Using Java EE Interceptors

This chapter discusses how to create interceptor classes and methods that interpose on method invocations or lifecycle events on a target class.

The following topics are addressed here:

- Overview of Interceptors
- Using Interceptors
- The `interceptor` Example Application

Overview of Interceptors

Interceptors are used in conjunction with Java EE managed classes to allow developers to invoke interceptor methods in conjunction with method invocations or lifecycle events on an associated target class.

Common uses of interceptors are logging, auditing, or profiling.

Interceptors can be defined within a target class as an interceptor method, or in an associated class called an interceptor class.

Interceptor classes contain methods that are invoked in conjunction with the methods or lifecycle events of the target class.

Interceptor classes and methods are defined using metadata annotations, or in the deployment descriptor of the application containing the interceptors and target classes.

Note - Applications that use the deployment descriptor to define interceptors are not portable across Java EE servers.

Interceptor methods within the target class or in an interceptor class are annotated with one of the metadata annotations defined in Table 48-1.

Table 48-1 Interceptor Metadata Annotations

Interceptor Metadata Annotation	Description
`javax.interceptor.AroundInvoke`	Designates the method as an interceptor method.
`javax.interceptor.AroundTimeout`	Designates the method as a timeout interceptor, for interposing on timeout methods for enterprise bean timers.
`javax.annotation.PostConstruct`	Designates the method as an interceptor method for post-construct lifecycle events.
`javax.annotation.PreDestroy`	Designates the method as an interceptor method for pre—destroy lifecycle events.

Interceptor Classes

Interceptor classes may be designated with the optional `javax.interceptor.Interceptor` annotation, but interceptor classes aren't required to be so annotated.

Interceptor classes must have a public, no-argument constructor.

The target class can have any number of interceptor classes associated with it.

The order in which the interceptor classes are invoked is determined by the order in which the interceptor classes are defined in the `javax.interceptor.Interceptors` annotation.

This order can be overridden in the deployment descriptor.

Interceptor classes may be targets of dependency injection.

Dependency injection occurs when the interceptor class instance is created, using the naming context of the associated target class, and before any `@PostConstruct` callbacks are invoked.

Interceptor Lifecycle

Interceptor classes have the same lifecycle as their associated target class.

When a target class instance is created, an interceptor class instance is also created for each declared interceptor class in the target class.

That is, if the target class declares multiple interceptor classes, an instance of each class is created when the target class instance is created.

The target class instance and all interceptor class instances are fully instantiated before any `@PostConstruct` callbacks are invoked, and any `@PreDestroy` callbacks are invoked before the target class and interceptor class instances are destroyed.

Interceptors and Contexts and Dependency Injection for the Java EE

Contexts and Dependency Injection for the Java EE Platform (CDI) builds on the basic functionality of Java EE interceptors.

For information on CDI interceptors, including a discussion of interceptor binding types, see Using Interceptors.

Using Interceptors

Interceptors are defined using one of the interceptor metadata annotations listed in Table 48-1 within the target class, or in a separate interceptor class.

The following code declares an `@AroundTimeout` interceptor method within a target class.

```
@Stateless
public class TimerBean {...
@Schedule(minute="*/1", hour="*")
public void automaticTimerMethod()
{ ... }
@AroundTimeout
public void timeoutInterceptorMethod
(InvocationContext ctx) { ... }
...
}
```

If interceptor classes are used, use the `javax.interceptor.Interceptors` annotation to declare one or more interceptors at the class or method level of the target class.

The following code declares interceptors at the class level.

```
@Stateless
@Interceptors({
PrimaryInterceptor.class,
SecondaryInterceptor.class
})
public class OrderBean { ... }
```

The following code declares a method-level interceptor class.

```
@Stateless
public class OrderBean {...
@Interceptors
(OrderInterceptor.class)
public void placeOrder(Order order)
{ ... }
...
}
```

Intercepting Method Invocations

The `@AroundInvoke` annotation is used to designate interceptor methods for managed object methods.

Only one around-invoke interceptor method per class is allowed.

Around-invoke interceptor methods have the following form:

```
@AroundInvoke
<visibility>
 Object
<Method name>
(InvocationContext)
throws Exception { ... }
```

For example:

```
@AroundInvoke
public void interceptOrder
(InvocationContext ctx) { ... }
```

Around-invoke interceptor methods can have public, private, protected, or package-level access, and must not be declared static or final.

