# Part V
# Contexts and Dependency Injection for the Java EE Platform

Part V explores Contexts and Dependency Injection for the Java EE Platform.

# This part contains the following chapters:

- Chapter 28, Introduction to Contexts and Dependency Injection for the Java EE Platform
- Chapter 29, Running the Basic Contexts and Dependency Injection Examples
- Chapter 30, Contexts and Dependency Injection for the Java EE Platform: Advanced Topics
- Chapter 31, Running the Advanced Contexts and Dependency Injection Examples

# Introduction to Contexts and Dependency Injection for the Java EE Platform

Contexts and Dependency Injection (CDI) for the Java EE platform is one of several Java EE 6 features that help to knit together the web tier and the transactional tier of the Java EE platform.

CDI is a set of services that, used together, make it easy for developers to use enterprise beans along with JavaServer Faces technology in web applications.

Designed for use with stateful objects, CDI also has many broader uses, allowing developers a great deal of flexibility to integrate various kinds of components in a loosely coupled but typesafe way.

CDI is specified by JSR 299, formerly known as Web Beans.

Related specifications that CDI uses include the following:

- JSR 330, Dependency Injection for Java
- The Managed Beans specification, which is an offshoot of the Java EE 6 platform specification (JSR 316)

# The following topics are addressed here:

- Overview of CDI
- About Beans
- About Managed Beans
- Beans as Injectable Objects
- Using Qualifiers
- Injecting Beans
- Using Scopes
- Giving Beans EL Names
- Adding Setter and Getter Methods
- Using a Managed Bean in a Facelets Page

- Injecting Objects by Using Producer Methods
- Configuring a CDI Application
- Further Information about CDI

# Overview of CDI

The most fundamental services provided by CDI are as follows:

- **Contexts:** The ability to bind the lifecycle and interactions of stateful components to well-defined but extensible lifecycle contexts

- **Dependency injection: The ability to inject components into an application in a typesafe way, including the ability to choose at deployment time which implementation of a particular interface to inject**

**In addition, CDI provides the following services:**

- Integration with the Expression Language (EL), which allows any component to be used directly within a JavaServer Faces page or a JavaServer Pages page
- The ability to decorate injected components
- The ability to associate interceptors with components using typesafe interceptor bindings
-

- An event-notification model

- A web conversation scope in addition to the three standard scopes (request, session, and application) defined by the Java Servlet specification

- A complete Service Provider Interface (SPI) that allows third-party frameworks to integrate cleanly in the Java EE 6 environment

# A major theme of CDI is loose coupling.

## CDI does the following:

- Decouples the server and the client by means of well-defined types and qualifiers, so that the server implementation may vary

- Decouples the lifecycles of collaborating components by doing the following:

- Making components contextual, with automatic lifecycle management
- Allowing stateful components to interact like services, purely by message passing

- Completely decouples message producers from consumers, by means of events

- Decouples orthogonal concerns by means of Java EE interceptors

# Along with loose coupling, CDI provides strong typing by

- Eliminating lookup using string-based names for wiring and correlations, so that the compiler will detect typing errors

- Allowing the use of declarative Java annotations to specify everything, largely eliminating the need for XML deployment descriptors, and making it easy to provide tools that introspect the code and understand the dependency structure at development time

# About Beans

CDI redefines the concept of a bean beyond its use in other Java technologies, such as the JavaBeans and Enterprise JavaBeans (EJB) technologies.

In CDI, a bean is a source of contextual objects that define application state and/or logic.

A Java EE component is a bean if the lifecycle of its instances may be managed by the container according to the lifecycle context model defined in the CDI specification.

More specifically, a bean has the following attributes:

- A (nonempty) set of bean types
- A (nonempty) set of qualifiers (see Using Qualifiers)

- A scope (see Using Scopes)
- Optionally, a bean EL name (see Giving Beans EL Names)
- A set of interceptor bindings
- A bean implementation

A bean type defines a client-visible type of the bean.

# Almost any Java type may be a bean type of a bean.

- A bean type may be an interface, a concrete class, or an abstract class and may be declared final or have final methods.

- A bean type may be a parameterized type with type parameters and type variables.

- A bean type may be an array type.

Two array types are considered identical only if the element type is identical.

- A bean type may be a primitive type.

# Primitive types are considered to be identical to their corresponding wrapper types in `java.lang`.

- A **bean** type may be a raw type.

# About Managed Beans

A managed bean is implemented by a Java class, which is called its bean class.

A top-level Java class is a managed bean if it is defined to be a managed bean by any other Java EE technology specification, such as the JavaServer Faces technology specification, or if it meets all the following conditions:

- It is not a nonstatic inner class.
- It is a concrete class or is annotated `@Decorator`.
- It is not annotated with an EJB component-defining annotation or declared as an EJB bean class in `ejb-jar.xml`.

- It has an appropriate constructor.

