Updated faces-config.xml

```

<?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
    http://java.sun.com/xml/ns/javaee/web-facesconfig_1_2.xsd"

```

```
version="1.2">
  <application>
    <message-bundle>messages</message-bundle>
  </application>

  <managed-bean>
    <managed-bean-name>calculatorController</managed-bean-name>
    <managed-bean-class>
      com.arcmind.jsfquickstart.controller.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>
      com.arcmind.jsfquickstart.model.Calculator
    </managed-bean-class>
    <managed-bean-scope>none</managed-bean-scope>
  </managed-bean>
</faces-config>
```

Now that you've modified the entire application, I'll break it down and cover the details.

Dependency injection with JSF

JSF supports dependency injection. You can inject beans into the properties of other beans. Because you're injecting the `calculator` bean into `calculatorController`, you can put it in scope `none`. Scope `none` means it will just be created and not put into a scope when it gets created. Listing 27 shows the portion of `faces-config.xml` that injects the managed `calculator` bean with a scope of `none`:

Listing 27. Managed calculator, scope none

```
<managed-bean>
  <managed-bean-name>calculator</managed-bean-name>
  <managed-bean-class>
    com.arcmind.jsfquickstart.model.Calculator
  </managed-bean-class>
  <managed-bean-scope>none</managed-bean-scope>
</managed-bean>
```

The `calculatorController` is mapped under `request` scope. You inject `calculator` into `calculatorController` by using `<managed-property>` and passing the `#{calculator}` expression. This ends up creating a `Calculator`

object and injecting it into the `CalculatorController` using the `CalculatorController` `setCalculator` method, as shown in Listing 28:

Listing 28. Managed calculator, scope request, and injection with managed-property

```
<managed-bean>
  <managed-bean-name>calculatorController</managed-bean-name>
  <managed-bean-class>
    com.arcmind.jsfquickstart.controller.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>
```

`CalculatorController` uses `calculator`, so you inject `calculator`. This lets you use `calculator` and keep it clean from JSF, which a good model object should be. JSFness is in `CalculatorController` only. This good separation of concerns helps makes your code more testable and more reusable.

CalculatorController's JSF binding components

`CalculatorController` knows a lot about JSF by design. `CalculatorController` binds three JSF components. One of them is the `resultsPanel`, which represents the panel that displays the results of the calculator, as shown in Listing 29:

Listing 29. CalculatorController's resultsPanel

```
private UIPanel resultsPanel;

...

public UIPanel getResultsPanel() {
    return resultsPanel;
}

public void setResultsPanel(UIPanel resultPanel) {
    this.resultsPanel = resultPanel;
}
```

The `resultsPanel` gets bound into `CalculatorController` by JSF. See the binding attribute in Listing 30:

Listing 30. Binding components into the controllers

```
<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>
```

In Listing 30, `binding="#{calculatorController.resultsPanel}"` associates the `resultsPanel` component with the binding. Essentially, JSF sees this expression, and when the page loads, it injects the `resultsPanel` component by calling the `calculatorController.setResultsPanel` method. This is a convenience that lets you manipulate the component's state programmatically without traversing the component tree.

Actually, what JSF does is call `calculatorController.getResultsPanel`. If that call returns a component, the JSF view uses that component. Otherwise, if `calculatorController.getResultsPanel` returns null, JSF creates the `resultPanel` component and then calls `calculatorController.setResultPanel` with the new component based on the binding expression.

Listing 31 shows how `CalculateController`'s `add()` method uses this technique:

Listing 31. CalculateController's add() method, turning off resultsPanel

```
public String add() {
    ...
    try {
        calculator.add();
        resultsPanel.setRendered(true);
        ...
    } catch (Exception ex) {
        ...
        resultsPanel.setRendered(false);
    }
    return null;
}
```

Need Ajax support? No worries! Partial page rendering to the rescue

If you use a framework such as JBoss Ajax4Jsf (see [Resources](#)), you can easily update the JSF calculator application with very few

changes to support partial page rendering. (This same support will come with JSF 2.0 out of the box. Ajax4JSF works with JSF 1.1 and JSF 1.2 so you can use it now.) From a user perspective, the applications would seem like an applet or a Flex application. And you don't need to write any JavaScript!

In Listing 31, if the call to the `calculator.add` method is successful, the `CalculateController.add` method calls `resultsPanel.setRendered(true)`, which turns on the results panel. If the call is unsuccessful, the `CalculateController.add` calls `resultsPanel.setRendered(false)`, and the panel no longer renders.

