JAKARTA EE

# Jakarta Interceptors

Jakarta Interceptors Team, https://projects.eclipse.org/projects/ee4j.interceptors

2.2, November 29, 2023: DRAFT

# Table of Contents

Specification: Jakarta Interceptors

Version: 2.2

Status: DRAFT

Release: November 29, 2023

# Copyright



# Eclipse Foundation Specification License



## Disclaimers

# Chapter 1. Overview

## 1.1. Specification Scope

Jakarta Interceptors defines a means of interposing on business method invocations and specific events—such as lifecycle events and timeout events—that occur on instances of Jakarta 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.

## 1.2. Revision History

This document is an update to the Jakarta Interceptors specification 2.0. Version 2.0 was part of the Jakarta EE 9 [6] javax to jakarta namespace change. Version 1.1 was based on the Interceptors chapter of the Enterprise JavaBeans™ 3.0 specification [1]. Version 1.2 included interceptor binding definitions that were originally defined in the Contexts and Dependency Injection for the Jakarta EE Platform (CDI) specification [3].

The change log for the current version is found in Change Log.

## 1.3. Relationship to Other Specifications

The Jakarta EE Platform specification requires support for interceptors. The use of interceptors defined by means of the `Interceptors` annotation is required to be supported for Jakarta Enterprise Beans 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 Jakarta EE Platform specification [10].

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

## 1.4. 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.

Java code and sample data fragments are formatted as shown in figure [1.1]:

```
package com.example.hello;

public class Hello {
```

```
    public static void main(String args[]) {
    System.out.println("Hello World");
    }
}
```

# Chapter 2. 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, method, or constructor.
- `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.
- `Business method` : a public method of an application component that is invoked by a Jakarta EE container/framework, e.g., Jakarta EJB, Jakarta CDI bean, Jakarta RESTful endpoint, etc.

## 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 Interceptor Ordering 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 `jakarta.interceptor.AroundInvoke` annotation). Around-invoke interceptor methods interpose on the invocation of business methods.
- Around-timeout interceptor methods (annotated with the `jakarta.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 `jakarta.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 `jakarta.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

`jakarta.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 Associating Interceptors with Classes and Methods using Interceptor Bindings) or the *jakarta.interceptor.Interceptors* annotation (see Associating Interceptors with Classes and Methods using the Interceptors Annotation). 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 Jakarta Enterprise Beans specification [2] requires support for the `ejb-jar.xml` deployment descriptor and the CDI specification [3,8] requires support 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 aroundConstruct lifecycle callback interceptor methods, no interceptor methods are invoked until after dependency injection has been completed on both the interceptor

instances and the target [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.
- 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. Invocation Context

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 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 `proceed` method in 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. 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.

The `getInterceptorBindings` method returns the set of interceptor binding annotations that were used to associate interceptors with the target instance that is being intercepted. The zero-parameter variant returns all interceptor binding annotations, while the variant with a `Class` parameter returns only interceptor binding annotations of given type.

The `getInterceptorBinding` method returns the single interceptor binding annotation of given type that was used to associate interceptors with the target instance that is being intercepted.

## 2.5. Exceptions

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)
```

*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, for example, 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 `Jakarta 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) {
        //... 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 or `final` . A lifecycle callback interceptor method must not be declared as `static` except in an application client.

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)
```

*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>()
```

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 Interceptor Ordering.

```
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

When invoked to interpose on lifecycle events, lifecycle callback interceptor methods may throw runtime exceptions, but—except for around-construct methods—may not throw checked exceptions.

In addition to the rules specified in section 2.5 Exceptions, 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 Jakarta Enterprise Beans, although Timer Service functionality may be extended to other specifications in the future, and extension specifications may define events that may be interposed on by around-timeout methods. The enterprise beans Timer Service, defined by the Jakarta Enterprise Beans 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)
```

*Note: An around-timeout interceptor method should 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.

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());
        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, 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—by using a method-level interceptor. 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) { ... }

     @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 or annotations to define default interceptors and their relative ordering.

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

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

[3] If the last parameter is a vararg parameter of type T, it is considered be equivalent to a parameter of type T{opening-bracket}{closing-bracket}.

[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.

[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 Jakarta Enterprise Beans specification [2]). The transaction context may be also changed by transactional interceptors in the invocation chain.

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

# 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 see [10].

## 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.

## 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 [8].

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 [1].

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 [8,3] adds the `jakarta.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 [2]. 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.

### 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 [3,8] adds the `jakarta.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 [3,8] 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, treats it as a definition error, and causes deployment to fail.

[1] This requirement follows the rules from the Jakarta Annotations specification, section 3.1 [9[.

[2] The Priority annotation also orders interceptors. See Chapter 5 Interceptor Ordering.

# 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.
- 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 Interceptor Ordering.

The `Interceptor` annotation is ignored during the processing of classes bound using the `Interceptors` annotation. It will continue to be observed on such classes when used in the context of interceptor binding.

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 Interceptor Ordering.

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

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

