
Oracle

JSR 318
Interceptors 1.2

Maintenance Lead:
Linda DeMichiel, Oracle

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

DRAFT

Specification: JSR-000318 Interceptors Specification ("Specification")**Version: 1.2 rev A****Status: Maintenance Release****Specification Lead: Oracle America, Inc. ("Specification Lead")****Release: February 18, 2016****Copyright 2016 Oracle America, Inc.****500 Oracle Parkway, Redwood City, California 94065, U.S.A.****All rights reserved.****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.

Table of Contents

Chapter 1	Overview.....	9
	1.1 Revision History.....	9
	1.2 Relationship to Other Specifications.....	10
	1.3 Document Conventions.....	10
Chapter 2	Interceptor Programming Contract.....	11
	2.1 Terminology.....	11
	2.2 Definition of Interceptor Classes and Interceptor Methods.....	11
	2.3 Interceptor Life Cycle.....	13
	2.3.1 Interceptor Environment.....	14
	2.4 InvocationContext.....	14
	2.5 Exceptions.....	16
	2.6 Business Method Interceptor Methods.....	16
	2.7 Interceptor Methods for Lifecycle Event Callbacks.....	17
	2.7.1 Exceptions.....	19
	2.8 Timeout Method Interceptor Methods.....	20
	2.9 Constructor- and Method-level Interceptors.....	21
	2.10 Default Interceptors.....	24
Chapter 3	Associating Interceptors with Classes and Methods using Interceptor Bindings.....	25
	3.1 Interceptor Binding Types.....	25
	3.1.1 Interceptor binding types with additional interceptor bindings.....	26
	3.1.2 Other sources of interceptor bindings.....	26
	3.2 Declaring the Interceptor Bindings of an Interceptor.....	26
	3.3 Binding an Interceptor to a Component.....	27
	3.4 Interceptor Resolution.....	28
	3.4.1 Interceptors with multiple bindings.....	29
	3.4.2 Interceptor binding types with members.....	29
Chapter 4	Associating Interceptors with Classes and Methods using the Interceptors Annotation.....	31
Chapter 5	Interceptor Ordering.....	33
	5.1 Enabling Interceptors.....	33
	5.2 Interceptor Ordering Rules.....	34
	5.2.1 Requirements for Extension Specifications.....	35
	5.2.2 Use of the Priority Annotation in Ordering Interceptors.....	36
	5.2.3 Examples.....	37
	5.3 Excluding Interceptors.....	37

Chapter 6	Related Documents	39
Appendix A	Change Log	41

Overview

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

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

1.1 Revision History

This document is an update to the Interceptors specification 1.2. Version 1.1 was based on the Interceptors chapter of the *Enterprise JavaBeansTM 3.0 (EJB)* specification (see [1]). Version 1.2 included interceptor binding definitions that were originally defined in the *Contexts and Dependency Injection for the Java EE Platform (CDI)* specification [3].

The change log for the current version is found in Appendix A.

1.2 Relationship to Other Specifications

The Java EE Platform specification requires support for interceptors. The use of interceptors defined by means of the `Interceptors` annotation is required to be supported for EJB and Managed Bean components, including in the absence of CDI. When CDI is enabled, the use of interceptors defined both by means of interceptor binding annotations and by means of the `Interceptors` annotation is required to be supported for component classes that support injection, as described in the section “Annotations and Injection” of the Java EE Platform specification [6].

Both the EJB and CDI specifications provide extensions to this specification. Other specifications may choose to do so in the future. Such specifications are referred to as extension specifications in this document. This document outlines permissible extensions to this specification and defines requirements for extension specifications.

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.

Interceptor Programming Contract

2.1 Terminology

The following terminology is used in this document:

Interceptor class: a class containing interceptor methods that is designed to be associated with a target class or method.

Interceptor: instance of an interceptor class.

Interceptor method: a method of an interceptor class or of a target class that is invoked to interpose on the invocation of a method of the target class, a constructor of the target class, a lifecycle event of the target class, or a timeout method of the target class.

2.2 Definition of Interceptor Classes and Interceptor Methods

An interceptor method for a target class may be declared in the target class, in an interceptor class associated with the target class, or in a superclass of the target class or interceptor class.

Any number of interceptor classes may be associated with a target class. See Chapter 5 for rules on interceptor ordering.

