



## CHAPTER 2

# LIFECYCLE AND PAGE NAVIGATION



## OBJECTIVES

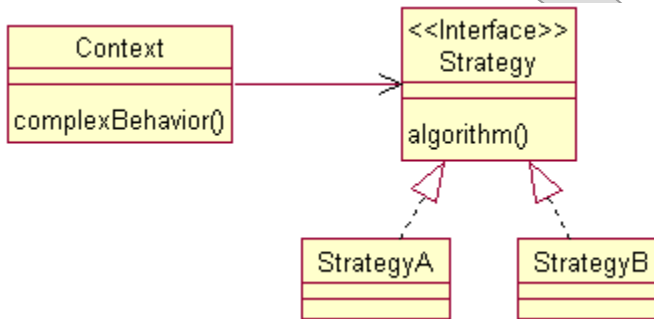
*After completing “Lifecycle and Page Navigation,” you will be able to:*

- Describe the JSF framework in terms of singleton objects that carry out tasks behind the scenes.
- Enumerate the six phases of the JSF request/response lifecycle, and describe their functions.
- Implement and declare a phase listener for a JSF application.
- Define navigation rules to cause JSF to move from one page to another in a complex web application.

# The Strategy Pattern

---

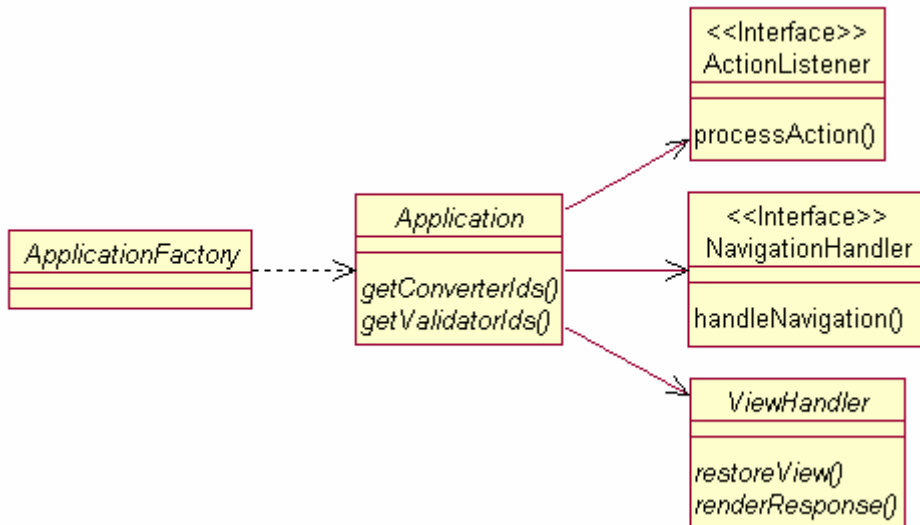
- The **Strategy** design pattern is a basic but often overlooked technique for factoring out pieces of a complex algorithm.



- The **Context** object has some complicated job to do.
  - It could carry out the whole task by itself, but that would make for terrible maintenance characteristics.
  - It could define a big pile of virtual methods – **onThis** and **onThat** – allowing subclasses to hook into its process and customize it.
  - This is in fact the **Template Method** pattern, and it's useful but it has its limits, especially since each unique set of customizations would require a fresh subclass.
- **Strategy** calls for a separate interface for each piece of the larger process that can be made reasonably discrete.
  - Then subtypes can implement the strategy and plug in to the main processor.

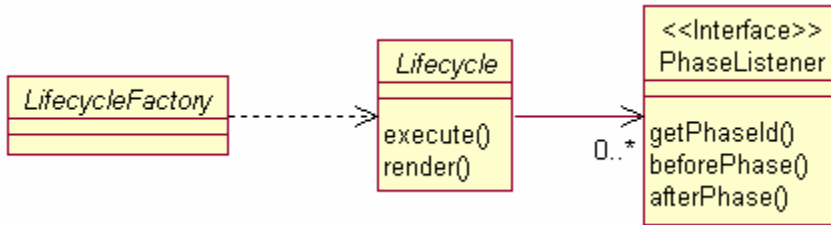
# The Framework of a JSF Application

- The **FacesServlet** carries out the complex task of coordinating an HTTP request/response cycle for a JSF application.
- For many of the major functions involved in this cycle, it delegates to some Strategy through factories defined in the configuration file.



- The **Application** object is a singleton that represents the application configuration as a whole. Various strategies are collected here, all of which have default implementations.

