Oracle

JSR 318
Interceptors 1.2

Maintenance Lead:

Marina Vatkina, Oracle

Please send comments to: users@interceptors-spec.java.net

Version 1.2, Maintenance Release

April 16, 2013

Specification: JSR-000318 Interceptors Specification ("Specification")
Version: 1.2
Status: Maintenance Release
Specification Lead: Oracle America, Inc. ("Specification Lead")
Release: April 30, 2013

LIMITED LICENSE GRANTS

1. License for Evaluation Purposes. Specification Lead hereby grants you a fully-paid, non-exclusive, non-transferable, worldwide, limited license (without the right to sublicense), under Specification Lead's applicable intellectual property rights to view, download, use and reproduce the Specification only for the purpose of internal evaluation. This includes (i) developing applications intended to run on an implementation of the Specification, provided that such applications do not themselves implement any portion(s) of the Specification, and (ii) discussing the Specification with any third party; and (iii) excerpting brief portions of the Specification in oral or written communications which discuss the Specification provided that such excerpts do not in the aggregate constitute a significant portion of the Specification.

2. License for the Distribution of Compliant Implementations. Specification Lead also grants you a perpetual, non-exclusive, non-transferable, worldwide, fully paid-up, royalty free, limited license (without the right to sublicense) under any applicable copyrights or, subject to the provisions of subsection 4 below, patent rights it may have covering the Specification to create and/or distribute an Independent Implementation of the Specification that: (a) fully implements the Specification including all its required interfaces and functionality; (b) does not modify, subset, superset or otherwise extend the Licensor Name Space, or include any public or protected packages, classes, Java interfaces, fields or methods within the Licensor Name Space other than those required/authorized by the Specification or Specifications being implemented; and (c) passes the Technology Compatibility Kit (including satisfying the requirements of the applicable TCK Users Guide) for such Specification ("Compliant Implementation"). In addition, the foregoing license is expressly conditioned on your not acting outside its scope. No license is granted hereunder for any other purpose (including, for example, modifying the Specification, other than to the extent of your fair use rights, or distributing the Specification to third parties). Also, no right, title, or interest in or to any trademarks, service marks, or trade names of Specification Lead or Specification Lead's licensors is granted hereunder. Java, and Java-related logos, marks and names are trademarks or registered trademarks of Oracle America, Inc. in the U.S. and other countries.

3. Pass-through Conditions. You need not include limitations (a)-(c) from the previous paragraph or any other particular "pass through" requirements in any license You grant concerning the use of your Independent Implementation or products derived from it. However, except with respect to Independent Implementations (and products derived from them) that satisfy limitations (a)-(c) from the previous paragraph, You may neither: (a) grant or otherwise pass through to your licensees any licenses under Specification Lead's applicable intellectual property rights; nor (b) authorize your licensees to make any claims concerning their implementation's compliance with the Specification in question.

4. Reciprocity Concerning Patent Licenses.

a. With respect to any patent claims covered by the license granted under subparagraph 2 above that would be infringed by all technically feasible implementations of the Specification, such license is conditioned upon your offering on fair, reasonable and non-discriminatory terms, to any party seeking it from You, a perpetual, non-exclusive, non-transferable, worldwide license under Your patent rights which are or would be infringed by all technically feasible implementations of the Specification to develop, distribute and use a Compliant Implementation.

b With respect to any patent claims owned by Specification Lead and covered by the license granted under subparagraph 2, whether or not their infringement can be avoided in a technically feasible manner when implementing the Specification, such license shall terminate with respect to such claims if You initiate a claim against Specification Lead that it has, in the course of performing its responsibilities as the Specification Lead, induced any other entity to infringe Your patent rights.

c Also with respect to any patent claims owned by Specification Lead and covered by the license granted under subparagraph 2 above, where the infringement of such claims can be avoided in a technically feasible manner when implementing the Specification such license, with respect to such claims, shall terminate if You initiate a claim against Specification Lead that its making, having made, using, offering to sell, selling or importing a Compliant Implementation infringes Your patent rights.

5. Definitions. For the purposes of this Agreement: "Independent Implementation" shall mean an implementation of the Specification that neither derives from any of Specification Lead's source code or binary code materials nor, except with an appropriate and separate license from Specification Lead, includes any of Specification Lead's source code or binary code materials; "Licensor Name Space" shall mean the public class or interface declarations whose names begin with "java", "javax", "com.sun" and “com.oracle” or their equivalents in any subsequent naming convention adopted by Oracle America, Inc. through the Java Community Process, or any recognized successors or replacements thereof; and "Technology Compatibility Kit" or "TCK" shall mean the test suite and accompanying TCK User's Guide provided by Specification Lead which corresponds to the Specification and that was available either (i) from Specification Lead's 120 days before the first release of Your Independent Implementation that allows its use for commercial purposes, or (ii) more recently than 120 days from such release but against which You elect to test Your implementation of the Specification.

This Agreement will terminate immediately without notice from Specification Lead if you breach the Agreement or act outside the scope of the licenses granted above.