Let's pause here. This is important: Because JSF is a component model, and components are stateful, the components remember their state. You don't need to include logic as you did in the earlier Calculator example. Tell the component not to render itself, and it does not render itself anymore. Tell a component to disable itself, and it's disabled every time you load the view as long as the view is active. JSF is much closer to a traditional GUI application than Model 2 style frameworks. The less code you need to write, the more quickly you can develop a Web application. JSF puts the *application* in *Web application*. Think about it.

CalculatorController working with messages

JSF has a mechanism for displaying status messages to users. `CalculateController` uses the `FacesContext` to add messages to the `FacesContext` so that they can be displayed to the user with the `<h:messages>` tag.

JSF stores a `FacesContext` in a `ThreadLocal` variable that you can access by calling the `FacesContext.getCurrentInstance()` method. The `add()` method uses the current `FacesContext` to add messages that will be available for that request, as shown in Listing 32:

Listing 32. CalculateController's add() method adding JSF messages

```
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));
        //Log the exception as well.
        ...
    }
    return null;
}
```

The code in Listing 32 adds a `FacesMessage` to the `facesContext` with a severity level of `INFO` if the add operation was successful or a `FacesMessage` of severity level `ERROR` if the add operation throws an exception.

The messages are displayed to the user with the `<h:messages>` tag, as shown in Listing 33:

Listing 33. Showing the end user error and status messages

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

Setting the `globalOnly` attribute to `true` shows only messages that are not tied to specific components, like the ones you added in Listing 32. Notice that you use different styles for status messages and error messages.

CalculatorController fixing the divide-by-zero exception

Because you are dealing with a component model, you can modify the values of components and initialize them based on display logic. When the new divide method throws a divide-by-zero exception, you'll cause it to recover by setting the `secondNumberInput` value to 1.

First, you need to bind the `secondNumberInput` to the `CalculatorController` class, as shown in Listing 34:

Listing 34. Binding input components: binding="#{calculatorController.resultsPanel}"

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

Next, you use the `secondNumberInput` component. If you get a divide-by-zero exception, you recover by setting the `secondNumberInput` to 1, as shown in Listing 35:

Listing 35. New divide() method

```
public String divide() {
    FacesContext facesContext = FacesContext.getCurrentInstance();

    try {
        calculator.divide();
        facesContext.addMessage(null, new FacesMessage(
            FacesMessage.SEVERITY_INFO, "Divided successfully", null));
        resultsPanel.setRendered(true);
    } catch (Exception ex) {
        if (ex instanceof ArithmeticException) {
            secondNumberInput.setValue(Integer.valueOf(1));
        }
        facesContext.addMessage(null,
            new FacesMessage(FacesMessage.SEVERITY_ERROR, ex.getMessage(), null));
    }
    return null;
}
```

It's important to think of JSF as more of a traditional GUI component model and not a fancy version of Model 2. Lots of possibilities emerge if you remember JSF is a component model. You can set the value of the `secondNumberInput` in Listing 35 because it is an object, not a piece of HTML in a JSP. You can manipulate it, and it will remember its value. It is stateful.

Working with attributes

Most JSF attributes take expressions, so if you wanted to turn the field label red when an error occurs, you can easily do so, as shown in Listing 36:

Listing 36. Turning the label red

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

Notice that the `styleClass` attribute is set to the expression `#{calculatorController.firstNumberStyleClass}`, which is bound to the method in Listing 37:

Listing 37. Returning the right style class if an error occurs

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



```

    } else {
      return "errorClass";
    }
  }
}

```

Listing 37 asks the `firstNumberedInput` component if its input is valid and then changes the `styleClass` that's returned based on whether it is valid.