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

This specification defines the interceptor method types listed below. Extension specifications may define additional interceptor method types.

- Around-invoke interceptor methods (annotated with the `javax.interceptor.AroundInvoke` annotation). Around-invoke interceptor methods interpose on the invocation of business methods.
- Around-timeout interceptor methods (annotated with the `javax.interceptor.AroundTimeout` annotation). Around-timeout interceptor methods interpose on the invocation of timeout methods, in response to timer events.
- Post-construct interceptor methods (annotated with the `javax.annotation.PostConstruct` annotation). Post-construct interceptor methods are invoked after dependency injection has been completed on the target instance.
- Pre-destroy interceptor methods (annotated with the `javax.annotation.PreDestroy` annotation). Pre-destroy interceptor methods are invoked before the target instance and all interceptor instances associated with it are destroyed by the container.
- Around-construct interceptor methods (annotated with the `javax.interceptor.AroundConstruct` annotation). Around-construct interceptor methods interpose on the invocation of the constructor of the target instance.

Post-construct, pre-destroy, and around-construct interceptor methods are collectively referred to as *lifecycle callback interceptor methods*. Extension specifications may define additional lifecycle callback events and lifecycle callback interceptor method types.

Up to one interceptor method of each interceptor method type may be defined in the same class. More specifically, up to one around-invoke interceptor method, one around-timeout interceptor method, and one lifecycle callback interceptor method for each of the different lifecycle events may be defined in the same class. Only the interceptor methods of the interceptor class that are relevant for the given invocation context are invoked. For example, when a business method is invoked, around-invoke interceptor methods are invoked, but any around-construct, around-timeout, post-construct, or pre-destroy methods are ignored.

A single interceptor method may be defined to interpose on any combination of business methods, timeout methods, and lifecycle callback events.

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

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

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

2.3 Interceptor Life Cycle

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

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

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.

With the exception of around-construct lifecycle callback interceptor methods, no interceptor methods are invoked until after dependency injection has been completed on both the interceptor instances and the target instance^[1].

Post-construct interceptor methods for the target instance are invoked after dependency injection has been completed on the target instance.

Pre-destroy interceptor methods are invoked before the target instance and all interceptor instances associated with it are destroyed.^[2]

The following rules apply specifically to around-construct lifecycle callback interceptor methods:

- Around-construct lifecycle callback interceptor methods are invoked after dependency injection has been completed on the instances of all interceptor classes associated with the target class. Injection of the target component into interceptor instances that are invoked during the around-construct lifecycle callback is not supported.
- The target instance is created *after* the last interceptor method in the around-construct interceptor chain invokes the `InvocationContext.proceed` method. If the constructor for the target instance supports injection, such constructor injection is performed. If the `InvocationContext.proceed` method is not invoked by an interceptor method, the target instance will not be created.

[1] If the `PostConstruct` callback is declared in the interceptor class or a superclass of the interceptor class, this callback it is not invoked when the interceptor instance itself is created.

[2] If the `PreDestroy` callback is declared in the interceptor class or a superclass of the interceptor class, it is not invoked when the interceptor instance itself is destroyed.

- An around-construct interceptor method can access the constructed instance using the `InvocationContext.getTarget` method after the `InvocationContext.proceed` method completes.
- Dependency injection on the target instance other than constructor injection is not completed until after the invocations of all interceptor methods in the around-construct interceptor chain complete successfully. Around-construct lifecycle callback interceptor methods should therefore exercise caution when invoking methods of the target instance since dependency injection on the target instance will not have been completed.

2.3.1 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. Around-construct interceptor methods run in the same Java thread as the target constructor.

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 information 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.

2.4 InvocationContext

The `InvocationContext` object provides information 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 is passed to each interceptor method for a given target class method or lifecycle event interception.

The `InvocationContext` instance allows an interceptor method to save information in the `Map` obtained via the `getContextData` method. This information can subsequently be retrieved and/or updated by other interceptor methods in the invocation chain, and thus serves as a means to pass contextual data between interceptors. The contextual data is not sharable across separate target class method or lifecycle callback event invocations. The lifecycle of the `InvocationContext` instance is otherwise unspecified.

If interceptor methods 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 `getTarget` method returns the associated target instance. For around-construct lifecycle callback interceptor methods, `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 interceptor method types other than around-timeout interceptor methods.

