Chapter 1 Interceptors

Interceptors are used to interpose on the method invocations and lifecycle events that occur on the instances of an associated target class.

1.1 Overview

An interceptor method may be defined on a target class or on an interceptor class associated with the target class. An interceptor class is a class (distinct from the target class) whose methods are invoked in response to method invocations and/or lifecycle events on the target class.

Any number of interceptor classes may be associated with a target class.

An interceptor class must have a public no-arg constructor.

Interceptor methods and interceptor classes may be defined for a class by means of metadata annotations or the deployment descriptor.

When annotations are used, one or more interceptor classes are denoted using the `Interceptors` annotation on the target class itself and/or on its methods. The `Interceptors` annotation can be applied to a method of the target class or of a superclass. If multiple interceptors are defined, the order in which they are invoked is determined by the order in which they are specified in the `Interceptors` annotation.

The `Interceptor` annotation may be used to explicitly designate a class as an interceptor class. Support for this annotation is not required. [XXX Why shouldn't support be required? In general, I think this spec needs some kind of positioning as to how it might be used as a building block by other specifications, e.g., along the lines of the Managed Beans spec.]

The `InterceptorBinding` annotation specifies that an annotation type is an interceptor binding type. Interceptor binding types are used to associated interceptors with interception targets. See [2] for further discussion. Support for this annotation is not required.[XXX Why shouldn't support be required?] [XXX When both `Interceptors` and interceptor binding annotations are specified, ordering rules need to be worked out for the more complex cases.]

The deployment descriptor may be used as an alternative to specify the invocation order of interceptors or to override the order specified in metadata annotations. An interceptor implementation is not required to support the use of a deployment descriptor to specify interceptor metadata.

Default interceptors may be defined using the deployment descriptor.

It is possible to carry state across multiple interceptor method invocations for a single method invocation or lifecycle callback event in the context data of the `InvocationContext` object. The `InvocationContext` object also provides metadata that enables interceptor methods to control the behavior of the interceptor invocation chain, including whether the next method in the chain is invoked and the values of its parameters and result.

An interceptor class shares the enterprise naming context of its associated target class. Annotations and/or XML deployment descriptor elements for dependency injection or for direct JNDI lookup refer to this shared naming context.

1.2 Interceptor Life Cycle

The lifecycle of an interceptor instance is the same as that of the target class instance with which it is associated. When the target instance is created, a corresponding interceptor instance is created for each associated interceptor class. These interceptor instances are destroyed when the target instance is removed.

Both the interceptor instance and the target instance are created before any `PostConstruct` callbacks are invoked. Any `PreDestroy` callbacks are invoked before the destruction of either the target instance or interceptor instance.

An interceptor instance may hold state. An interceptor instance may be the target of dependency injection. Dependency injection is performed when the interceptor instance is created, using the naming context of the associated target class. Any `PostConstruct` interceptor callback methods are invoked after this dependency injection has taken place on both the interceptor instances and the target instance.

1.3 Method Interceptors

Interceptor methods that interpose on business method invocations are denoted by the `AroundInvoke` annotation or `around-invoke` deployment descriptor element.

Around-invoke methods may be defined on interceptor classes and/or the target class (or superclass). Only one around-invoke method may be defined on a given class.

Around-invoke methods can have `public`, `private`, `protected`, or `package` level access. An around-invoke method must not be declared as `final` or `static`.

Around-invoke methods have the following signature:

```
Object <METHOD>(InvocationContext) throws Exception
```

An around-invoke method can invoke any component or resource that the method it is intercepting can invoke.

An around-invoke method invocation occurs within the same transaction and security context as the method on which it is interposing.

1.3.1 Multiple Method Interceptor Methods

If multiple method interceptor methods are defined for a target class, the following rules govern their invocation order. The deployment descriptor may be used to override the interceptor invocation order specified in annotations.