DISCLAIMER OF WARRANTIES

THE SPECIFICATION IS PROVIDED "AS IS". SPECIFICATION LEAD MAKES NO REPRESENTATIONS OR WARRANTIES, EITHER EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO, WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, NON-INFRINGEMENT (INCLUDING AS A CONSEQUENCE OF ANY PRACTICE OR IMPLEMENTATION OF THE SPECIFICATION), OR THAT THE CONTENTS OF THE SPECIFICATION ARE SUITABLE FOR ANY PURPOSE. This document does not represent any commitment to release

or implement any portion of the Specification in any product. In addition, the Specification could include technical inaccuracies or typographical errors.

LIMITATION OF LIABILITY

TO THE EXTENT NOT PROHIBITED BY LAW, IN NO EVENT WILL SPECIFICATION LEAD OR ITS LICENSORS BE LIABLE FOR ANY DAMAGES, INCLUDING WITHOUT LIMITATION, LOST REVENUE, PROFITS OR DATA, OR FOR SPECIAL, INDIRECT, CONSEQUENTIAL, INCIDENTAL OR PUNITIVE DAMAGES, HOWEVER CAUSED AND REGARDLESS OF THE THEORY OF LIABILITY, ARISING OUT OF OR RELATED IN ANY WAY TO YOUR HAVING, IMPELEMENTING OR OTHERWISE USING USING THE SPECIFICATION, EVEN IF SPECIFICATION LEAD AND/OR ITS LICENSORS HAVE BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

You will indemnify, hold harmless, and defend Specification Lead and its licensors from any claims arising or resulting from: (i) your use of the Specification; (ii) the use or distribution of your Java application, applet and/or implementation; and/or (iii) any claims that later versions or releases of any Specification furnished to you are incompatible with the Specification provided to you under this license.

RESTRICTED RIGHTS LEGEND

U.S. Government: If this Specification is being acquired by or on behalf of the U.S. Government or by a U.S. Government prime contractor or subcontractor (at any tier), then the Government's rights in the Software and accompanying documentation shall be only as set forth in this license; this is in accordance with 48 C.F.R. 227.7201 through 227.7202-4 (for Department of Defense (DoD) acquisitions) and with 48 C.F.R. 2.101 and 12.212 (for non-DoD acquisitions).

REPORT

If you provide Specification Lead with any comments or suggestions concerning the Specification ("Feedback"), you hereby: (i) agree that such Feedback is provided on a non-proprietary and non-confidential basis, and (ii) grant Specification Lead a perpetual, non-exclusive, worldwide, fully paid-up, irrevocable license, with the right to sublicense through multiple levels of sublicensees, to incorporate, disclose, and use without limitation the Feedback for any purpose.

GENERAL TERMS

Any action related to this Agreement will be governed by California law and controlling U.S. federal law. The U.N. Convention for the International Sale of Goods and the choice of law rules of any jurisdiction will not apply.

The Specification is subject to U.S. export control laws and may be subject to export or import regulations in other countries. Licensee agrees to comply strictly with all such laws and regulations and acknowledges that it has the responsibility to obtain such licenses to export, re-export or import as may be required after delivery to Licensee.

This Agreement is the parties' entire agreement relating to its subject matter. It supersedes all prior or contemporaneous oral or written communications, proposals, conditions, representations and warranties and prevails over any conflicting or additional terms of any quote, order, acknowledgment, or other communication between the parties relating to its subject matter during the term of this Agreement. No modification to this Agreement will be binding, unless in writing and signed by an authorized representative of each party.

Chapter 1 Overview

Interceptors are used to interpose on business method invocations and specific events, such as timeout events or lifecycle events, that occur on instances of Java EE components and other managed classes.

An interceptor is either a method on the component class (called the target class) or a method on a separate class (called the interceptor class) associated with the target class.

1.1 Revision History

This document is an update to the Interceptors specification 1.1. Version 1.1 was based on the Interceptors chapter of the *Enterprise JavaBeans™ 3.0 (EJB)* specification (see [1]). The current version also includes interceptor binding definitions that had been originally defined in the *Contexts and Dependency Injection for the Java EE Platform (CDI)* specification (see [3]).

The full change log for version 1.2 is found in Appendix B.

1.2 Relationship to Other Specifications

The EJB specification requires support for the chapters 2, 4, and 5 of this specification (excluding sections 5.3 and 5.4) and the CDI specification requires support for the chapters 2, 3, and 5 (excluding section 5.5). Both the EJB and CDI specifications provide extensions to this specification. Other specifications may choose to do so in the future.

1.3 Document Conventions

The regular Times font is used for information that is prescriptive by the Interceptors specification.

The italic Times font is used for paragraphs that contain descriptive information, such as notes describing typical use, or notes clarifying the text with prescriptive specification.

`The Courier font is used for code examples.`

Chapter 2

Interceptor Programming Contract

2.1 Interceptor Definition

An interceptor method may be defined on a target class or on an interceptor class associated with the target class.

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

An interceptor class must not be `abstract` and must have a `public` no-arg constructor.

An around-invoke interceptor method, an around-timeout interceptor method, and lifecycle callback interceptor methods for different lifecycle events may be defined on the same interceptor class. An interceptor class may have superclasses. See section 5.2.2 on the invocation order of interceptors with superclasses.