The `getMethod` method returns the method of the target class for which the current interceptor method was invoked. The `getMethod` returns null in a lifecycle callback interceptor method for which there is no corresponding lifecycle callback method declared in the target class (or inherited from a superclass) or in an around-construct lifecycle callback interceptor method.

The `getConstructor` method returns the constructor of the target class for which the current around-construct interceptor method was invoked. The `getConstructor` method returns null for interceptor method types other than around-construct interceptor methods.

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 invocation of the target class method or constructor. 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 to the `setParameters` call.

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 around-construct 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—in the case of around-construct interceptor methods—the target instance will not be created. The `proceed` method returns the result of the next method invoked.

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

If a method is of type `void`, the invocation of the `proceed` method returns `null`. For around-construct lifecycle callback interceptor methods, 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 lifecycle callback interceptor 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.

2.5 Exceptions

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

OPEN ISSUE: Should the above statement be extended to constructors as well? I.e., should the specification state the following?: Interceptor methods are allowed to throw runtime exceptions or any checked exceptions that the associated target method or constructor allows within its `throws` clause.

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 throws the same exception as any thrown by the associated target method unless an interceptor method further down the Java call stack has caught it and thrown a different exception or suppressed the exception. Exceptions and initialization and/or cleanup operations should typically be handled in `try/catch/finally` blocks around the `proceed` method.

2.6 Business Method Interceptor Methods

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

Around-invoke methods may be declared in interceptor classes, in the superclasses of interceptor classes, in the target class, and/or in superclasses of the target class. However, only one around-invoke method may be declared in 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
```

[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.

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 interceptor methods in the invocation chain, such as those defined by the *Java Transaction API* specification [7].

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)
        throws Exception {

        //... log invocation data ...

        return ctx.proceed();
    }
}

@Monitored
public class ShoppingCart {

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

2.7 Interceptor Methods for Lifecycle Event Callbacks

The `AroundConstruct` annotation specifies a lifecycle callback interceptor method that interposes on the invocation of the target instance's constructor.

The `PostConstruct` annotation specifies a lifecycle callback interceptor method that is 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, such as a timer event, is invoked on the target instance.

The `PreDestroy` annotation specifies a lifecycle callback interceptor method that interposes on the target instance's removal by the container.

Extension specifications are permitted to define additional lifecycle events and lifecycle callback interceptor methods types.

Around-construct interceptor methods may be only declared in interceptor classes and/or superclasses of interceptor classes. Around-construct interceptor methods must not be declared in the target class or in its superclasses.

All other lifecycle callback interceptor methods can be declared in an interceptor class, superclass of an interceptor class, in the target class, and/or in a superclass of the target class.

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

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 methods declared in a given class.

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].

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`.

Lifecycle callback interceptor methods declared in an interceptor class or superclass of an interceptor class must have one of the following signatures:

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

Note: A lifecycle callback interceptor method 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 such an interceptor method returns a value, the value is ignored by the container when the method is invoked to interpose on a lifecycle event.

Lifecycle callback interceptor methods declared in a target class or in a superclass of a target class must have the following signature:

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

[5] In general, 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 lifecycle callback interceptor methods (see the *Enterprise JavaBeans™* specification [2]). The transaction context may be also changed by transactional interceptors in the invocation chain.

The following example declares lifecycle callback interceptor methods in both the interceptor class and the target class. Rules for interceptor ordering are described in Chapter 5.

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

2.7.1 Exceptions

Lifecycle callback interceptor methods (when invoked to interpose on lifecycle events) may throw runtime exceptions, but not checked exceptions.

OPEN ISSUE: What if an around-construct method intercepts a constructor that throws a checked exception? Is the around-construct interceptor method allowed to throw checked exceptions? Is it allowed to catch/ignore them?

In addition to the rules specified in section 2.5, the following rules apply to the lifecycle callback interceptor methods:

- A lifecycle callback interceptor method declared in an interceptor class or in a superclass of an interceptor class may catch an exception thrown by another lifecycle callback interceptor method in the invocation chain, and clean up before returning.
- Pre-destroy interceptor methods 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.

2.8 Timeout Method Interceptor Methods

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

NOTE: Timeout methods are currently specific to Enterprise JavaBeans, although Timer Service functionality may be extended to other specifications in the future. 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 timer callbacks registered with the Timer Service are called timeout methods.

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

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();  
    }  
}
```

2.9 Constructor- and Method-level Interceptors

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

For example, by using a method-level interceptor, an around-invoke interceptor method may be applied only to a specific business method of the target class, independent of the other methods of the target class. Likewise, an around-timeout interceptor method may be applied only to a specific timeout method on the target class, independent of the other timeout methods of the target class.