- Default interceptors, if any, are invoked first. Default interceptors can only be specified in the deployment descriptor. Default interceptors are invoked in the order of their specification in the deployment descriptor.
- If there are any interceptor classes associated with the target class using the `Interceptors` annotation, the interceptor methods defined by those interceptor classes are invoked before any interceptor methods defined on the target class itself.
- The around-invoke methods defined on those interceptor classes are invoked in the same order as the specification of the interceptor classes in the `Interceptors` annotation.
- If an interceptor class itself has superclasses, the interceptor methods defined by the interceptor class's superclasses are invoked before the interceptor method defined by the interceptor class, most general superclass first.
- After the interceptor methods defined on interceptor classes have been invoked, then, in order:
  - If any method-level interceptors are defined for the target class method that is to be invoked, the methods defined on those interceptor classes are invoked in the same order as the specification of those interceptor classes in the `Interceptors` annotation applied to that target class method.
  - If a target class has superclasses, any around-invoke methods defined on those superclasses are invoked, most general superclass first.
  - The around-invoke method, if any, on the target class itself is invoked.
- If an around-invoke method is overridden by another method (regardless of whether that method is itself an around-invoke method), it will not be invoked.

The `InvocationContext` object provides metadata that enables interceptor methods to control the behavior of the invocation chain, including whether the next method in the chain is invoked and the values of its parameters and result.

1.3.2 Exceptions

Around-invoke interceptor methods may throw any exceptions that are allowed in the `throws` clause of the target class method on which they are interposing.

Around-invoke methods are allowed to catch and suppress exceptions and to recover by calling the `proceed` method. Around-invoke methods are allowed to throw runtime exceptions or any checked exceptions that the associated target method allows within its `throws` clause.

Around-invoke methods run in the same Java call stack as the associated target method. The invocation of the `InvocationContext.proceed` method will throw the same exception as any thrown by the associated target method unless an interceptor further down the Java call stack has caught it and thrown a different exception. Exceptions and initialization and/or cleanup operations should typically be handled in try/catch/finally blocks around the `proceed` method.

1.4 Timeout Method Interceptors

Interceptor methods that interpose on timeout methods are denoted by the `AroundTimeout` annotation or `around-timeout` deployment descriptor element.

Around-timeout methods may be defined on interceptor classes and/or the target class (or superclass). Only one around-timeout method may be defined on a given class.

Around-timeout methods can have `public`, `private`, `protected`, or `package` level access. An around-timeout method must not be declared as `final` or `static`.

Around-timeout methods have the following signature:

```
Object <METHOD>(InvocationContext) throws Exception
```

An around-timeout method can invoke any component or resource that its corresponding timeout method can invoke.

An around-timeout method invocation occurs within the same transaction and security context as the timeout method on which it is interposing.

The `InvocationContext.getTimer` method allows an around-timeout method to retrieve the timer object associated with the timeout.

1.4.1 Multiple Timeout Method Interceptor Methods

If multiple timeout method interceptor methods are defined for a target class, the following rules govern their invocation order. The deployment descriptor may be used to override the interceptor invocation order specified in annotations.

- Default interceptors, if any, are invoked first. Default interceptors can only be specified in the deployment descriptor. Default interceptors are invoked in the order of their specification in the deployment descriptor.
- If there are any interceptor classes associated with the target class using the `Interceptors` annotation, the interceptor methods defined by those interceptor classes are invoked before any interceptor methods defined on the target class itself.
- The around-timeout methods defined on those interceptor classes are invoked in the same order as the specification of the interceptor classes in the `Interceptors` annotation.

- If an interceptor class itself has superclasses, the interceptor methods defined by the interceptor class's superclasses are invoked before the interceptor method defined by the interceptor class, most general superclass first.

- After the interceptor methods defined on interceptor classes have been invoked, then, in order:
  - If any method-level interceptors are defined for the target class method that is to be invoked, the methods defined on those interceptor classes are invoked in the same order as the specification of those interceptor classes in the `Interceptors` annotation applied to that target class method.
  - If a target class has superclasses, any around-timeout methods defined on those superclasses are invoked, most general superclass first.
  - The around-timeout method, if any, on the target class itself is invoked.

- If an around-timeout method is overridden by another method (regardless of whether that method is itself an around-timeout method), it will not be invoked.

The `InvocationContext` object provides metadata that enables interceptor methods to control the behavior of the invocation chain, including whether the next method in the chain is invoked and the values of its parameters and result.

1.4.2 Exceptions

Around-timeout interceptor methods may throw any exceptions that are allowed in the `throws` clause of the timeout method on which they are interposing.

