JSF, CDI and Bean Validation

JavaServer Faces, JSF

`javax.faces`

JSF Home page:

http://www.oracle.com/technetwork/java/javaee/overview/index.html

JSF tag library documentation:

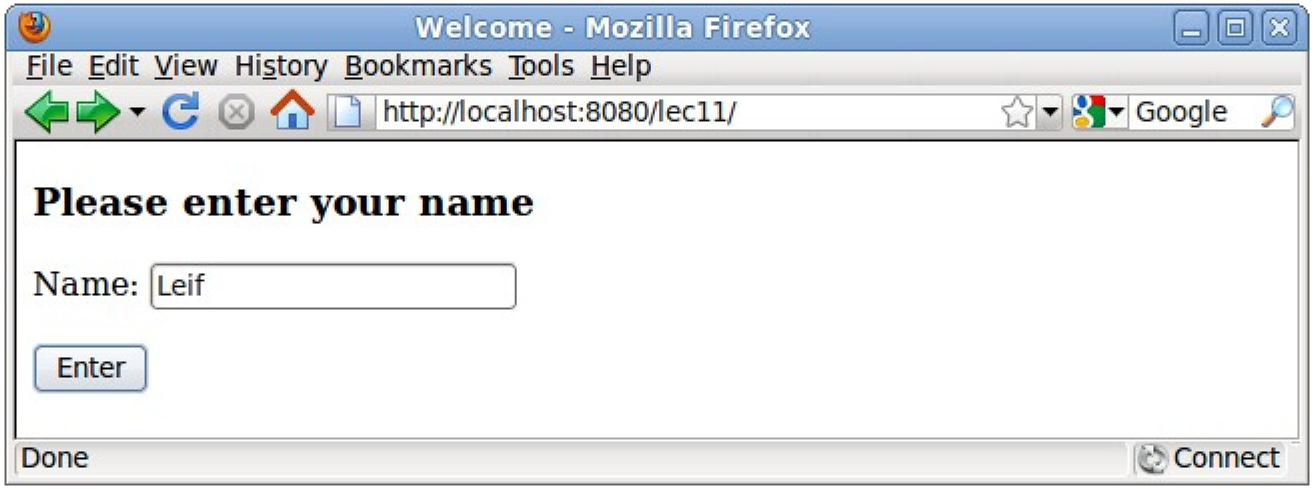
http://docs.oracle.com/javaee/7/javaserverfaces/2.2/vdldocs/facelets/

JSF, Content

- JSF Introduction
- JSF tags
- Managed Beans
- Expression language
- JSP Standard Tag Library (JSTL)

A Simple Example

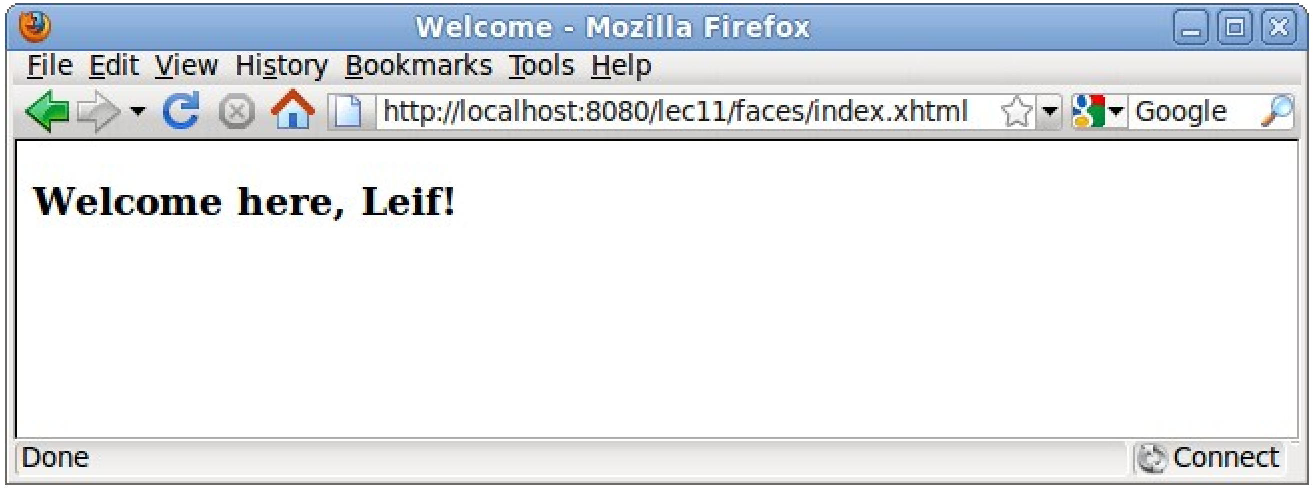
- The example has two views.


Welcome - Mozilla Firefox
File Edit View History Bookmarks Tools Help
Google
Enter
Done
Connect