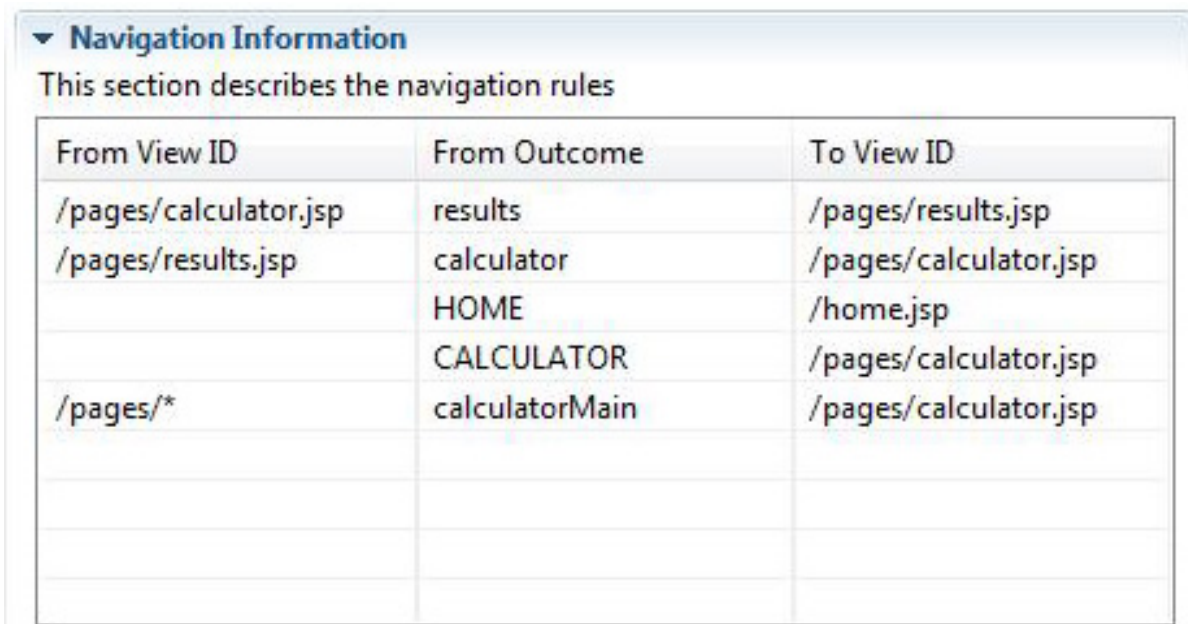
Section 6. Navigation in JSF

JSF has a navigation mechanism (similar to Struts). JSF's navigation mechanism has logical outcomes that you map to the next logical view. In this section, you'll add navigation to the Calculator application.

Navigation rules

Figure 11 shows the navigation rules that you'll add to the Calculator application:

Figure 11. Navigation rules to add to the Calculator application

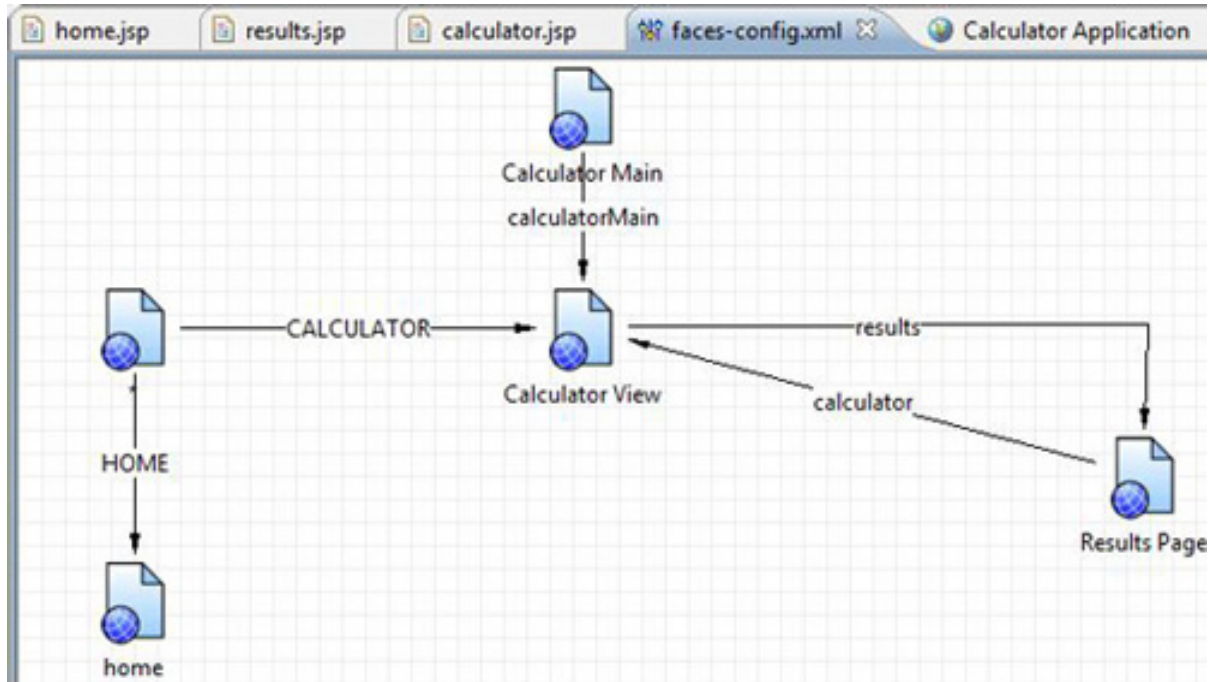


From View ID	From Outcome	To View ID
/pages/calculator.jsp	results	/pages/results.jsp
/pages/results.jsp	calculator	/pages/calculator.jsp
	HOME	/home.jsp
	CALCULATOR	/pages/calculator.jsp
/pages/*	calculatorMain	/pages/calculator.jsp

It's nice to have tools that help you lay out your Web application's flow. Many IDEs provide tools for drawing the navigation rules of an JSF application. Figure 12 shows

the Navigation Rule Layout Tool from Eclipse JEE:

Figure 12. Navigation rules layout in Eclipse



After you work through this section, Figures 11 and 12 will make a lot more sense.

Maybe you don't need navigation

There are times when you need to navigate from one view to the next. But many JSF component frameworks have tree views and tab views. One view might have 10 tabs, and clicking on a tab loads a different part of the view. Using this design, you could probably easily create a large application without ever writing navigation rules because the whole application is in one view. The ramifications of this approach, including workarounds and best practices, are out of this tutorial's scope.

You'll first add a home page that links to the calculator page. Then you'll split the calculator page into two pages: one to show the calculator view, and one page to show the results view. You'll also need navigation rules back and forth among the calculator page, results page, and the home page.

Linking from the home page to the calculator page

You can link from the home page to the calculator page in three ways:

- Via a `commandLink` and a navigation rule
- Via a `commandLink` and a navigation rule using a `redirect`
- Via an `outputLink`

Linking via `commandLink` and a navigation rule is done by adding a navigation rule in the `faces-config.xml` file, as shown in Listing 38:

Listing 38. Navigation rule defined in `faces-config.xml`

```
<navigation-rule>
  <navigation-case>
    <from-outcome>CALCULATOR</from-outcome>
    <to-view-id>/pages/calculator.jsp</to-view-id>
  </navigation-case>
</navigation-rule>
```

Listing 38 states that any action that returns `CALCULATOR` will cause JSF to load `/pages/calculator.jsp` as the next view. This rule is dynamic, in that any action from anywhere in the application that returns `CALCULATOR` will go to `/pages/calculator.jsp` (unless a more-specific rule applies — in JSF you can add a `<from-view-id>` that then makes the rule apply only to that view; I'll cover the more-specific case later in this section.) Listing 38 is similar to a *global forward* in Struts-speak.

In the form, add the `commandLink` shown in Listing 39, which must be inside of a `<h:form>`:

Listing 39. Using `commandLink` in the home page

```
<h:form>
  <h:panelGrid columns="1">
    <h:commandLink action="CALCULATOR" value="Calculator Application"/>
    ...
  </h:panelGrid>
</h:form>
```

This works as you would expect: the Calculator application loads in the browser. But the user might find it confusing that the URL in the browser still says `http://localhost:8080/calculator3/home.jsf` even though the calculator view is showing. This can be disconcerting, especially to Web-savvy users. And any user who wants to bookmark the calculator part of the application won't know the true link.

One way to fix this is by using a `redirect` in the `faces-config.xml` navigation rule, as shown in Listing 40:

Listing 40. Navigation rule with `redirect` element

```
<navigation-rule>
  <navigation-case>
    <from-outcome>CALCULATOR_REDIRECT</from-outcome>
    <to-view-id>/pages/calculator.jsp</to-view-id>
    <redirect/>                                <!-- LOOK HERE -->
  </navigation-case>
</navigation-rule>
```

A `commandLink` can use the navigation rule by specifying the outcome string as the action attribute. When clicked, the link in Listing 41 takes the user to the calculator page:

Listing 41. Using navigation rule with redirect

```
<h:commandLink action="CALCULATOR_REDIRECT" value="Calculator Application (redirect)"/>
```

This solves the problem, but at the cost of a small extra hit to the server, which on a slow connection might be pretty long. But if you're building an internal application, then this hit won't amount to much.