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

Interceptors classes may be associated with the target class using either interceptor binding (see Chapter 3) or the `Interceptors` annotation (see Chapter 4). Typically only one interceptor association type is used for any component.

A deployment descriptor may be used to specify the invocation order of interceptors or to override the order specified in metadata annotations. A deployment descriptor can be optionally used to define default interceptors or to associate interceptors with the target class. The EJB specification requires support for the `ejb-jar.xml` deployment descriptor (see [2]) and the CDI specification requires supports for the `beans.xml` deployment descriptor (see [3]).

2.2 Interceptor Life Cycle

The lifecycle of an interceptor instance is the same as that of the target class instance with which it is associated.

Except as noted below, when the target instance is created, a corresponding interceptor instance is created for each associated interceptor class. These interceptor instances are destroyed if the target instance fails to be created or when it is removed.

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.

When a `AroundConstruct` lifecycle callback interceptor is used, the following rules apply:

- The `AroundConstruct` lifecycle callback is invoked after the dependency injection has been completed on instances of all interceptor classes associated with the target class. Injection of the target component into interceptor instances that are invoked during the `AroundConstruct` lifecycle callback is not supported.
- The target instance is created and its constructor injection is performed, if applicable, when the last interceptor method in the `AroundConstruct` interceptor chain invokes the `InvocationContext.proceed` method. If the `InvocationContext.proceed` method is not invoked by an interceptor method, the target instance will not be created.
- The `AroundConstruct` interceptor method can access the constructed instance using the `InvocationContext.getTarget` method after the `InvocationContext.proceed` completes.
- Dependency injection on the target instance is not completed until after invocation of all interceptor methods in the `AroundConstruct` interceptor chain complete successfully.
- The `PostConstruct` lifecycle callback chain for the target instance, if any, will be invoked after the dependency injection has been completed on the target instance.
- An `AroundConstruct` lifecycle callback interceptor method should exercise caution when invoking methods of the target instance since its dependency injection may not have been completed.

With the exception of `AroundConstruct` lifecycle callback interceptors, all interceptor methods, including `PostConstruct` callbacks are invoked after dependency injection has been completed on both the interceptor instances and the target instance[1].

`PreDestroy` callbacks, if any, are invoked before the target instance and all interceptor instances associated with it are destroyed.[2]

2.2.1 Interceptor Environment

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.

Around-invoke and around-timeout interceptor methods run in the same Java thread as the associated target method.

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 (see Chapter 5 for the rules on interceptors ordering).

2.3 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 Constructor<?> getConstructor();
     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` [4]. The lifecycle of the `InvocationContext` instance is otherwise unspecified.

[1] If the `PostConstruct` callback is defined on the interceptor class, this callback it is not invoked when the interceptor instance itself is created.

[2] If the `PreDestroy` callback is defined on the interceptor class, it is not invoked when the interceptor instance itself is destroyed.

The `getTarget` method returns the associated target instance. For the `AroundConstruct` lifecycle callback interceptor method, `getTarget` returns null if called before the `proceed` method returns.

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. In a lifecycle callback interceptor for which there is no corresponding lifecycle callback method on the target class or in the `AroundConstruct` lifecycle callback interceptor method, `getMethod` returns null.

The `getConstructor` method returns the constructor of the target class for which the `AroundConstruct` interceptor was invoked. For around-invoke, around-timeout and all other lifecycle callback interceptor methods, `getConstructor` returns null.

The `getParameters` method returns the parameters of the method or constructor 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 or constructor invocation. Modifying the parameter values does not affect the determination of the method or the constructor that is invoked on the target class. The parameter types must match the types for the target class method or constructor, and the number of parameters supplied must equal the number of parameters on the target class method or constructor[3], 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. For `AroundConstruct` lifecycle callback interceptor methods, the invocation of the last interceptor method in the chain causes the target instance to be created. Interceptor methods must always call the `InvocationContext.proceed` method or no subsequent interceptor methods, target class method, or lifecycle callback methods will be invoked, or the target instance will not be created. 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 `AroundConstruct` lifecycle callback interceptor method, the invocation of `proceed` in the last interceptor method in the chain causes the target instance to be created. For all other 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[4], and `null` is returned.

2.4 Exceptions

Interceptor methods are allowed to throw runtime exceptions or any checked exceptions that the associated target method allows within its `throws` clause.

[3] If the last parameter is a vararg parameter of type `T`, it is considered be equivalent to a parameter of type `T[]`.

[4] In case of the `PostConstruct` interceptor, if there is no callback method defined on the target class, the invocation of `InvocationContext.proceed` method in the last interceptor method in the chain validates the target instance.

Interceptor methods are allowed to catch and suppress exceptions and to recover by calling the `InvocationContext.proceed` 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.

2.5 Business Method Interceptors

Interceptor methods that interpose on business method invocations are denoted by the `AroundInvoke` annotation.

Around-invoke methods may be defined on interceptor classes and/or the target class and/or superclasses of the target class or the interceptor classes. However, 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 `abstract`, `final` or `static`.