That is, one of the following is the case:

- The class has a constructor with no parameters.
- The class declares a constructor annotated `@Inject`.

No special declaration, such as an annotation, is required to define a managed bean.

# Beans as Injectable Objects

The concept of injection has been part of Java technology for some time.

Since the Java EE 5 platform was introduced, annotations have made it possible to inject resources and some other kinds of objects into container-managed objects.

CDI makes it possible to inject more kinds of objects and to inject them into objects that are not container-managed.

The following kinds of objects can be injected:

- (Almost) any Java class
- Session beans
- Java EE resources: data sources, Java Message Service topics, queues, connection factories, and the like

- Persistence contexts (JPA `EntityManager` objects)
- Producer fields
- Objects returned by producer methods
- Web service references
- Remote enterprise bean references

For example, suppose that you create a simple Java class with a method that returns a string:

```
package greetings;
public class Greeting {
public String greet(String name)
{ return "Hello, " + name + "."; }
}
```

This class becomes a bean that you can then inject into another class.

This bean is not exposed to the EL in this form.

# Giving Beans EL Names explains how you can make a bean accessible to the EL.

# Using Qualifiers

You can use qualifiers to provide various implementations of a particular bean type.

A qualifier is an annotation that you apply to a bean.

A qualifier type is a Java annotation defined as `@Target({METHOD, FIELD, PARAMETER, TYPE})` and `@Retention(RUNTIME)`.

For example, you could declare an `@Informal` qualifier type and apply it to another class that extends the `Greeting` class.

To declare this qualifier type, you would use the following code:

```
package greetings;
import static java.lang.annotation.
ElementType.FIELD;
import static java.lang.annotation.
ElementType.METHOD;
import static java.lang.annotation.
ElementType.PARAMETER;
import static java.lang.annotation.
ElementType.TYPE;
import static java.lang.annotation.
RetentionPolicy.RUNTIME;
```

```
import
java.lang.annotation.Retention;
import java.lang.annotation.Target;
import javax.inject.Qualifier;
@Qualifier
@Retention(RUNTIME)
@Target
({TYPE, METHOD, FIELD, PARAMETER})
public @interface Informal {}
```

# You can then define a bean class that extends the Greeting class and uses this qualifier:

```
package greetings;
@Informal
public class InformalGreeting
extends Greeting {
public String greet(String name)
{ return "Hi, " + name + "!"; }
}
```

Both implementations of the bean can now be used in the application.

If you define a bean with no qualifier, the bean automatically has the qualifier `@Default`.

The unannotated `Greeting` class could be declared as follows:

```
package greetings;
import
javax.enterprise.inject.Default;
@Default
public class Greeting {
public String greet(String name)
{ return "Hello, " + name + "."; }
}
```

# Injecting Beans

In order to use the beans you create, you inject them into yet another bean that can then be used by an application, such as a JavaServer Faces application.

For example, you might create a bean called `Printer` into which you would inject one of the `Greeting` beans:

```
import javax.inject.Inject;
public class Printer {
@Inject Greeting greeting;
...
```

This code injects the `@Default Greeting` implementation into the bean.

The following code injects the `@Informal` implementation:

```
import javax.inject.Inject;
public class Printer {
@Inject
@Informal Greeting greeting;
...
```

More is needed for the complete picture of this bean.

Its use of scope needs to be understood.

In addition, for a JavaServer Faces application, the bean needs to be accessible through the EL.

# Using Scopes

For a web application to use a bean that injects another bean class, the bean needs to be able to hold state over the duration of the user's interaction with the application.

The way to define this state is to give the bean a scope.

You can give an object any of the scopes described in Table 28-1, depending on how you are using it.

Table 28-1 Scopes

| Scope | Annotation | Duration |
|---|---|---|
| Request | `@RequestScoped` | A user's interaction with a web application in a single HTTP request. |
| Session | `@SessionScoped` | A user's interaction with a web application across multiple HTTP requests. |
| Application | `@ApplicationScoped` | Shared state across all users' interactions with a web application. |
| Dependent | `@Dependent` | The default scope if none is specified; it means that an object exists to serve exactly one client (bean) and has the same lifecycle as that client (bean). |

| | | |
|---|---|---|
| Conversation | `@ConversationScoped` | A user's interaction with a JavaServer Faces application, within explicit developer-controlled boundaries that extend the scope across multiple invocations of the JavaServer Faces lifecycle.<br><br>All long-running conversations are scoped to a particular HTTP servlet session and may not cross session boundaries. |

**The first three scopes are defined by both JSR 299 and the JavaServer Faces API.**

The last two are defined by JSR 299.

You can also define and implement custom scopes, but that is an advanced topic.

Custom scopes are likely to be used by those who implement and extend the CDI specification.

A scope gives an object a well-defined lifecycle context.

A scoped object can be automatically created when it is needed and automatically destroyed when the context in which it was created ends.

Moreover, its state is automatically shared by any clients that execute in the same context.