Around-invoke interceptors can call any component or resource callable by the target method on which it interposes, have the same security and transaction context as the target method, and run in the same Java virtual machine call-stack as the target method.

Around-invoke interceptors can throw any exception allowed by the throws clause of the target method.

They may catch and suppress exceptions, and then recover by calling the `InvocationContext.proceed` method.

Using Multiple Method Interceptors

Use the `@Interceptors` annotation to declare multiple interceptors for a target method or class.

```
@Interceptors({
PrimaryInterceptor.class,
SecondaryInterceptor.class,
LastInterceptor.class})
public void updateInfo(String info)
{ ... }
```

The order of the interceptors in the `@Interceptors` annotation is the order in which the interceptors are invoked.

Multiple interceptors may also be defined in the deployment descriptor.

The order of the interceptors in the deployment descriptor is the order in which the interceptors will be invoked.

```
...
<interceptor-binding>
<target-name>
myapp.OrderBean
```

```
</target-name>
<interceptor-class>
myapp.PrimaryInterceptor.class
</interceptor-class>
<interceptor-class>
myapp.SecondaryInterceptor.class
</interceptor-class>
<interceptor-class>
myapp.LastInterceptor.class
</interceptor-class>
```

```
<method-name>
updateInfo
</method-name>
</interceptor-binding>
...
```

To explicitly pass control to the next interceptor in the chain, call the `InvocationContext.proceed` method.

Sharing Data Across Interceptors

The same `InvocationContext` instance is passed as an input parameter to each interceptor method in the interceptor chain for a particular target method.

The `InvocationContext` instance's `contextData` property is used to pass data across interceptor methods.
The `contextData` property is a

`java.util.Map<String, Object>` object.

Data stored in `contextData` is accessible to interceptor methods further down the interceptor chain.

The data stored in `contextData` is not sharable across separate target class method invocations.
That is, a different `InvocationContext`

object is created for each invocation of the method in the target class.

Accessing Target Method Parameters From an Interceptor Class

The `InvocationContext` instance passed to each around-invoke method may be used to access and modify the parameters of the target method.

The `parameters` property of `InvocationContext` is an array of `Object` instances that corresponds to the parameter order of the target method.

For example, for the following target method:

```
@Interceptors
(PrimaryInterceptor.class)
public void updateInfo
(String firstName, String lastName,
Date date) { ... }
```

The `parameters` property, in the `InvocationContext` instance passed to the around-invoke interceptor method in `PrimaryInterceptor`,

is an `Object` array containing a `String` object (`firstName`), a `String` object (`lastName`), and a `Date` object (`date`).

The parameters can be accessed and modified using the `InvocationContext.getParameters` and `InvocationContext.setParameters` methods, respectively.

Intercepting Lifecycle Callback Events

Interceptors for lifecycle callback events (post-create and pre-destroy) may be defined in the target class or in interceptor classes.

The `@PostCreate` annotation is used to designate a method as a post-create lifecycle event interceptor.

The `@PreDestroy` annotation is used to designate a method as a pre-destroy lifecycle event interceptor.

Lifecycle event interceptors defined within the target class have the following form:

```
void <Method name>() { ... }
```

For example:

```
@PostCreate
void initialize() { ... }
```

Lifecycle event interceptors defined in an interceptor class have the following form:

```
void <Method name>
(InvocationContext) { ... }
```

For example:

```
@PreDestroy
void cleanup(InvocationContext ctx)
{ ... }
```

Lifecycle interceptor methods can have public, private, protected, or package-level access, and must not be declared static or final.

Lifecycle interceptor methods are called in an unspecified security and transaction context.

That is, portable Java EE applications should not assume the lifecycle event interceptor method has access to a security or transaction context.

Only one interceptor method for each lifecycle event (post-create and pre-destroy) is allowed per class.

Using Multiple Lifecycle Callback Interceptors

Multiple lifecycle interceptors may be defined for a target class by specifying the interceptor classes in the `@Interceptors` annotation:

```
@Interceptors({
PrimaryInterceptor.class,
SecondaryInterceptor.class,
LastInterceptor.class})
@Stateless public class OrderBean{...}
```

The order in which the interceptor classes are listed in the `@Interceptors` annotation defines the order in which the interceptors are invoked.

Data stored in the `contextData` property of `InvocationContext` is not sharable across different lifecycle events.

Intercepting Timeout Events

Interceptors for EJB timer service timeout methods may be defined using the `@AroundTimeout` annotation on methods in the target class or in an interceptor class.