# The Lifecycle Class



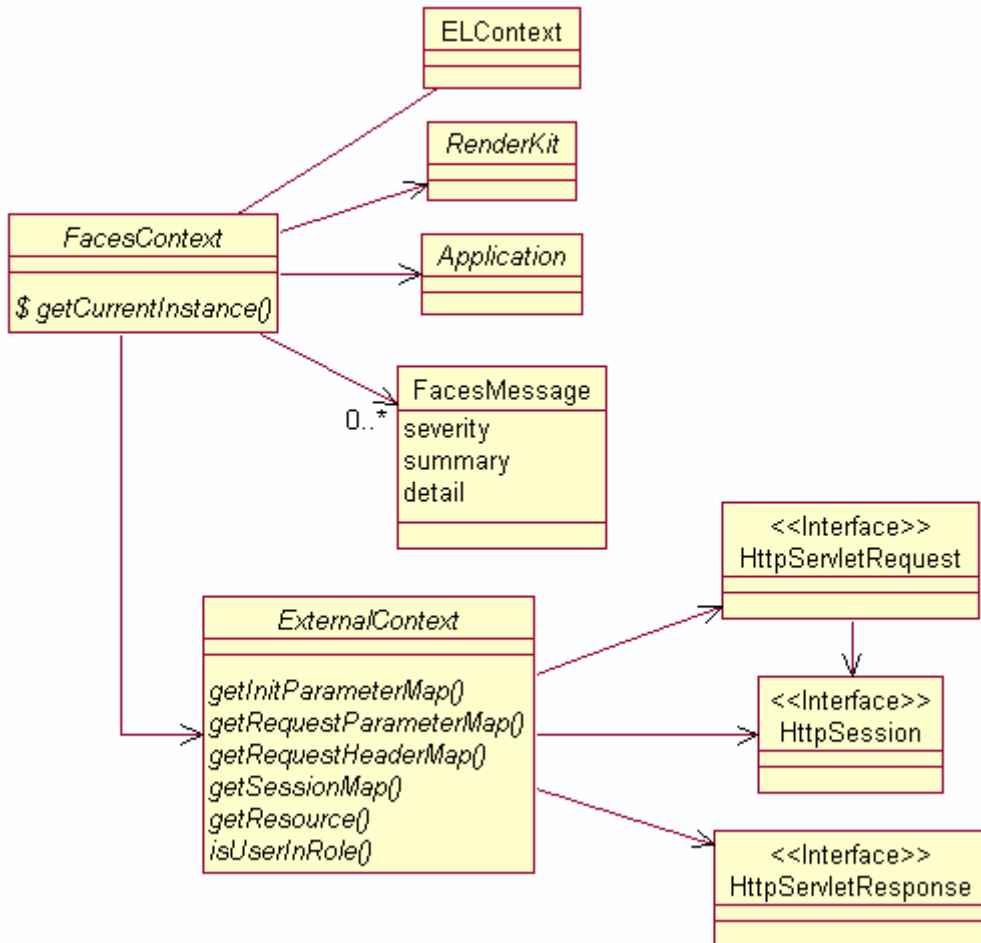
- **Lifecycle** is the object that carries out request handling, with the help of many of the other objects shown on the previous page.

```
public abstract class Lifecycle
{
    public abstract void execute (FacesContext ctx);
    public abstract void render (FacesContext ctx);
    ...
}
```

- The **execute** method creates or re-creates the UI tree, reads in request parameters, carries out validation, transfers values to the backing bean(s), and invokes the application's control logic.
- **render** goes the other way, creating a new HTML page as the HTTP response, based partly on the results of **execute**.
- Both methods fire **PhaseEvents** before and after each of six formally defined lifecycle phases.
  - You can register **PhaseListeners** with the **Lifecycle** object programmatically or declaratively; we'll see the latter approach by example in a moment.
  - The **UIViewRoot** instance that exists at the root of every JSF view can also register phase listeners, through a slightly different mechanism.

# The FacesContext Class

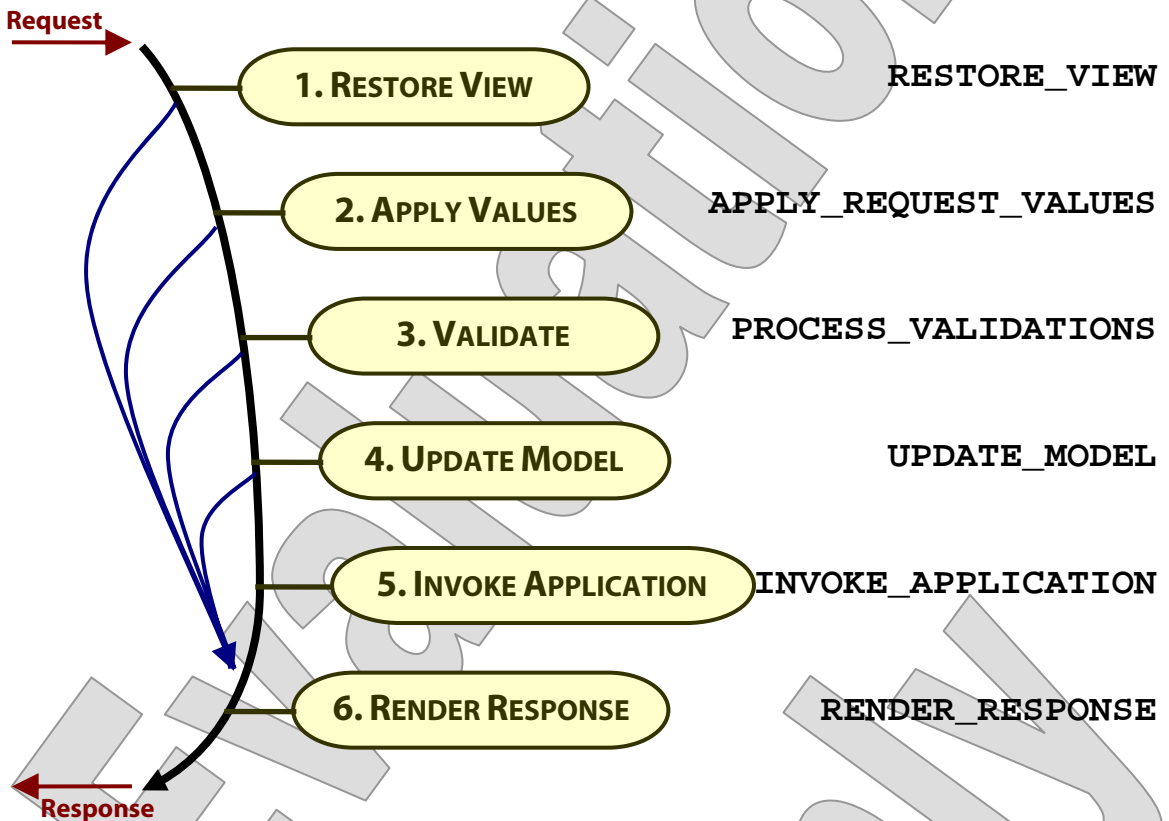
- Notice the **FacesContext** parameter on those lifecycle methods.
- This class is the centerpiece of the JSF framework:



- It is passed as an argument to many JSF API methods.
- When not given, it can be derived by any piece of code by calling **FacesContext.getCurrentInstance**.
- It is a gateway to many other useful objects and values, as shown above.