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

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

If a method-level interceptor is applied to more than one method of a associated target class this 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 the `placeOrder` method will be monitored:

```
public class ShoppingCart {  
    @Monitored  
    public void placeOrder() {...}  
}
```

In the following example, the `MyInterceptor` interceptor is applied to a subset of the business methods of the session bean. Note that the `created` and `removed` methods of the `MyInterceptor` interceptor will not be invoked:

```
public class MyInterceptor {  
    ...  
    @AroundInvoke  
    public Object around_invoke(InvocationContext ctx)  
        throws Exception {...}  
  
    @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, the `ValidationInterceptor` interceptor interposes on the bean constructor only, and the `validateMethod` interceptor method will not be invoked:

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

@ValidateSpecial
public class ValidationInterceptor {

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

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

}

public class SomeBean {

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

    public void someMethod() {
        ...
    }

}
```

In the following example, the `validateConstructor` method of the `ValidationInterceptor` interceptor interposes on the bean constructor, and the `validateMethod` method of the interceptor interposes on the `anotherMethod` business method of the bean.

```
public class SomeBean {

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

    public void someMethod() {
        ...
    }

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

}
```

2.10 Default Interceptors

Default interceptors are interceptors that apply to a set of target classes. An extension specification may support the use of a deployment descriptor to define default interceptors and their relative ordering.

OPEN ISSUE: What about annotations to define default interceptors? I.e., should we state the following?: An extension specification may support the use of a deployment descriptor or annotations to define default interceptors and their relative ordering.

Chapter 3

Associating Interceptors with Classes and Methods using Interceptor Bindings

Interceptor bindings are intermediate annotations that may be used to associate interceptors with any component that is not itself an interceptor or decorator [6].

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 subset of its 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 by CDI stereotypes.

3.2 Declaring the Interceptor Bindings of an Interceptor

The interceptor bindings of an interceptor are specified by annotating the interceptor class with the interceptor 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.

If an interceptor does not declare an `Interceptor` annotation, it can be bound to components using the `Interceptors` annotation.

An extension specification may define other ways of declaring an interceptor and binding an interceptor to a component, such as by means of a deployment descriptor.

An interceptor 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 around-construct lifecycle callback interceptors, interceptors for lifecycle callbacks may only declare interceptor binding types that are defined as `Target (TYPE)`.

OPEN ISSUE: What if that same interceptor contains an around-invoke method? Does that same restriction apply? (It seems too restrictive.)

3.3 Binding an Interceptor to a Component

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

In the following example, the `MonitoringInterceptor` is applied to the target class. It will therefore apply to all business methods of the class.

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

In this example, the `MonitoringInterceptor` is applied to a single method:

```
public class ShoppingCart {
    @Monitored
    public void placeOrder() { ... }
}
```

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

The set of interceptor bindings for a method or constructor are those applied to the target class combined with those applied at method level or constructor level. Note that the interceptor bindings applied to the target class may include those inherited from its superclasses. The CDI specification rules for the inheritance of type-level metadata apply to the inheritance of interceptor bindings from superclasses of the target class. See [3].

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].

OPEN ISSUE: What is meant by “of the same type”? Does this include annotation members? Need to clarify.

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

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

If a 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, and causes deployment to fail.

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, and causes deployment to fail.

3.4 Interceptor Resolution

The process of matching interceptors to a given business method, timeout method, or lifecycle event of a component is called *interceptor resolution*.

For a lifecycle event other than instance construction, 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 event or method.
- The interceptor is enabled. An interceptor is enabled if the `Priority` annotation is applied to the interceptor class^[8]. An extension specification may define other means of enabling interceptors. For example, the CDI specification enables an interceptor if the interceptor class is listed under the `<interceptors>` element of the `beans.xml` file for the bean archive.

[8] The `Priority` annotation also orders interceptors. See Chapter 5.

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 CDI stereotypes [3] 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, and causes deployment to fail.

Chapter 4

Associating Interceptors with Classes and Methods 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.