http://localhost:8080/lec11/

Please enter your name

Name: Leif

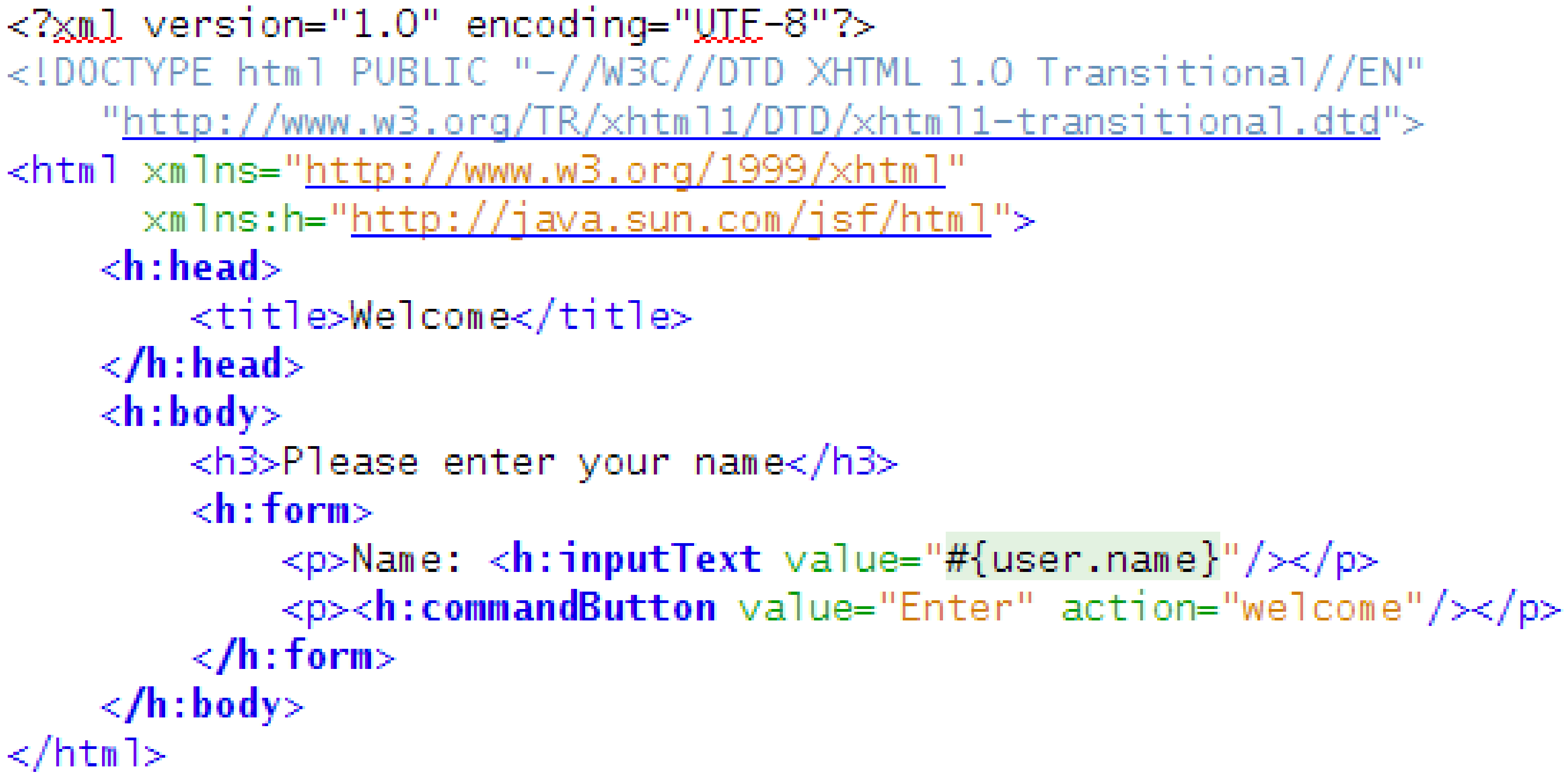

Welcome - Mozilla Firefox
File Edit View History Bookmarks Tools Help
Google
Done
Connect

http://localhost:8080/lec11/faces/index.xhtml

Welcome here, Leif!

A Simple Example, Cont'd

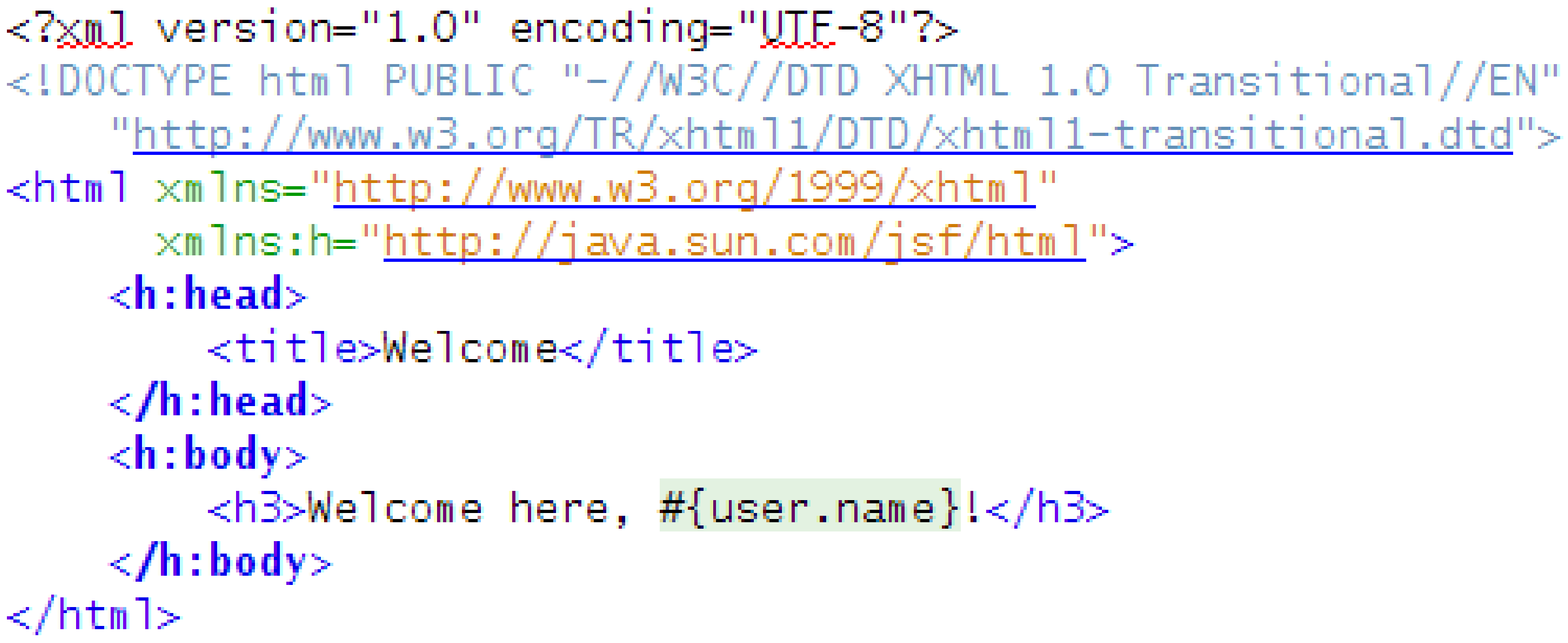
- The first JSF page, index.xhtml.



```
<?xml version="1.0" encoding="UTF-8"?>
<!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"
      xmlns:h="http://java.sun.com/jsf/html">
    <h:head>
        <title>Welcome</title>
    </h:head>
    <h:body>
        <h3>Please enter your name</h3>
        <h:form>
            <p>Name: <h:inputText value="#{user.name}"/></p>
            <p><h:commandButton value="Enter" action="welcome"/></p>
        </h:form>
    </h:body>
</html>
```