# The Request/Response Cycle

- **Lifecycle** handles each HTTP request in a series of **phases**.
- The **PhaseId** class encapsulates these as a Flyweight (Java 1.4 enumerated type) – the identifiers are shown at right:

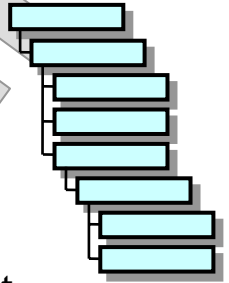


- The **Lifecycle.execute** method actually manages the first five of these phases; JSF makes thin slices of the request handling, as that's the part that's unknown and unpredictable.
- Then **Lifecycle.render** deals with the sixth and final phase.
- Note that a fatal error might occur in any phase, and the framework can respond to certain indicators by moving directly to render a response.

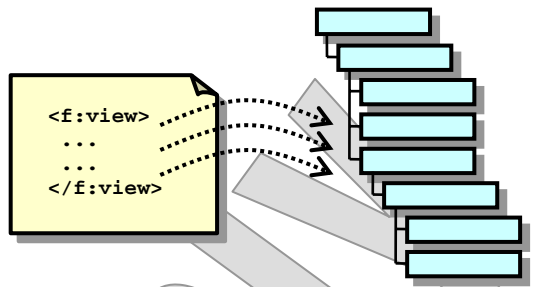
# Lifecycle Phases

---

- In the **Restore View** phase, JSF classes build the tree of UI components for the incoming request.
  - This is one of the trickiest parts of JSF: a view must be selected or created if necessary, and there is a great deal of state management by JSF to track the status of the view – typically using HTML hidden input fields.
  - Fortunately this is also the phase that requires the least intervention (typically, none) by application code.
  - Most JSF applications will define their views in JSP (or Facelets) and supply navigation rules that wire multiple pages into flows.
  - The default **ViewHandler** takes it from there.



- During **Apply Request Values** (is the full name), the request parameters are read and their values are used to set the values of the corresponding UI components.

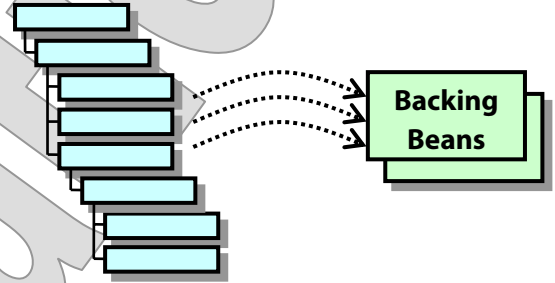


- The **Apply Validations** phase triggers calls to all registered validators.
  - Any input can be scanned by any number of validators.
  - Any validation errors will abort the request-handling process and skip to rendering the response.

# Lifecycle Phases

- The **Update Model** phase brings a transfer of state from the UI component tree to any and all backing beans, according to the **value expressions** defined for the components themselves.

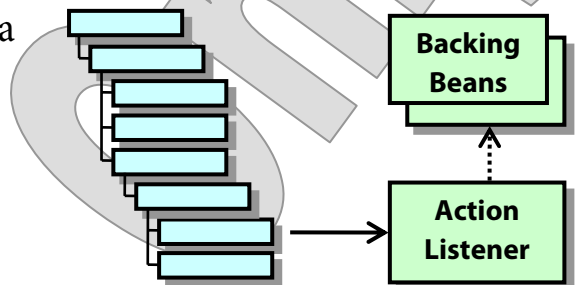
- Note the difference between this phase and Apply Request Values: that phase moves values from client-side HTML form controls to server-side UI components; while in this phase the information moves from the UI components to the backing beans.



- It is in this phase that **converters** are invoked to parse string representations of various values to their proper primitive or object types.

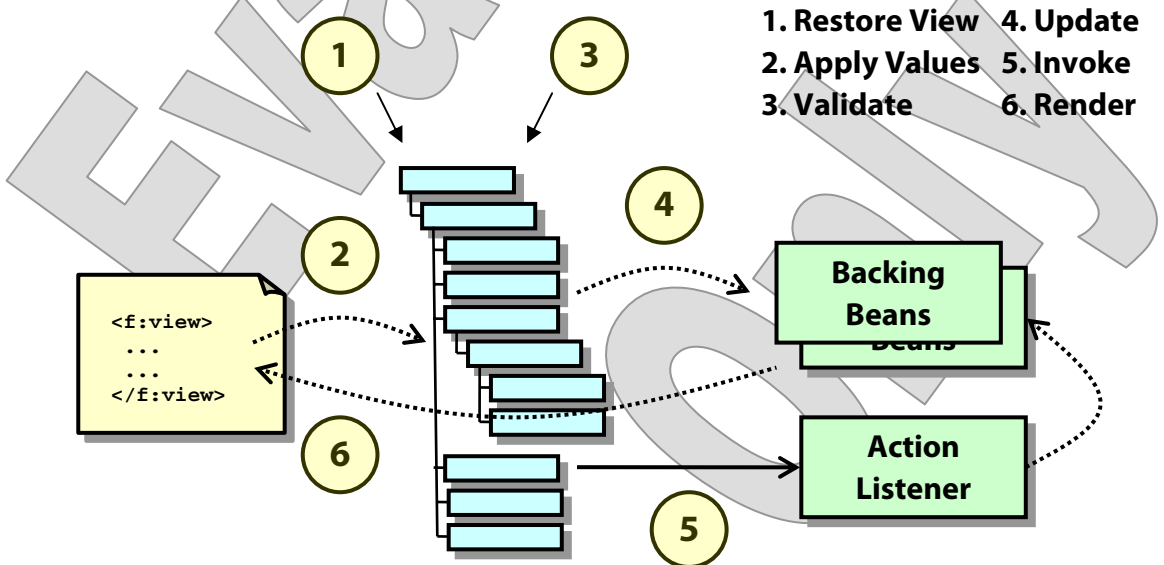
- **Invoke Application** does exactly that; this is the primary location at which the application code responds to the request proper.