Around-invoke methods have the following signature:

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

Note: An around-invoke interceptor method may be declared to throw any checked exceptions that the associated target method allows within its throws clause. It may be declared to throw the `java.lang.Exception` if it interposes on several methods that can throw unrelated checked exceptions.

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

In general, an around-invoke method invocation occurs within the same transaction and security context as the method on which it is interposing. However, note that the transaction context may be changed by transactional interceptors in the invocation chain.

The following example defines `MonitoringInterceptor`, which is used to interpose on `ShoppingCart` business methods:

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

@Monitored @Interceptor
public class MonitoringInterceptor {

     @AroundInvoke
     public Object monitorInvocation(InvocationContext ctx){

          //... log invocation data ...

          return ctx.proceed();
     }

}

@Monitored
public class ShoppingCart {

     public void placeOrder(Order o) {
          ...
     }
}
```

2.6 Interceptors for Lifecycle Event Callbacks

The `AroundConstruct` annotation denotes lifecycle callback interceptor methods that interpose on invocation of the target instance's constructor.

The `PostConstruct` annotation denotes lifecycle callback interceptor methods that are invoked after the target instance has been constructed and dependency injection on that instance has been completed, but before any business method or other event can be invoked on the target instance.

The `PreDestroy` annotation denotes lifecycle callback interceptor methods that interpose on the target instance's removal.

Specifications that support lifecycle callback interceptors are permitted to define additional lifecycle events.

The `AroundConstruct` interceptor methods may be only defined on interceptor classes and/or superclasses of interceptor classes. The `AroundConstruct` callback should not be defined on the target class.

All other interceptor methods that interpose on the target instance lifecycle event callbacks can be defined on an interceptor class and/or directly on the target class.

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

Lifecycle callback interceptor methods are invoked in an unspecified security context. Lifecycle callback interceptor methods are invoked in a transaction context determined by their target class and/or method[5].

`AroundConstruct` lifecycle callback interceptor methods may be defined on superclasses of interceptor classes. All other lifecycle callback interceptor methods may be defined on superclasses of the target class or of 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 target class or interceptor class.

Lifecycle callback interceptor methods defined on an interceptor class must have one of the following signatures:

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

Note: A lifecycle callback interceptor method must not throw application exceptions, but it may be declared to throw checked exceptions including the `java.lang.Exception` if the same interceptor method interposes on business or timeout methods in addition to lifecycle events. If a lifecycle callback interceptor method returns a value, it is ignored by the container.

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 `abstract`, `final` or `static`.

The following example defines lifecycle interceptor methods on both the interceptor class and the target class (see Chapter 5 for the rules on interceptors ordering):

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

[5] In general, such a lifecycle callback interceptor method will be invoked in an unspecified transaction context. Note however that singleton and stateful session beans support the use of a transaction context for the invocation of their lifecycle call interceptor methods (see the *Enterprise JavaBeans™* specification [2]). The transaction context may be also changed by transactional interceptors in the invocation chain.

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

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

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

2.6.1 Exceptions

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

In addition to the rules specified in section 2.4 the following rules apply to the lifecycle callback interceptor methods:

- A lifecycle callback interceptor method (other than a method on the target class or its superclasses) may catch an exception thrown by another lifecycle callback interceptor method in the invocation chain, and clean up before returning.
- 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.

2.7 Timeout Method Interceptors

The EJB Timer Service, defined by the *Enterprise JavaBeans™* specification [2], is a container-provided service that allows the Bean Provider to register enterprise beans for timer callbacks according to a calendar-based schedule, at a specified time, after a specified elapsed time, or at specified intervals. The timeout callbacks registered with the Timer Service are called timeout methods.

Interceptor methods that interpose on timeout methods are denoted by the `AroundTimeout` annotation.

Around-timeout methods may be defined on interceptor classes and/or the target class and/or superclasses of the target class or interceptor classes. However, only one around-timeout method may be defined on a given class.

Note: If a timeout method is also exposed as a business method, the around-timeout interceptor methods will be invoked only when the target method is executed by the container as a result of the timeout event. When the same method is invoked as a business method by the client, and the target method has around-invoke interceptors associated with it, only around-invoke interceptor methods will be invoked.

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

Around-timeout methods have the following signature:

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

Note: An around-timeout interceptor method must not throw application exceptions, but it may be declared to throw checked exceptions or the `java.lang.Exception` if the same interceptor method interposes on business methods in addition to the timeout methods.

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[6] 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.

[6] Note that the transaction context may be changed by transactional interceptors in the invocation chain.

In the following example around-timeout interceptor is associated with two timeout methods:

```
public class MyInterceptor {

     private Logger _logger = ...;

     @AroundTimeout
     private Object aroundTimeout(InvocationContext ctx)
                                  throws Exception {
          _logger.info("processing: " + ctx.getTimer().getInfo());
          return ctx.proceed();
          ...
     }
}

@Interceptors(MyInterceptor.class)
@Singleton
public class CacheBean {

     private Data data;

     @Schedule(minute="*/30",hour="*",info="update-cache")
     public void refresh(Timer t) {
          data.refresh();
     }