Around-timeout methods are allowed to catch and suppress exceptions and to recover by calling the `proceed` method. Around-timeout methods are allowed to throw runtime exceptions or any checked exceptions that the associated target timeout method allows within its throws clause.

Around-timeout methods run in the same Java call stack as the associated target timeout method. The invocation of the `InvocationContext.proceed` method will throw the same exception as any thrown by the associated target method unless an interceptor further down the Java call stack has caught it and thrown a different exception. Exceptions and initialization and/or cleanup operations should typically be handled in try/catch/finally blocks around the `proceed` method.

1.5 Interceptors for Lifecycle Event Callbacks

Interceptor methods for lifecycle event callbacks can be defined on an interceptor class and/or directly on the target class. The `PostConstruct` and `PreDestroy` annotations are used to define an interceptor method for a lifecycle callback event. If the deployment descriptor is used to define interceptors, the `post-construct` and `pre-destroy` elements are used.

Lifecycle callback interceptor methods, around-invoke interceptor methods, and around-timeout interceptor methods may be defined on the same interceptor class. [XXX I assume the omission of around-timeout here in the Interceptors 1.1 specification was inadvertant. This point needs clarification.]

Lifecycle callback interceptor methods are invoked in an unspecified security context. Lifecycle callback interceptor methods are invoked in an transaction context determined by their target class and/or method[1]. [XXX previous statement was not true for singleton session beans.]

Lifecycle callback interceptor methods may be defined on superclasses of the target class or interceptor classes. However, a given class may not have more than one lifecycle callback interceptor method for the same lifecycle event. Any subset or combination of lifecycle callback annotations may otherwise be specified on a given class.

A single lifecycle callback interceptor method may be used to interpose on multiple callback events.

Lifecycle callback interceptor methods defined on an interceptor class have the following signature:

```
void <METHOD>(InvocationContext)
```

Lifecycle callback interceptor methods defined on a target class have the following signature:

```
void <METHOD>()
```

Lifecycle callback interceptor methods can have `public`, `private`, `protected`, or `package` level access. A lifecycle callback interceptor method must not be declared as `final` or `static`.

Example:

```
@Interceptors(MyInterceptor.class)
@Stateful public class ShoppingCartBean implements ShoppingCart {
    private float total;
    private Vector productCodes;
    ...
    public int someShoppingMethod() {
        ...
    }

    @PreDestroy void endShoppingCart() {
        ...
    }
}

public class MyInterceptor {
    ...
    @PostConstruct
    public void someMethod(InvocationContext ctx) {
        ...
        ctx.proceed();
        ...
    }
```

[1] In general, such a lifecycle callback interceptor method will be invoked in an unspecified transaction context. Note however that singleton session beans support the use of a transaction context for the invocation of the PostConstruct and PreDestroy lifecycle call interceptor methods. See [1].

```
    @PreDestroy
    public void someOtherMethod(InvocationContext ctx) {
        ...
        ctx.proceed();
        ...
    }
}
```

1.5.1 Multiple Callback Interceptor Methods for a Lifecycle Callback Event

If multiple callback interceptor methods are defined for a lifecycle event for a target class, the following rules governing their invocation order apply. The deployment descriptor may be used to override the interceptor invocation order specified in annotations.

- Default interceptors, if any, are invoked first. Default interceptors can only be specified in the deployment descriptor. Default interceptors are invoked in the order of their specification in the deployment descriptor.
- If there are any interceptor classes associated with the target class using the `Interceptors` annotation, the lifecycle callback interceptor methods defined by those interceptor classes are invoked before any lifecycle callback interceptor methods defined on the target class itself.
- The lifecycle callback interceptor methods defined on those interceptor classes are invoked in the same order as the specification of the interceptor classes in the `Interceptors` annotation.
- If an interceptor class itself has superclasses, the lifecycle callback interceptor methods defined by the interceptor class's superclasses are invoked before the lifecycle callback interceptor method defined by the interceptor class, most general superclass first.
- After the lifecycle callback interceptor methods defined on interceptor classes have been invoked, then:
    - If a target class has superclasses, any lifecycle callback interceptor methods defined on those superclasses are invoked, most general superclass first.
    - The lifecycle callback interceptor method, if any, on the target class itself is invoked.
- If a lifecycle callback interceptor method is overridden by another method regardless of whether that method is itself a lifecycle callback interceptor method of the same or different type, it will not be invoked.