Only one `@AroundTimeout` method per class is allowed.

Timeout interceptors have the following form:

```
Object <Method name>
(InvocationContext)
throws Exception { ... }
```

For example:

```
@AroundTimeout
protected void
timeoutInterceptorMethod
(InvocationContext ctx) { ... }
```

Timeout interceptor methods can have public, private, protected, or package-level access, and must not be declared static or final.

Timeout interceptors can call any component or resource callable by the target timeout method, and are invoked in the same transaction and security context as the target method.

Timeout interceptors may access the timer object associated with the target timeout method through the `InvocationContext` instance's `getTimer` method.

Using Multiple Timeout Interceptors

Multiple timeout interceptors may be defined for a given target class by specifying the interceptor classes containing `@AroundTimeout` interceptor methods in an `@Interceptors` annotation at the class level.

If a target class specifies timeout interceptors in an interceptor class, and also has a `@AroundTimeout` interceptor method within the target class itself, the timeout interceptors in the interceptor classes are called first, then the timeout interceptors defined in the target class.

For example, in the following example, assume that the `PrimaryInterceptor` and `SecondaryInterceptor` class have timeout interceptor methods.

```
@Interceptors({
PrimaryInterceptor.class,
SecondaryInterceptor.class})
@Stateful
public class OrderBean {
...
@AroundTimeout
private void last
(InvocationContext ctx) { ... }
...
}
```

The timeout interceptor in `PrimaryInterceptor` will be called first, then the timeout interceptor in `SecondaryInterceptor`, and finally the `last` method defined in the target class.

The `interceptor` Example Application

The `interceptor` example demonstrates how to use an interceptor class, containing an `@AroundInvoke` interceptor method, with a stateless session bean.

The `HelloBean` stateless session bean is a simple enterprise bean with a two business methods, `getName` and `setName` to retrieve and modify a string.

The `setName` business method has an `@Interceptors` annotation that specifies an interceptor class, `HelloInterceptor`, for that method.

```
@Interceptors
(HelloInterceptor.class)
public void setName(String name)
{ this.name = name; }
```

The `HelloInterceptor` class defines an `@AroundInvoke` interceptor method, `modifyGreeting`, that converts the string passed to `HelloBean.setName` to lower case.

```
@AroundInvoke
public Object modifyGreeting
(InvocationContext ctx)
throws Exception {
Object[] parameters =
ctx.getParameters();
String param =
(String) parameters[0];
param = param.toLowerCase();
parameters[0] = param;
ctx.setParameters(parameters);
```

```
try {
return ctx.proceed();
}catch (Exception e) {
logger.warning("Error calling
ctx.proceed in modifyGreeting()");
return null;
}
}
```

The parameters to `HelloBean.setName` are retrieved and stored in an `Object` array by calling the `InvocationContext.getParameters` method.

Because `setName` only has one parameter, it is the first and only element in the array.

The string is set to lower case, and stored in the `parameters` array, then passed to `InvocationContext.setParameters`.

To return control to the session bean, `InvocationContext.proceed` is called.

The user interface of `interceptor` is a JavaServer Faces web application that consists of two Facelets views, `index.xhtml`, which has a form for entering the name, and `response.xhtml`, which displays the final name.

Running the `interceptor` Example Application in NetBeans IDE

1. From the File menu, choose Open Project.

2. In the Open Project dialog, navigate to *tut-install*`/examples/ejb/`.

3. Select the `interceptor` folder and click Open Project.

4. In the Projects tab, right-click the `interceptor` project and select Run.

This will compile, deploy, and run the `interceptor` example, opening a web browser page to `http://localhost:8080/interceptor/`.

5. Type a name into the form and select Submit.

The name will be converted to lowercase by the method interceptor defined in the `HelloInterceptor` class.

Running the `interceptor` Example Applications Using Ant

1. Go to the following directory:

   `tut-install``/examples/ejb/interceptor/`

2. To compile the source files and package the application, use the following command: `ant`

This command calls the `default` target, which builds and packages the application into a WAR file, `interceptor.war`, located in the `dist` directory.

3. To deploy and run the application using Ant, use the following command:

```
ant run
```

This command deploys and runs the `interceptor` example, opening a web browser page to `http://localhost:8080/interceptor/`.

4. Type a name into the form and select Submit.

The name will be converted to lowercase by the method interceptor defined in the `HelloInterceptor` class.