- Typically this takes the form of a call to process the action event generated by the submit button that the user clicked, but there are several variations on this theme.



# Lifecycle Phases

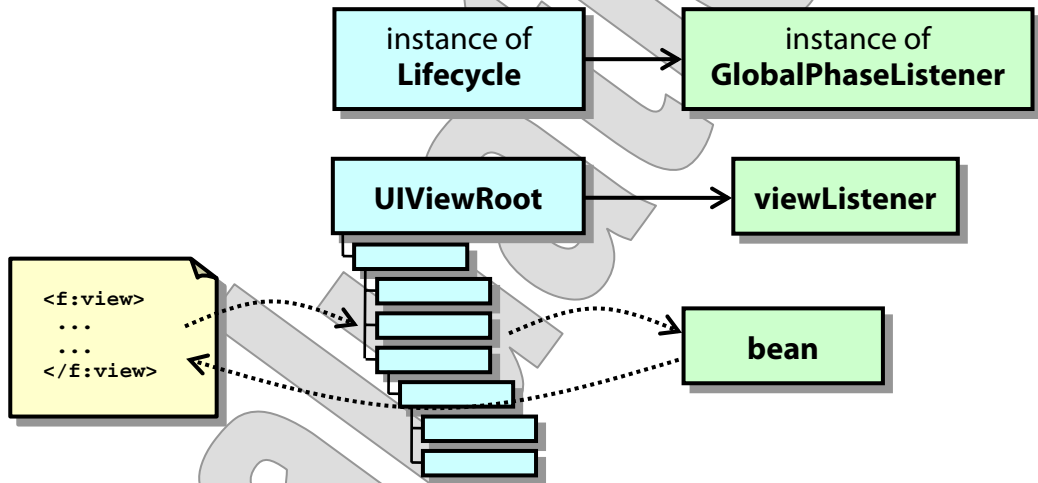
- Finally, **Render Response** brings several inverse behaviors together in one process:
  - **Values are transferred back** to the UI components from the bean – including any modifications that may have been made by the bean itself or by the controller.
  - The UI components **encode** their HTML representations as part of an overall view-generation process.
  - The UI components **save their state** – not just their values, but other attributes having to do with the presentation itself. This can happen server-side, but by default state is written into the HTML as hidden input fields and thus returns to the JSF implementation with the next request.
- A summary of the process is shown here:



# Tracing a Request

EXAMPLE

- In **Examples/Lifecycle/Step1** there is an application that does nothing but trace an HTTP request through the JSF lifecycle.
- We're going to set up a few "listening posts" and see what they can tell us:



- We'll register a **PhaseListener** with the global **Lifecycle** object.
- We'll register a different sort of listener with the view root itself.
- We'll trace calls to get/set methods on a simple backing bean.

- Start, as we usually will, with the configuration file:
  - There is a `<managed-bean>` declaration for a generic **bean**; this is just here to trace calls to its own accessor and mutator methods.
  - Another bean is **viewListener** – this will be registered on the **UIViewRoot** and be notified at the start and end of each phase.

```
<managed-bean>
  <managed-bean-name>viewListener
  </managed-bean-name>
  <managed-bean-class>cc.jsf.ViewPhaseListener
  </managed-bean-class>
  <managed-bean-scope>application
  </managed-bean-scope>
</managed-bean>
```

- Here's a top-level element we've not seen before: `<lifecycle>` can declare phase-listener classes and JSF will automatically instantiate and register them:

```
<lifecycle>
  <phase-listener>cc.jsf.GlobalPhaseListener
  </phase-listener>
</lifecycle>
```

- Note that we are now taking the tack of forwarding from **index.jsp** to a **.jsf** URL:

```
<body>
  <jsp:forward page="lifecycle.jsf" />
</body>
```

- This way we get the auto-mapping to **index.jsp** without the hang-ups of needing JSF to intervene that we saw in the Ellipsoid demo in the previous chapter.

- **lifecycle.jsf** uses a JSF **method expression** to assign specific methods of a specific bean as phase listeners:

```
<f:view
  beforePhase="#{viewListener.before}"
  afterPhase="#{viewListener.after}"
>
  <h:form id="form" >
    <h:selectOneMenu value="#{bean.selection}" >
      <f:selectItems value="#{bean.selections}" />
    </h:selectOneMenu>
    <h:commandButton value="Choose" />
    <h:commandButton value="Jump the Gun"
      immediate="true" />
  </h:form>
</f:view>
```

- Note that the bean class does not have to implement **PhaseListener** in this approach; instead JSF uses reflection to find the right method of a predefined signature.
- We'll talk more about method expressions in Chapter 4.

- Our formal **PhaseListener** implementation is found in **src/cc/jsf/GlobalPhaseListener.java**:

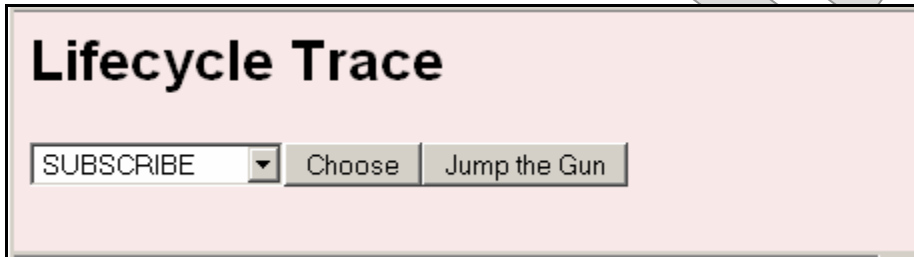
```
public void beforePhase (PhaseEvent ev)
{
    if (ev.getPhaseId ().equals
        (PhaseId.RESTORE_VIEW))
    {
        // print a separator for the report
    }
    System.out.println ();
    System.out.println ("GlobalPhaseListener.before("
        + ev.getPhaseId () + ")");
}

public void afterPhase (PhaseEvent ev)
{
    System.out.println ("GlobalPhaseListener.after("
        + ev.getPhaseId () + ")");
}
```

- The **ViewPhaseListener** class (found in the same package) does more or less the same thing.
  - However, note that this class does not implement the JSF **PhaseListener** interface, while **GlobalPhaseListener** does.
  - This is just a different approach that is friendly to JSPs, using the new EL method expression to identify a method name rather than using traditional Java polymorphism. Again, we'll delve into this technique later in the course.