The `Interceptors` annotation can be applied to the target class or to a method or a constructor declared in the target class or in 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 around-invoke or around-timeout interceptor methods are to be invoked.
OPEN ISSUE: The above point requires clarification with regard to applying the `Interceptors` annotation at constructor-level to a superclass constructor. What should happen when an instance of the target class is created? Should the superclass interceptor be invoked or not?
- Constructor-level interceptors can be defined by applying the `Interceptors` annotation to the constructor for which the around-construct interceptor methods are to be invoked.

Constructor- and method-level interceptors are invoked in addition to any interceptors declared in the target class, in an interceptor class associated with the target class, or in a superclass of the target class or interceptor class, and in addition to any default interceptors (if supported).

If multiple interceptor classes are specified in the `Interceptors` annotation, the interceptor methods of these classes are invoked in the order in which the classes are specified. The ordering rules for interceptors are defined in Chapter 5.

The `Interceptor` annotation is ignored on classes bound using the `Interceptors` annotation.

An extension specification may support the use of a deployment descriptor to associate interceptor classes with a target class, and/or method or constructor of a target class, and to specify the order of interceptor invocation or override metadata specified by annotations.

In the following example, the `around-invoke` methods specified by both the `MyInterceptor` and the `MyOtherInterceptor` classes will be invoked when the `otherMethod` method is called. The rules for ordering these interceptors are defined in Chapter 5.

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

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


Interceptor Ordering

5.1 Enabling Interceptors

Only interceptors that are enabled are eligible to be invoked.

Interceptors declared using interceptor bindings are enabled using the `Priority` annotation (see Section 5.2.2). The `Priority` annotation also controls interceptor ordering (see Section 5.2).

Interceptors declared using the `Interceptors` annotation are enabled by that annotation. Using the `Interceptors` annotation to associate interceptor classes with a target class or a method or constructor of a target class enables them for that target class, method, or constructor. The order in which the interceptor classes are specified in the `Interceptors` annotation controls interceptor ordering (see Section 5.2). Interceptor methods declared in the target class or in a superclass of the target class are enabled unless overridden.

An extension specification may define alternative mechanisms (e.g., a deployment descriptor such as the CDI `beans.xml` deployment descriptor [3] or the EJB `ejb-jar.xml` deployment descriptor [2]) to enable and order interceptors or to override the order specified by means of annotations.

Note: The `InvocationContext` object allows 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. See Section 2.4.

5.2 Interceptor Ordering Rules

For each interceptor method type (i.e., around-invoke, around-timeout, post-construct, etc.), the following interceptor invocation ordering rules apply, except as specified otherwise by an extension specification. See Section 5.2.1 for permissible exceptions to these rules on the part of extension specifications.

- Default interceptors are invoked first.
 - Default interceptors are invoked in the order defined by the extension specification (e.g., by their order in the deployment descriptor).
 - If a default interceptor class has superclasses, interceptor methods declared in the interceptor class's superclasses are invoked before the interceptor method declared in the interceptor class itself, most general superclass first.
- Interceptors declared by applying the `Interceptors` annotation *at class-level* are invoked next.
 - **OPEN ISSUE:** What happens if a superclass of the target class specifies the `Interceptors` annotation? Should those interceptors also be run? Previous versions of this specification did not state so, and the RI is inconsistent in its treatment of this case.
 - Interceptor methods declared in the interceptor classes listed in the `Interceptors` annotation are invoked in the same order as the specification of the interceptor classes in that annotation.
 - If an interceptor class declared by applying the `Interceptors` annotation *at class-level* has superclasses, interceptor methods declared in the interceptor class's superclasses are invoked before the interceptor method declared in the interceptor class itself, most general superclass first.
- Interceptors declared by applying the `Interceptors` annotation *at method- or constructor-level* are invoked next.
 - Interceptor methods declared in the interceptor classes listed in the `Interceptors` annotation are invoked in the same order as the specification of the interceptor classes in that annotation.
 - If an interceptor class declared by applying the `Interceptors` annotation *at method- or constructor-level* has superclasses, interceptor methods declared in the interceptor class's superclasses are invoked before the interceptor method declared in the interceptor class itself, most general superclass first.
- Interceptors declared using interceptor bindings are invoked next.
 - All interceptors specified using interceptor binding annotations visible on the target class (e.g., specified on the class or visible on the class because they were declared with the `Inherited` annotation) are combined with all binding annotations on the target method and sorted by the priorities specified by the `Priority` annotation; and then the interceptor methods are invoked in order of priority. The `Priority` annotation is described in Section 5.2.2.