     @Schedule(dayOfMonth="1",info="validate-cache")
     public void validate(Timer t) {
          data.validate();
     }

}
```

2.8 Constructor- and Method-level Interceptors

Method-level interceptors are interceptor classes associated with a specific business or timeout method of the target class.

Constructor-level interceptors are interceptor classes associated with a constructor of the target class.

For example, 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 on the target class, independent of the other timeout methods of the target class.

Only the interceptor methods of the interceptor class that are relevant for the context in which the interceptor is bound are invoked. For example, if the interceptor is bound only to a business method, any `AroundConstruct`, `AroundTimeout`, `PostConstruct`, or `PreDestroy` methods on the interceptor are not called.

Method-level interceptors may not be associated with a lifecycle callback method of the target class.

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.

In the following example only `placeOrder` method will be monitored:

```
public class ShoppingCart {

     @Monitored
     public void placeOrder() {...}

}
```

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

In the following example `MyInterceptor` interceptor interposes on a subset of business methods of the session bean. **Note** that `created` and `removed` methods of the `MyInterceptor` interceptor will **not** be called:

```
public class MyInterceptor {
    ...
     @AroundInvoke
     public Object around_invoke(InvocationContext ctx) {...}

     @PostConstruct
    public void created(InvocationContext ctx) {...}

     @PreDestroy
    public void removed(InvocationContext ctx) {...}
}

@Stateless
public class MyBean {

     @PostConstruct
     void init() {...}

     public void notIntercepted() {...}

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

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

In the following example `ValidationInterceptor` interceptor interposes on the bean constructor **only**:

```
@Inherited
@InterceptorBinding
@Target({CONSTRUCTOR, METHOD})
@Retention(RUNTIME)
public @interface ValidateSpecial {}

@ValidateSpecial
public class ValidationInterceptor {

     @AroundConstruct
     public void validate_constructor(InvocationContext ctx){...}

     @AroundInvoke
     public Object validate_method(InvocationContext ctx){...}

}

public class SomeBean {

     @ValidateSpecial
     SomeBean(...) {
          ...
     }

     public void someMethod() {
          ...
     }
}
```

In the following example `ValidationInterceptor` interceptor interposes on the bean constructor and *one* of the business methods of the bean:

```
public class SomeBean {

     @ValidateSpecial
     SomeBean(...) {
          ...
     }

     public void someMethod() {
          ...
     }

     @ValidateSpecial
     public void anotherMethod() {
          ...
     }
}
```

Chapter 3

Associating Interceptors using Interceptor Bindings

Interceptor bindings are intermediate annotations that may be used to associate interceptors with any component that is not itself an interceptor. See the *Contexts and Dependency Injection Specification* (CDI) [3] for additional restrictions on which components can have interceptors.

3.1 Interceptor Binding Types

An interceptor binding type is a Java annotation defined as `Retention(RUNTIME)`. Typically an interceptor binding is defined as `Target({TYPE, METHOD, CONSTRUCTOR})` or any subset of valid target types.

An interceptor binding type may be declared by specifying the `InterceptorBinding` meta-annotation.

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

3.1.1 Interceptor binding types with additional interceptor bindings

An interceptor binding type may declare other interceptor bindings.

```
@Inherited
@InterceptorBinding
@Target({TYPE, METHOD})
@Retention(RUNTIME)
@Monitored
public @interface DataAccess {}
```

Interceptor bindings are transitive — an interceptor binding declared by an interceptor binding type is inherited by all components and other interceptor binding types that declare that interceptor binding type.

An interceptor binding type can only be applied to an interceptor binding type defining a superset of it's target types. For example interceptor binding types declared `Target(TYPE)` may not be applied to interceptor binding types declared `Target({TYPE, METHOD})`.

3.1.2 Other sources of interceptor bindings

An extension specification may define other sources of interceptor bindings, such as via a CDI stereotype.

3.2 Declaring Interceptor Bindings of an Interceptor

The interceptor bindings of an interceptor are specified by annotating the interceptor class with the binding types and the `Interceptor` annotation and are called the set of interceptor bindings for the interceptor.

```
@Monitored @Interceptor
public class MonitoringInterceptor {

     @AroundInvoke
     public Object monitorInvocation(InvocationContext ctx)
               throws Exception { ... }

}
```

An interceptor class may declare multiple interceptor bindings.

Multiple interceptors may declare the same interceptor bindings.

An extension specification may define other ways of declaring an interceptor.

If an interceptor does not declare an `Interceptor` annotation, it could be bound to components using `Interceptors` annotation or a deployment descriptor file.

All interceptors declared using the `Interceptor` annotation should specify at least one interceptor binding. If an interceptor declared using the `Interceptor` annotation does not declare any interceptor binding, non-portable behavior results.

With the exception of `AroundConstruct` lifecycle callback interceptors, an interceptor for lifecycle callbacks may only declare interceptor binding types that are defined as `Target(TYPE)`.

3.3 Binding an Interceptor to a Component

An interceptor may be bound to a component by annotating the component class, a method, or constructor of the component class, with the interceptor binding type.