A Simple Example, Cont'd

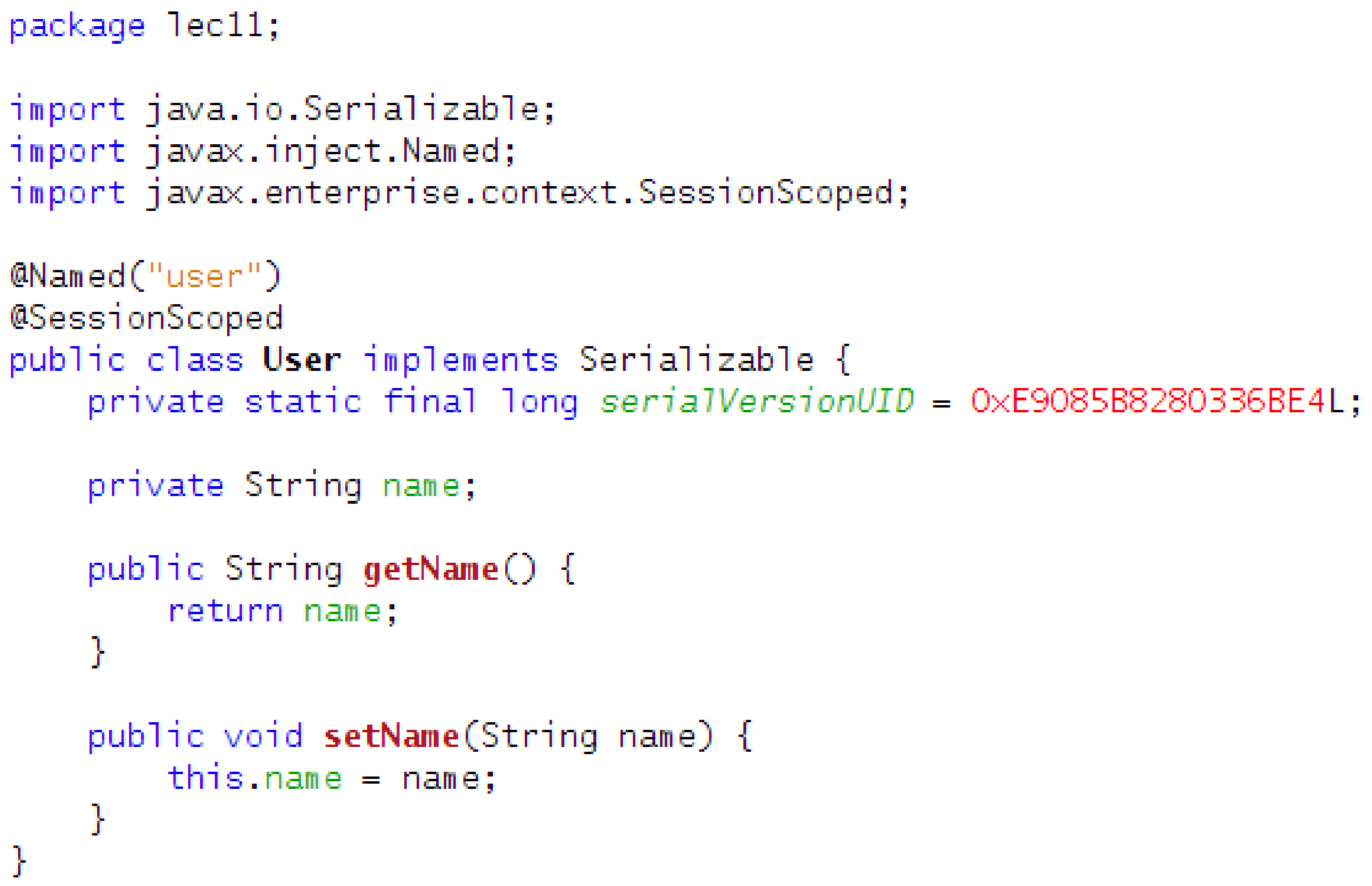
- The second JSF page, welcome.xhtml.



```
<?xml version="1.0" encoding="UTF-8"?>
<!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"
      xmlns:h="http://java.sun.com/jsf/html">
    <h:head>
        <title>Welcome</title>
    </h:head>
    <h:body>
        <h3>Welcome here, #{user.name}!</h3>
    </h:body>
</html>
```