```
        ...
    }
}
```

# Chapter 5. 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 Use of the Priority Annotation in Ordering Interceptors). The *Priority* annotation also controls interceptor ordering (see Section 5.2 Interceptor Ordering Rules).

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 Ordering Rules). 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 [3,8] or the Jakarta Enterprise Beans *ejb-jar.xml* deployment descriptor [2]) to enable and order interceptors, to override the order specified by means of annotations, or to disable interceptors.

*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 Invocation Context.*

## 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.

- 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* to the target class 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 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.

This specification does not define the semantics of applying the Interceptors annotation to a superclass of the target class, and thus the corresponding interceptor methods may or may not be invoked. Applications that specify the Interceptors annotation on a superclass of the target class will not be portable.

- 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 Use of the Priority Annotation in Ordering Interceptors.
- 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. 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 when computing the invocation order of 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 Jakarta 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 Jakarta 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 { ... }
```

## 5.3. 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.

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() {
        ...
    }
}
```

# Appendix A: Bibliography


- Enterprise JavaBeansTM, version 3.0 . http://jcp.org/en/jsr/detail?id=220.
- Jakarta™ Enterprise Beans, version 4.0 . https://jakarta.ee/specifications/enterprise-beans/4.0/.
- Jakarta™ Contexts and Dependency Injection (CDI specification) version 3.0 . https://jakarta.ee/specifications/cdi/3.0/ .
- Jakarta™ XML Web Services version 3.0 . https://jakarta.ee/specifications/xml-web-services/3.0/ .
- Jakarta™ Annotations version 2.0 . https://jakarta.ee/specifications/annotations/2.0/.
- Jakarta™ EE Platform 9 . https://jakarta.ee/specifications/platform/9/.
- Jakarta™ Transaction version 2.0 . https://jakarta.ee/specifications/transactions/2.0/ .
- Jakarta™ Contexts and Dependency Injection (CDI specification) version 4.0 . https://jakarta.ee/specifications/cdi/4.0/ .
- Jakarta™ Annotations version 2.1 . https://jakarta.ee/specifications/annotations/2.1/.
- Jakarta™ EE Platform 10 . https://jakarta.ee/specifications/platform/10/ .

# Appendix B: Change Log

## B.1. Changes for 2.1

Updated dependencies for Jakarta EE 10.

Added JPMS module-info.

## B.2. Changes for 2.0

Clarified Relationship to Other Specifications to be consistent with the Jakarta 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 Terminology and Definition of Interceptor Classes and Interceptor Methods.

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

Clarified that around-construct interceptor methods may throw checked exceptions.

Clarified distinction between core requirements and the latitude available to extension specifications.

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

Clarified that interceptor binding may not be used to associate interceptors with decorators.

Corrected bug in section Interceptor binding types with additional interceptor bindings: An interceptor binding type can only be applied to an interceptor binding type defining a subset of its target types.

Removed inconsistency whereby only around-construct lifecycle callback interceptors could declare interceptor binding types defined other than as *Target(TYPE)* .

Clarified when *Priority* annotation is ignored.

Added section Enabling Interceptors to Interceptor Ordering to centralize existing requirements on enabling interceptors and separate concept of the enabling of interceptors from the ordering of interceptors.

Combined interceptor ordering rules into a single algorithm in section Interceptor Ordering Rules.

Factored out section Excluding Interceptors on excluding interceptors.

Clarified that *ExcludeClassInterceptors* applies only to interceptors defined by means of the *Interceptors* annotation.

Made numerous editorial cleanup changes, and reorganized document for clarity.