- Build and deploy the application (**ant**) and test at the URL:

`http://localhost:8080/Lifecycle`



- Before doing anything else, look at the server console, and see this report from the request that JSF just handled:

```
GlobalPhaseListener.before(RESTORE_VIEW 1)
GlobalPhaseListener.after(RESTORE_VIEW 1)

GlobalPhaseListener.before(RENDER_RESPONSE 6)
  ViewPhaseListener.before(RENDER_RESPONSE 6)
    Menu.getSelections()
    ...
    Menu.getSelection()
    ...
  ViewPhaseListener.after(RENDER_RESPONSE 6)
GlobalPhaseListener.after(RENDER_RESPONSE 6)
```

- So immediately we learn at least three things:
  - The phase listener registered with **Lifecycle** gets the first look on the way in and the last look on the way out; the methods registered with the **UIViewRoot** are called further “inside” the cycle.
  - The **UIViewRoot** can’t inform us about the Restore View phase at all – no surprise, since this phase is dedicated to creating the **UIViewRoot**!
  - Only the first and last phase are executed for an initial request – or in fact for any request that isn’t associated with a previously-served JSF view. Since there was no pre-existing view, there is nothing to which to apply values, nothing to validate, etc.
- Now select a value and click **Choose**. In the server console, see the report on a full JSF lifecycle, as shown on the following page.
  - All six phases are executed.
  - The **UIViewRoot** is involved in all but the first phase.
  - The bean is involved in Validate (mostly because it provides a list of legal values to the HTML `<select>` control), Update Model (writing values to the bean), and Render Response (reading them).
- You can try the **Jump the Gun** button now if you like, but we’ll discuss this in a later chapter; it will be more interesting once we’ve talked more about the JSF event model and registered a few more listeners.

# Tracing a Request

EXAMPLE

```
GlobalPhaseListener.before(RESTORE_VIEW 1)
GlobalPhaseListener.after(RESTORE_VIEW 1)

GlobalPhaseListener.before(APPLY_REQUEST_VALUES
  ViewPhaseListener.before(APPLY_REQUEST_VALUES
  ViewPhaseListener.after(APPLY_REQUEST_VALUES 2
GlobalPhaseListener.after(APPLY_REQUEST_VALUES 2)

GlobalPhaseListener.before(PROCESS_VALIDATIONS 3)
  ViewPhaseListener.before(PROCESS_VALIDATIONS 3)
    Menu.getSelections()
    Menu.getSelection()
  ViewPhaseListener.after(PROCESS_VALIDATIONS 3)
GlobalPhaseListener.after(PROCESS_VALIDATIONS 3)

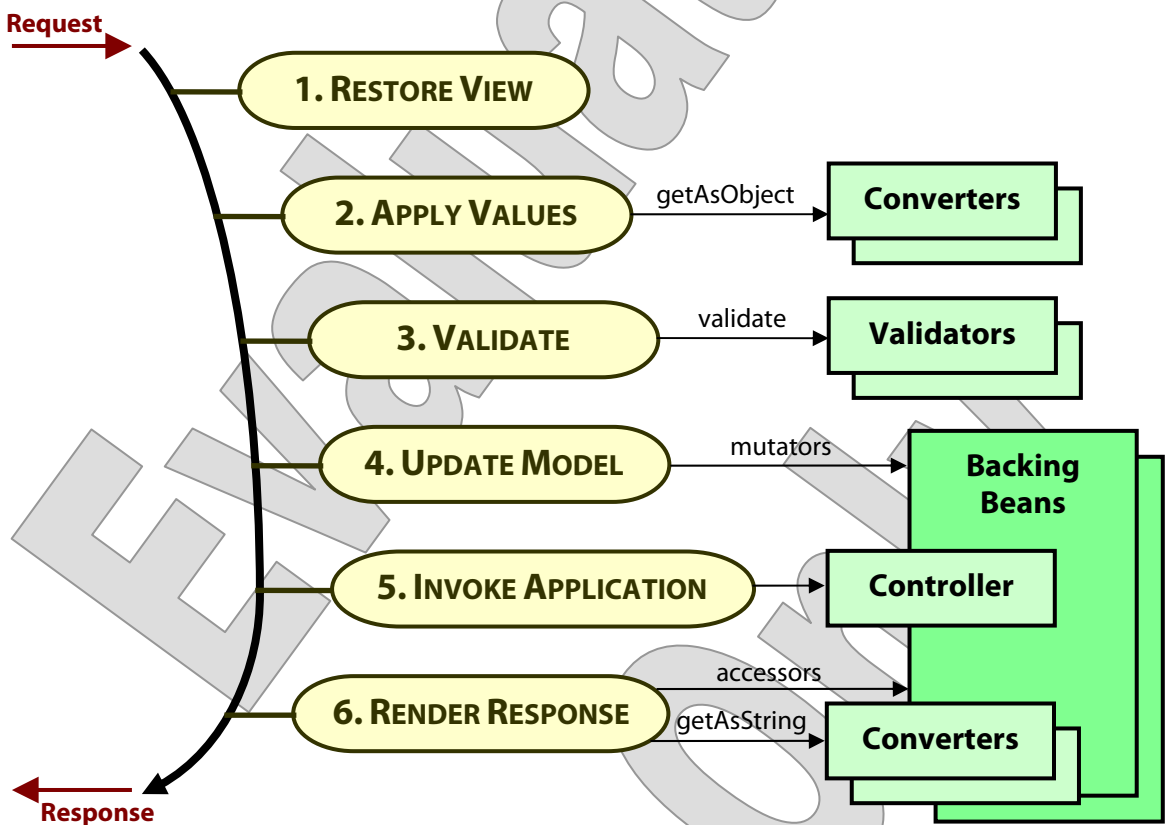
GlobalPhaseListener.before(UPDATE_MODEL_VALUES 4)
  ViewPhaseListener.before(UPDATE_MODEL_VALUES 4)
    Menu.setSelection()
  ViewPhaseListener.after(UPDATE_MODEL_VALUES 4)
GlobalPhaseListener.after(UPDATE_MODEL_VALUES 4)

GlobalPhaseListener.before(INVOKE_APPLICATION 5)
  ViewPhaseListener.before(INVOKE_APPLICATION 5)
  ViewPhaseListener.after(INVOKE_APPLICATION 5)
GlobalPhaseListener.after(INVOKE_APPLICATION 5)

GlobalPhaseListener.before(RENDER_RESPONSE 6)
  ViewPhaseListener.before(RENDER_RESPONSE 6)
    Menu.getSelections()
    Menu.getSelections()
    Menu.getSelection()
    Menu.getSelection()
    Menu.getSelection()
    Menu.getSelection()
  ViewPhaseListener.after(RENDER_RESPONSE 6)
GlobalPhaseListener.after(RENDER_RESPONSE 6)
```