Java EE components, such as servlets and enterprise beans, and JavaBeans components do not by definition have a well-defined scope.

# These components are one of the following:

- Singletons, such as Enterprise JavaBeans singleton beans, whose state is shared among all clients

- Stateless objects, such as servlets and stateless session beans, which do not contain client-visible state

- Objects that must be explicitly created and destroyed by their client, such as JavaBeans components and stateful session beans, whose state is shared by explicit reference passing between clients

If, however, you create a Java EE component that is a managed bean, it becomes a scoped object, which exists in a well-defined lifecycle context.

# The web application for the Printer bean will use a simple request and response mechanism, so the managed bean can be annotated as follows:

```
import javax.inject.Inject;
import javax.enterprise.context.
RequestScoped;
@RequestScoped
public class Printer {

```

```
@Inject
@Informal Greeting greeting;
...
```

Beans that use session, application, or conversation scope must be serializable, but beans that use request scope do not have to be serializable.

# Giving Beans EL Names

To make a bean accessible through the EL, use the @Named built-in qualifier:

```
import javax.inject.Inject;
import javax.enterprise.context.
RequestScoped;
import javax.inject.Named;
@Named
```

```
@RequestScoped
public class Printer {
@Inject
@Informal Greeting greeting;
...
```

The `@Named` qualifier allows you to access the bean by using the bean name, with the first letter in lowercase.

For example, a Facelets page would refer to the bean as `printer`.

You can specify an argument to the `@Named` qualifier to use a nondefault name:

```
@Named("MyPrinter")
```

With this annotation, the Facelets page would refer to the bean as `MyPrinter`.

# Adding Setter and Getter Methods

To make the state of the managed bean accessible, you need to add setter and getter methods for that state.

The `createSalutation` method calls the bean's `greet` method, and the `getSalutation` method retrieves the result.

Once the setter and getter methods have been added, the bean is complete.

The final code looks like this:

```
package greetings;
import javax.inject.Inject;
import javax.enterprise.context.
RequestScoped;
import javax.inject.Named;
```

```
@Named
@RequestScoped
public class Printer {
@Inject
@Informal Greeting greeting;
private String name;
private String salutation;
public void createSalutation() {
this.salutation =
greeting.greet(name);
}
```

```
public String getSalutation()
{ return salutation; }
public String setName(String name)
{ this.name = name; }
public String getName()
{ return name; }
}
```

# Using a Managed Bean in a Facelets Page

To use the managed bean in a Facelets page, you typically create a form that uses user interface elements to call its methods and display their results.

This example provides a button that asks the user to type a name, retrieves the salutation, and then displays the text in a paragraph below the button:

```
<h:form id="greetme">
<p>
<h:outputLabel
value="Enter your name: "
for="name"/>
```

```
<h:inputText id="name"
value="#{printer.name}"/></p><p>
<h:commandButton value="Say Hello"
action=
"#{printer.createSalutation}"/></p>
<p>
<h:outputText
value="#{printer.salutation}"/>
</p>
</h:form>
```

# Injecting Objects by Using Producer Methods

Producer methods provide a way to inject objects that are not beans, objects whose values may vary at runtime, and objects that require custom initialization.

For example, if you want to initialize a numeric value defined by a qualifier named @MaxNumber, you can define the value in a managed bean and then define a producer method, getMaxNumber, for it:

```
private int maxNumber = 100;...
@Produces
@MaxNumber int getMaxNumber()
{  return maxNumber;  }
```

When you inject the object in another managed bean, the container automatically invokes the producer method, initializing the value to 100:

```
@Inject
@MaxNumber private int maxNumber;
```

If the value can vary at runtime, the process is slightly different.

# For example, the following code defines a producer method that generates a random number defined by a qualifier called @Random:

```
private java.util.Random random =
new java.util.Random
( System.currentTimeMillis() );
java.util.Random getRandom()
{ return random; }
```

```
@Produces
@Random int next() {
return
getRandom().nextInt(maxNumber);
}
```

When you inject this object in another managed bean, you declare a contextual instance of the object:

```
@Inject
@Random
Instance<Integer> randomInt;
```

You then call the `get` method of the `Instance`:

```
this.number = randomInt.get();
```

# Configuring a CDI Application

An application that uses CDI must have a file named `beans.xml`.

The file can be completely empty (it has content only in certain limited situations), but it must be present.

For a web application, the `beans.xml` file can be in either the `WEB-INF` directory or the `WEB-INF/classes/META-INF` directory.

For EJB modules or JAR files, the `beans.xml` file must be in the `META-INF` directory.

# Further Information about CDI

For more information about CDI for the Java EE platform, see

- Contexts and Dependency Injection for the Java EE platform specification:

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

- An introduction to Contexts and Dependency Injection for the Java EE platform:

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

- Dependency Injection for Java specification:

http://jcp.org/en/jsr/detail?id=330