All lifecycle callback interceptor methods for a given lifecycle event run in the same Java call stack. The `InvocationContext` object provides metadata that enables interceptor methods to control the invocation of further methods in the chain. If there is no corresponding callback method on the target class (or any of its superclasses), the invocation of the `InvocationContext.proceed` method by the last interceptor method defined on an interceptor class in the chain will be a no-op.

1.5.2 Exceptions

Lifecycle callback interceptor methods may throw runtime exceptions, but not checked exceptions.

The invocation of the `InvocationContext.proceed` method will throw the same exception as any thrown by another lifecycle callback interceptor method unless an interceptor further down the Java call stack has caught it and thrown a different exception. A lifecycle callback interceptor method (other than a method on the target call or its superclasses) may catch an exception thrown by another lifecycle callback interceptor method in the invocation chain, and clean up before returning. Exceptions and initialization and/or cleanup operations should typically be handled in try/catch/finally blocks around the `proceed` method.

The `PreDestroy` callbacks are not invoked when the target instance and the interceptors are discarded as a result of such exceptions: the lifecycle callback interceptor methods in the chain should perform any necessary clean-up operations as the interceptor chain unwinds.

1.6 InvocationContext

The `InvocationContext` object provides metadata that enables interceptor methods to control the behavior of the invocation chain.

```
public interface InvocationContext {
    public Object getTarget();
    public Object getTimer();
    public Method getMethod();
    public Object[] getParameters();
    public void setParameters(Object[] params);
    public java.util.Map<String, Object> getContextData();
    public Object proceed() throws Exception;
}
```

The same `InvocationContext` instance will be passed to each interceptor method for a given target class method or lifecycle event interception. This allows an interceptor to save information in the context data property of the `InvocationContext` that can be subsequently retrieved in other interceptors as a means to pass contextual data between interceptors. The contextual data is not sharable across separate target class method invocations or lifecycle callback events. If interceptors are invoked as a result of the invocation on a web service endpoint, the map returned by `getContextData()` will be the JAX-WS `MessageContext` [3]. The lifecycle of the `InvocationContext` instance is otherwise unspecified.

The `getTarget` method returns the associated target instance.

The `getTimer` method returns the timer object associated with a timeout method invocation. The `getTimer` method returns null for around-invoke methods and lifecycle callback interceptor methods.

The `getMethod` method returns the method of the target class for which the interceptor was invoked. For around-invoke and around-timeout methods, this is the method on the associated target class; for lifecycle callback interceptor methods, `getMethod` returns null. [XXX I am assuming that the omission of around-timeout methods was inadvertant. Need to verify.]

The `getParameters` method returns the parameters of the method invocation. If the `setParameters` method has been called, `getParameters` returns the values to which the parameters have been set.

The `setParameters` method modifies the parameters used for the target class method invocation. Modifying the parameter values does not affect the determination of the method that is invoked on the target class. The parameter types must match the types for the target class method, and the number of parameters supplied must equal the number of parameters on the target class method, or the `IllegalArgumentException` is thrown.

The `proceed` method causes the invocation of the next interceptor method in the chain, or, when called from the last around-invoke or around-timeout interceptor method, the target class method. Interceptor methods must always call the`InvocationContext.proceed` method or no subsequent interceptor methods or target class method or lifecycle callback methods will be invoked. [XXX I assume that the omission of around-timeout in the original was inadvertant.] The `proceed` method returns the result of the next method invoked. If a method is of type `void`, the invocation of the `proceed` method returns `null`. For lifecycle callback interceptor methods, if there is no callback method defined on the target class, the invocation of `proceed` in the last interceptor method in the chain is a no-op, and `null` is returned. If there is more than one such interceptor method, the invocation of the `proceed` method causes the container to execute those methods in order.

1.7 Default Interceptors

Default interceptors may be defined to apply to a set of target classes. The deployment descriptor is used to define default interceptors and their relative ordering.

The default interceptors are invoked before any other interceptors for a target class. The `interceptor-order` deployment descriptor element may be used to specify alternative orderings.