In the following example, the `MonitoringInterceptor` will be applied at the class level, and therefore applies to all business methods of the class:

```
@Monitored
public class ShoppingCart { ... }
```

In this example, the `MonitoringInterceptor` will be applied at the method level:

```
public class ShoppingCart {

     @Monitored
     public void placeOrder() { ... }

}
```

A component class, method, or constructor of a component class may declare multiple interceptor bindings.

The set of interceptor bindings for a method or constructor are those declared at class level combined with those declared at method or constructor level.

An interceptor binding declared on a method or constructor replaces an interceptor binding of the same type declared at class level or inherited from a superclass[7].

An extension specification may define additional rules for combining interceptor bindings, such as interceptors defined via a CDI stereotype.

If the component class declares or inherits a class level interceptor binding, it must not be declared `final`, or have any non-`static`, non-`private`, `final` methods. If a component has a class-level interceptor binding and is declared `final` or has a non-`static`, non-`private`, `final` method, the container automatically detects the problem and treats it as a definition error.

[7] This requirement follows the rules from the *Common Annotations* specification, section 2.1 (see [5]).

If a non-`static`, non-`private` method of a component class declares a method level interceptor binding, neither the method nor the component class may be declared `final`. If a non-`static`, non-`private`, `final` method of a component has a method level interceptor binding, the container automatically detects the problem and treats it as a definition error.

3.4 Interceptor Resolution

The process of matching interceptors to a certain lifecycle callback method, timeout method, business method or a constructor of a certain component is called interceptor resolution.

For a lifecycle callback, the interceptor bindings include the interceptor bindings declared or inherited by the component at the class level, including, recursively, interceptor bindings declared as meta-annotations of other interceptor bindings.

For a business method, timeout method, or constructor the interceptor bindings include the interceptor bindings declared or inherited by the component at the class level, including, recursively, interceptor bindings declared as meta-annotations of other interceptor bindings, together with all interceptor bindings declared on the constructor or method, including, recursively, interceptor bindings declared as meta-annotations of other interceptor bindings.

An interceptor is bound to a method or constructor if:

- The method or constructor has all the interceptor bindings of the interceptor. A method or constructor has an interceptor binding of an interceptor if it has an interceptor binding with (a) the same type and (b) the same annotation member value for each member. An extension specification may further refine this rule. For example the CDI specification [3] adds the `javax.enterprise.util.Nonbinding` annotation, causing member values to be ignored by the resolution process.
- The interceptor intercepts the given kind of lifecycle callback or business method.
- The interceptor is enabled.

3.4.1 Interceptors with multiple bindings

An interceptor class may specify multiple interceptor bindings.

```
@Monitored @Logged @Interceptor @Priority(1100)
public class MonitoringLoggingInterceptor {
     @AroundInvoke
     public Object aroundInvoke(InvocationContext context)
               throws Exception { ... }

}
```

This interceptor will be bound to all methods of this component:

```
@Monitored @Logged
public class ShoppingCart { ... }
```

The `MonitoringLoggingInterceptor` will not be bound to methods of this component, since the `Logged` interceptor binding does not appear:

```
@Monitored
public class ShoppingCart {

     public void placeOrder() { ... }

}
```

However, the `MonitoringLoggingInterceptor` will be bound to the `placeOrder` method of this component:

```
@Monitored
public class ShoppingCart {

     @Logged
     public void placeOrder() { ... }

}
```

3.4.2 Interceptor binding types with members

Interceptor binding types may have annotation members.

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

     boolean persistent();

}
```

Any interceptor with that interceptor binding type must select a member value:

```
@Monitored(persistent=true) @Interceptor @Priority(2100)
public class PersistentMonitoringInterceptor {

     @AroundInvoke
     public Object monitorInvocation(InvocationContext ctx)
              throws Exception { ... }

}
```

The `PersistentMonitoringInterceptor` applies to this component:

```
@Monitored(persistent=true)
public class ShoppingCart { ... }
```

But not to this component:

```
@Monitored(persistent=false)
public class SimpleShoppingCart { ... }
```

Annotation member values are compared using the `equals` method.

Array-valued or annotation-valued members of an interceptor binding type are not supported. An extension specification may add support for these member types. For example the CDI specification [3] adds the `javax.enterprise.util.Nonbinding` annotation, allowing array-valued or annotation-valued members to be used on the annotation type, but ignored by the resolution process

If the set of interceptor bindings of a component class or interceptor, including bindings inherited from stereotypes and other interceptor bindings, has two instances of a certain interceptor binding type and the instances have different values of some annotation member, the container automatically detects the problem and treats it as a definition error.

Chapter 4

Associating Interceptors using the Interceptors Annotation

The `Interceptors` annotation can be used to denote interceptor classes and associate one or more interceptor classes with a target class, and/or one or more of its methods, and/or a constructor of the target class.

If multiple interceptors are defined for the target class in the `Interceptors` annotation, they are invoked in the order in which they are specified. See Chapter 5 for the rules on interceptors ordering.

If a deployment descriptor is supported, it can be used to associate interceptor classes with the target class, and/or methods, and/or a constructor of the target class and specify the order of interceptor invocation or override metadata specified by annotations.

The `Interceptors` annotation can be applied to a method or a constructor of the target class or of a superclass of the target class:

- Method-level 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.
- Constructor-level interceptors can be defined by applying the `Interceptors` annotation to the constructor for which the interceptors are to be invoked.

If more than one constructor- or method-level interceptor is defined for a target class constructor or method, the interceptors are invoked in the order specified. Constructor- and method-level interceptors are invoked in addition to any default interceptors and interceptors defined for the target class (and its superclasses).

Example:

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

     @Interceptors(org.acme.MyOtherInterceptor.class)
     public void otherMethod() {
          ...
     }
}
```

4.1 Default Interceptors

Default interceptors are interceptors that apply to a set of target classes. If a deployment descriptor is supported, it may be used to define default interceptors and their relative ordering.