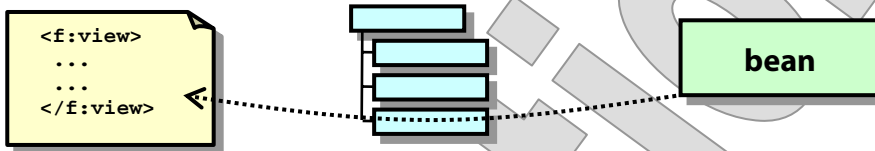
# Who Does What

- We're starting to see that JSF applications really are more event-driven than they are simply request-driven.
- Designing and implementing in JSF means breaking down application logic into many small classes: beans, controllers, validators, and so on.
- The primary type of actor in each of the lifecycle phases is shown below:



# View Selection: Initial Request

- How does JSF decide which view to instantiate (or “restore”)?
- On the initial request to a JSF application, there is no view to restore – which is why JSF jumps to rendering.

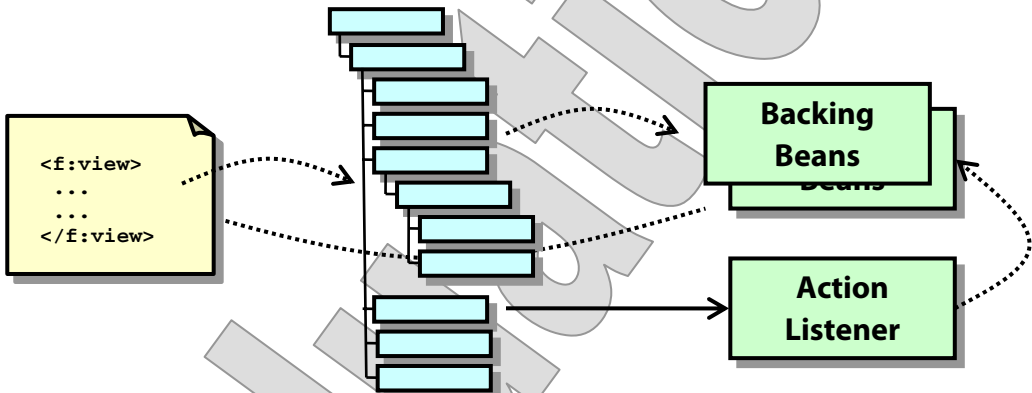


- The view selected for rendering is defined in the JSP.
  - Remember that JSF maps incoming URLs to `*.jsf` for this purpose.
- Since they are triggered by user gestures on this JSF-rendered page, successive requests will carry identifiers that allow JSF to reconstitute the previously-rendered view.
  - You may have noticed that the HTML `<form>`s we’re using have no **action** attributes.
  - The JSF custom tags use the JSF API to generate an **action** attribute; this is enough to identify the class of view to be created on a request from this form, and additional identifiers connect specific HTML tags to UI components on the server side.
  - View the HTML source of any JSF page to see this for yourself:

```
<form id="form" name="form" method="post"
  action="/Lifecycle/lifecycle.jsf"
  enctype="application/x-www-form-urlencoded">
<input type="hidden" name="form" value="form" />
<input type="hidden" name="javax.faces.ViewState"
id="javax.faces.ViewState" value="j_id4:j_id5" />
<select name="form:j_id_jsp_1467299309_2" size="1">
  <option value="SUBSCRIBE">SUBSCRIBE</option>
  ...
```