The `ExcludeDefaultInterceptors` annotation or `exclude-default-interceptors` deployment descriptor element is used to exclude the invocation of default interceptors for a target class.

1.8 Method-level Interceptors

An around-invoke interceptor method may be defined to apply only to a specific target class method invocation, independent of the other methods of the target class. Likewise, an around-timeout interceptor method may be defined to apply only to a specific timeout method, independent of the other timeout methods of the target class. Method-level interceptors are used to specify method interceptor methods or timeout interceptor methods. If an interceptor class that is *only* used as a method-level interceptor defines lifecycle callback interceptor methods, those lifecycle callback interceptor methods are not invoked.

Method-specific around-invoke and around-timeout interceptors can be defined by applying the `Interceptors` annotation to the method for which the interceptors are to be invoked, or by means of the `interceptor-binding` deployment descriptor element. If more than one method-level interceptor is defined for a target class method, the interceptors are invoked in the order specified. Method-level target class method interceptors are invoked in addition to any default interceptors and interceptors defined for the target class (and its superclasses). The deployment descriptor may be used to override this ordering.

The same interceptor may be applied to more than one method of the target class.

Example:

```
@Stateless
public class MyBean {

     public void notIntercepted() {...}

     @Interceptors(org.acme.MyInterceptor.class)
     public void someMethod() {
          ...
     }

     @Interceptors(org.acme.MyInterceptor.class)
     public void anotherMethod() {
          ...
     }
}
```

The applicability of a method-level interceptor to more than one method of an associated target class does not affect the relationship between the interceptor instance and the target class—only a single instance of the interceptor class is created per target class instance.

The `ExcludeDefaultInterceptors` annotation or `exclude-default-interceptors` deployment descriptor element, when applied to a target class method, is used to exclude the invocation of default interceptors for that method. The `ExcludeClassInterceptors` annotation or `exclude-class-interceptors` deployment descriptor element is used similarly to exclude the invocation of the class-level interceptors.

In the following example, if there are no default interceptors, only the interceptor `MyInterceptor` will be invoked when `someMethod` is called.

```
@Stateless
@Interceptors(org.acme.AnotherInterceptor.class)
public class MyBean {
     ...
     @Interceptors(org.acme.MyInterceptor.class)
     @ExcludeClassInterceptors
     public void someMethod() {
          ...
     }
}
```

If default interceptors have been defined for the target class, they can be excluded for the specific method by applying the `ExcludeDefaultInterceptors` annotation on the method.

```
@Stateless
@Interceptors(org.acme.AnotherInterceptor.class)
public class MyBean {
     ...
     @ExcludeDefaultInterceptors
     @ExcludeClassInterceptors
     @Interceptors(org.acme.MyInterceptor.class)
     public void someMethod() {
          ...
     }
}
```

1.9 Specification of Interceptors in the Deployment Descriptor

The deployment descriptor can be used as an alternative to metadata annotations to specify interceptors and their binding to target classes or to override the invocation order of interceptors as specified in annotations. An interceptor implementation is not required to support use of the deployment descriptor to specify interceptor metadata.

The `interceptor` deployment descriptor element is used to specify the interceptor methods of an interceptor class. The interceptor methods are specified by using the `around-invoke`, `around-timeout`, `post-construct`, and `pre-destroy` elements.

At most one method of a given interceptor class can be designated as an around-invoke method, an around-timeout method, a post-construct method, or pre-destroy method, regardless of whether the deployment descriptor is used to define interceptors or whether some combination of annotations and deployment descriptor elements is used.

1.9.1 Binding of Interceptors to Target Classes

The `interceptor-binding` element is used to specify the binding of interceptor classes to target classes and their methods. The subelements of the `interceptor-binding` element are as follows:

- The `target-name` element must identify the associated target class or the wildcard value `"*"` (which is used to define interceptors that are bound to all target classes).

- The `interceptor-class` element specifies the interceptor class. The interceptor class contained in an `interceptor-class` element must either be declared in the `interceptor` deployment descriptor element or appear in at least one `Interceptor` annotation on a target class. The `interceptor-order` element is used as an optional alternative to specify a total ordering over the interceptors defined for the given level and above.

- The `exclude-default-interceptors` and `exclude-class-interceptors` elements specify that default interceptors and class interceptors, respectively, are not to be applied to a target class and/or method.