A Simple Example, Cont'd

- The managed bean, User.java.



```java
package lec11;

import java.io.Serializable;
import javax.inject.Named;
import javax.enterprise.context.SessionScoped;

@Named("user")
@SessionScoped
public class User implements Serializable {
    private static final long serialVersionUID = 0xE9085B8280336BE4L;

    private String name;

    public String getName() {
        return name;
    }

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

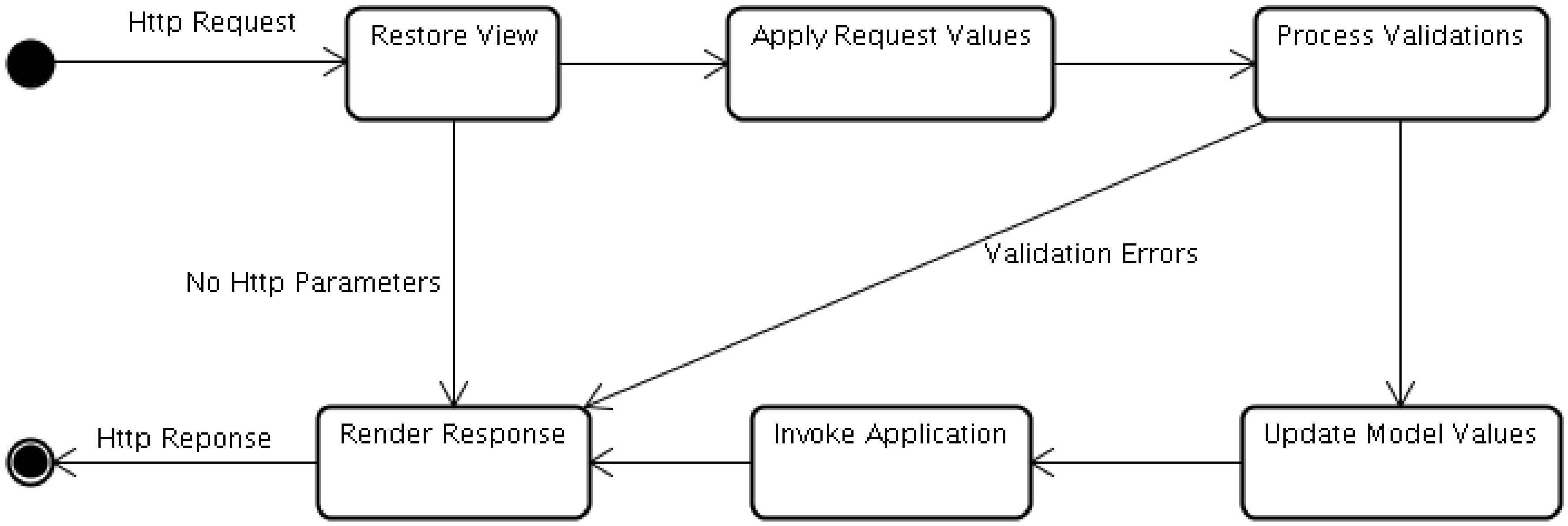
Overview of JSF Architecture

- JSF has a component based architecture
  - Treats view parts as UI components, not as HTML elements.
  - Maintains an internal *component tree*.
  - Think of it as a Swing or AWT UI.
  - **`index.xhtml`** in the initial example has three components. The first, a form, is the ancestor of the other two, a button and a text field.

Overview of JSF Architecture, Cont'd

- Each tag in a page has an internal associated tag handler class inside JSF.
  - The tag handler classes are organized according to the component tree.
- The internal JSF classes handles translation of JSF tags to HTML tags, interpretation of Http requests, calls to managed beans etc.

The Phases of a JSF Request



The Phases of a JSF Request

- *Restore View* Phase
  - Retrieves the component tree (i.e. tree of internal tag handler classes) for the page if it was displayed previously. It the page is displayed the first time the component tree is instead created.
  - If there are no Http parameters in the request JSF skips directly to the *Render Response* phase.

The Phases of a JSF Request, Cont'd

- *Apply Request Values* Phase
  - The Http request parameters are placed in a hash table that is passed to all objects in the component tree.
  - Each object identifies the parameters belonging to the component it represents and stores those parameter values.
  - Values stored in objects in the component tree are called *local values*.

The Phases of a JSF Request, Cont'd

- *Process Validations* Phase
  - It is possible to attach validators to user editable components (typically text fields) in a JSF page, using JSF tags.
  - Example of validators are that a field is not empty, that a parameter is an integer, that it is a string of a certain length etc.
  - In this phase, the validators are executed to check that the local values are correct.
  - If some validation fails JSF skips to the *Render Response* phase and redisplays the current page with error messages about the failed validations.

The Phases of a JSF Request, Cont'd

- *Update Model Values* Phase
  - The local values are used to update managed beans by invoking setter methods.
  - Managed beans and their properties are identified by their names, in the **`index.html`** page in the initial example the user enters their name in a text field that has the value **`user.name`**. This means the name is sent to the method **`setName`** in the managed bean that is named **`user`**.

The Phases of a JSF Request, Cont'd

- *Invoke Application* Phase
  - Here the method specified by the **`action`** attribute of the component that caused the Http request is called.

The Phases of a JSF Request, Cont'd

- *Render Response* Phase
  - Here the next view is created.
  - Everything in the XHTML page except JSF tags is unchanged.
  - JSF tags are transformed to XHTML tags by the objects in the component tree.
  - Getter methods in managed beans are called in order to retrieve values. In the **`welcome.xhtml`** page in the initial example the value **`user.name`** is retrieved by a call to the method **`getName`** in the managed bean that is named **`user`**.

Tag Libraries in JSF

- HTML
  - Used to create HTML elements.
  - The recommended prefix is *h:*
  - Some important tags are covered below.
- Core
  - Used to add objects , such as validators, listeners and AJAX support, to HTML elements.
  - The recommended prefix is *f:*
  - Example in the slides explaining validation.

Tag Libraries in JSF, Cont'd

- Facelets
  - Used to create composite views, e.g. views that have common components like header, footer and menu, without using duplicated code.
  - The recommended prefix is *ui:*

Tag Libraries in JSF, Cont'd

- Composite Components
  - Used to create custom components.
  - The recommended prefix is *composite:*

Tag Libraries in JSF, Cont'd

- JSTL (JSP Standard Tag Library) Core
  - Utility tags managing for example flow control.
  - The recommended prefix is *c:*
  - Some important tags are covered below.
- JSTL (JSP Standard Tag Library) Functions
  - Utility functions mainly for handling strings.
  - The recommended prefix is fn*:*
  - Some example tags are covered below.

Tag Library Declaration

- Tag libraries must be declared in the XHTML page where they are used.
- This is done in the `<HTML>` tag.
- The **`index.xhtml`** in the initial example uses the HTML tag library. It is declared as follows.

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://java.sun.com/jsf/html">
```

Some Tags in the HTML Tag Library

- **`head`**, renders the head of the page.
- **`body`**, renders the body of the page.
- **`form`**, renders an HTML form.
- **`inputText`**, renders an HTML text field.
- **`inputSecret`**, renders an HTML password field.
- **`outputLabel`**, renders a plain text label for another component.
- **`outputText`**, renders plain text.
- **`commandButton`**, renders a submit button.

Attributes for The HTML Tags

- All tags mentioned on the preceding page, except **`head`** and **`body`**, have the following attributes.
  - **`id`**, gives a unique name to the component. All components have a unique name. It is assigned by JSF if not stated explicitly with the `id` tag.
  - **`value`**, specifies the component's currently displayed value. This can be an expression that refers to a property in a managed bean. If so, the value will be read from the bean when the component is displayed and stored to the bean when the component is submitted.
  - **`rendered`**, a boolean expression that tells whether the component is displayed or not.