# View Selection: Recycling One Page

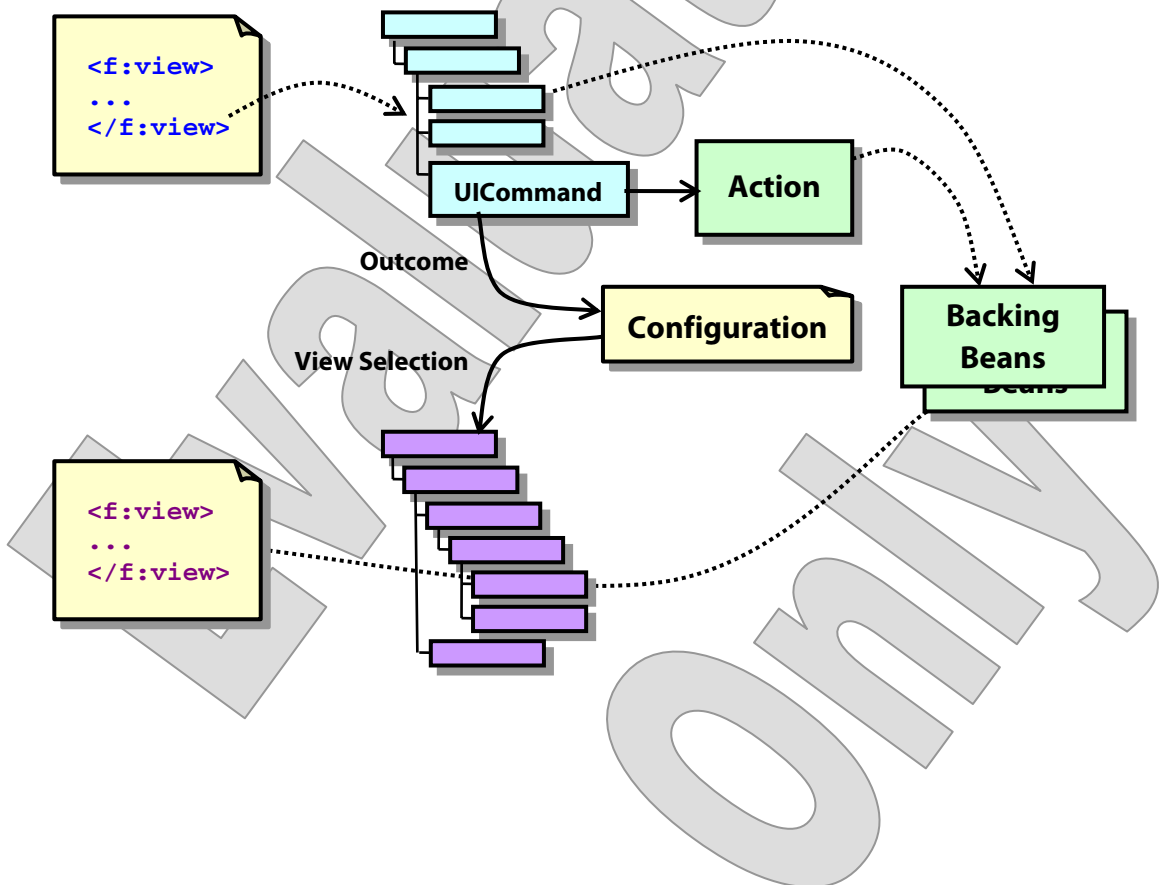
- By default, successive requests from a JSF page will forward to the same JSP, *ad infinitum*.
  - This means that the same tree of UI components will be used as the view for request processing and rendering a response.



- This is convenient for UI designs that seek to minimize the total number of screens that the user encounters and to offer a look and feel more like a desktop application.
- It can also be handy in other styles of application.

# View Selection: From Page to Page

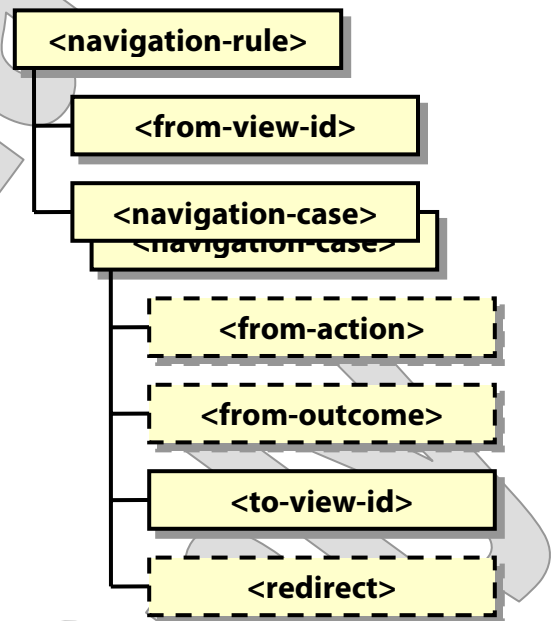
- To cause JSF to proceed from one page to another, you must inform it of the page flows of your application.
  - Do this with **navigation rules**, which predicate the choice of response view on the **outcome** of a request.
  - Now the view that's used to process the request (in the first five phases of the lifecycle) may not be the one used to render a response:



# Navigation Rules

- Define navigation rules in your configuration file.
  - As with everything in JSF, there is an option to create them programmatically, but it's a rare application that uses the JSF API for this purpose; page flows themselves are usually pre-defined.
- A navigation rule tells the JSF runtime:
  - For what **origin** we're setting a rule – this is `<from-view-id>`
  - **Where** to direct traffic in **what cases**:

```
<navigation-rule>  
  <from-view-id>  
    /index.jsp  
  </from-view-id>  
  <navigation-case>  
    <from-outcome>  
      normal  
    </from-outcome>  
    <to-view-id>  
      /results.jsp  
    </to-view-id>  
  </navigation-case>  
</navigation-rule>
```



- So JSF takes the decision of what page follows what other page away from the page definition, and puts it in the configuration.
  - It decides where to go next based on the identity of the requesting view, plus the outcome.
  - By contrast, simple HTML forms, and Struts, and Spring, all let the view define the request URLs that emanate from various HTML forms, buttons, and links.

# Outcomes

---

- How is the outcome of a request determined?
- The HTML `<submit>` button that triggered the request will be generated by a certain kind of UI component called a **command**.
  - This will be a subclass of **UICommand**.
  - It is most often represented by the JSP custom tag `<h:commandButton>`.
- A command can have an **action** attribute, and the value of this attribute shapes the outcome.
  - If it is a literal string, then this string is the outcome value – recall `index.jsp` from the Ellipsoid demo:

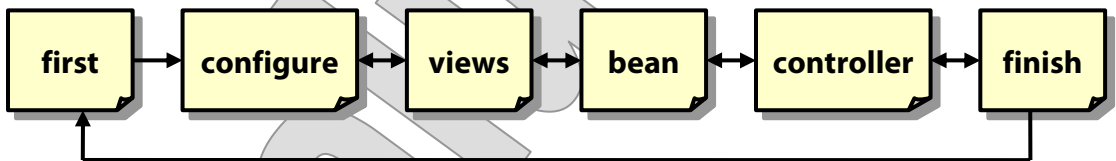
```
<h:commandButton value="Analyze" action="normal" />
```

- **action** is yet another possible place for an EL method expression, and thus a specific method on a specific bean can be called as part of the request/response cycle.

```
<h:commandButton action="#{controller.execute}" />
```

- In this case, the return value from the method will be the outcome; this of course allows for dynamic choices of outcome and hence what page might be served next.
- We'll see more of this in later chapters, but for now we'll work mostly with static outcomes, such that each button or link, combined with the navigation rules in the configuration, pre-determine the page flow for all user gestures.