- The `method-name` element specifies the method name for a method-level interceptor; and the optional `method-params` elements identify a single method among multiple methods with an overloaded method name.

Interceptors bound to all target classes using the wildcard syntax "*" are default interceptors. In addition, interceptors may be bound at the level of the target class (class-level interceptors) or at the level of the methods of the target class (method-level interceptors).

The binding of interceptors to classes is additive. If interceptors are bound at the class level and/or default level as well as at the method level, both class-level and/or default-level as well as method-level interceptors will apply. The deployment descriptor may be used to augment the interceptors and interceptor methods defined by means of annotations. When the deployment descriptor is used to augment the interceptors specified in annotations, the interceptor methods specified in the deployment descriptor will be invoked after those specified in annotations, according to the ordering specified in sections 1.3.1, 1.4.1, and 1.5.1. The `interceptor-order` deployment descriptor element may be used to override this ordering.

The `exclude-default-interceptors` element disables default interceptors for the level at which it is specified and lower. That is, `exclude-default-interceptors` when applied at the class level disables the application of default interceptors for all methods of the class. The `exclude-class-interceptors` element applied to a method disables the application of class-level interceptors for the given method. Explicitly listing an excluded higher-level interceptor at a lower level causes it to be applied at that level and below.

It is possible to override the ordering of interceptors by using the `interceptor-order` element to specify a total ordering of interceptors at class level and/or at method level. If the `interceptor-order` element is used, the ordering specified at the given level must be a total order over all interceptor classes that have been defined at that level and above (unless they have been explicitly excluded by means of one of the `exclude-` elements described above).

There are four possible styles of the `interceptor-binding` element syntax:

Style 1:

```
<interceptor-binding>
    <target-name>*</target-name>
    <interceptor-class>INTERCEPTOR</interceptor-class>
</interceptor-binding>
```

Specifying the `target-name` element as the wildcard value "*" designates default interceptors.

Style 2:

```
<interceptor-binding>
    <target-name>TARGETNAME</target-name>
    <interceptor-class>INTERCEPTOR</interceptor-class>
</interceptor-binding>
```

This style is used to refer to interceptors associated with the specified target class (class-level interceptors).

Style 3:

```
<interceptor-binding>
     <target-name>TARGETNAME</target-name>
     <interceptor-class>INTERCEPTOR</interceptor-class>
     <method-name>METHOD</method-name>
</interceptor-binding>
```

This style is used to associate a method-level interceptor with the specified method of the specified target class. If there are multiple methods with the same overloaded name, the element of this style refers to all the methods with the overloaded name. Note that the wildcard value "`*`" cannot be used to specify method-level interceptors.

Style 4:

```
<interceptor-binding>
     <target-name>TARGETNAME</target-name>
     <interceptor-class>INTERCEPTOR</interceptor-class>
     <method-name>METHOD</method-name>
     <method-params>
          <method-param>PARAM-1</method-param>
          <method-param>PARAM-2</method-param>
          ...
          <method-param>PARAM-n</method-param>
     </method-params>
<interceptor-binding>
```

This style is used to associate a method-level interceptor with the specified method of the specified target class. This style is used to refer to a single method within a set of methods with an overloaded name. The values `PARAM-1` through `PARAM-n` are the fully-qualified Java types of the method's input parameters (if the method has no input arguments, the `method-params` element contains no `method-param` elements). Arrays are specified by the array element's type, followed by one or more pair of square brackets (e.g. `int[][]`).

If both styles 3 and 4 are used to define method-level interceptors for the same bean, the relative ordering of those method-level interceptors is undefined.

1.9.1.1 Examples

Examples of the usage of the `interceptor-binding` syntax are given below.