Attributes for The HTML Tags, Cont'd

- The **`outputLabel`** tag also has the **`for`** attribute.
  - Specifies for which other component this component is a label. The label is normally displayed immediately to the left of that other component.

Attributes for The HTML Tags, Cont'd

- The **`commandButton`** tag also has the **`action`** attribute.
  - Tells what to do when the user clicks the button.
  - Can be the name of a XHTML page, without the **`.xhtml`** extension. In this case the specified page is displayed.
  - Can also be the name of a method in a managed bean, in this case that method is invoked.

Plain HTML Tags Instead of HTML Tag Library

- It Is allowed to use plain html tags instead of the HTML tag library and the **`h:`** prefix.
- In this case, tags that shall be managed by JSF must have attributes in the **`http://xmlns.jcp.org/jsf`** namespace.
- The following slide illustrates this for the HTML5 **`datalist`** tag, which has no corresponding tag in the JSF HTML tag library.

Plain HTML Tags Instead of HTML Tag Library

-
  ```
  <?xml version='1.0' encoding='UTF-8' ?>
  <!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"
        xmlns:jsf="http://xmlns.jcp.org/jsf">
      <body>
          <form>
              <input list="browsers"/>
              <datalist id="browsers">
                  <option value="Internet Explorer"/>
                  <option jsf:id="abc" value="#{data.browser}"/>
              </datalist>
          </form>
      </body>
  </html>
  ```

JSTL (JSP Standard Tag Library)

Core Tags

- `choose`, an if statement.

```
<c:choose>
  <c:when test="#{condition}">
    The condition was true.
  </c:when>
  <c:otherwise>
    The condition was false.
  </c:otherwise>
 </c:choose>
```

- If the boolean condition specified in the **`test`** attribute is true, the **`when`** block is executed, if not the **`otherwise`** block is executed.

JSTL (JSP Standard Tag Library) Core Tags, Cont'd

- **`forEach`**, a loop statement.

```
<c:forEach var="element" items="#{myList}"
           varStatus="loopCount" >
   Element number #{loopCount.count} is #{element}
</c:forEach>
```

- The **`var`** attribute specifies the name of the variable holding the current element's value. This variable is used when the value shall be displayed.
- The **`items`** attribute refers to the collection that shall be iterated over.
- The **`varStatus`** attribute defines a variable that holds information like the current element's index in the collection.

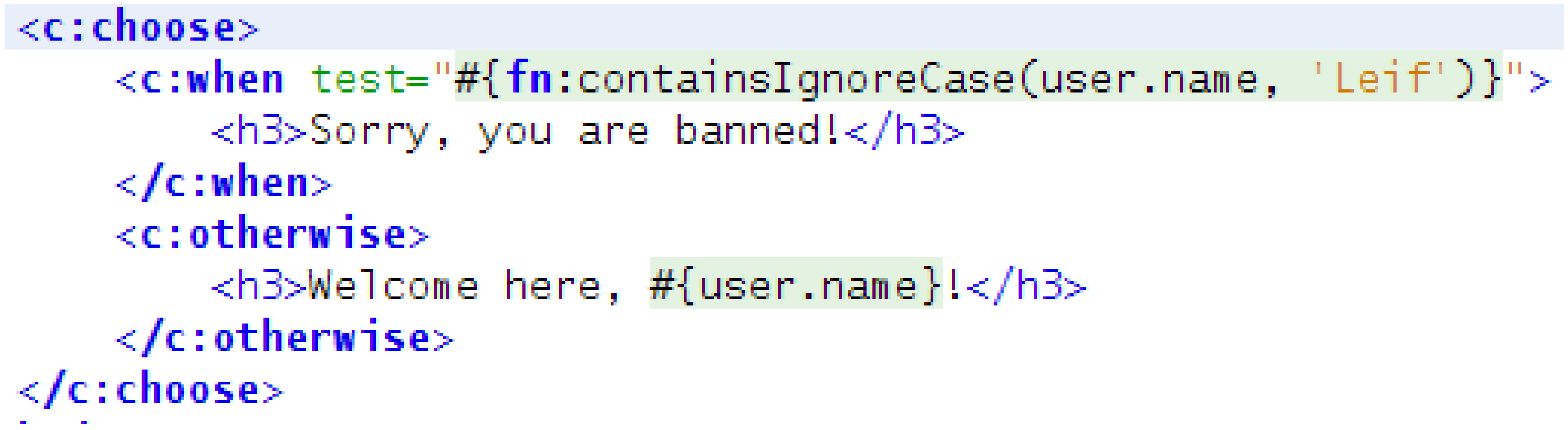
Functions In the JSTL (JSP Standard Tag Library) Functions Library

- Note that these are functions, not tags.
- **`contains(str, substr)`**, returns **`true`** if **`str`** contains **`substr`**.
- **`startsWith(str, substr)`**, returns **`true`** if **`str`** starts with **`substr`**.
- **`length(str)`**, returns the length of **`str.`**
- And many more.

JSTL (JSP Standard Tag Library)

Functions Functions, Cont'd

- Example:



```
<c:choose>
    <c:when test="#{fn:containsIgnoreCase(user.name, 'Leif')}">
        <h3>Sorry, you are banned!</h3>
    </c:when>
    <c:otherwise>
        <h3>Welcome here, #{user.name}!</h3>
    </c:otherwise>
</c:choose>
```