You don't have to hit the server twice if you don't mind directly linking to the page you want to load, as shown in Listing 42:

Listing 42. Linking directly with outputLink

```
<h:outputLink value="pages/calculator.jsf">
  <h:outputText value="Calculator Application (outputlink)"/>
</h:outputLink>
```

Listing 42 links directly to the next view. In a Model 2 architecture and in JSF, it is considered bad form to link directly from one view to the next view. Typically, the controller should have a chance to initialize the model for the next view, so it is better to go through an action method. But Listing 42 does create a link, and the right URL will show up in the browser.

The ability to bookmark JSF applications has been a problem for a while. Several frameworks have addressed this, including JBoss Seam. This is a small nit that will also be addressed in JSF 2.

Navigating to the results page

After you perform a calculation in the calculator, you want the next view to be the new results page. To accomplish this, add the navigation rule in Listing 43:

Listing 43. Navigation rule from all actions in calculator view to results page

```
<navigation-rule>
  <display-name>Calculator View</display-name>
  <from-view-id>/pages/calculator.jsp</from-view-id>
  <navigation-case>
    <from-outcome>results</from-outcome>
    <to-view-id>/pages/results.jsp</to-view-id>
  </navigation-case>
</navigation-rule>
```

Listing 43 says if the current view is the calculator view (`calculator.jsp`), and any action returns `results`, then the next JSF view will be the results page (`results.jsp`). JSF looks at the return value from the action methods and converts them to a string (if they are not null) and uses that string to pick the next view. The return type can be any object because its `toString` method will get called. (A lot of folks use `Enums`.)

Change all of your operations to return `results`, as in the `add()` method shown in Listing 44:

Listing 44. Action methods now return results

```
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));
    }
    return "results";
}
```

Notice that the `add()` method returns `results`, and that `results` is mapped to `results.jsp`. Note that the Cancel button does not return `results`. If an action returns a value that is not mapped to a navigation rule, JSF stays on the current view.

You can get more specific, as in Listing 45:

Listing 45. Mappings matched to actions

```
<navigation-rule>
  <display-name>Calculator View</display-name>
  <from-view-id>/pages/calculator.jsp</from-view-id>
```

```
<navigation-case>
  <from-action>#{calculatorController.add}</from-action>
  <from-outcome>results</from-outcome>
  <to-view-id>/pages/results.jsp</to-view-id>
</navigation-case>
</navigation-rule>
```

But now you'd need to have a mapping per method. Yawn! That is not very DRY.

You also want to add a return-home `commandButton` to the calculator page, as shown in Listing 46:

Listing 46. Calculator actions with return-home link

```
<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"/>
  <h:commandButton action="HOME" value="Home" immediate="true"/>
</div>
```

Note that the home link specifies the action value of `HOME`. Listing 47 shows the navigation rule that maps the `HOME` outcome that the home link uses to the home page:

Listing 47. Home mapping navigation rule

```
<navigation-rule>
  <navigation-case>
    <from-outcome>HOME</from-outcome>
    <to-view-id>/home.jsp</to-view-id>
    <redirect/>
  </navigation-case>
</navigation-rule>
```

Note that the `commandButtons` and `commandLinks` work the same with respect to action handling and navigation rules.

The results page has a link back to the calculator page and a link back to the home page, as shown in Listing 48:

Listing 48. Results page can go home or back to the calculator

```
<h:panelGrid columns="1" rowClasses="oddRow, evenRow">
  <h:commandLink action="calculator" value="Return to the calculator page"/>
  <h:commandLink action="HOME" value="Go to the home page"/>
  <h:commandLink action="calculatorMain" value="Go to main calculator page"/>
</h:panelGrid>
```

You can go back to the calculator two ways. One is similar to the way you've already seen, shown in Listing 49:

Listing 49. Mapping back to the calculator

```
<navigation-rule>
  <display-name>Results Page</display-name>
  <from-view-id>/pages/results.jsp</from-view-id>
  <navigation-case>
    <from-outcome>calculator</from-outcome>
    <to-view-id>/pages/calculator.jsp</to-view-id>
  </navigation-case>
</navigation-rule>
```

Listing 49 says if you're on the results page (/pages/results.jsp) and an action returns `calculator`, then go the calculator page (/pages/calculator.jsp). If you want something that is less specific than Listing 49 but more specific than the global forward type that I covered first, you could use Listing 50:

Listing 50. Another mapping back to the calculator from anywhere under pages/*

```
<navigation-rule>
  <from-view-id>/pages/*</from-view-id>
  <navigation-case>
    <from-outcome>calculatorMain</from-outcome>
    <to-view-id>/pages/calculator.jsp</to-view-id>
    <redirect/>
  </navigation-case>
</navigation-rule>
```

Listing 50 allows you to define logical areas of your application and have outcomes that apply to just those places.