Style 1: The following interceptors are default interceptors. They will be invoked in the order specified.

```
<interceptor-binding>
   <target-name>*</target-name>
   <interceptor-class>org.acme.MyDefaultIC</interceptor-class>
   <interceptor-class>org.acme.MyDefaultIC2</interceptor-class>
</interceptor-binding>
```

Style 2: The following interceptors are the class-level interceptors of the `EmployeeService` class. They will be invoked in the order specified after any default interceptors.

```
<interceptor-binding>
     <target-name>EmployeeService</target-name>
     <interceptor-class>org.acme.MyIC</interceptor-class>
     <interceptor-class>org.acme.MyIC2</interceptor-class>
</interceptor-binding>
```

Style 3: The following interceptors apply to all the `myMethod` methods of the `EmployeeService` class. They will be invoked in the order specified after any default interceptors and class-level interceptors.

```
<interceptor-binding>
     <target-name>EmployeeService</target-name>
     <interceptor-class>org.acme.MyIC</interceptor-class>
     <interceptor-class>org.acme.MyIC2</interceptor-class>
     <method-name>myMethod</method-name>
</interceptor-binding>
```

Style 4: The following interceptor element refers to the `myMethod(String firstName, String LastName)` method of the `EmployeeService` class.

```
<interceptor-binding>
     <target-name>EmployeeService</target-name>
     <interceptor-class>org.acme.MyIC</interceptor-class>
     <method-name>myMethod</method-name>
     <method-params>
          <method-param>java.lang.String</method-param>
          <method-param>java.lang.String</method-param>
     </method-params>
</interceptor-binding>
```

The following example illustrates a **Style 3** element with more complex parameter types. The method `myMethod(char s, int i, int[] iar, mypackage.MyClass mycl, mypackage.MyClass[][] myclaar)` would be specified as:

```
<interceptor-binding>
     <target-name>EmployeeService</target-name>
     <interceptor-class>org.acme.MyIC</interceptor-class>
     <method-name>myMethod</method-name>
     <method-params>
          <method-param>char</method-param>
          <method-param>int</method-param>
          <method-param>int[]</method-param>
          <method-param>mypackage.MyClass</method-param>
          <method-param>mypackage.MyClass[][]</method-param>
     </method-params>
</interceptor-binding>
```

The following example illustrates the total ordering of interceptors using the `interceptor-order` element:

```
<interceptor-binding>
   <target-name>EmployeeService</target-name>
   <interceptor-order>
      <interceptor-class>org.acme.MyIC</interceptor-class>
      <interceptor-class>org.acme.MyDefaultIC</interceptor-class>
      <interceptor-class>org.acme.MyDefaultIC2</interceptor-class>
      <interceptor-class>org.acme.MyIC2</interceptor-class>
   </interceptor-order>
</interceptor-binding>
```

Chapter 2 Related Documents

[1] Enterprise JavaBeans, version 3.1. *http://jcp.org/en/jsr/detail?id=318*.

[2] Contexts and Dependency Injection for the Java EE Platform . *http://jcp.org/en/jsr/detail?id=299*.

[3] Java API for XML Web Services (JAX-WS 2.0). *http://jcp.org/en/jsr/detail?id=224*.

Appendix A Revision History

This appendix lists the significant changes that have been made during the development of the Interceptors 1.1 Specification.

A.1 Proposed Final Draft

Initial Version.

A.2 Final Draft

Changed official specification title to Interceptors 1.1.

Added optional Interceptor and InterceptorBinding annotations.

Clarified that deployment descriptor support is not required.

Appendix B

Change Log

Editorial cleanup and conversion to standard template.

Section 1.5, second paragraph. Included around-timeout methods in the list.

Section 1.5. Clarified statement regarding transaction context of lifecycle callback methods, as it was not correct for singleton session beans. Expanded footnote to clarify.

Section 1.6. I assumed the omission of around-timeout in the description of the getMethod and proceeed methods was inadvertant, and added to the list.

Appendix C Open Issues and To-Do Items

Section 1.1. Why shouldn't support for Interceptor annotation be required? This could use some clarification. In general, I think this spec needs some kind of positioning as to how it might be used as a building block by other specifications.

Section 1.1. Similarly, why shouldn't support for InterceptorBinding annotation be required?

Section 1.1. When both `Interceptors` and interceptor binding annotations are specified, ordering rules need to be worked out for the more complex cases.

Section 1.5. I assume the omission of around-timeout here in the Interceptors 1.1 specification was inadvertant. This point needs clarification.

Section 1.5. Clarified statement regarding transaction context of lifecycle callback methods. Need to double check.

Section 1.6. I assume the omission of around-timeout in the description of the getMethod and proceed methods was inadvertant. Need to double-check.