The `ExcludeDefaultInterceptors` annotation is used to exclude the invocation of default interceptors for a target class, or when applied to a target class constructor or method, is used to exclude the invocation of default interceptors for that constructor or that method.

Example:

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

Chapter 5

Interceptor Ordering

5.1 Defining Interceptor Order

Associating interceptors with the target class or method of the target class using the `Interceptors` annotation enables them for the target class, a method, or a constructor of the target class. The order in which they are invoked is determined by the order in which they are specified in the annotation (see section 5.5). If a deployment descriptor is supported, it may be used to to override the association, specify the interceptor invocation order, or override the order specified in annotations.

Interceptors declared using interceptor bindings are enabled and ordered using the `Priority` annotation (see section 5.3) or the `beans.xml` deployment descriptor (see the CDI specification [3]).

An extension specification may define alternative mechanisms to enable and order interceptors.

For the same interceptor method type, interceptors declared using interceptor bindings are called after interceptors declared using the `Interceptors` annotation (or using the corresponding element of a deployment descriptor) and before an interceptor method of the same interceptor type declared on the target class or any superclass of the target class.

The `InvocationContext` object (see section 2.3) 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.

5.2 Common Ordering Rules

The following rules apply regardless of how an interceptor is associated with the target class. Additional ordering rules for interceptors declared using interceptor bindings are described in Section 5.3. Additional ordering rules for interceptors declared using the `Interceptors` annotation are described in Section 5.5.

5.2.1 Invocation Order of Interceptors Declared on the Target Class

Interceptor methods declared on the target class or its superclasses are invoked in the following order:

- If a target class has superclasses, any interceptor methods defined on those superclasses are invoked, most general superclass first.
- The interceptor method, if any, on the target class itself is invoked.

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

5.2.2 Invocation Order of Interceptors with Superclasses

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.

5.3 Ordering Interceptors using the Priority Annotation

An interceptor bound to a component, a component method, or constructor using interceptor binding may be enabled for the entire application by applying the `Priority` annotation, along with a priority value, on the interceptor class. Interceptors with smaller priority values are called first. If more than one interceptor has the same priority, the relative order of these interceptor is undefined.

```
@Monitored @Interceptor @Priority(100)
public class MonitoringInterceptor {

     @AroundInvoke
     public Object monitorInvocation(InvocationContext ctx)
               throws Exception { ... }

}
```

The `Priority` annotation is ignored on interceptors bound to a component using the `Interceptors` annotation.

Extension specifications may define other ways of assigning priorities to an interceptor.

5.4 Interceptor Priority

The following priority values are defined for interceptor ordering when used with the `Priority` annotation:

- `Interceptor.Priority.PLATFORM_BEFORE = 0`
- `Interceptor.Priority.LIBRARY_BEFORE = 1000`
- `Interceptor.Priority.APPLICATION = 2000`
- `Interceptor.Priority.LIBRARY_AFTER = 3000`
- `Interceptor.Priority.PLATFORM_AFTER = 4000`

These values define the following interceptor ranges to order interceptors for a specific interposed method or event in the interceptor chain:

- Interceptors defined by the Java EE Platform specifications that are to be executed at the beginning of the interceptor chain should have priority values in the range `PLATFORM_BEFORE` up until `LIBRARY_BEFORE`.
- Interceptors defined by extension libraries that are intended to be executed earlier in the interceptor chain should have priority values in the range `LIBRARY_BEFORE` up until `APPLICATION`.
- Interceptors defined by applications should be in the range `APPLICATION` up until `LIBRARY_AFTER`.
- Interceptors defined by extension libraries that are intended to be executed later in the interceptor chain should have priority values in the range `LIBRARY_AFTER` up until `PLATFORM_AFTER`.
- Interceptors defined by the Java EE Platform specifications that are to be executed at the end of the interceptor chain should have priority values at `PLATFORM_AFTER` or higher.
- An interceptor that must be invoked before or after another defined interceptor can choose any appropriate value.

Negative priority values are reserved for future use by this specification and should not be used.