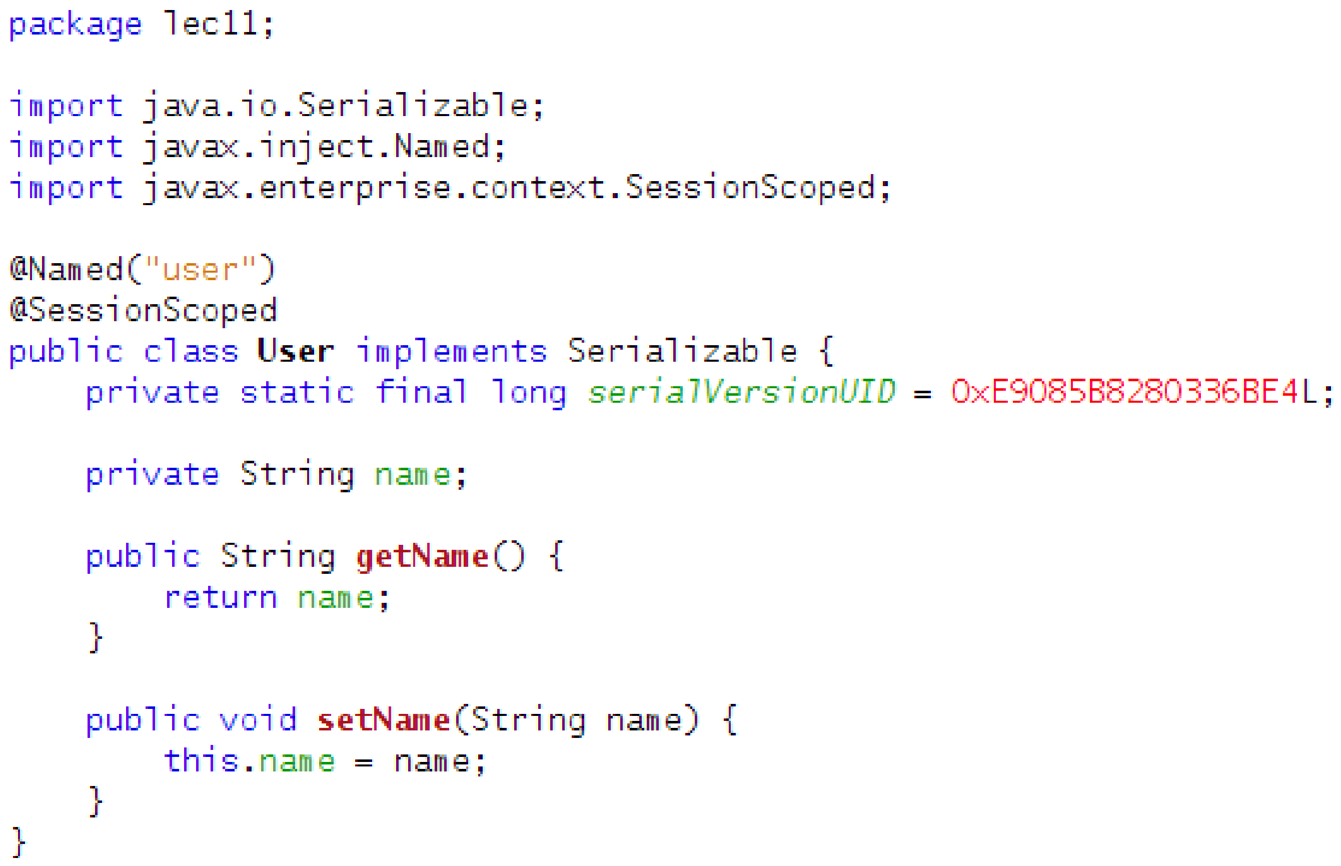
Managed Beans

- Managed beans are plain Java classes.
  - Must have a public no-arg constructor.
  - Must have a scope annotation, e.g. **`@SessionScoped`**, see next slide for more examples.
  - May be annotated **`@Named("myName")`**, where **`myName`** becomes the name of the bean.
- The beans are managed by the *CDI* (*Context and Dependency Injection*) container.
  - Part of Java EE
  - Creates and connects objects according to specifications in annotations.

Managed Beans, Cont'd

- All managed beans have a scope which defines their life time. Some scope annotations are:
  - **`ApplicationScoped`**, the object will exist for the entire application life time.
  - **`SessionScoped`**, the object will be discarded when the current Http session ends.
  - **`ConversationScoped`**, a conversation can be started and stopped manually in the code. If it is not, it has the life time of a Http request. Unlike sessions, conversations are unique for each browser tab and therefore thread safe.
  - **`RequestScoped`**, the object will be discarded when the current Http request ends.

Managed Beans, Example



```
package lec11;

import java.io.Serializable;
import javax.inject.Named;
import javax.enterprise.context.SessionScoped;

@Named("user")
@SessionScoped
public class User implements Serializable {
    private static final long serialVersionUID = 0xE9085B8280336BE4L;

    private String name;

    public String getName() {
        return name;
    }

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

More JSF Features, Flows and Contracts

- There is much more to JSF than what is covered here. Two interesting features are *flows* and *contracts*.
- A flow is a set of views (jsf pages) grouped together. Flows can be used instead of views when specifying navigation rules, outcomes, action targets etc. Typical examples of views would be checkout from a web shop or to register at a community.
- A contract defines a layout template together with style sheets and other resources. It is possible to change the site layout by changing contract. A typical use of contracts would be for themes.

Expression Language

- The *expression language* is used in dynamic expressions in JSF (and JSP) pages.
  - Stateless, variables can not be declared.
  - Statements are written between **`#{`** and **`}`**
  - The result of an EL statement is a string.

Expression Language, Cont'd

- The syntax is **`#{something.somethingElse}`**, where **`something`** is for example one of the following:
  - The name of a managed bean.
  - **`param`**, which is a **`java.util.Map`** containing all HTTP request parameters. If there are more parameters with the same name the first is returned.
  - **`paramValues`**, which is a **`java.util.Map`** containing all HTTP request parameters. No matter how many parameters there are with the specified name a **`java.util.List`** with all of them is returned.

Expression Language, Cont'd

- **`header`**, which is a **`java.util.Map`** containing all HTTP headers. If there are more headers with the same name the first is returned.
- **`headerValues`**, which is a **`java.util.Map`** containing all HTTP headers. No matter how many headers there are with the specified name a **`java.util.List`** with all of them is returned.
- **`cookie`**, which is a **`java.util.Map`** containing all HTTP cookies.

EL, The Operators . and []

- If the operator **`.`** is used (**`#{something.somethingElse}`**) the following must be true.
  - **`something`** is a **`java.util.Map`** or a managed bean.
  - **`somethingElse`** is a key in a **`java.util.Map`** or a property in a managed bean or a method in a managed bean.

EL, The Operators . and [], Cont'd

- If the operator `[]` is used (**`#{something["somethingElse"]}`**) the following must be true.
  - **`something`** is a **`java.util.Map`**, a managed bean, an array or a **`java.util.List`**.
  - If **`somethingElse`** is a string (surrounded by double quotes, "") it must be a key in a **`java.util.Map`**, a property in a managed bean, an index to an array or an index to a **`java.util.List`**.
  - If **`somethingElse`** is not surrounded by double quotes it must be a valid EL statement.

EL Examples

- If these are managed beans:

```
   @Named(“person”)
   public class PersonBean {
       @Inject private DogBean dog;

       public DogBean getDog() {
           return dog;
       }
   }
@Named(“dog”)
   public class DogBean {
       private String name;

       public String getName() {
           return name;
       }
   }
```

- Then it is allowed to write **`#{person.dog.name}`** or **`#{person[dog["name"]]}`**.

EL Examples, Cont'd

- Input from an HTML form:

```
<form>
  Address: <input type="text" name="address">
  Phone1: <input type="text" name="phone">
  Phone2: <input type="text" name="phone">
</form>
```

- Can be read like this:

```
The address is #{param.address}
Phone1 is #{param.phone}
Phone1 is #{paramValues.phone[0]}
Phone2 is #{paramValues.phone[1]}
```

- However, there is seldom any need for this since request parameters are normally handled by managed beans.

The EL Operators

- Remember that JSF/JSP pages are views and thus not the place for a lot of calculations.
- Arithmetic
  - *addition:* **`+`**
  - *subtraction:* **`-`**
  - *multiplication:* **`*`**
  - *division:* **`/`** *or* **`div`**
  - *remainder:* **`%`** *or* **`mod`**
- Logical
  - and: **`&&`** *or* **`and`**
  - or: **`||`** *or* **`or`**
  - not: **`!`** *or* **`not`**

The EL Operators, Cont'd

- Relational
  - *equals:* **`==`** *or* **`eq`**
  - ***not equals: != or ne***
  - *less than:* **`<`** *or* **`lt`**
  - *greater than:* **`>`** *or* **`gt`**
  - *less than or equal to:* **`<=`** *or* **`le`**
  - *greater than or equal to:* **`>=`** *or* **`ge`**
- String Operations
  - *String concatenation:* **`+=`**

EL, Null Values

- Since EL is used for user interfaces it produces the most user friendly output.
- This means that (like HTML) it tries to silently ignore errors.
- Null values does not generate any output at all, no error messages are produced.

EL, Lambda Expressions

- A lambda expression is an anonymous function.
  - **`(x->x+1)`** This is a function that takes one parameter, **`x`**, and increments it by one.
  - Useful for collection manipulation, see next slide.

EL, Collection Handling

- EL collection manipulation is based on *streams* of items from a collection.
- Operations are chained together to form a *pipeline*.
- A pipeline has three parts:
  - Starts with a source stream created from a collection.
  - Continues with functions that operate on items in the stream.
  - Ends with a function that generates a new collection.

EL, Collection Handling

- The following EL statement extracts all categories in a book store: **`inventory.allBooks.stream().map(b>b.category).distinct().toList()`**
  - The source stream, **`inventory.allBooks.stream()`**, contains all **`Book`** objects contained in the list returned by the method **`getAllBooks()`** in the **`inventory`** managed bean.
  - Two functions operate on the stream:
    - **`map(b->b.category)`** maps the stream of **`Book`** objects to a stream of the **`category`** properties in the **`Book`** objects.
    - **`distinct()`** produces a stream containing the elements of the source stream that are distinct, according to **`Object.equals`**.
    - **`toList()`** returns a list containing all elements in the final stream.

Java Contexts and Dependency Injection for the Java EE platform, CDI

`javax.annotation`, **`javax.decorator`**, **`javax.inject`**, **`javax.enterprise.inject`**, **`javax.enterprise.context`**, **`javax.inteceptor`**

Specification:

http://www.jcp.org/en/jsr/summary?id=299

Documentation:

http://docs.jboss.org/weld/reference/1.0.0/en-US/html/

CDI, Contents

Introduction

- The CDI container manages the life cycle of Java objects and their relations and interactions.
  - This relieves the application from the burden of creating and discarding objects at the right time, which might be problematic in web applications.
- Since the CDI container manages all relations, it can intercept calls between objects and do additional handling, for example add interceptors.

Introduction, Cont'd

- More or less all problems regarding object management can be solved by CDI.
- Do not reinvent the wheel, do not forget CDI. Whenever you write code managing object life cycles, check if it can be done easier with CDI.

Bean Characteristics

- Practically all Java objects that have a scope annotation and either a no-arg constructor or a constructor annotated **`@Inject`** are beans, i.e. can be managed by the CDI container.
- A bean has one or more types.
  - The types are the names of all classes and interfaces in the bean's class declaration. If the declaration is **`public class MyBean extends SuperClass implements MyInterface`** the bean's types are **`java.lang.Object`**, **`MyBean`**, **`SuperClass`** and **`MyInterface`**.

Bean Characteristics, Scope

- A bean has a scope.
  - The scope determines when the bean is instantiated and discarded.
  - An object is always created when it is first accessed, independent of its scope.
  - **`@ApplicationScoped`**, the object remains for the application's entire life time.
  - **`@SessionScoped`**, the object will be discarded when the current Http session ends.

Bean Characteristics, Scope, Cont'd

– **`@ConversationScoped`**, a conversation can be started and stopped manually in the code, using a **`javax.enterprise.context.Conversation`** object. If it is not, it has the life time of a Http request. Note that a conversation scoped object is always discarded when the Http session ends. Unlike sessions, conversations are unique for each browser tab and therefore thread safe.

Bean Characteristics, Scope, Cont'd

- **`@RequestScoped`**, the object will be discarded when the current Http request is handled.
- **`@TransactionScoped`**, the object will be discarded when the current transaction is done.
- **`@Dependent`**, the bean has the same scope as the bean into which it is injected. If it is referenced in an EL expression it lives only during that expression. A **`@Dependent`** bean is never shared between other beans.

Bean Characteristics, Qualifier

- A bean has one more more qualifiers.
- A bean type alone might not provide enough information for the container to know which bean to inject. Suppose we have two implementations of for example a **`Payment`** interface: **`CreditCardPayment`** and **`CashPayment`**. Injecting a field of type Payment (**`@Inject private Payment payment`**) does not tell CDI which implementation to inject. In these cases, we must specify which of them to inject. This is done using a qualifier.

Bean Characteristics, Qualifier, Cont'd

- A qualifier is a user-defined annotation that is itself annotated **`@Qualifer`**:
  - The qualifier declaration:

```
@Qualifier
@Target({TYPE, METHOD, PARAMETER, FIELD})
@Retention(RUNTIME)
public @interface CreditCard {}
```

  - The declaration of a class using the qualifier.