Section 7. Conclusion

This tutorial covered JSF and showed that it is a component model. Because JSF is a component framework like Swing, SWT, or AWT, it makes Web application development more like traditional GUI development and less like traditional Web development. Applications written with JSF are shorter and easier to understand and maintain, versus applications written in a typical Model 2 style. There seems to be a

lot of interest in JSF in the Java community and growing job demand to match.

JSF 2, which is forming, incorporates Facelets concepts, adds native Ajax support, and makes JSF component development easier. JSF 2 should further enhance your investment in JSF. The new model has partial page rendering via Ajax, which is available today from tools like Ajax4JSF.

This is not to say that JSF does not have competitors. Among the server-side component models, Tapestry 5 looks solid yet is not compatible with Tapestry 4. Wicket is also very interesting but has not gained enough momentum for many to consider it.

Then there are the noncomponent server-side Java frameworks. It has been said that Struts 2.x has done a great job in improving WebWork, and some expect Struts 2.1.x to open the door to bigger and better things, yet Struts 2 is really based on WebWork and not Struts 1.x. Spring MVC continues to grow by leaps and bounds and is a good choice if you pick a non-GUI component server-side Web framework.

Lastly, there are pure client-side plays that delegate to services on the server, such as the Google Web Toolkit (GWT) and Adobe Flex. (The architecture is different from JSF but the target applications are the same.) Each has its advantages and drawbacks but could impact JSF adoption.

Yet JSF is likely to continue to do well because it is the standard for Java EE and has quite an active community behind it. Demand for JSF exceeds demand for Tapestry, Spring MVC, Java Flex, and GWT. JSF 2.0 could propel JSF even further.

Part 2 in this tutorial series will cover the JSF life cycle and working with validators, converters, phase listeners, and other advanced JSF features.

Downloads

Description	Name	Size	Download method
Sample code for this article ¹	j-jsf1.zip	112KB	HTTP

[Information about download methods](#)

Note

1. For instructions and JAR files to run this tutorial's source code with Eclipse JEE, Tomcat, and Maven 2, see [Resources](#).

Resources

Learn

- [Instructions and JARs for running this tutorial's source code with Eclipse JEE, Tomcat, and Maven 2.](#)
- ["Web Tier to Go With Java EE 5: Summary of New Features in JavaServer Faces 1.2 Technology"](#) (Jennifer Ball and Ed Burns, java.sun.com, February 2006): Read about the new features in JSF 1.2.
- ["Web Tier to Go With Java EE 5: Summary of New Features in JSP 2.1 Technology"](#) (Pierre Delisle and Jennifer Ball, java.sun.com, February 2006): Get an overview of the new features introduced in version 2.1 of JSP technology.
- [Facelets](#): Facelets is a view technology that focuses on building JSF component trees.
- ["Facelets fits JSF like a glove"](#) (Richard Hightower, developerWorks, February 2006): In this article, the author introduces Facelets' easy HTML-style templating and reusable composition components.
- ["Advanced Facelets programming"](#) (Richard Hightower, developerWorks, May 2006): Read about advanced ways to bridge the gap between JSF and EL.
- [JSFToolbox for Dreamweaver](#): A suite of design and coding extensions for Dreamweaver that lets Java developers design and build JSF pages with Adobe's Web authoring tool.
- [JSF Tag Library Documentation](#): Official documentation for the `html` and `core` tag libraries.
- [JSF API](#): API Javadocs for the JSF specification.
- [JBoss Ajax4jsf](#): Partial page rendering with JSF and Ajax.
- [Crank](#): Crank is a master/detail, CRUD, and annotation-driven validation framework built with JPA, JSF, Facelets, and Ajax. Crank is a use-case analysis of what is possible with the new JSF 2.0 stack.
- Browse the [technology bookstore](#) for books on these and other technical topics.
- [developerWorks Java technology zone](#): Hundreds of articles about every aspect of Java programming.

Discuss

- Check out [developerWorks blogs](#) and get involved in the [developerWorks community](#).

About the author

Richard Hightower

Rick Hightower serves as chief technology officer for [ArcMind Inc.](#). He is coauthor of the popular book *Java Tools for Extreme Programming* and coauthor of *Professional Struts* and [Struts Live](#). Rick is the founding developer on the [Crank project](#), a JSF/Facelets, Ajax, CRUD framework for idiomatically developing GUIs.

Trademarks

Java and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.