- Let's take a closer look at page navigation techniques.
- In **Examples/Wizard/Step1** there is a simple wizard-style application that carries the user through the linear process ... of developing a JSF application.
  - Later we'll write a song about songwriting.
- Open the configuration file, and notice that for this exercise we're working with an application that defines no backing beans, no phase listeners, etc. – only navigation rules.
  - The page flow we're describing, in toto, is:



- The first rule lets the user proceed from the home page to **configure.jsp** when she clicks the **Next** button (which has **action="next"**):

```
<navigation-rule>
  <from-view-id>/start.jsp</from-view-id>
  <navigation-case>
    <from-outcome>next</from-outcome>
    <to-view-id>/configure.jsp</to-view-id>
  </navigation-case>
</navigation-rule>
```

- The second rule lets the user move either forward or back from **configure.jsp**, depending on which button is clicked:

```
<navigation-rule>
  <from-view-id>/configure.jsp</from-view-id>
  <navigation-case>
    <from-outcome>back</from-outcome>
    <to-view-id>/start.jsp</to-view-id>
  </navigation-case>
  <navigation-case>
    <from-outcome>next</from-outcome>
    <to-view-id>/views.jsp</to-view-id>
  </navigation-case>
</navigation-rule>
```

- The rest of the rules are about the same, with the rule for **finish.jsp** allowing the user either to go back or to start over at **start.jsp**.

- A typical page defines the JSF view to include a bit of text, a supporting image, and the **Next** and **Back** buttons.
  - See `views.jsp`:

```
<body>
  <h2>JSF Wizard: Create Views</h2>
  <f:view>
    <h:form id="views" >
      <p></p>
      <p>Build one or more JSF view trees,
        using JSP custom tags.</p>
      <p>
        <h:commandButton value="Back"
          action="back" />
        <h:commandButton value="Next"
          action="next" />
      </p>
    </h:form>
  </f:view>
</body>
```

- Build and deploy with Ant, and test the application at this URL.  
You can run to the end of the process to see:

<http://localhost:8080/Wizard>

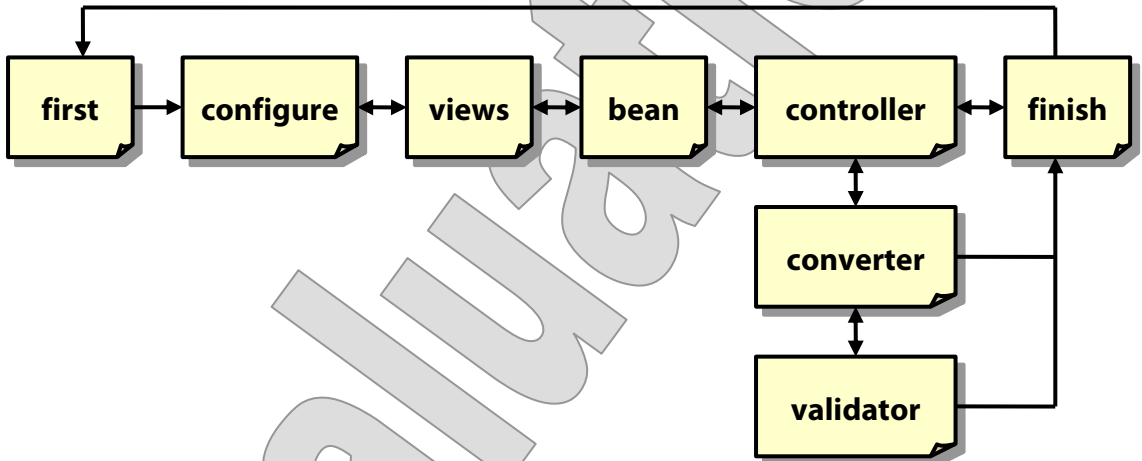
## JSF Wizard: Congratulations!

The diagram illustrates the JSF architecture. At the top, `web.xml` is linked to `faces-config.xml` by a dashed arrow. Below this, a `View` component (represented as a document icon) receives a `Request` and `invokes` the `Controller`. The `Controller` interacts with a `JavaBean` component via bidirectional dashed arrows. The `Controller` then `selects` another `View` component, which outputs a `Response`.

You've built a complete JSF application!

**Suggested time: 15-30 minutes**

In this lab you will add navigation rules to the JSF configuration file for the Wizard application that will allow the user to branch out to one or two optional steps in the page flow:



This will make for some more interesting choices in the pages, and by the end of the lab we will have started to see why most applications need dynamic outcomes at one point or another. We'll leave a couple of issues to be resolved in a later exercise.

Detailed instructions are found at the end of the chapter.

## SUMMARY

- Quite a lot goes on “under the hood” of a JSF application.
- The **FacesServlet** is the entry point for each request, but it quickly delegates to concrete strategies:
  - The **Lifecycle** object carries out the six handling phases.
  - The **FacesContext** keeps track of important request-scope actors, including the view(s), the external context with familiar servlet API object references, and the application configuration.
- Though most JSF development happens at a finer grain, it is possible to do application work at this level, by listening for the changing of the lifecycle phases and operating on objects top-down, starting from **FacesContext**.
- Navigation rules make JSF work with multiple views; by default it will always serve up the same view – until the user types a new request URL into the browser’s location bar.
  - Navigation rules define almost all the page-flow logic of a JSF application.
  - The rest is declared in the **action** attributes of commands written into the views, or implemented algorithmically in Java classes that listen for action events.