```
@CreditCard
public class CraditCardPayment implements Payment {
    //Payment methods.
}
```

  - In some bean where the qualified class is injected:

```
@Inject @CreditCard private Payment payment
```

Bean Characteristics, EL Name

- If the bean shall be accessed from a JSF page it must have a Expression Language (EL) name.
  - The EL name can be left out if the bean is not accessed from a JSF page.
- The EL name is specified with the **`@Named`** annotation, i.e.

```
@Named("ELBean")
public class MyBean {
```

Injection Points

- The **`@Inject`** annotation can be placed at a field, a constructor or a set method (called *initializer method*).
- The CDI container guarantees that if a matching bean exists it will be passed to the injection point.

Which bean instance can be injected?

- A bean instance that is to be passed to another bean's injection point must:
  - Be of the type specified at the injection point.
  - Match all qualifiers at the injection point (if there are any).
- There will be an error at deploy time (not execution time) if there is no bean matching an **`@Inject`** annotation.

The Injected Bean's Scope

- The scope of an injected bean depends on the scope of the *injected* bean, not the scope of the bean *into which it is* injected.
- To make sure that the injected instance is not shared with other beans and that it has the same scope as the bean into which it is injected, add **`@New`** to the injection point:

```
@Inject @New private MyBean myBean;
```

Interceptors

- An interceptor is a method that is called before and/or after another method. The call of the interceptor is not performed in the application code, but by the container.
- In CDI, an interceptor definition is an annotation:

```
@InterceptorBinding
@Target({METHOD, TYPE})
@Retention(RUNTIME)public @interface Logging {}
```

Interceptors, Cont'd

- The interceptor implementation is an ordinary class, annotated **`@Interceptor`** and with the interceptor definition:

```
@Logging
@Interceptor
public class Logger implements Serializable {
    @AroundInvoke
    public Object logInvocation(InvocationContext ctx)
                                                throws Exception {
        //Print a log message.
    }
}
```

- The **`@AroundInvoke`** annotation means that the method is called before and after any method annotated with the interceptor annotation (i.e. **`@Logging`**).

Interceptors, Cont'd

- Annotating a method with the interceptor annotation causes a call to the interceptor before and after calls to the annotated method.
- Annotating a class with the interceptor annotation causes a call to the interceptor before and after calls to any method in that class:

```
@Logging
public class User implements Serializable {
      ...
}
```

Interceptors, Cont'd

- Interceptors must be enabled in the **`beans.xml`** configuration file:

```
<interceptors>
    <class>util.Logger</class>
</interceptors>
```

Life Cycle Callbacks

- A method annotated **`@PostConstruct`** is executed after dependency injection is done but before the class is put into service.
  - There can only be one such method per class. The method must be void and take no arguments and may not throw a checked exception. All accessibilities are possible.
- The **`@PreDestroy`** annotation is used on methods as a callback notification to signal that the instance is in the process of being removed by the container.
  - The same rules apply as for post construct methods.

More CDI Features

- CDI has many more features besides those covered here. Some examples follows.
- *Alternatives*, a possibility to chose bean implementation at deployment time. Useful for example for testing.
- *Events*, resembles the Observer design pattern but provides additional functionality.
- *Stereotypes*, a way to avoid duplicated code by placing properties shared by bean classes in one common location.
- *Specialization*, allows one bean type to replace another bean type.
- *Decorators*, objects that intercepts method calls to beans and extend their functionality. This is the same design pattern as is used by the streams in the **`java.io`** package.

Bean Validation

`javax.validation.*`

Specification:

http://www.jcp.org/en/jsr/summary?id=303

Documentation Summary:

http://www.oracle.com/technetwork/articles/javaee/javaee6overview-141808.html#beanval

Bean Validation, Contents

- Introduction
- Built-In Constraints
- Custom Constraints

Introduction, the Problem

- Validating data is a common task that occurs throughout an application, for example, in the presentation layer using JSF tags.
- The same validations as in the presentation are probably required in methods in the model and integration layers, since those layers can not rely on validations done in other layers.
- This introduces the risk of coding the same validation logic in multiple layers, which means duplicated code.

Introduction, the Solution

- Bean Validation removes this code duplication by offering a standard framework for validation.
- The same set of validators can be shared by all the layers of an application.
- The validations are specified as annotations or in XML configuration files.
- Fields, method parameters and method return values can be validated.

Built-In Constraints

- The built-in constraints are found in the package **`javax.validation.constraints`**. Below is an example that specifies that the length of the string must be at least 1 character. It also shows one possible way to specify an error message.

```
@Size(min=1, message="Please enter a value")
private String street;
```

  – Note that a string in an Html field is never null so the **`@NotNull`** constraint can not be used here.

Built-In Constraints, Cont'd

- We can not validate JSF pages using JSF validation tags if exactly the same validation shall be used in all layers.
  - Instead, bean validations must be placed in the managed beans, as on the previous slide.

Custom Constraints

- A custom constraint is an annotation with the **`@Constraint`** annotation:

```
@Constraint(validatedBy = ZipCodeValidator.class)
@Documented
@Target({ANNOTATION_TYPE, METHOD, FIELD})
@Retention(RUNTIME)
public @interface ZipCode {
    String message() default "Not a valid zip code";
    java.lang.Class<?>[] groups() default {};
    java.lang.Class<? extends Payload>[] payload() default {};
}
```

Custom Constraints, Cont'd

- The validation of the custom constraint is performed by the class specified in the **`@Context`** annotation (see previous slide):

```
public class ZipCodeValidator implements
ConstraintValidator<ZipCode, String> {
    @Override
    public void initialize(ZipCode constraintAnnotation) {}

    @Override
    public boolean isValid(String value,
                           ConstraintValidatorContext context) {
        try {
            Integer.parseInt(value);
        } catch(NumberFormatException nfe) {
            return false;
        }
        return value.length() == 5;
    }
}
```

Custom Constraints, Cont'd

- The custom constraint is used just like built-in constraints:

```
@ZipCode
private String zipCode;
```