The following example defines an extension library interceptor that is to be executed before any application interceptor, but after any early platform interceptor:

```
@Priority(Interceptor.Priority.LIBRARY_BEFORE+10)
@Interceptor
public class ValidationInterceptor { ... }
```

5.5 Invocation Order for Multiple Interceptors Defined Using the Interceptors Annotation

If multiple business or timeout method interceptor methods are defined for a target class or multiple callback interceptor methods are defined for a lifecycle event for a target class using the `Interceptors` annotation, the following rules govern their invocation order for each interceptor type. A 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 a 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 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 interceptor methods are invoked according to the rules described in the section 5.2.2.
- After the interceptor methods defined on interceptor classes have been invoked, then in order:
    - If any constructor- or around-invoke or around-timeout method-level interceptors are defined for the target class constructor or method that is to be invoked, the corresponding 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 constructor or method.
    - Interceptors declared on the target class or its superclasses are invoked according to the rules described in the section 5.2.1.

In the following example interceptors will be invoked in the following order when `someMethod` is called: `SomeInterceptor`, `AnotherInterceptor`, `MyInterceptor`.

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

}
```

The `ExcludeClassInterceptors` annotation can be used to exclude the invocation of the class-level interceptors.

In the next example 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() {
          ...
     }

}
```

Chapter 6 Related Documents

[1] Enterprise JavaBeans™, version 3.0. *http://jcp.org/en/jsr/detail?id=220*.

[2] Enterprise JavaBeans™, version 3.2. *http://jcp.org/en/jsr/detail?id=345*.

[3] Contexts and Dependency Injection for the Java EE Platform 1.1 (CDI specification) *http://jcp.org/en/jsr/detail?id=346*.

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

[5] Common Annotations for the Java Platform Specification 1.2. *http://jcp.org/en/jsr/detail?id=250*.

Appendix A

Revision History

This appendix lists the significant changes that have been made during the development of the Interceptors 1.2 Specification.

A.1 Draft 1

Editorial cleanup and conversion to standard template.

Assigned chapter numbers to sections and rearranged them for consistency.

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

Added Chapter 1 (Overview)

Added Chapter 3, derived from Chapter 9 of the CDI specification.

Rearranged various sections and examples for better flow.

Added examples with interceptor bindings to common sections

Added standard `Priority` ranges

Added a note on a timeout method that is also a business method and around-timeout and around-invoke interceptors

Removed deployment descriptors definitions

A.2 Draft 2

Fixed page numbers in the book

Minor editorial changes

Moved Section 5.5 to the end of Chapter 5

Moved rule on invocation order of interceptors with superclasses to Section 5.2.2

Replaced rules on the `Nonbinding` annotation with references to the CDI spec

Clarified that an around-invoke interceptor method, around-timeout interceptor method, and lifecycle callback interceptor methods for different lifecycle events may be defined on the same interceptor class

Added the `AroundConstruct` lifecycle callback interceptor

Extended `InvocationContext` with `getConstructor` method; adjusted rules on the `InvocationContext.getTarget` return value, and `InvocationContext.proceed` result

Added notes to the around-invoke and around-timeout about the `throw Exception` clause in their signatures

Added a note that transaction context of interceptors may be changed by transactional interceptors in the invocation chain

Explained how interceptors are enabled

A.3 Draft 3

Editorial changes

Moved sections 5.2 and 5.3 under section 5.2, “Common Ordering Rules”

Created section 2.2.1, “Interceptor Environment”

Added an option for lifecycle callback methods defined on interceptor classes to have the same signature as around-invoke and around-timeout methods

A.4 Final Draft 1

Minor editorial changes

Enabled method-level interceptors for lifecycle callbacks. Added constructor-level interceptors that follow the same rules as method-level interceptors.

Changed `InvocationContext.getMethod` method to return null if there is no corresponding lifecycle callback method on the bean class and for the `AroundConstruct` lifecycle callback interceptor method.

A.5 Final Draft 2

Minor editorial changes

Editorial changes for consistency in constructor-level interceptors descriptions

Removed support of method-level interceptors for lifecycle callbacks.

Appendix B Change Log

Editorial cleanup and conversion to standard template.

Assigned chapter numbers to sections and rearranged them for consistency.

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

Added Chapter 1 (Overview)

Added Chapter 3, derived from Chapter 9 of the CDI specification.

Removed deployment descriptors definitions

Rearranged various sections and examples for better flow.

Added examples with interceptor bindings to common sections.

Added standard `Priority` ranges

Added a note on a timeout method that is also a business method and around-timeout and around-invoke interceptors

Added the `AroundConstruct` lifecycle callback interceptor

Extended `InvocationContext` with `getConstructor` method; adjusted rules on the `InvocationContext.getTarget` return value, and `InvocationContext.proceed` result

Added notes to the around-invoke and around-timeout about the `throw Exception` clause in their signatures

Added a note that transaction context of interceptors may be changed by transactional interceptors in the invocation chain

Explicitly noted that around-invoke and around-timeout methods may be defined on superclasses of the target class or interceptor classes. However, only one method of each type may be defined on a given class.

Explained how interceptors are enabled

Added an option for lifecycle callback methods defined on interceptor classes to have the same signature as around-invoke and around-timeout methods

Added constructor-level interceptors that follow the same rules as method-level interceptors.

Changed `InvocationContext.getMethod` method to return null if there is no corresponding lifecycle callback method on the bean class and for the `AroundConstruct` lifecycle callback interceptor method.