- If an interceptor class declared using interceptor bindings has superclasses, interceptor methods declared in the interceptor class's superclasses are invoked before the interceptor method declared in the interceptor class itself, most general superclass first.
- Interceptor methods declared in the target class or in any superclass of the target class are invoked last.
 - If the target class has superclasses, interceptor methods declared in the target class's superclasses are invoked before an interceptor method declared in the target class itself, most general superclass first.

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

5.2.1 Requirements for Extension Specifications

In general, extension specifications are permitted to override the default ordering rules specified in Section 5.2—for example, by means of deployment descriptors. However, the following rules must be observed by all extension specifications.

OPEN ISSUE: The following rules are currently observed by both the EJB and the CDI specifications for interceptors defined/enabled by deployment descriptors. Should they be *required* to apply to extension specifications?

- If an interceptor class has superclasses, interceptor methods declared in the interceptor class's superclasses are invoked before the interceptor method declared in the interceptor class itself, most general superclass first. There must be no interleaving of other methods (interceptor or otherwise) in this ordering.
- If the target class has superclasses, interceptor methods declared in the target class's superclasses are invoked before an interceptor method declared in the target class itself, most general superclass first. There must be no interleaving of other methods (interceptor or otherwise) in this ordering.
- The interceptor methods declared in the target class and its superclasses are invoked after all other interceptor methods.

5.2.2 Use of the Priority Annotation in Ordering Interceptors

The `Priority` annotation can be used to enable and order interceptors associated with components that use interceptor bindings. The required `value` element of the `Priority` annotation determines the ordering. Interceptors with smaller priority values are called first. If more than one interceptor has the same priority, the relative order of those interceptors 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.

The following priority values are defined for interceptor ordering when used with the `Priority` annotation. Interceptors with lower priority values are invoked earlier in the interceptor chain.

- `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, but after interceptors in the range up until `LIBRARY_BEFORE` 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 { ... }
```

5.2.3 Examples

TO DO: MORE COMPLEX EXAMPLES TO GO HERE

5.3 Excluding Interceptors

Interceptors may be excluded from execution by means of the `ExcludeClassInterceptors` annotation and the `ExcludeDefaultInterceptors` annotation.

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

OPEN ISSUE: The intent here needs to be clarified. The javadocs seem to imply that `ExcludeClassInterceptors` is not restricted to interceptors defined via `Interceptors`.

The `ExcludeDefaultInterceptors` annotation can be used to exclude the invocation of default interceptors for a target class or—when applied to a target class constructor or method—to exclude the invocation of default interceptors for a particular constructor or method.

An extension specification may define other means for excluding interceptors from execution, such as by means of a deployment descriptor.

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() {
        ...
    }
}
```

In the following example only the interceptor `MyInterceptor` will be invoked when `someMethod` is called. The `ExcludeClassInterceptors` annotation is used to exclude the invocation of the class-level interceptors.

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

In the next example, only the interceptor `MyInterceptor` will be invoked when `someMethod` is called. The `ExcludeDefaultInterceptors` annotation is used to exclude the invocation of the default interceptors (if any).

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

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>.
- [6] Java Platform, Enterprise Edition (Java EE), v7. <http://jcp.org/en/jsr/detail?id=342>.
- [7] Java™ Transaction API (JTA), version 1.2. <http://jcp.org/en/jst/detail?=907>.

Appendix A

Change Log

Clarified Section 1.2 to be consistent with the Java EE Platform specification with regard to when interceptors defined by means of the `Interceptors` annotation and interceptors defined by means of interceptor bindings are required to be supported.

Clarified terminology in sections 2.1 and 2.2.

Noted that around-construct interceptors run in the same thread as the target constructor in section 2.3.1.

Reworded to indicate that deployment descriptors are specific to extension specifications.

Flagged open issues in sections 2.5, 2.7.1, 2.10, 3.2, 4, 5.2.1, 5.3.

Combined interceptor ordering rules into a single algorithm in section 5.2.

Added section 5.3 on excluding interceptors.

Made miscellaneous editorial changes.

